Il "Bibbione di Glk"

Lorenzo Marcantonio

24 agosto 2007

Tratto e tradotto da "*The Glk API Specifications 0.7.0*" di
Andrew Plotkin
con contribuzione da Autori Vari

Indice

Parte 1.

Solo Glk

Introduzione

❖ ❖ ❖

La parte prima contiene sostanzialmente la traduzione delle *Glk API Specifications 0.7.0* in lingua italiana. Contiene anche miei commenti personali riguardo la loro attuale interpretazione e il loro utilizzo, in particolar modo in congiunzione con l'implementazione denominata garglk-mod (o Gargoyle-mod).

Tutto il materiale tradotto rimane ovviamente copyright di Andrew Plotkin (per la prima parte). Gli errori di traduzioni sono miei (a dire il vero nel documento originale *ci sono* degli errori... *quelli* sono static corretti!).

Le sezioni successive sono invece frutto di ricerca e integrazione in base alla varia documentazione e ai tutorial reperibili in rete.

Le fonti sono in particolare la documentazione di `infglk` di Marnie Parker, *Gull* di Adam Cadre, *Inform Glk/glulx for Dunces* di Doe, alcuni articoli residuati da *Just Enough Glulx* di Roger Firth (il testo originale è stato discontinuato) e anche la *Game Author's Guide to Glulx Inform* dello stesso Plotkin (che a quanto pare sembra inventare tutto ciò che gravita attorno ad Inform...).

Le versioni utilizzate come riferimento durante la stesura sono:

- Inform 6.31, con target glulx, ovviamente. Inform utilizzato per generare Z-code non è in grado di sfruttare al meglio Glk, anche se in congiunzione con un interprete Glk, in quanto vincolato alle specifiche della Z-machine. Quindi, per intendersi, solo barra di stato superiore e finestra della storia. Nulla di più...
- La libreria standard Inform 6/11 (bipiattaforma). Si può ormai presumere che eventuali revisioni rimangano compatibili, dato che lo sviluppo attuale sembra unicamente rivolto verso Inform 7.
- infglk 0.7.0 (aggiornato per Glk 0.7). Anche se specifico per Glk 0.7 si può utilizzare senza problemi per sviluppare applicazioni (giochi) per Glk 0.6.1 (la

versione precedente senza supporto Unicode, vedere il commento al Capitolo 2).

- Altre librerie che verranno menzionate mano a mano, in genere all'ultima versione disponibile al momento della stesura del testo.

Se non fosse ancora chiaro le annotazioni sono separate dal testo principale da queste simpatiche cornici (con un carattere più piccolo). Nella seconda parte queste annotazioni indicano sezioni relative a dettagli secondari o di utilizzo avanzato.

◆ ◆ ◆

Cosè Glk

Glk è un tentativo di definire una API (interfaccia di programmazione) portabile per applicazioni con interfacce utente a testo.

Piuttosto che spiegare dettagliatamente il significato di tutto ciò, si guardino le tipiche Avventure Testuali. TADS e la Z-machine di Infocom hanno funzionalità di interfaccia praticamente identiche; entrambe permettono ad un programma di:

- Visualizzare una quantità indefinita di testo in un buffer su schermo permettendo di controllarne lo stile;
- Richiedere all'utente una linea di testo;
- Visualizzare alcune linee di testo in una (piccola) finestra separata;
- Memorizzare informazioni in un file o leggerne.

In ogni caso, l'implementazione di queste funzionalità varia largamente tra piattaforme e sistemi operativi differenti. Queste variazioni sono inoltre invisibili al programma (l'interprete del gioco). Al gioco non interessa se l'output è visualizzato attraverso un emulatore di terminale a caratteri oppure in una finestra; e nemmeno se l'input utilizza l'editing del mouse come sul Mac oppure dei tasti di controllo come EMACS.

D'altra parte, l'utente è probabilmente estremamente interessato a tutte queste particolarità dell'interfaccia. Questo è il motivo per cui ci sono interpreti nativi per Macintosh, interpreti controllati con lo stilo sui palmari Newton e Palm e così via; d'altra parte questo è il motivo per cui sono state ideate tutte queste piattaforme differenti in primo luogo.

Visti da *ancora* un altro punto di vista, TADS e Inform non sono soli; esiste storicamente un gran numero di sistemi per Avventure Testuali. Di questi, la maggior parte sono obsoleti o addirittura deceduti; ma è inevitabile che ne appaiano altri. Gli utenti vogliono che ogni sistema esistente venga portato sulla piattaforma da loro in uso; preferirebbero inoltre che tutti

questi sistemi utilizzino una interfaccia il più simile possibile a quella a cui sono abituati.

Tutto ciò ha come risultato una enorme mole di lavoro.

Glk tenta di separare tutto ciò che in una Avventura Testuale è identico in ogni sistema (ma differente su differenti sistemi operativi) dalle parti uniche ai sistemi (ma identiche in tutti i sistemi operativi). Il confine fra questi due mondi sono le API Glk.

La speranza è che un nuovo sistema per Interactive Fiction, e anche quelli esistenti che sono meno supportati (Hugo, AGT, ecc.) possano essere scritti utilizzando Glk per tutte le funzioni di input/output. Il sistema sarebbe quindi scritto in C *veramente* portabile. Dall'altra parte ci sarebbe una libreria Glk per ogni sistema operativo e interfaccia (Macintosh, X-windows, terminali curses, ecc.). Portare il sistema su ogni piattaforma diventerebbe banale; basterebbe ricompilare il sistema con la libreria corretta.

Glk può anche servire come interfaccia per applicazioni che non siano necessariamente giochi: macinatori di dati, programmini di comodo o qualunque altra cosa che normalmente mancherebbe delle comodità tipiche dell'ambiente operativo della macchina.

E questa non sarebbe una Virtual Machine?

Potete pensare a Glk come ad una virtual machine, senza la parte della virtual machine. La *macchina*" è semplicemente codice C portabile.

❖ ❖ ❖

Sinceramente non ho mai capito come si possa confondere Glk con una virtual machine... Personalmente definirei Glk come:

> "Una libreria multi-piattaforma che fornisce servizi di input/output e di supporto, ottimizzata in particolar modo per le virtual machine e il tipo di operazioni tipiche dell'Interactive Fiction."

◆ ◆ ◆

È stata progettata appositamente anche una virtual machine nata per funzionare con Glk. Questa VM, chiamata glulx, usa Glk come interfaccia; ogni chiamata a Glk corrisponde ad un opcode di input/output della VM.

Ovviamente, Glk può essere utilizzato anche con altri sistemi per Interactive Fiction. Il vantaggio di glulx è quello di fornire all'autore l'accesso diretto e completo alle API Glk. Altri sistemi tipicamente utilizzano una astrazione dell'I/O, che si può applicare solo in parte a Glk. Per questi sistemi, Glk tende a fornire il minimo comune denominatore: altamente portabile, ma non necessariamente con tutte le funzionalità (nonostante la presenza di

una funzione in Glk, potrebbe non essere disponibile attraverso l'astrazione fornita dal sistema).

Cosa non fa Glk?

Glk non gestisce quelle cose che dovrebbero essere gestite dal programma (o dal sistema IF o dalla macchina virtuale) che è compilato con Glk. Questo significa che Glk non contiene parti rilevanti a:

- Parsing
- Memorizzazione degli oggetti di gioco
- Computazioni
- Compressione del testo

Convenzioni utilizzate in questo documento

Questo documento definisce le API Glk. Qui si tenta di specificare esattamente tutto ciò che viene fatto, quel che è legale, quel che non lo è e il perché di tutto ciò.

Il destinatario principale è il programmatore di giochi: la persona che intende utilizzare Glk per visualizzare testo, accettare tasti e così via. Specificando quello che fa la libreria Glk, però, il documento definisce anche il lavoro del programmatore Glk, ovver la persona che vuole scrivere una implementazione della libreria Glk per una nuova piattaforma o sistema operativo.

Se la libreria Glk garantisce qualcosa, il programmatore del gioco vi può fare affidamento e il programmatore Glk è obbligato a supportarla. D'altra parte, se la libreria *non* garantisce qualcosa, il programmatore Glk può gestirla nel modo che preferisce e il programmatore del gioco non può farvi affidamento. Se qualcosa è illegale, il programmatore del gioco non può farlo e il programmatore Glk non se ne deve preoccupare.[1]

❖ ❖ ❖

L'implementazione in garglk-mod nella stragrande maggioranza dei casi effettua dei controlli sommari sui parametri passati. Sfortunatamente *non* è una libreria

[1] È preferibile, anche se non richiesto, che la libreria Glk rilevi le richieste illegali e mostri messaggi di errore appropriati. Ma potrebbe benissimo bloccarsi atrocemente quando il programma fa qualcosa di illegale. Ed è per questo che il programmatore del gioco non deve farlo.

a prova di bomba. Con abbastanza impegno e buona volontà (ma forse anche con molto meno) è possibile farla crashare in modo indecoroso.

D'altra parte gli errori più probabili e/o semplici da verificare sono gestiti.

Sfortunatamente sono in circolazione alcune implementazioni Glk non del tutto conformi e, ancora più sfortunatamente, esistono giochi che *sfruttano* queste non-conformità (generalmente in modo benigno).

Una implementazione troppo fiscale delle specifiche rischia quindi di rivelarsi nociva per la compatibilità con il software pre-esistente.

◆ ◆ ◆

In seguito, "Glk" o "la libreria" si riferirà alla libreria Glk, mentre "il programma" sarà il gioco (o che altro) che utilizza la libreria Glk per mostrare testo, accettare testo o altro. "Noi" siamo quelli che scrivono il programma. "Il giocatore" è la persona che utilizzerà la combinazione programma/Glk per giocare (o chissà che altro).

❖ ❖ ❖

Aggiungo inoltre che con "autore" intendo il programmatore Inform/glulx che vuole scrivere un gioco utilizzando Glk.

◆ ◆ ◆

Le API Glk sono dichiarate in un header C chiamato `glk.h`. È conveniente tenerlo sotto mano mentre si legge questo documento.

❖ ❖ ❖

Il file `glk.h` di garglk *non è* identico agli altri. Contiene infatti alcune estensioni alla libreria Glk. La possibilità di estendere in modo privato la libreria è, fra l'altro, *esplicitamente* concessa! Vedere la sottosezione 10.2.4.

Per quanto riguarda l'utilizzo in Inform un primo e *indispensabile* aiuto viene da `infglk` (scaricabile liberamente da if-archive).

Sebbene sia solo un semplice wrapper intorno all'opcode `@glk` di glulx, permette di migliorare enormemente la leggibilità (e anche la scrittura) del codice; per esempio la chiamata:

```
glk($0044, gg_commandstr, 0);
```

(una delle prime chiamate a Glk, nella libreria standard Inform; sia in `parserm.h` che in `verblibm.h` le chiamate Glk sono *tutte* in questo formato, se non peggio). Utilizzando `infglk` è semplicemente possibile scrivere:

```
glk_stream_close(gg_commandstr);
```

Incidentalmente, `$0044` è il selettore di dispatch della funzione (vedere il Capitolo 11 e in particolare la sezione 11.6), mentre (grazie alle regole relative al passaggio dei parametri in Inform) è possibile omettere gli ultimi parametri finché valgono `0`.

A `infglk` è comunque riservato un capitolo a parte (Capitolo 13). Alcuni autori coraggiosi però hanno avuto il coraggio di scrivere *intere librerie* soltanto utilizzando i codici di dispatch in esadecimale!

◆ ◆ ◆

Crediti

Glk è frutto del lavoro di molte persone per molti anni. Se provassi ad elencare tutti coloro che mi hanno offerto commenti e suggerimenti, impallidirei immediatamente, dimenticherei il nome di tutti e diventerei una creatura muta ed eremita che abita in un tunnel ferroviario sotto ad Oakland. Ma devo ringraziare queste persone che hanno scritto le librerie Glk e i sistemi di link: Matt Kimball, Ross Raszewsi, David Kinder, John Elliott, Joe Mason, Stark Springs, e, ehm, quelli di cui mi sono dimenticato.

Evin Robertson scrisse la proposta originale per le funzioni Glk Unicode che ho importato quasi alla lettera in questo documento. Grazie.

1 Struttura generale

1.1. La funzione main del programma

La funzione principale del programma (quella che in C è nota come `main()`) fa parte di Glk.[1]

È obbligatorio definire una funzione chiamata `glk_main()`, che la libreria chiama per iniziare l'esecuzione effettiva del programma. `glk_main()` deve rimanere in esecuzione fino al termine del programma e poi ritornare.

❖ ❖ ❖

Per gli autori (che molto probabilmente saranno interessati solo a Inform/glulx) la parte relativa a `glk_main()` non è di alcun interesse. Nel loro caso questa funzione fa parte dell'interprete glulx (per fare due nomi: glulxe e git, ma ne esistono altri, anche in Java!). Per lo stesso motivo quasi tutto il materiale del Capitolo 10, del Capitolo 11 e gran parte del Capitolo 12 è pressoché inutile per un autore.

◆ ◆ ◆

Glk effettua tutta la gestione dell'interfaccia utente (oltre ad altre cose importanti) in una funzione chiamata `glk_select()`. Questa funzione attende un evento (tipicamente l'input del giocatore) e ritorna una struttura che lo rappresenta. In quasi tutti i casi il programma necessita di un ciclo degli eventi. Nel caso più semplice si potrebbe scrivere qualcosa di simile a quello riportato in Figura 1.1.

Il programma in figura è compatibile con Glk e perfettamente legale. Come si può facilmente immaginare, non fa niente. Il giocatore vedrà una finestra

[1] Questo significa anche che, tecnicamente, Glk non è esattamente una libreria. In un certo senso, si scrive una libreria, che viene chiamata da Glk. Questa situazione al contrario è bizzarra e possibile fonte di mal di testa, quindi è meglio non pensarci.

Figura 1.1. Un programma Glk minimale

```
void glk_main()
{
    event_t ev;
    while (1) {
        glk_select(&ev);
        switch (ev.type) {
        default:
            /* nulla */
            break;
        }
    }
}
```

vuota, che potrà solo osservare o distruggere con una procedura definita dalla piattaforma in uso.[23]

❖ ❖ ❖

Al contrario di `glk_main()` la funzione `glk_select()` *è essenziale* per un programma Inform/glulx. Anche se non sono facilmente identificabili, la libreria standard contiene ben *tre* cicli di gestione degli eventi!

La questione verrà approfondita nella sezione 14.12.

◆ ◆ ◆

1.2. Uscire dal programma

Volendo chiudere il programma prima di uscire dalla funzione `glk_main()`, si può chiamare `glk_exit()`.

[2] CMD-punto sul Macintosh; una voce nel menù finestra di un X window manager; CTRL-C in una finestra terminale.

[3] D'altra parte, questo programma non gira follemente consumando tempo di CPU. La funzione `glk_select()` attende un evento. Poiché non può ritornare senza eventi attenderà per sempre, cedendo tempo di calcolo agli altri processi (se significativo per la macchina del giocatore).

In realtà, esistono alcuni eventi che sono sempre segnalati. Altri potranno essere definiti in future versioni delle API Glk. Questo è il motivo per cui la risposta di default ad un evento è di non fare nulla. Se non si riconosce un evento, meglio ignorarlo.

```
void glk_exit(void);
```

Questa funzione non ritorna.

Se si mostra del testo in una finestra e poi si termine il programma, si può assumere tranquillamente che il giocatore sarà in grado di leggerlo. Molto probabilmente la libreria Glk aspetterà con un messaggio del tipo *Premi un tasto per uscire*; le possibilità sono aperte, ovviamente; una versione per terminale di Glk potrebbe semplicemente uscire e lasciare visibile l'ultima schermata sullo schermo.[4]

❖ ❖ ❖

Sebbene esista un metodo standard per uscire da un programma Inform/glulx (lo statement quit), in realtà non ci sono motivi particolari per cui non si possa uscire con glk_exit(). Non ci sono però nemmeno vantaggi nel farlo (quit esce in modo altrettanto brutale).

◆ ◆ ◆

1.3. Il gestore di interrupt

La maggior parte delle piattaforme ha un metodo per interrompere un programma (CMD-punto sul Macintosh, CTRL-C sotto Unix, una voce del menù del window manager di X o altre possibilità). L'utente può farlo in ogni momento, anche durante l'esecuzione di una funzione o durante una chiamata di libreria Glk.

Se esiste la necessità di chiudere risorse critiche, è possibile registrare una funzione per gestire l'interrupt.

```
void glk_set_interrupt_handler(void (*func)(void));
```

L'argomento passato a glk_set_interrupt_handler() deve essere un puntatore ad una funzione che non prende argomenti e non ritorna nulla. Se Glk riceve un interrupt, ed è stata impostata una funzione di gestione, questa verrà richiamata prima di chiudere il processo.

Inizialmente non vi è alcun gestore definito. Questa situazione si può ripristinare in seguito con glk_set_interrupt_handler(NULL).

Se si chiama glk_set_interrupt_handler() con una nuova funzione mentre ne era definita un'altra, la nuova sostituisce la vecchia. Glk non tenta di accodare i gestori, solo l'ultimo definito rimane in effetto.

[4] È possibile uscire dal programma *solo* con glk_exit() o ritornando da glk_main(). Utilizzando la funzione exit() o altre funzioni specifiche della piattaforma, possono accadere *brutte cose*. Alcune versioni della libreria Glk potrebbero essere progettate per sessioni multiple, per esempio, e terminerebbero quindi tutte le sessioni e non solo la corrente. Inoltre probabilmente il testo finale non rimarrebbe visibile al giocatore come richiesto.

Non si può tentare di interagire con l'utente nel gestore di interrupt. In particolare, non bisogna chiamare `glk_select()` o `glk_select_poll()`. Qualunque cosa si provi a visualizzare potrebbe non essere visibile al giocatore.

❖ ❖ ❖

Questa è probabilmente l'unica funzione Glk che non è possibile esportare ad un programma interpretato, dato che il parametro è un puntatore a funzione! E nonostante tutto, lo strato di dispatch ha un selettore assegnato...

Come per la gestione del tick (vedere la sezione 1.4) questa è una di quelle parti di Glk di cui l'autore non si dovrà mai preoccupare. Tutte le *risorse* a cui ha accesso vengono gestite automaticamente da Glk alla chiusura; del resto se ne farà carico l'interprete.

◆ ◆ ◆

14. Il tick

Molte piattaforme hanno delle seccature a cui bisogna adempiere ogni tanto, per evitare che arrivino i folletti del bosco a mangiare il computer.

Il rischio non è *proprio* quello. Ma conviene chiamare `glk_tick()` ogni tanto, per sicurezza. Può essere necessario cedere tempo ad altre applicazioni in un sistema operativo cooperativo, oppure controllare gli interrupt durante un ciclo infinito.

```
void glk_tick(void);
```

La chiamata è rapida, molto rapida; anzi, è così rapida che, in media, non fa nulla. Quindi si può fare spesso.[5]

`glk_tick()` non aggiorna lo schermo, non controlla l'input del giocatore né gestisce eventi di qualsivoglia tipo. Per fare queste ed altre cose è necessario utilizzare `glk_select()` o `glk_select_poll()`.[6] Vedere il Capitolo 4

❖ ❖ ❖

Il programmatore Inform/glk non si dovrà *mai* preoccupare di questa funzione. Primo, perché in realtà fa *veramente* poco, nella maggior parte delle piattaforme

[5] In una virtual machine, una volta per istruzione è appropriato. In un programma con un sacco di calcoli, usate una frequenza simile.

[6] Alcuni critici hanno fatto notare che nel programma di esempio `model.c` la funzione `glk_tick()` non viene proprio utilizzata. Il motivo è che non sono presenti cicli pesanti in quel programma. Si fa un po' di lavoro per ogni comando e poi si torna in cima al ciclo degli eventi. La chiamata `glk_select()`, ovviamente, attende nuovi eventi e quindi fa tutti le considerazioni al riguardo che si possano immaginare; ma nella prossima versione di `model.c` ci sarà una chiamata a `glk_tick()` nel ciclo che stampa il testo di `verb_yada()`. Per una questione di principio.

In sostanza, è necessario garantire un limite superiore fisso di computazione fra le chiamate a `glk_tick()`. In un interprete per VM, dove il codice potrebbe contenere un ciclo infinito, questo problema è critico. In un programma C si può generalmente notare ad occhio il punto in cui inserire la chiamata.

non fa proprio nulla (è stata concepita per le necessità di MacOS Classic, che è cooperativo). E, secondo, viene già chiamata regolarmente dall'interprete glulx.

◆ ◆ ◆

1.5. Tipi di base

Per semplicità, tutti gli argomenti utilizzati nelle chiamate Glk sono solo di una manciata di tipi:

- Interi senza segno a 32 bit. Questi interi vengono utilizzati appena possibile, ovvero quasi sempre. Questo tipo viene chiamato `glui32`.
- Interi con segno a 32 bit. Chiamato `glsi32` e usato di rado.
- Riferimenti a oggetti di libreria. In pratica sono puntatori a strutture C opache; ogni libreria utilizzerà strutture differenti, quindi non si può né si deve tentare di manipolare il loro contenuto. Vedere la sezione 1.6.
- Puntatori a uno dei tipi di cui sopra.
- Puntatori a strutture che consistono interamente dei tipi di cui sopra.
- Caratteri senza segno. Vengono utilizzati solo per i caratteri del testo con codifica Latin-1; vedere il Capitolo 2.
- Puntatori a void. Quando non c'è niente di meglio da usare.

1.6. Oggetti opachi

Glk definisce una manciata di classi di oggetti speciali. Questi oggetti non sono utilizzabili direttamente dal programma; vanno sempre referenziati attraverso puntatori a strutture C opache.

Al momento, sono definite le seguenti classi:[7]

Finestre Sezioni di schermo, utilizzate per mostrare e acquisire informazioni.

Stream Flussi di dati, sui quali si può fare input/output di testo.[8]

Riferimenti a file Puntatori a file in memoria permanente (su disco).[9]

[7] In futuro potrebbero venir definite altre classi di oggetti.

[8] Ci sono stream su finestra e stream su file, visto che i dati possono essere mandati su file o finestre. E anche stream in memoria.

[9] Sotto Unix un riferimento a file è un pathname; su Mac un `FSSpec`. In realtà sono presenti anche altre informazioni, come il tipo di file e se è un file di testo o binario. Vedere il Capitolo 6.

Canali audio Canali sul quale vengono emessi suoni e musica.[10]

Quando si crea uno di questi oggetti, è sempre possibile che la creazione fallisca (a causa di mancanza di memoria o di risorse o di qualche altro errore del sistema operativo). In questi casi, la funzione di allocazione ritorna `NULL` al posto di un puntatore valido. È sempre buona idea tener conto di questa possibilità.

`NULL` non è mai l'identificatore valido di un oggetto (finestra, stream, riferimento a file o canale audio). Il valore `NULL` è spesso utilizzato per indicare "nessun oggetto" oppure "niente", ma non è un riferimento valido. Se una funzione Glk richiede un riferimento ad oggetto come argomento, non è legale passare `NULL` (a meno che la definizione della funzione specifichi altrimenti).

In `glk.h` sono definiti i tipi `winid_t`, `strid_t`, `frefid_t` e `schanid_t` per contenere i riferimenti ad oggetti. Queste typedef sono in realtà puntatori alle struct `glk_window_struct` (finestre), `glk_stream_struct` (stream), `glk_fileref_struct` (riferimenti a file) e `glk_schannel_struct` (canali audio, se supportati), rispettivamente.[11] È ovviamente illegale passare un tipo di puntatore ad una funzione che se ne aspetta un altro.

❖ ❖ ❖

Se si utilizza Inform/glulx *tutto* viene gestito come un intero. Compresi i vari puntatori opachi. In particolare sembra che Inform abbia la tendenza a considerare le proprietà con valori > 65534 come proprietà funzionali.

◆ ◆ ◆

1.6.1. Sassi

Ogni istanza di questi oggetti (finestre, stream, file reference o canali audio) ha un *sasso* associato. Sotto al sasso vi è semplicemente un intero di 32 bit che viene fornito, per qualunque scopo, quando si crea l'oggetto.[12]

❖ ❖ ❖

Nella sua estrema fantasia (basti guardare i nomi Glk e glulx), Andrew Plotkin ha deciso di mettere sotto i sassi quelli che, in tutto il resto del mondo, sono semplicemente gli *user defined values*; per intenderci, Windows 3.0 già li utilizzava per tener

[10] Non tutte le librerie Glk supportano l'audio.

[11] Questo è il modo in cui vengono gestiti gli oggetti opachi programmando in C. Se si utilizza Glk attraverso una virtual machine, le cose saranno probabilmente diverse. Gli oggetti opachi saranno rappresentati con interi o come oggetti della VM di un qualche tipo.

[12] La libreria, per intenderci, tiene al sicuro questo valore sotto un sasso e lo ritorna quando lo si richiede.

Ma a cosa servono i sassi? Se non viene in mente un buon utilizzo, meglio usare 0 e dimenticarsi della loro esistenza.

Figura 1.2. Iterare sugli oggetti (forma generica)

```
obj = glk_CLASS_iterate(NULL, NULL);
while (obj) {
    /* ...fa qualcosa con obj... */
    obj = glk_CLASS_iterate(obj, NULL);
}
```

traccia delle finestre aperte! La metafora di un intero sotto ad un sasso è comunque divertente...

L'intera sezione 14.6 è dedicata all'argomento. Per Inform i sassi sono *vitali*, come si vedrà.

◆ ◆ ◆

1.6.2. Iterare sugli oggetti opachi

Per ogni classe di oggetto opaco, esiste una funzione di iterazione che è possibile utilizzare per ottenere una lista di tutti gli oggetti esistenti di quella classe. Questa funzione è sempre nella forma

```
CLASSid_t glk_CLASS_iterate(CLASSid_t obj,
    glui32 *rockptr);
```

...dove CLASS rappresenta una delle classi di oggetti opachi.[13]

Durante la prima chiamata, glk_CLASS_iterate(NULL, r), si ottiene il primo oggetto; chiamando glk_CLASS_iterate(obj, r) si ottiene l'oggetto successivo. Quando non ci sono più oggetti su cui iterare, la funzione ritorna NULL.

L'argomento rockptr è un puntatore a glui32; se esso non è NULL per ogni chiamata a glk_CLASS_iterate(), oltre all'oggetto ritornato, il valore sotto al suo sasso viene messo nella locazione (*rockptr). Se non interessa il valore sotto al sasso si può tranquillamente passare NULL per rockptr.

Un utilizzo tipico è mostrato in figura Figura 1.2.

Se vengono creati o distrutti oggetti dentro al ciclo, i risultati possono essere imprevedibili. È sempre legale chiamare glk_CLASS_iterate(obj, r), sempre che obj sia un oggetto valido oppure NULL

L'ordine in cui vengono ritornati gli oggetti è assolutamente arbitrario. La libreria è libera di riordinarli ogni volta che viene creato o distrutto un oggetto di una certa classe. Finché gli oggetti sono sempre gli stessi (ovvero non ne vengono né creati né distrutti), la regola è che glk_CLASS_iterate(obj,

[13] Attualmente ci sono: glk_window_iterate(), glk_stream_iterate(), glk_fileref_iterate () e glk_schannel_iterate(). Ci potranno essere in futuro altre funzioni per altre classi, ma si comporteranno nello stesso modo.

r) ha risultati fissi e predeterminati e che iterare nel modo sopraindicato elenca ogni oggetto una e una sola volta.

❖ ❖ ❖

Per un esempio su come iterare in Inform/glulx, vedere la funzione di libreria standard GGRecoverObjects in parserm.h. L'unica parte che manca è l'iterazione sui canali audio, ma vedere anche il commento relativo alla sottosezione 1.6.1. Altre informazioni nella sezione 14.7. E ovviamente l'esempio sezione 15.3.

◆ ◆ ◆

1.7. Il sistema Gestalt

Il meccanismo "gestalt" (allegramente *preso in prestito* da MacOS) è un sistema con il quale le API Glk possono essere aggiornate senza rendere la vita impossibile ai programmatori. Nuove funzionalità (grafica, audio e così via) possono essere aggiunte senza cambiare le specifiche di base. Il sistema permette anche di definire funzionalità facoltative (quelle che non tutte le implementazioni Glk implementeranno) e consente di stabilirne la presenza senza tentare di capirlo da un numero di versione.

❖ ❖ ❖

Una nota importante per i programmatori Inform/glulx: attenzione a non confondere la chiamata glk_gestalt con l'opcode glulx @gestalt!

Anche se sono sostanzialmente identiche e hanno lo stesso principio di funzionamento (d'altra parte Glk e glulx sono dello stesso autore e create l'una per l'altra) esiste una differenza fondamentale:

- glk_gestalt ritorna informazioni sulle funzionalità di Glk.
- @gestalt ritorna informazioni sulle funzionalità di glulx.

Sono quindi due funzioni completamente distinte (anche i selettori sono, ovviamente, diversi e incompatibili).

◆ ◆ ◆

L'idea di base è che si possono richiedere informazioni sulle funzionalità delle API chiamando le funzioni gestalt:

```
glui32 glk_gestalt(glui32 sel, glui32 val);
glui32 glk_gestalt_ext(glui32 sel, glui32 val,
    glui32 *arr, glui32 arrlen);
```

Il selettore (l'argomento sel) indica di quale funzionalità si stanno richiedendo informazioni; gli altri tre argomenti sono dati aggiuntivi, che possono essere significativi oppure no. Gli argomenti arr e arrlen della funzione glk_gestalt_ext() sono sempre facoltativi; è sempre valido indicare

Tabella 1.1. Selettori gestalt definiti in Glk 0.7

Selettore	Scopo
Version	Versione di Glk in uso.
CharInput	Tasti supportati per input a carattere.
LineInput	Caratteri supportati nell'input a linea.
CharOutput	Caratteri visualizzabili.
MouseInput	Supporto del click del mouse.
Timer	Supporto degli eventi del timer.
Graphics	Presenza delle funzioni grafiche.
DrawImage	Supporto del disegno di immagini.
GraphicsTransparency	Supporto per il canale alpha (trasparenza) nelle immagini
Sound	Presenza delle funzioni audio.
SoundVolume	Possibilità di regolare il volume.
SoundNotify	Supporto degli eventi di notifica audio.
SoundMusic	Supporto per la riproduzione di MOD.
Hyperlinks	Presenza delle funzioni per i collegamenti.
HyperlinkInput	Supporto degli eventi dei collegamenti.
Unicode	Presenza delle funzioni Unicode (Glk 0.7)

NULL o 0, se non interessa l'informazione che rappresentano. glk_gestalt() è semplicemente una scorciatoia per questo ultimo caso; glk_gestalt(x, y) è esattamente equivalente a glk_gestalt_ext(x, y, NULL, 0).

Il punto critico (nonché il *trucco* che fa funzionare il sistema) è che se la libreria Glk non ha mai sentito nominare il selettore sel, ritorna 0. È *sempre* sicuro chiamare glk_gestalt(x, y) (o glk_gestalt_ext(x, y, NULL, 0)).[14] Anche impiegando una vecchia libreria, compilata ancor prima che una data funzionalità venisse immaginata, è possibile verificarne la disponibilità chiamando glk_gestalt(); la libreria indicherà correttamente di non supportarla ritornando 0.

Una lista riassuntiva di tutti i selettori attualmente definiti è in Tabella 1.1.

[14] È anche sicuro chiamare glk_gestalt_ext(x, y, z, zlen) per un selettore sconosciuto x, quando z non è NULL, purché z punti ad un array di almeno zlen elementi. Il selettore sta attento a non scrivere oltre quel punto nell'array (se scrive nell'array).

Se un selettore non richiede il secondo argomento, è bene passare sempre 0; è meglio non assumere che il secondo argomento venga semplicemente ignorato: il selettore potrebbe venir esteso in una versione futura. Si continuerà ad ottenere il comportamento corrente passando 0 come secondo argomento, ma altri valori potrebbero produrre altri comportamenti.

1.8. Il numero di versione

Consideriamo ora, per esempio, il selettore gestalt_Version.

```
glui32 res;
res = glk_gestalt(gestalt_Version, 0);
```

res sarà impostato ad un numero di 32 bit che codifica la versione delle specifiche Glk implementate dalla libreria. I 16 bit superiori contengono il numero di versione maggiore; i successivi 8 il numero di versione minore; gli 8 bit meno significativi un numero di versione ancora minore, se c'è.[15]

La versione corrente (quella documentata in queste specifiche) è la 0.7.0, quindi questo selettore ritornerà 0x00000700.

❖ ❖ ❖

Ai fini dell'utilizzo pratico, attualmente, tutte le librerie sono conformi allo standard 0.6.1 (o almeno 0.6.0).

Le uniche funzionalità che si *perdono* della versione 0.7 sono quelle relative all'Unicode. Vedere il commento al Capitolo 2.

◆ ◆ ◆

```
glui32 res;
res = glk_gestalt_ext(gestalt_Version, 0, NULL, 0);
```

Questo segmento fa esattamente la stessa cosa. Notare che, in entrambi i casi, il secondo argomento non viene usato; per evitare sorprese in futuro è bene passare sempre 0.

1.9. Altre convenzioni

Il file glk.h è sempre lo stesso su tutte le piattaforme, con la sola eccezione delle typedef di glui32 e glsi32. Questi due tipi saranno sempre definiti come interi a 32 bit, che potrebbero essere long, int o qualche altra appropriata definizione C.

Tutte le costanti sono #define e tutte le funzioni sono dichiarazioni di funzioni (e non macro).[16]

FALSE è 0; TRUE è 1. Anche NULL è, ovviamente, 0.

❖ ❖ ❖

True e false non sono *esattamente* costanti in Inform (certo, esiste la convenzione di rtrue e rfalse). Vale comunque la regola TRUE (1), FALSE (0).

[15] In questo modo la versione 78.2.11 viene codificata come 0x004E020B.

[16] Ci sono alcuni punti in cui le macro sarebbero più efficienti (glk_gestalt() per esempio) ma non sono possibili colli di bottiglia ed è stata preferita la leggibilità.

Ma *attenzione*, per Inform `NULL` non vale 0, ma -1 Per completezza di cronaca, `infglk.h` definisce `GLK_NULL` come 0.

◆ ◆ ◆

Come già stabilito, è illegale passare `NULL` ad una funzione che si aspetta un oggetto valido, a meno che la definizione della funzione specifichi altrimenti.

Alcune funzioni hanno parametri puntatore, utilizzati come passaggio per riferimento; l'intento della funzione è di ritornare un valore nello spazio puntato dall'argomento. A meno che la definizione indichi altrimenti, è legale passare un puntatore `NULL` per indicare che non interessa il valore ritornato.

2

Codifica dei caratteri

Glk ha due API, separate ma parallele, per gestire l'input/output del testo. Le funzioni di base hanno a che fare interamente con caratteri di 8 bit; i loro argomenti sono array di byte (ottetti). Queste funzioni assumono la codifica caratteri Latin-1. Si può anche dire che utilizzano i codepoint U+00–U+FF di Unicode.

Latin-1 è una codifica caratteri a 8 bit; mappa codici numerici nell'intervallo 0–255 in caratteri stampabili. I valori da 32 a 126 sono i soliti caratteri standard ASCII (da ' ' a '~'). I valori negli intervalli 0–31 e 127–159 sono riservati per i caratteri di controllo e non hanno equivalenti visualizzabili.[1]

Le funzioni estese, o Unicode, gestiscono unicamente con word di 32 bit. Prendono array di word, non di byte, come argomenti. Sono di conseguenza in grado di gestire qualunque codepoint Unicode. Le funzioni estese hanno nomi che terminano in _uni.[2]

[1] Notare che le API Glk di base *non* usano UTF-8 né altre forme Unicode. Ciascun carattere è rappresentato da un singolo byte, compresi i caratteri inclusi nell'intervallo 128–255.

[2] Poiché queste funzioni gestiscono array di word di 32 bit, si può dire che utilizzano la forma di codifica UTF-32, ma *non* lo *schema* UTF-32, visto che le funzioni Glk operano su word, e non a livello di byte. UTF-32 è anche conosciuto come UCS-4, stando alle specifiche Unicode (appendice C.2), a parte qualche requisito semantico di cui non tratteremo. Per gli scopi pratici dell'utilizzo sotto Glk si possono ignorare le problematiche di codifica e assumere di utilizzare sequenze di codepoint numerici.

Ma perché non UTF-8? È un algoritmo di compressione ragionevole per i flussi Unicode; ma i sistemi per IF hanno tipicamente il loro modello di compressione per il testo. Mischiarli provoca più problemi di quanti ne risolva. L'altro vantaggio dell'UTF-8 è che i caratteri ASCII a 7 bit sono automaticamente validi, ma questo non è essenziale, in quanto i compilatori possono generare senza problemi dati testuali nella forma desiderata. Il problema è che UTF-8 è una codifica a lunghezza variabile. Nessuno ha mai pianto all'idea di evitare un simile abominio.

E cosa dire del testo bidirezionale? È una buona idea, che potrebbe comparire nelle ver-

❖ ❖ ❖

Le funzionalità Unicode sono le *uniche* variazioni nelle specifiche Glk dalla versione 0.6 alla versione 0.7.

La stragrande maggioranza (per non dire la totalità, una volta esclusa l'implementazione di riferimento di Plotkin) delle librerie Glk in circolazione al momento implementano solamente la versione 0.6 delle specifiche.

A quanto pare il supporto per Unicode sembra si rivelerà un buco nell'acqua, e questo per un serie di motivi:

- Implementare *correttamente* lo standard Unicode è tutt'altro che banale. Implementarlo in modo portabile è ancora più complesso (basti pensare al solo algoritmo di normalizzazione in forma C...).
- Il fatto che alcune caratteristiche fondamentali (quali il testo bidirezionale o il supporto per alcune forme combinanti) sia lasciato *a discrezione* della libreria rende il sistema non affidabile per alcuni sistemi di scrittura che trarrebbero veramente beneficio da Unicode (due a caso: arabo ed ebraico). Allo stato attuale ne potrebbero usufruire le lingue con uno script pseudo-latino (per esempio, l'Europa dell'est, grazie agli accenti compositi) ma non molto altro.
- La stragrande maggioranza dell'IF in circolazione, e in fase di sviluppo, è in lingue per cui la codifica Latin-1 è ampiamente sufficiente.
- Anche se glulx è potenzialmente in grado di gestire i caratteri Unicode, convincere Inform a fare lo stesso è un altro problema (e non mi risulta che gli altri sistemi di sviluppo siano più avanti da questo punto di vista).

  Tecnicamente parlando, esistono delle patch per il compilatore Inform 6.31 che aggiungono il supporto Unicode, e Glulx dalla versione 3 è in grado di utilizzarle direttamente (Glulx 3 è l'adeguamento per l'infrastruttura Unicode, per l'appunto). La diffusione di queste tecnologie è comunque ancora ridotta.

Nel frattempo, finché qualche cinese o giapponese (o comunque straniero con uno strano linguaggio) si deciderà a fabbricare delle implementazioni *decenti* di Glk con supporto Unicode, meglio dimenticarsi di tutto questo e continuare ad usare lo stra-collaudato Latin-1. Oppure usare la libreria per MacOS X di Plotkin e vedere cosa succede...

Per meglio distinguere le nuove funzioni relative alla codifica Unicode, è stato apposto un apposito bollino a margine. "Rimuovendo" queste definizioni, le specifiche diventano sostanzialmente identiche alla versione 0.6.1 di Glk.

---

sioni future di questo documento. Non è in questa versione perché vogliamo qualcosa di semplice, e subito. Per il momento, meglio visualizzare tutto il testo in ordine di lettura (non necessariamente da sinistra a destra) e sperare per il meglio. Il suggerimento di stile relativo alla direzione potrebbe applicarsi solo a paragrafi interi (come per la giustificazione); oppure ad ogni sezione di testo, richiedendo quindi l'implementazione dell'algoritmo *BiDi* a zig-zag. È un campo tutto da esplorare, l'impaginazione è ardua. Un'altra possibilità sarebbe di lasciare determinare alla libreria la direzionalità del testo in base al set di caratteri. Non è impossibile: MacOS X è in grado di farlo. Ma può essere troppo complicato. Nel frattempo vale la pena notare che la libreria Windows Glk *non* rileva automaticamente la direzione, ma la CheapGlk su MacOS X lo fa. Di conseguenza al momento non vi è un mezzo indipendente dalla piattaforma per gestire la scrittura da destra a sinistra.

Tabella 2.1. Funzioni Unicode (solo in Glk 0.7)

glk_buffer_to_lower_case_uni
glk_buffer_to_upper_case_uni
glk_buffer_to_title_case_uni
glk_put_char_uni
glk_put_string_uni
glk_put_buffer_uni
glk_put_char_stream_uni
glk_put_string_stream_uni
glk_put_buffer_stream_uni
glk_get_char_stream_uni
glk_get_buffer_stream_uni
glk_get_line_stream_uni
glk_request_char_event_uni
glk_request_line_event_uni
glk_stream_open_file_uni
glk_stream_open_memory_uni

Tra l'altro... il 90% del lavoro necessario per *l'utilizzo* delle funzioni Unicode consiste nell'aggiungere `_uni` al nome delle funzione (credo!). Implementarle, come si è detto, è un'altra storia.

◆ ◆ ◆

2.1. Verificare la presenza delle funzionalità Unicode

Le funzioni di base sono obbligatorie e disponibili in ogni libreria Glk. Le funzioni Unicode, non necessariamente. Prima di chiamarle, bisogna verificarne la presenza con il seguente selettore gestalt:

```
glui32 res;
res = glk_gestalt(gestalt_Unicode, 0);
```

(in quanto chiamata gestalt è disponibile anche su Glk 0.6, quindi non è segnalata come funzione Unicode).

Se le funzioni Unicode sono supportate, viene ritornato `1`. Se ritorna `0` non bisogna provare ad utilizzarle. Potrebbero non visualizzare nulla, mostrare delle schifezze o generare un errore. La liste delle funzioni Unicode è presentata in Tabella 2.1.

Scrivendo un programma in C, c'è una complicazione aggiuntiva. Una libreria che non supporta Unicode potrebbe non implementare le funzioni. Anche testando con il gestalt, il programma potrebbe non linkare correttamente. Se il file `glk.h` è così vecchio da non dichiarare le funzioni e le costanti Unicode, probabilmente non compilerebbe neppure.

Per evitare questi problemi, si può effettuare un test del preprocessore per l'esistenza di GLK_MODULE_UNICODE. Se è definita, allora la definizione delle funzioni e costanti Unicode è presente. In caso contrario, no.

2.2. Output

Quando si manda testo ad una finestra, o ad un file aperto in modo testo, è lecito utilizzare qualunque carattere stampabile Latin-1: da 32 a 126 e da 160 a 255. È possibile utilizzare anche il carattere newline (linefeed, CTRL-J, decimale 10, esadecimale 0x0A).

Non è legale utilizzare qualunque altro carattere di controllo (0–9, 11–31, 127–159). Non si possono utilizzare nemmeno caratteri di formattazione standard quali tabulazione (CTRL-I), ritorno carrello (CTRL-M) o salto pagina (CTRL-L).[3]

❖ ❖ ❖

In particolare esistono in circolazione (ma magari sono stati corretti nel frattempo) alcuni programmi che utilizzano tab e carriage return, nonostante siano *esplicitamente* proibiti.

garglk-mod si comporta in modo intelligente al riguardo: la prima volta che viene impiegato uno di questi caratteri di controllo l'utente viene avvisato (nella tradizione di TEX). In ogni caso vengono interpretati rispettivamente come uno spazio e un newline.

Per quanto riguarda gli *altri* caratteri di controllo, vengono lasciati alla mercé del font utilizzato (che generalmente mostra il glifo sconosciuto).

Se i font utilizzati contengono i glifi necessari, il sistema di visualizzazione converte automaticamente le legature ff (ff), fi (fi), fl (fl), ffi (ffi), ffl (ffl), – (--), — (---) e . . . (...).

Un'ultima cosa: se il parametro di configurazione quotes è diverso da 0, il motore di formattazione trasforma automaticamente i caratteri "‘", "’" e "‟" nelle relative varianti tipografiche (anche tristemente note come *virgolette inglesi*).

◆ ◆ ◆

Visualizzare caratteri Unicode oltre 255 è una questione più complicata; troppo complicata per essere coperta con precisione in questa sede. Si leggano le specifiche Unicode e buona fortuna.[4]

[3] Come al solito, il comportamento della libreria di fronte ad un carattere illegale è indefinito. Sarebbe preferibile mostrare un codice numerico, come 0177 o 0x7F per avvisare l'utente che qualcosa non va. D'altra parte è completamente lecito ignorare i caratteri illegali; ma è meglio non farci affidamento.

[4] I caratteri combinanti in particolare sono una grana. Mostrare un combinante può variare l'apparenza del precedente carattere visualizzato. La libreria deve essere preparata a gestirlo, anche se i caratteri sono stati passati a chiamate glk_put_char_uni() separate.

Notare che, nel caso di un file aperto in modalità binaria, si possono inviare byte di qualunque valore senza restrizioni. Vedere la sottosezione 5.5.4

❖ ❖ ❖

Si ricorda una cosa importante: anche se il grosso dell'utilizzo di Glk nel campo delle avventure testuali riguarda l'output di testo in una finestra, esiste anche la possibilità di aprire e utilizzare file *esterni*, su disco. Anche solo per salvare e ricaricare la partita!

Il discorso, più ampio, riguarda gli stream, trattati a fondo nel Capitolo 5. Ma l'output su finestra, anche se può non risultare evidente grazie al concetto di *stream corrente*, è effettuato per mezzo di uno stream.

Dal punto di vista concettuale non vi è differenza fra scrivere su una finestra o scrivere su uno stream (anche se nell'ultimo caso ci sono più opzioni). *Leggere*, d'altra parte, comporta due meccanismi completamente distinti: eventi per le finestre e stream per tutti gli altri tipi di stream.

◆ ◆ ◆

Una particolare implementazione Glk potrebbe non essere in grado di visualizzare tutti i caratteri stampabili. È garantito il funzionamento dei caratteri ASCII (32–126 e il newline 10). Altri caratteri potrebbero essere mostrati correttamente, oppure come combinazione di più caratteri (come ae per la legatura æ), o addirittura con un qualche carattere sostitutivo (per esempio, un pallino o un punto interrogativo).

È possibile verificare il comportamento per ogni carattere utilizzando il selettore `gestalt_CharOutput`. Se `ch` è un codice carattere (Latin-1 o Unicode), chiamando in questo modo

```
glui32 res, len;
res = glk_gestalt_ext(gestalt_CharOutput, ch, &len, 1);
```

...il valore di `res` sarà uno dei seguenti:

`gestalt_CharOutput_CannotPrint` Il carattere non può essere visualizzato. Il giocatore potrebbe non vedere nulla oppure un segnaposto.

`gestalt_CharOutput_ExactPrint` Il carattere viene mostrato esattamente come definito.

`gestalt_CharOutput_ApproxPrint` La libreria utilizza una qualche approssimazione del carattere. Sarà più o meno corretta ma non necessariamente precisa né distinguibile da altri caratteri similari.

In tutti i casi, `len` (il valore `glui32` puntato dal terzo argomento) sarà il numero effettivo di glifi che saranno utilizzati per rappresentare il carattere.

Nel caso di gestalt_CharOutput_ExactPrint, varrà sempre 1;[5] il numero di caratteri nel caso di gestalt_CharOutput_CannotPrint potrebbe essere 0 (non viene mostrato nulla) o anche più; e infine, come caso limite (per esempio in presenza di legature espanse) per gestalt_CharOutput_ApproxPrint potrebbe essere 1 o più. Questa informazione diventa utile nel momento in cui si usa un font a spaziatura fissa.[6]

Se ch è un carattere non stampabile (0–9, 11–31, 127–159) questo selettore ritorna sempre gestalt_CharOutput_CannotPrint.

Attenzione ai valori di byte con segno. Se si dichiara una variabile char ch con valore 0xFE e poi si chiama

```
res = glk_gestalt(gestalt_CharOutput, ch);
```

...in tal caso (per definizione) ch verrà *estesa in segno* a 0xFFFFFFFE, che *non* è un carattere valido, nemmeno in Unicode. La forma corretta da usare è

```
res = glk_gestalt(gestalt_CharOutput,
    (unsigned char)ch);
```

❖ ❖ ❖

Finché si utilizzano solamente caratteri ASCII non c'è bisogno di preoccuparsi: sono tutti garantiti funzionanti. Ma nelle lingue europee (fra cui l'Italiano!) si fa ampio uso di caratteri accentati, facenti parte del banco superiore della codifica Latin-1.

Anche se potrebbe essere ragionevole considerarli come disponibili (e in effetti sarebbe veramente poco pratico gestire correttamente la situazione in cui non siano supportati dalla libreria) esiste comunque la possibilità che vengano simulati (per esempio sostituendo à con a').

Nel caso di scrittura su una griglia di testo quest'ultima circostanza può creare seri problemi di allineamento (un carattere accentato potrebbe essere visualizzato da *due* caratteri consecutivi, come già visto, ma non necessariamente).

Il selettore gestalt_CharOutput è quindi consigliata per calcolare la lunghezza *effettiva* del testo in una griglia di testo. In altre situazioni la sua utilità è relativamente limitata.

Volendo fare *veramente* i precisini si potrebbe controllare all'inizio che siano disponibili tutti i caratteri estesi utilizzati e dare una segnalazione (o uscire con un errore) in caso contrario.

◆ ◆ ◆

[5] Unicode include il concetto di caratteri non spaziatori e di caratteri combinanti che non rappresentano glifi; e anche caratteri a larghezza doppia i cui glifi occupano due spazi in un font a spaziatura fissa. Versioni future potrebbero gestire questi concetti ritornando 0 o 2, ma per il momento aderiamo al principio del *cominciamo con le cose semplici*

[6] Come descritto nella sezione 1.9 questa informazione può essere omessa passando NULL come terzo argomento in glk_gestalt_ext() oppure utilizzando glk_gestalt().

2.3. Input di linea

È possibile richiedere al giocatore di inserire una linea di testo. Vedere la sezione 4.2.

Il testo digitato verrà messo in un buffer specificato. Non sono presenti c'è né campo di lunghezza, né terminatore NUL, nel buffer. La lunghezza del testo è però ritornata come parte dell'evento.

Utilizzando le API di base, il buffer conterrà unicamente caratteri Latin-1 stampabili (32–126, 160–255).

Una particolare implementazione di Glk potrebbe non essere in grado di accettare tutti i caratteri Latin-1 stampabili. È garantito il supporto dei caratteri ASCII (32–126).

Per stabilire se un carattere è digitabile si deve utilizzare il selettore `gestalt_LineInput`. Dato in `ch` un codice carattere, chiamando

```
glui32 res;
res = glk_gestalt(gestalt_LineInput, ch);
```

`res` sarà TRUE (1) se il carattere può essere digitato dal giocatore sulla linea di input oppure FALSE (0) in caso contrario. Ovviamente se `ch` è un carattere Latin-1 non stampabile, il risultato è garantito FALSE (0).

24. Input a carattere

È possibile richiedere al giocatore di premere un singolo tasto all'interno di una finestra. Vedere la sezione 4.1.

Utilizzando le API di base, il codice carattere ritornato può essere qualunque valore da 0 a 255. I caratteri stampabili sono già stati descritti. I codici rimanenti sono tipicamente codici di controllo: da CTRL-A a CTRL-Z e pochi altri.

Esistono anche alcuni codici speciali, rappresentanti specifici tasti, che possono essere ritornati in un evento di input carattere. Sono rappresentati da interi a 32 bit, iniziando da `0xFFFFFFFF` e proseguendo verso il basso. Questi codici speciali sono definiti nel file `glk.h` ed elencati in Tabella 2.2.

Differenti implementazioni di Glk varieranno l'insieme dei caratteri digitabili dal giocatore. La limitazione più evidente è che alcuni caratteri sono mappati su altri. Per esempio, la maggior parte delle tastiere utilizza il carattere CTRL-I quando viene premuto il tasto TAB. La libreria Glk, sempre che sia in grado di riconoscerlo, genererà un evento `keycode_Tab` (valore `0xFFFFFFF7`) in questo caso. Di conseguenza, in queste tastiere, *nessun*

Tabella 2.2. Tasti speciali in input a carattere e loro nomi comuni

Costante	Tasto associato
keycode_Left	Cursore a sinistra
keycode_Right	Cursore a destra
keycode_Up	Cursore in alto
keycode_Down	Cursore in basso
keycode_Return	Tasto Return o Invio
keycode_Delete	Tasto delete (Canc) o backspace
keycode_Escape	Tasto escape (Esc)
keycode_Tab	Tasto Tab (tabulatore)
keycode_PageUp	Tasto pagina precedente (PagSu)
keycode_PageDown	Tasto pagina successiva (PagGiù)
keycode_Home	Tasto Home (inizio riga o inizio pagina)
keycode_End	Tasto End (Fine)
keycode_Func1	Tasto funzione F1
keycode_Func2	Tasto funzione F2
⋮	⋮
keycode_Func12	Tasto funzione F12
keycode_Unknown	Qualunque altro tasto senza significato in Latin-1 a cui non sia assegnato un codice speciale.

tasto genererà mai un evento CTRL-I (valore 9). Simili mappature saranno probabilmente presenti per altri codici speciali.[7]

Alcuni caratteri potrebbero non essere digitabili semplicemente perché non esistono.[8]

Altri caratteri potrebbero non essere utilizzabili in quanto riservati all'interfaccia. Per esempio, la libreria Mac Glk riserva il tasto TAB per passare da una finestra Glk all'altra. In questo caso, su Mac, la libreria non genererà mai né un evento keycode_Tab e neppure un evento CTRL-I.

Il carattere linefeed o CTRL-J, che è l'unico carattere di controllo *stampabile* probabilmente non sarà *digitabile*. Nella maggior parte delle librerie verrà convertito in keycode_Return. Anche in questo caso se si chiede di "*premere il tasto invio*" il codice da verificare è keycode_Return.

I tasti delete e backspace sono raggruppati sotto un unico codice in quanto

[7] D'altra parte, una libreria potrebbe essere in grado do distinguere il tasto TAB da CTRL-I. Questo è legale. L'idea di fondo è che se il programma chiede "*premi il tasto tab*", il valore da verificare è quello di keycode_Tab e non quello di CTRL-I.

[8] Non tutte le tastiere hanno i tasti Home e End. Una piattaforma basata su penna potrebbe addirittura non avere nessun carattere di controllo.

sono stati storicamente confusi più volte. Queste specifiche negano formalmente ogni desiderio di distinzione. Ovviamente una libreria è libera di gestirli separatamente durante l'input di linea, che è il caso in cui fa la differenza. Gestirli come un unico tasto durante l'input a carattere non dovrebbe essere un grande problema, ed è consigliato.

È possibile stabilire se un carattere (o un tasto speciale) è digitabile con il selettore `gestalt_CharInput`. Sia `ch` un codice carattere o un codice speciale (da `0xFFFFFFFF` in giù), chiamando

```
glui32 res;
res = glk_gestalt(gestalt_CharInput, ch);
```

...come al solito `res` sarà `TRUE` (1) se il carattere può essere digitato oppure `FALSE` (0) in caso contrario.[9]

❖ ❖ ❖

L'input a carattere nella libreria standard di Inform è circoscritto alla gestione dei menù (`DoMenu` in verblib).

È stato utilizzato un accorgimento geniale, che ha permesso di evitare il controllo per mezzo di `gestalt_CharInput`. Anche se su schermo vengono indicati i tasti alfabetici (per esempio "`N`" per successivo), la routine accetta *anche* i tasti di spostamento del cursore corrispondenti; se non sono disponibili? niente, al limite la libreria non li genererà. I tasti alfabetici essendo ASCII sono comunque garantiti, quindi non vi è alcun rischio a richiederli.

◆ ◆ ◆

[9] Nota per i programmatori Glk: l'obiettivo non è di generare ogni possibile evento carattere. Se la libreria indica che è in grado di generare un tasto, gli autori assumeranno che è disponibile e *chiederanno al giocatore di usarlo*. Se un evento `keycode_Home` può essere generato solo premendo ESC-CTRL-A, e il giocatore non lo sa, sarà nei guai quando il gioco indicherà "*Premi il tasto Home per il prossimo suggerimento*". In questo caso sarebbe meglio per la libreria indicare di *non* essere in grado di produrre quel tasto; in tal modo il gioco potrebbe adeguarsi e richiedere per esempio di usare il tasto H.

Ovviamente è meglio non affidarsi a tasti particolarmente oscuri. I tasti di movimento cursore e l'invio saranno quasi certamente disponibili; gli altri saranno progressivamente meno disponibili ed è meglio (e questo è rivolto agli autori) non farne affidamento. *È necessario* verificare ogni tasto che si intende usare *inclusi* i tasti di movimento cursore e il return, ed essere pronti ad usare tasti differenti se `gestalt_CharInput` indica che non sono disponibili.

2.5. Maiuscole e minuscole

È possibile convertire i caratteri Latin-1 tra maiuscolo e minuscolo con due funzioni Glk:

```
unsigned char glk_char_to_lower(unsigned char ch);
unsigned char glk_char_to_upper(unsigned char ch);
```

Queste funzioni sono simili ma hanno alcuni vantaggi rispetto alle standard ANSI tolower() e toupper(): per prima cosa sono definite su tutto il set Latin-1, accentate comprese; inoltre funzionano allo stesso modo su ogni piattaforma (dato che sono parte di Glk); e, per finire, sono valide su tutti i caratteri. In altre parole, utilizzando glk_char_to_lower() su un carattere minuscolo o un carattere che non è una lettera, il risultato sarà lo stesso carattere non modificato.

Simili conversioni in Unicode sono più problematiche e devono essere applicate ad intere stringhe di caratteri, non ai singoli.

Unicode
Glk 0.7

```
glui32 glk_buffer_to_lower_case_uni(glui32 *buf,
    glui32 len, glui32 numchars);
glui32 glk_buffer_to_upper_case_uni(glui32 *buf,
    glui32 len, glui32 numchars);
glui32 glk_buffer_to_title_case_uni(glui32 *buf,
    glui32 len, glui32 numchars, glui32 lowerrest);
```

Queste funzioni hanno due argomenti lunghezza poiché una stringa Unicode può allungarsi quando cambia. L'argomento len è la lunghezza di buffer disponibile; numchars è il numero di caratteri presenti inizialmente nel buffer. Di conseguenza numchars$\leq$len. Il contenuto del buffer a seguire di numchars non influisce sull'operazione.

Le funzioni ritornano il numero di caratteri dopo la conversione. Se questo è maggiore di len i caratteri saranno stati troncati a len ma il valore ritornato è quello effettivo. Il contenuto del buffer a seguire del valore ritornato non è definito.

Le funzioni lower_case e upper_case fanno l'operazione ovvia: convertono ogni carattere nel buffer (i primi numchars presenti) nel loro equivalente minuscolo o maiuscolo, sempre che esista.

La funzione title_case ha un flag aggiuntivo. L'operazione di base è di cambiare il primo carattere del buffer in maiuscolo e lasciare il resto inalterato. Se lowerrest è TRUE (1) anche tutti i caratteri non iniziali sono convertiti in minuscolo (invece di lasciarli stare).

Vedere la specifica Unicode (capitoli 3.13, 4.2 e altri) per l'esatta definizione di queste trasformazioni.[10]

❖ ❖ ❖

I programmatori Inform possono tranquillamente utilizzare le routine di libreria `LowerCase` e `UpperCase`. Quando si compila per Inform/glulx la loro definizione è *esattamente* quella delle funzioni Glk corrispondenti:

```
[ LowerCase c; return glk($00A0, c); ];
[ UpperCase c; return glk($00A1, c); ];
```

Questo vale ovviamente *solo* per le funzioni standard, non per quelle Unicode (che rappresentano un incubo a parte).

◆ ◆ ◆

[10] Unicode ha alcuni casi peculiari. Per esempio, un carattere combinato che sembra "ss" potrebbe diventare in maiuscolo utilizzando *due* caratteri "S". Il title casing è ancora più strano: "ss" a inizio buffer potrebbe essere trasformato in due caratteri simili a "Ss". La funzione `glk_buffer_to_title_case_uni()` è formalmente il title casing del primo carattere del buffer.

Bozze precedenti comprendevano una funzione che effettuava il title casing di ogni singola *parola* nel buffer. È stata rimossa dopo aver letto il terribile Annesso #29 dello standard Unicode, che specifica come dividere una stringa in parole.

3 Finestre

Nella maggior parte delle piattaforme, la combinazione programma/libreria apparirà al giocatore in una finestra: o una finestra che copre lo schermo intero, o una che condivide lo schermo con altre finestre. Ovviamente il programma non se ne deve preoccupare. Lo spazio dello schermo reso disponibile a Glk è un rettangolo, ed è possibile dividerlo in pannelli per vari scopi. È di questi pannelli a cui ci si riferirà in seguito con il termine "finestre".

Ogni finestra è identificata da un puntatore ad una struttura C opaca. Vedere la sezione 1.6.

Una finestra ha un tipo. Attualmente sono definiti quattro tipi di finestra:

Buffer di testo Un flusso di testo.[1] È possibile aggiungere testo solo in fondo alla finestra e, nello stesso punto, richiedere una linea di testo dal giocatore.

Griglia di testo Una griglia di caratteri visualizzata con un font a spaziatura fissa.[2] È possibile modificare il testo in qualunque punto della griglia.

Finestra grafica Una griglia di pixel colorati. Le finestre grafiche non sono in grado di effettuare input o output di testo, ma in compeno esistono comandi per disegnarvi sopra.[3]

Finestra vuota Non contiene nulla. Non supporta né input né output, di alcun tipo.[4]

---

[1] La *finestra della storia* di un gioco Infocom.

[2] La *finestra di stato* di un gioco Infocom.

[3] Le finestre grafiche sono facoltative; non tutte le implementazioni Glk le supportano; vedere la sezione 7.4.

[4] Le finestre vuote sono più che altro un esempio di finestra *generica*. Difficilmente serviranno a qualcosa.

Poiché Glk è un sistema in espansione, altri tipi di finestra potrebbero essere aggiunti in futuro. È importante ricordare che non tutti i tipi di finestra saranno necessariamente disponibili su ogni libreria Glk.

Esiste anche un tipo di finestra speciale, il paio. Le finestre paio sono create da Glk per gestire la disposizione delle finestre. Non si possono creare esplicitamente. Vedere la sottosezione 3.5.2.

Ogni finestra ha un sasso. Sotto questo viene messo un valore fornito quando si crea la finestra; si può utilizzare a piacere. Vedere la sottosezione 1.6.1.

Quando Glk viene avviata non ci sono finestre.[5]

Senza una finestra non si può fare nessun tipo di input/output; la prima cosa da fare sarà quindi crearne una. Vedere la sezione 3.2.

È possibile creare un numero arbitrario di finestre, di qualunque tipo. È anche possibile controllare la loro disposizione con un sistema di chiamate abbastanza flessibile. Vedere la sezione 3.1.

È possibile chiudere qualunque finestra. Si possono anche chiudere tutte le finestre, tornando allo stato originale dell'avvio.

È possibile richiedere input da una o più finestre. L'input può essere generato dal mouse (sulle piattaforme che lo supportano), oppure costituito da singoli caratteri o da intere linee di testo. È legale richiedere input da diverse finestre allo stesso tempo. La libreria deve avere un qualche meccanismo per permettere al giocatore di scegliere in quale finestra digitare.

3.1. Disposizione delle finestre

Il sistema di disposizione delle finestre è abbastanza complicato. Questo è un tentativo di spiegarlo in maniera coerente.[6]

❖ ❖ ❖

Il meccanismo di divisione delle finestre è in effetti diverso da tutti quelli comunemente impiegati. Mentre normalmente si parla di *finestre che contengono altre finestre*, Glk restringe la struttura al concetto di *finestra partizionata in due finestre*. E poi il modo in cui viene definito *come* partizionare una finestra ha le sue peculiarità...

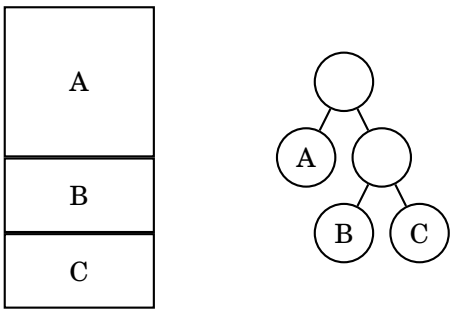
Questa sezione è *effettivamente* un po' complicata. Tuttavia è necessario comprenderla per creare layout di finestre personalizzati.

◆ ◆ ◆

[5] Con *non ci sono finestre* si intende che non ci sono finestre Glk. In un ambiente quale X o MacOS potrebbe esserci una finestra applicativa visibile; questo è lo spazio di schermo che conterrà tutte le finestre Glk che verranno create. Ma, all'inizio, questo spazio è vuoto e inutilizzato.

[6] Se si sta leggendo per avere solo una infarinatura di Glk meglio staltare a leggere la sezione 3.5 e tornare in seguito.

Figura 3.1. Prima si divide A, poi si divide B



All'inizio non ci sono finestre. Si può quindi creare una finestra, che occuperà l'intera area di schermo disponibile. In seguito, si può dividere questa finestra in due. Una delle metà è la finestra originale; l'altra metà è nuova, del tipo che si vuole. Bisogna indicare se la nuova finestra è a sinistra, a destra, sopra o sotto l'originale. Bisogna anche indicare il modo in cui viene suddivisa. La divisione della superficie disponibilie può essere $50/50$ o $70/30$ o qualsiasi altro rapporto. Oppure si può dare una dimensione fissa alla nuova finestra e lasciare alla vecchia lo spazio rimanente. Oppure si può (ancora!) dare dimensione fissa alla *vecchia* finestra per lasciare spazio alla *nuova*.

A questo punto ci sono due finestre. Esattamente nello stesso modo è possibile suddividerne una: quella originale o quella appena creata. La finestra viene suddivisa in due che, assieme, occupano lo stesso posto di quella originale.

È possibile ripetere tutto questo a piacimento. Ogni volta che viene divisa una finestra, ne viene creata una nuova. In conseguenza a questo fatto, la chiamata che fa ciò si chiama `glk_window_open()`.[7]

È importante ricordare che l'ordine di suddivisione è significativo. Se si divide due volte, non esiste un trio di finestre; esiste un paio con un altro paio in un lato. Matematicamente, la struttura delle finestre è un albero binario, come mostrato in Figura 3.1.

Oppure, si potrebbe dividere A in A e B, poi dividere A ancora in A e C (Figura 3.2).

Negli esempi sopra è stato utilizzato il tipo di divisione più semplice. Ogni divisione è $50/50$ e la nuova finestra del paio è sempre *sotto* quella originale (quella che viene divisa). È possibile fare cose più interessanti. Ecco altri tre modi per ottenere il primo esempio; tutti e tre hanno la *stessa* struttura ad albero, ma sono diversi sullo schermo (Figura 3.3).

[7] Il nome `glk_split_window` potrebbe essere stato più appropriato, oppure aver confuso ulteriormente. Ne è stato scelto uno.

Figura 3.2. Prima si divide A, poi si divide ancora una volta A

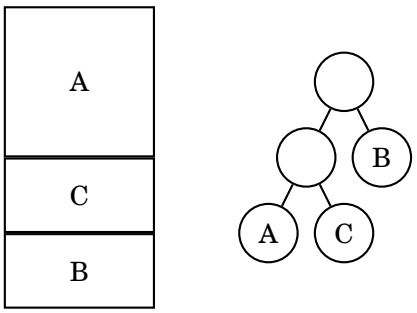
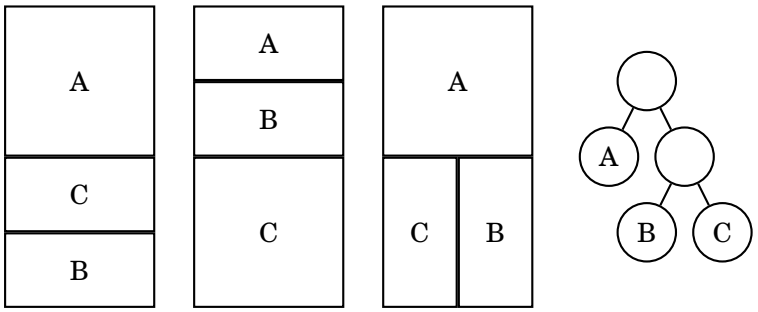


Figura 3.3. Stesso ordine di divisione, diversi posizionamenti



Sulla sinistra, si effettua la seconda divisione (B in B/C) al contrario; la nuova finestra (C) viene messa sopra la vecchia (B).

Al centro si alterano le percentuali. La prima divisione (A in A/B) è una divisione $25/75$, il che rende B tre volte la dimensione di A. La seconda divisione (B in B/C) è una divisione $33/66$, che rende C il doppio della dimensione di B. Il risultato assomiglia al secondo esempio, ma ha una differente struttura interna.

❖ ❖ ❖

Anche se l'apparenza è la stessa, la differente struttura interna dell'albero delle finestre potrebbe provocare un diverso comportamento in caso di ridimensionamento, come si vedrà in seguito.

◆ ◆ ◆

Sulla destra, la seconda divisione (B in B/C) è verticale invece che orizzontale, con la nuova finestra (C) sulla sinistra della vecchia.

Le finestre visibili sullo schermo Glk sono le *foglie* dell'albero binario; sono appese in fondo ai rami nel diagramma. Ci sono anche dei *nodi interni*, quelli alle biforcazioni, senza identificatore. Queste sono le misteriose finestre paio.

Figura 3.4. La prima finestra solitaria

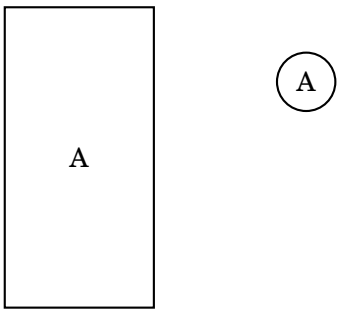
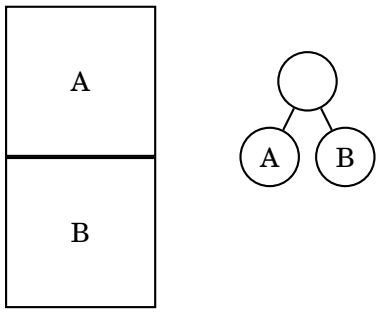


Figura 3.5. La prima finestra solitaria viene divisa orizzontalmente



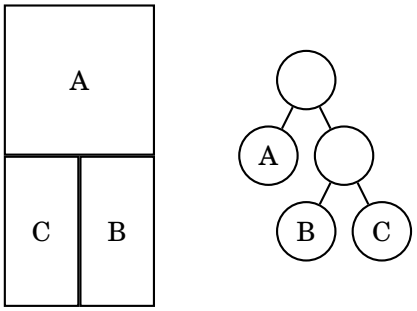
Le finestre paio non si creano direttamente; sono create come conseguenza di una divisione. Quando si crea una nuova finestra, una nuova finestra paio viene creata automaticamente. Nel seguente processo, si può vedere che, quando una finestra viene divisa, è sostituita da una nuova finestra paio e si muove per divenire una delle due foglie del paio.

Non è possibile disegnare in una finestra paio. La sua superficie è completamente riempita dalle due finestre che contiene. È in quelle che bisogna disegnare.

Perché avere le finestre paio nel sistema? Sono comode per certe operazioni. Per esempio, si può chiudere qualsiasi finestra in ogni momento; ma talvolta è desiderabile chiudere un'intera gerarchia di finestre in una volta. Nel terzo passo mostrato, se si chiude la finestra paio inferiore, vengono chiusi anche tutti i suoi discendenti (sia B che C) e rimane solo una singola finestra, A, che è quella con cui è iniziato tutto.

Visto che già si sta utilizzando della terminologia matematica, ecco alcune spiegazioni. La *radice* dell'albero è in cima (gli alberi matematici, come quelli genealogici, crescono capovolti, verso il basso). Se c'è solo una finestra, è la radice; altrimenti la radice è la finestra paio in cima. Ogni finestra paio ha

Figura 3.6. La finestra inferiore viene ulteriormente divisa, ma questa volta in verticale



esattamente due *figlie*. Altri tipi di finestre sono le *foglie* dell'albero, che non hanno figli. I *discendenti* di una finestra, ovviamente, sono figli, nipoti, e così via. Il *padre* e gli *antenati* di una finestra sono esattamente quello che ci si aspetta. Di conseguenza la radice è l'antenata di ogni altra finestra.

Esistono funzioni Glk per determinare la finestra radice e per conoscere il padre di una data finestra. Notare che il padre di ogni finestra è una finestra paio (eccetto per la radice che non ha padre).

3.2. Aprire, chiudere e dimensionare le finestre

```
winid_t glk_window_open(winid_t split, glui32 method,
    glui32 size, glui32 wintype, glui32 rock);
```

Se non ci sono finestre, i primi tre argomenti non sono significativi, `split` deve essere zero e `method` e `size` sono ignorati. `wintype` è il tipo di finestra che si intende creare, `rock` è il sasso (vedere la sottosezione 1.6.1).

Se già esistono altre finestre, la nuova finestra deve essere creata dividendone una preesistente. `split` è la finestra che si vuole dividere e in questo caso *non può* essere zero. `method` è una combinazione di costanti che specifica la direzione e il metodo di divisione (vedere sotto). `size` è la dimensione della divisione. Come prima, `wintype` è il tipo di finestra da creare mentre `rock` è il valore da tenere sotto un sasso.

Le costanti per `method` sono:

- `winmethod_Above`, `winmethod_Below`, `winmethod_Left`, `winmethod_Right`: la nuova finestra sarà rispettivamente sopra, sotto, a sinistra oppure a destra di quella da dividere.

- `winmethod_Fixed`, `winmethod_Proportional`: La nuova finestra sarà di dimensione fissa o in una data proporzione della vecchia (vedere sotto).

È importante ricordare la possibilità in cui la libreria non sia in grado di creare una nuova finestra, nel qual caso `glk_window_open()` ritornerà `NULL`.[8]

Gli esempi visti fino ad ora hanno utilizzato la tipologia di divisione più semplice (ovvero *al di sotto*). Ogni paio è stata una divisione in percentuale con x percento da un lato e $100 - x$ percento dall'altro lato. Se il giocatore ridimensiona la finestra, l'intera struttura si espande, contrae o allunga in modo uniforme.

Come detto prima, è anche possibile effettuare divisioni a dimensione fissa. Queste sono leggermente più complicate, in quanto bisogna sapere come viene misurata una dimensione in una finestra.

La dimensione è misurata in maniera diversa a seconda del tipo di finestra di cui si parla. Per esempio, una finestra a griglia di testo è misurata in base alla dimensione del suo font a larghezza fissa. È possibile definire una griglia di testo con una altezza fissa di quattro righe oppure di dieci colonne. Una finestra a buffer di testo è misurata con la dimensione del *suo* font.[9] Le finestre grafiche sono misurate in pixel, non in caratteri. Le finestre vuote non sono misurabili; non esiste un modo sensato per misurarle e quindi non è possibile creare una finestra vuota di dimensione fissa, solo in misura proporzionale.

Per esempio per creare una finestra a buffer di testo che occupa il 40% superiore della finestra originale, si può eseguire

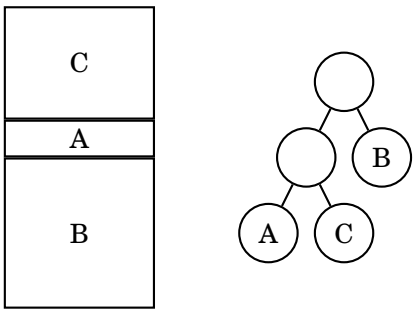
```
newwin = glk_window_open(win,
    winmethod_Above | winmethod_Proportional,
    40, wintype_TextBuffer, 0);
```

Per creare una griglia di testo che è sempre alta cinque linee, nella parte bassa della finestra originali

[8] È accettabile uscire se la finestra che si sta creando è una finestra importante: la prima per esempio. Ma non si provi ad effettuare un'operazione su una finestra senza essere sicuri che il suo riferimento non sia zero.

[9] Attenzione, diversi tipi di finestra possono utilizzare font di diversa dimensione; e anche due griglie di testo possono utilizzare font di dimensioni diverse.

Figura 3.7. La situazione prima del ridimensionamento




```
newwin = glk_window_open(win,
    winmethod_Below | winmethod_Fixed,
    5, wintype_TextGrid, 0);
```

Attenzione: l'interpretazione del parametro `size` è condizionata dall'argomento `method`. Se il metodo è `winmethod_Fixed`, dipende anche dall'argomento `wintype`. La nuova finestra viene chiamata la della divisione, poiché il suo tipo determina come viene calcolata la divisione.[10]

Questo sistema funziona piuttosto bene, finché tutte le impostazioni sono coerenti. Cosa succede se c'è un conflitto? Le regole sono semplici. Il dimensionamento scorre sempre verso il basso dell'albero, e il giocatore è in cima. Partendo dalla situazione in Figura 3.7.

Per prima cosa dividiamo A in A e B, con una proporzione del 50%. Dopodiché dividiamo A in A e C, dove C al di sopra, C è una griglia di testo e C ha una dimensione fissa di due righe (misurata nel proprio font). A ottiene lo spazio rimanente del 50% che aveva prima.

Ora se il giocatore allunga la finestra in verticale il risultato è approssimativamente quello mostrato in Figura 3.8.

La libreria determina: la prima suddivisione, l'originale divisione A-B, è $50/50$. Quindi B ottiene metà dello spazio su schermo, e il suo paio adiacente prende l'altra metà. Dopodiché per la seconda divisione C prende due righe; A prende quello che avanza. Tutto fatto.

A questo punto l'utente malignamente comincia a restringere la finestra, poco alla volta (Figura 3.9).

La logica rimane invariata. B ha sempre metà dello spazio a disposizione. Al punto 3, non c'è più posto per A, che finisce quindi ad altezza zero; Nulla viene mostrato in A. Al punto 4, il 50% dello spazio superiore non è più

[10] Anche per le divisioni effettuate con `winmethod_Proportional` si può chiamare la nuova finestra *chiave*. Ma in questo caso non è così importante poiché la dimensione sarà un semplice rapporto dello spazio disponibile e non la dimensione fissa di una nuova finestra.

Figura 3.8. Il giocatore ha allargato lo spazio disponibile

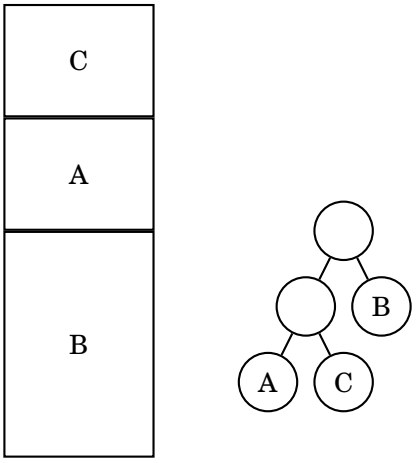


Figura 3.9. Il giocatore restringe progressivamente lo spazio disponibile.

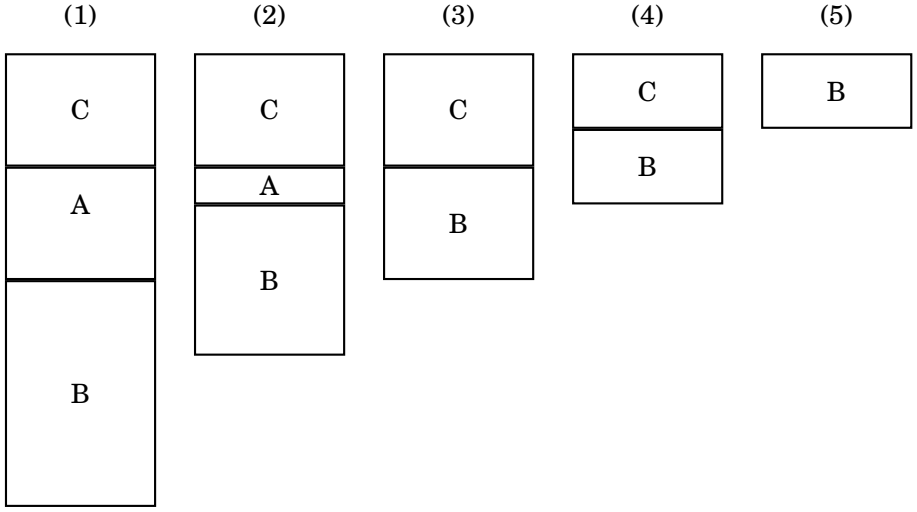
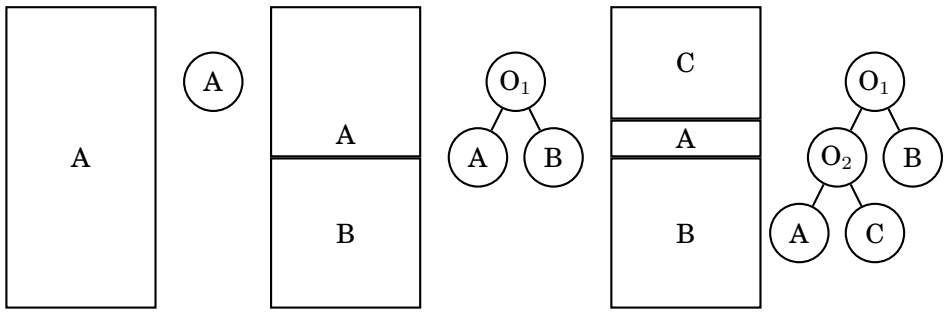


Figura 3.10. Le prime tre divisioni, ai fini del dimensionamento e le prime due finestre paio conseguenti



sufficiente per mostrare due righe in C, che ne ottiene quindi solo una. Al punto 5, anche C viene ridotta a 0.

Quando una finestra finisce sottodimensionata, si ricorda comunque quanto dovrebbe essere grande. Nell'esempio sopra, C si ricorda che dovrebbe essere di due righe; se l'utente in seguito espande la finestra alla dimensione originale, il layout ritorna come prima.

La restrizione del flusso di controllo verso il basso non è ottimale. Dopo tutto, nel punto 4, potrebbe esserci posto per entrambe le linee in C se B cedesse una parte del suo 50% ottenuto inizialmente. Ma questo non avviene.[11]

Cosa succede dividendo una finestra di dimensione fissa? La risultante finestra paio (ovvero, le due nuove finestre messe assieme) mantiene la stessa configurazione della finestra che è stata divisa. La finestra chiave della divisione originaria è ancora la finestra chiave di quella divisione, anche se ora è una *nipote*.

Il modo semplice (e corretto) per pensare al funzionamento è che il dimensionamento è memorizzato dal padre della finestra, non dalla finestra stessa; e il dimensionamento consiste in un puntatore alla finestra chiave e al valore della dimensione.

❖ ❖ ❖

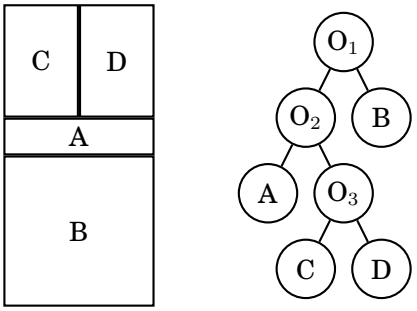
Secondo me, non è assolutamente semplice... Però è corretto per definizione, dato che è il modo in cui è implementato il meccanismo di dimensionamento nell'implementazione di riferimento.

◆ ◆ ◆

Nella situazione di Figura 3.10, dopo la prima divisione, la nuova finestra

[11] Questo rende la vita molto più semplice per la libreria Glk. Per determinare la configurazione della finestra deve solo tener conto degli antenati e mai dei discendenti. In questo modo il meccanismo di layout è un semplice algoritmo ricorsivo, senza backtracking.

Figura 3.11. Un'altra divisione, un'altra finestra, un altro paio



paio (O_1, che copre l'intero schermo) sa che il suo primo figlio (A) è sopra al secondo e ottiene il 50% della propria area. A è la finestra chiave di questa divisione, ma essendo proporzionale, ciò non è importante.

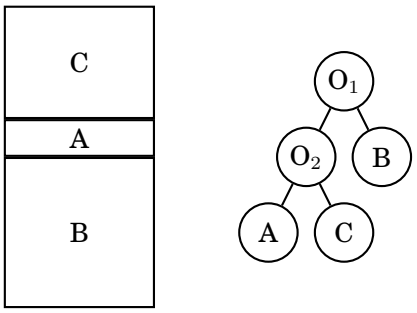
Dopo la *seconda* divisione, tutto ciò rimane vero; O_1 sa che il primo figlio prende il 50% dello spazio e che la sua finestra chiave (di O_1) è A. Ma ora il primo figlio di O_1 è O_2 al posto di A. La nuova finestra paio (O_2) sa che il *suo* primo figlio (C) è sopra al secondo e prende una dimensione fissa di due righe (misurate nel font di C, poiché C è la finestra chiave di O_2.)

Se ora dividiamo C, il paio risultante sarà sempre alto come due righe del font di CR: alto quanto basta per mostrare due linee nel font che C sta utilizzando. In questo esempio, lo faremo verticalmente (Figura 3.11).

O_3 ora sa che i suoi figli hanno una divisione sinistra/destra di $50 - 50$. O_2 è ancora impegnata a dare al suo figlio superiore, O_3, due righe nel font di C. Questo perché C è la finestra chiave di O_2.[12]

[12] In questo caso è una buona idea, poiché il risultato è che C, la griglia di testo, è ancora alta due righe. Se O_3 fosse stata una divisione in orizzontale, le cose potrebbero essere state problematiche. Ma le regole rimarrebbero le stesse. Si può sempre cambiare metodo se il risultato non è di gradimento.

Figura 3.12. È stata chiusa una finestra (la D)




```
void glk_window_close(winid_t win,
    stream_result_t *result);
```

Questa funzione chiude una finestra, il che è sostanzialmente l'opposto di aprirne una. È legale chiudere tutte le finestre, o anche chiudere la finestra radice (che è in sostanza la stessa cosa di chiuderle tutte).

Nell'argomento `result` viene messo il numero di caratteri dello stream associato alla finestra. Vedere il Capitolo 5 e in particolare la sezione 5.3.

Quando si chiude una finestra (e questa non è la radice), l'altra finestra del suo paio si prende tutta l'area che si è liberata. Provando a chiudere D, nell'esempio in corso (Figura 3.12).

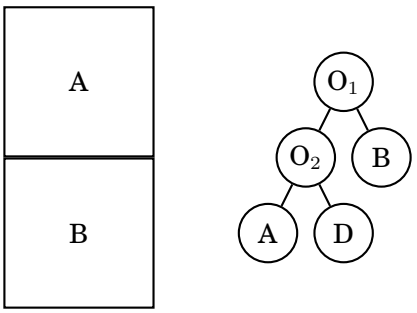
Cosa è successo? D se ne è andata. Anche O_3 non c'è più, e con essa la divisione verticale $50/50$. Le altre impostazioni sono inalterate; O_2 continua a dare alla sua figlia superiore due righe, misurate nel font della sua finestra chiave, che è C. Convenientemente la figlia superiore di O_2 *è* C, proprio come era prima che venisse creata D. Guardando meglio, ora che non c'è più D, tutto è tornato nella situazione precedente alla creazione di D.

Cosa sarebbe successo se si fosse chiusa C invece di D? Il risultato sarebbe stato questo:

Anche questa volta O_3 è scomparsa. Ma D è collassata ad altezza zero. Il motivo è il seguente: l'altezza di D è controllata da O_2, e la finestra chiave di O_2 era C, ma C ora non c'è più. O_2 non ha più una finestra chiave, per cui non può calcolare l'altezza per la sua figlia superiore. Il default è quindi zero.[13]

[13] Potrebbe sembrare una scelta poco pratica. In realtà è deliberato. Non è bene lasciare una finestra paio senza chiave, e il default di zero lo fa notare. Si può comunque usare `glk_window_set_arrangement()` per correggere la situazione. Vedere la sezione 3.3.

Figura 3.13. Questa volta è stata chiusa la finestra C, non la D



3.3. Cambiare la dimensione delle finestre

Esistono funzioni di libreria per cambiare e misurare le dimensioni effettiva di una finestra:

```
void glk_window_get_size(winid_t win,
    glui32 *widthptr, glui32 *heightptr);
void glk_window_set_arrangement(winid_t win,
    glui32 method, glui32 size, winid_t keywin);
void glk_window_get_arrangement(winid_t win,
    glui32 *methodptr, glui32 *sizeptr,
    winid_t *keywinptr);
```

glk_window_get_size() ritorna semplicemente la dimensione attuale di una finestra, nel relativo sistema di misura. Come descritto in la sezione 1.9, sia widthptr che heightptr possono essere NULL, se interessa solo una delle due misure.[14]

glk_window_set_arrangement() modifica le dimensioni di una divisione preesistente, ovvero i parametri di divisione di una data finestra paio.

Per contro, glk_window_get_arrangement() ritorna i parametri di un dato paio.

Considerando l'esempio sopra, dove D è collassata ad altezza zero. Supponendo che D sia un buffer di testo, si potrebbe ottenere un layout più utile facendo

```
winid_t o2;
o2 = glk_window_get_parent(d);
glk_window_set_arrangement(o2,
    winmethod_Above | winmethod_Fixed,
```

[14] Possono essere anche entrambi NULL se si vuol perdere tempo.

```
    3, d);
```

In questo modo D (il figlio superiore di O_2) diventa la finestra chiave di O_2, e prende una dimensione fissa di 3 righe.

Se in seguito si volesse espandere D, si potrebbe fare

```
glk_window_set_arrangement(o2,
    winmethod_Above | winmethod_Fixed, 5, NULL);
```

Questa espande D a cinque righe. Notare che, dato che la finestra chiave di O_2 è già D non è necessario fornire l'argomento `keywin`; si può passare `NULL` per indicare "la finestra chiave è immutata".

Se si cambia la finestra chiave di un paio, la nuova chiave *deve* essere una discendente di quel paio. Nell'esempio corrente, si potrebbe cambiare la finestra chiave di O_2 in A, ma non in B. La finestra chiave inoltre non può essere una finestra paio.

```
glk_window_set_arrangement(o2,
    winmethod_Below | winmethod_Fixed,
    3, NULL);
```

Questo cambia il dimensionamento in modo che sia calcolato sulla figlia *inferiore* di O_2, che è A. La finestra chiave rimane D; in questo modo A sarebbe alta tre righe misurata nel font di D, e D otterrebbe lo spazio residuo di O_2. Non necessariamente l'effetto è quello desiderato. Per far si che A sia alta tre righe, ma nel font di A, bisogna usare

```
glk_window_set_arrangement(o2,
    winmethod_Below | winmethod_Fixed,
    3, a);
```

È anche possibile trasformare O_2 in una divisione proporzionale:

```
glk_window_set_arrangement(o2,
    winmethod_Below | winmethod_Proportional,
    30, NULL);
```

...oppure

```
glk_window_set_arrangement(o2,
    winmethod_Above | winmethod_Proportional,
    70, NULL);
```

Il risultato è esattamente lo stesso, visto che una divisione al 30% sopra è identica ad una 70% sotto. Non è necessario specificare una finestra chiave con una divisione proporzionale, quindi l'argomento `keywin` è `NULL`.[15]

[15] Si potrebbe anche specificare A o D come finestra chiave, ma il risultato rimarrebbe lo stesso.

Qualunque dimensione si sia specificata, `glk_window_get_size()` indicherà la dimensione effettiva della finestra.

Notare che è possibile ridimensionare le finestre, ma non scambiarle né muoverle. Non si può spostare A sopra a D o cambiare O_2 in una divisione verticale con A a sinistra o a destra di D.[16]

❖ ❖ ❖

Se il layout delle finestre è stato studiato con accuratezza non dovrebbe essere necessario l'uso di `glk_window_set_arrangement()`. Inoltre, nei casi che non è in grado di gestire, la tecnica di *chiudere e riaprire* le finestre provoca la perdita del contenuto delle finestre stesse!

È anche vero che questo sistema di layout *non è in grado* di gestire qualunque combinazione di finestre: si provi a pensare, per esempio, a come realizzare una finestra di dimensione fissa centrata fra due finestre proporzionali (e quindi della stessa dimensione!).

◆ ◆ ◆

34. Una nota sullo stile di visualizzazione

Il modo in cui le finestre sono visualizzate è, ovviamente, interamente a carico della libreria Glk; dipende oltremodo da ciò che è naturale sulla macchina del giocatore. I bordi fra le finestre potrebbero essere linee nere, barre 3D, righe di caratteri "#"; potrebbero anche *non* esserci bordi.[17]

Potrebbero anche essere presenti altre decorazioni. Una finestra a buffer di testo spesso avrà una barra di scorrimento. La libreria (o il giocatore) potrebbero preferire margini più larghi attorno alle finestre di testo, e così via.

La libreria è responsabile della gestione di tutte queste decorazioni, margini, spazi e bordi. Non è necessario preoccuparsene. La garanzia è che se si richiede una dimensione fissa di due righe, la griglia potrà contenere due righe di caratteri; ovviamente se c'è abbastanza spazio residuo. Ogni margine e bordo viene automaticamente conteggiato. Se *non* c'è abbastanza spazio (come nei punti 4 e 5, molto più sopra), pazienza, non c'è nulla da fare.

Come sapere se il dimensionamento è andato male?

Si può chiamare `glk_window_get_size()` per determinare la dimensione che avete effettivamente ottenuto. Ovviamente, bisogna disegnare nelle finestre in base alla dimensione reale, non a quella che è stata richiesta. Se c'è abbastanza spazio, la dimensione richiesta e quella reale saranno identiche; ma non necessariamente. Bisogna utilizzare `glk_window_get_size()` e verificare.

[16] Per ottenere questo effetto si potrebbe chiudere una delle finestre e suddividere l'altra nel modo desiderato con `glk_window_open()`.

[17] E questa è una possibilità importante di cui tenere conto.

3.5. I tipi di finestra

Ecco una descrizione tecnica di tutti i tipi di finestra, e di come si comportano esattamente.

3.5.1. Finestre vuote

Una finestra vuota è sempre vuota. Non gestisce né input né output (è possibile utilizzare `glk_window_get_stream()` su di essa, come su qualunque altra finestra, ma scrivere su questo stream non ha effetto). Una finestra vuota non ha dimensione; `glk_window_get_size()` ritornerà sempre $(0, 0)$ ed è illegale utilizzarla come finestra chiave per una divisione a larghezza fissa.[18]

❖ ❖ ❖

L'*inutilità* delle finestre vuote è sempre più evidente: dato che non hanno dimensione non possono servire nemmeno per forzare un dimensionamento di qualche genere.

Le finestre vuote non servono *veramente* a niente. Se qualcuno ne trova un utilizzo plausibile mi piacerebbe saperlo...

◆ ◆ ◆

3.5.2. Finestre paio

Una finestra paio è completamente riempita dalle due finestre che contiene. Non supporta né input né output e non ha dimensioni.

Non si può creare direttamente una finestra paio; ne viene creata una automaticamente ogni volta che si divide una finestra con `glk_window_open()`. Le finestre paio sono sempre create con un valore 0 sotto al sasso.

È possibile chiudere una finestra paio con `glk_window_close()`; questo provoca la chiusura di tutte le finestre discendenti dal paio.

È legale dividere una finestra paio con `glk_window_open()`.

❖ ❖ ❖

Dividere una finestra paio provoca la riduzione simultanea di entrambe le figlie. Non è proprio intuitivo ma chi ha qualche conoscenza al riguardo degli alberi binari può facilmente comprendere quel che accade in questa circostanza. Una finestra paio però non può essere finestra chiave, in quanto non ha dimensioni.

D'altra parte studiando un layout *a priori* non dovrebbe essere mai necessario richiedere la divisione di una finestra paio.

◆ ◆ ◆

[18] Una finestra vuota non è come non avere finestre. Quando Glk viene avviata non ci sono finestre, neppure finestre vuote.

3.5.3. *Finestre a buffer di testo*

Una finestra a buffer di testo contiene un flusso lineare di testo. Gestisce l'output; quando le viene inviato testo, il nuovo testo viene aggiunto in coda. Non è possibile modificare in alcun modo il testo che è già stato inserito. Non ci sono garanzie su quanto testo la finestra possa contenere; il testo vecchio potrebbe rimanere memorizzato per sempre in modo che l'utente possa rileggerlo, oppure potrebbe essere buttato via appena esce dalla finestra.[19]

La visualizzazione del testo in un buffer di testo è a carico della libreria. Le linee probabilmente non saranno interrotte nel mezzo delle parole; ma se lo sono, la libreria non sta facendo nulla di illegale, solo qualcosa di indecoroso. La selezione del testo e la gestione degli appunti, se disponibili, sono gestiti nel modo preferito dalla piattaforma in uso. I paragrafi (definiti da caratteri di newline nell'output) potrebbero essere indentati.[20]

❖ ❖ ❖

I buffer di testo rappresentano il succo dell'interfaccia di qualunque Avventura Testuale ed è su queste finestre che è concentrata la maggior parte degli algoritmi di garglk-mod. È in queste finestre che avviene il grosso dell'"azione", in particolar modo a riguardo della formattazione dei paragrafi.

Si ricorda che un paragrafo è delimitato da un carattere di newline.

◆ ◆ ◆

Quando un buffer di testo viene cancellato (con `glk_window_clear()`, la libreria farà qualcosa di appropriato; i dettagli possono variare. Potrebbe pulire la finestra con il testo successivo che appare in cima (o anche dal fondo); potrebbe mostrare una quantità di linee bianche sufficiente per far scorrere il testo corrente fuori dalla finestra; potrebbe mostrare un divisore o simbolo distintivo per indicare la fine della pagina.

La dimensione di una finestra a buffer di testo non è necessariamente imprecisa. Chiamando `glk_window_get_size()` si ottiene il numero di righe e di colonne che sarebbero disponibili *se* la finestra fosse riempita di caratteri "0" (zero) nel font *normale*. Poiché d'altra parte la finestra può impiegare caratteri a spaziatura variabile il numero di caratteri disponibili per linea può non essere preciso. La finestra potrebbe anche supportare testo ad altezza variabile (per esempio se il giocatore utilizza un carattere più grande per enfasi); in questo caso anche il numero di righe disponibili sarà impreciso.

Similmente, quando si richiede una divisione a larghezza fissa in base ad un buffer di testo, si richiede una finestra in grado di contenere un numero

[19] Di conseguenza potrebbe esserci o meno una barra di scorrimento controllabile dal giocatore o qualche strumento simile.

[20] Non bisogna in generale simulare l'indentazione mettendo degli spazi in testa ad ogni paragrafo. Questa formattazione è compito della libreria in base alle preferenze del giocatore. Ovviamente sono a carico del programmatore i casi speciali (come le liste indentate).

fisso di righe (o di colonne) di caratteri "0". Il numero effettivo di righe (o di caratteri) visualizzati dipenderà dai font e dal testo visualizzato, come già detto.

❖ ❖ ❖

L'utilizzo del carattere "0" (zero) per determinare la dimensione delle finestre di testo è un'idiozia tipografica che risale agli interpreti originali Infocom per i giochi v6 (quelli grafici).

Delle metriche più appropriate potrebbero essere gli spazi em o en (più rappresentativi del carattere medio di una cifra numerica). Ma le specifiche impongono lo zero, quindi garglk-mod calcola le dimensioni sulla dimensione del carattere zero del font standard. Fortunatamente, in molti font, i caratteri numerici sono studiati per essere grandi quanto un en (in questo modo è più semplice allineare le cifre in colonna).

Le griglie di testo, per loro stessa definizione, sono immuni da questo problema (specialmente per quando riguarda la divisione fissa).

◆ ◆ ◆

In input a caratteri ci sarà un qualche segnale visibile che indichi che la finestra è in attesa della pressione di un tasto (tipicamente un cursore alla fine del testo). Quando il giocatore preme un tasto in quella finestra, viene generato un evento ma il tasto *non* viene visualizzato nella finestra.

❖ ❖ ❖

La stragrande maggioranza degli interpreti *non* è conforme, da questo punto di vista. Sfortunatamente durante l'input a carattere in genere non appaiono segnalazioni per cui spesso gli autori sono costretti a visualizzare in proprio il prompt.

garglk-mod segue la regola, ma in questo modo è possibile che si vedano dei prompt *doppi* (prima quello visualizzato dal gioco, poi quello di garglk-mod).

◆ ◆ ◆

Anche durante l'input a linea ci sarà un segnale visibile. Il modo più comune per il giocatore di comporre una linea di testo è nella finestra stessa, alla fine del testo (questo è quello che viene generalmente fatto nell'IF). Ma non è strettamente necessario. Un approccio alternativo potrebbe essere quello sfruttato dai client MUD: esiste una finestra dedicata di una linea, al di fuori dello spazio Glk, e l'utente digita l'input in quella finestra.[21]

Quando il giocatore finisce la sua linea di input, la libreria mostrerà il testo inserito alla fine del buffer di testo (se non era già presente). Sarà seguita da un newline, di conseguenza il testo visualizzato in seguito inizierà su una linea (paragrafo) successiva a quella dell'input.

[21] Se viene utilizzato questo approccio, bisogna prevedere un modo per permettere la gestione dell'input in due o più finestre. È responsabilità della libreria rendere questa funzionalità disponibile all'utente. Per l'autore basta richiedere l'input a linea sulle finestre desiderate e attendere il risultato.

Se viene chiamata `glk_cancel_line_event()` accade la stessa cosa; il testo in composizione viene reso visibile in coda al buffer, seguito da un newline.

❖ ❖ ❖

Per pura curiosità: la necessità di distinguere lo stile del testo digitato e di rendere editabile il testo pre-inserito rendono impossibile la piena conformità con le specifiche della Z-machine: esiste un'oscura regola al riguardo per compensare uno pseudo-bug negli ultimi giochi Infocom (Z-Machine Standard 1.0, sezione 15 opcode `read`):

> "Moreover, if byte 1 contains a positive value at the start of the input, then read assumes that number of characters are left over from an interrupted previous input, and writes the new characters after those already there. Note that the interpreter does not redisplay the characters left over: the game does this, if it wants to. This is unfortunate for any interpreter wanting to give input text a distinctive appearance on-screen, but *Beyond Zork*, *Zork Zero* and *Shogun* clearly require it."
>
> ("Just a tremendous pain in my butt" – Andrew Plotkin; "the most unfortunate feature of the Z-machine design" – Stefan Jokisch.)

La mia risposta personale: chi se ne frega in fondo di uno o due giochi (per la precisione *tre*) storici? Oltretutto è solo un bug di doppia visualizzazione, nulla di fatale.

◆ ◆ ◆

3.5.4. Finestre a griglia di testo

Una griglia di testo contiene una matrice rettangolare di caratteri, scritti con un font a spaziatura fissa. La sua dimensione è il numero di righe e di colonne della matrice.

Una griglia di testo supporta l'output. Mantiene internamente la posizione del cursore. Quando la finestra viene aperta viene ripulita (con dei caratteri spazio) e il cursore viene posizionato in alto a sinistra, nella posizione $(0, 0)$. Se la finestra viene pulita con `glk_window_clear()`, la finestra viene nuovamente riempita con spazi e il cursore ritorna nell'angolo in alto a sinistra.

Quando viene inviato testo alla finestra, i caratteri sono inseriti nella griglia in ordine, da sinistra verso destra e dall'alto verso il basso, nel normale ordine di lettura. Quando il cursore raggiunge la fine della linea, riparte dall'inizio della linea successiva. La libreria *non* fa alcun tentativo per gestire la divisione delle linee in corrispondenza della fine delle parole.[22]

È possibile spostare il cursore con `glk_window_move_cursor()`.

[22] La visualizzazione di caratteri particolari può causare l'avanzamento del cursore di più di una posizione per carattere. Per esempio la legatura "æ" può essere visualizzata come "ae". Vedere la sezione 2.2 per come gestire questa circostanza.

```
void glk_window_move_cursor(winid_t win,
    glui32 xpos, glui32 ypos);
```

Se il cursore viene spostato a destra oltre alla fine della linea, va a capo; il prossimo carattere verrà visualizzato all'inizio della linea successiva.

Se il cursore viene spostato oltre l'ultima linea, o quando il cursore raggiunge la fine dell'ultima linea, finisce *fuori dallo schermo* e scritture successive non hanno effetto alcuno. Per riportare il cursore all'interno della regione visibile in questi casi è necessario chiamare `glk_window_move_cursor()` o `glk_window_clear()`.[23]

Quando una griglia di testo viene ridimensionata e ristretta, l'area in basso e/o a destra viene scartata, ma l'area visibile rimane inalterata. Quando al contrario viene espansa, la nuova area in basso/a destra viene pulita con degli spazi.[24]

Le griglie di testo gestiscono input a linea, a carattere e perfino da mouse (se il mouse è disponibile, ovviamente).

L'input da mouse ritorna la posizione del carattere su cui è stato cliccato, da $(0, 0)$ a $(larghezza - 1, altezza - 1)$.

❖ ❖ ❖

La libreria garglk originale (pre-mod) ha un grave difetto al riguardo: nelle griglie di testo gli eventi del mouse vengono riportati in pixel, come per le finestre grafiche.

Ed è per questo motivo che non funziona il menù di conversazione di *City of Secrets*: anche se *apparentemente* usa i collegamenti, in realtà viene utilizzato il click del mouse e non un evento di collegamento per la scelta delle frasi. In questo modo, anche se il supporto per i collegamenti non è completo il menù di conversazione è funzionante con garglk-mod.

◆ ◆ ◆

L'input a carattere è come già descritto nella sezione precedente.

L'input a linea è invece leggermente diverso; viene garantito all'interno della finestra, a partire dalla posizione del cursore. Il giocatore può digitare solo fino al margine destro della finestra; di conseguenza la dimensione massima della linea sarà $(larghezza - cursore - 1)$. Se l'argomento `maxlen`

[23] Gli argomenti di `glk_window_move_cursor()` sono senza segno. Di fatto, non è un problema visto che non esistono posizioni negative. Se si prova ad utilizzare un valore negativo, Glk lo interpreterà come positivo (e molto grande) per cui si ricadrà nella situazioni *a capo* o *fuori schermo*.

Inoltre il cursore delle griglie di testo *non* è necessariamente visibile. In particolare, quando si richiede input (a linea o a carattere) in una griglia di testo, non necessariamente sarà presente un cursore per guidare il giocatore. Conviene quindi mostrare un prompt nel punto indicato: un carattere ">" per esempio.

[24] Può essere utile gestire gli eventi `evtype_Arrange` per ridisegnare il contenuto delle griglie di testo quando cambiano dimensione.

passato a `glk_request_line_event()` è più piccolo, la libreria non consentirà l'input oltre a `maxlen` caratteri dal punto di partenza.[25]

Quando il giocatore conferma la sua linea di input, questa rimane visibile nella finestra, e il cursore viene posizionato all'inizio della riga *successiva*. Lo stesso accade se viene utilizzata `glk_cancel_line_event()`.

❖ ❖ ❖

L'input a linea su griglie di testo è inteso per la compilazione di moduli a schermo (l'unico in esistenza credo sia all'inizio di *Bureocracy* per esempio). Ma non ho ancora visto nessun esempio di questo utilizzo, in circolazione.

◆ ◆ ◆

3.5.5. *Finestre grafiche*

Una finestra grafica contiene un matrice rettangolare di pixel. La sua dimensione è il numero di colonne e righe della matrice.

Ogni finestra grafica ha un colore di sfondo, che è inizialmente bianco; è possibile però cambiarlo, vedere la sezione 7.2.

Quando una finestra grafica viene ridimensionata e ridotta, viene scartata la parte che non è più visibile, a destra e/o in basso. Quando viene invece ingrandita, la nuova sezione esposta (a destra e/o in basso) viene riempita con il colore di sfondo.[26]

In alcune librerie è possibile ricevere in qualunque momento un evento di ridisegno grafico (`evtype_Redraw`). In questo caso, la finestra è già stata ripulita con il colore di sfondo e deve essere ridisegnata. Se si creano finestre grafiche è *obbligatorio* gestire questi eventi.[27]

Per una descrizione delle funzioni di disegno che si applicano alle finestre grafiche vedere la sezione 7.2.

Le finestra grafiche non supportano né input né output di testo.

---

[25] In questo modo è possibile far compilare un campo a larghezza fissa, senza che il giocatore possa sovrascrivere altre parti della finestra.

[26] È possibile gestire gli eventi `evtype_Arrange` per ridisegnare le finestre grafiche quando cambiano dimensione.

[27] Gli eventi di ridisegno possono essere emessi quando la finestra Glk viene scoperta o resa in altro modo visibile. Altre librerie Glk possono gestire il problema in maniera automatica (per esempio salvandone il contenuto) e quindi non inviare eventi di ridisegno. Comunque, in una varietà di situazioni questa memoria potrebbe non essere adeguata, per esempio se cambia la risoluzione o il numero di colori dello schermo. Di conseguenza gli eventi di ridisegno sono sempre una possibilità, anche nelle librerie più evolute. Bisogna essere sempre preparati per gestirli.

Quando viene creata la finestra, *non* viene inviato un evento. Si presume che il disegno avverrà immediatamente. Inoltre gli eventi di ridisegno non vengono emessi quando la finestra viene ridimensionata: se ne è responsabile il giocatore il caso è coperto dall'evento di ridimensionamento, altrimenti è il programma stesso che deve averlo richiesto e ridisegnare di conseguenza.

Non tutte le librerie supportano le finestre grafiche. È possibile verificare la disponibilità delle funzioni grafiche con il sistema gestalt. In un programma C è anche possibile verificare se le funzioni grafiche sono disponibili in fase di compilazione. Vedere la sezione 7.4.[28]

3.6. Stream di eco

Ogni finestra ha uno stream di finestra associato; si scrive sulla finestra scrivendo su questo stream. È anche possibile collegare un secondo stream ad una finestra. Ogni testo visualizzato sulla finestra viene anche rimandato su questo secondo stream che prende quindi il nome di *stream di eco*.

In pratica, ogni chiamata a `glk_put_char()` (o altri comandi di output) diretti allo stream della finestra viene replicata sullo stream di eco. Questo vale anche per comandi di stile quali `glk_set_style()`.

Notare che la replicazione è a senso unico. È ancora possibile mandare testo direttamente sullo stream di eco, e finirà correttamente nel posto dove deve finire (a seconda del tipo di stream) ma non finirà sulla finestra.

```
void glk_window_set_echo_stream(winid_t win,
    strid_t str);
strid_t glk_window_get_echo_stream(winid_t win);
```

Inizialmente ogni finestra non ha alcuno stream di eco associato; in questa situazione `glk_window_get_echo_stream(win)` ritorna `NULL`. È in seguito possibile impostare lo stream di eco di una finestra ad un qualsiasi stream valido con la funzione `glk_window_set_echo_stream(win, str)`.

La chiamata `glk_window_set_echo_stream(win, NULL)` permette di rimuovere lo stream di eco associato ad una finestra.

Uno stream di eco può essere di qualunque tipo, anche lo stream di un'altra finestra.[29]

Una finestra può avere solo uno stream di eco. Ma uno stream può essere lo stream di eco di una o più finestre, anche simultaneamente.

Se una finestra viene chiusa, il suo stream di eco rimane aperto; in questo caso *non* viene chiuso automaticamente.[30]

[28] Inoltre, come tutte le finestre, bisogna controllare che non venga ritornato `NULL` durante la creazione di una finestra grafica.

[29] Anche se sarebbe una cosa abbastanza stupida, visto che il testo mostrato in una finestra verrebbe duplicato in un'altra. Generalmente, l'eco di una finestra viene utilizzato con uno stream su file per ottenere la trascrizione di quella finestra.

[30] Attenzione a non confondere lo stream della finestra con il suo stream di eco. Lo stream della finestra *fa parte* della finestra, e con essa viene chiuso. Lo stream di eco è solo temporaneamente associato con essa.

Se uno stream viene chiuso, ed è lo stream di eco di una o più finestre, queste lo perdono come stream di eco. Chiamando quindi in seguito con una di queste finestre glk_window_get_echo_stream() si otterrà NULL.

È illegale settare lo stream di eco di una finestra al proprio stream di finestra. Questo creerebbe un ciclo infinito con risultati probabilmente catastrofici. È similmente illegale creare un ciclo più lungo (due o più finestre che si fanno eco l'una con l'altra).

3.7. Altre funzioni sulle finestre

```
winid_t glk_window_iterate(winid_t win,
    glui32 *rockptr);
```

Questa funzione può essere impiegata per iterare sulla lista di tutte le finestre aperte (comprese le finestre paio). Vedere la sottosezione 1.6.2.

Come descritto, l'ordine in cui sono ritornate le finestre è arbitrario. La finestra radice non è necessariamente la prima, né necessariamente l'ultima.

```
glui32 glk_window_get_rock(winid_t win);
```

Ritorna il valore sotto al sasso della finestra. Le finestre paio hanno sempre il valore 0. Le altre finestre hanno il valore passato durante la creazione.

```
glui32 glk_window_get_type(winid_t win);
```

Viene ritornato il tipo della finestra (wintype_...).

```
winid_t glk_window_get_parent(winid_t win);
```

La funzione ritorna la finestra che è padre della finestra indicata. Se win è la finestra radice, ritorna NULL, visto che la radice non ha padre. Il padre di ogni finestra è sempre una finestra paio; tutti gli altri tipi di finestra non hanno figli.

```
winid_t glk_window_get_sibling(winid_t win);
```

Ritorna l'altra figlia della finestra padre della finestra specificata (la sorella, in altre parole). Se win è la radice, ritorna NULL.

```
winid_t glk_window_get_root(void);
```

Ritorna la finestra radice. Se non ci sono finestre, ritorna NULL.

```
void glk_window_clear(winid_t win);
```

Pulisce la finestra. Il significato dell'operazione varia a seconda del tipo di finestra:

Buffer di testo possono accadere diverse cose, come la cancellazione di tutto il testo nella finestra o far scorrere il contenuto con delle linee vuote o anche inserire un marcatore di salto pagina;

Griglia di testo pulisce la finestra riempiendo ogni carattere con degli spazi. Il cursore viene portato in alto a sinistra (posizione $(0, 0)$);

Grafica pulisce la superficie dell'intera finestra con il colore di sfondo. Vedere la sottosezione 3.5.5;

Altri tipi di finestra nessun effetto.

È illegale tentare di pulire una finestra che ha in corso una richiesta di input a linea.

```
strid_t glk_window_get_stream(winid_t win);
```

Ritorna lo stream associato con la finestra (vedere la sottosottosezione 5.5.3). Ogni finestra ha uno stream su cui si può inviare testo, anche se, a seconda del tipo di finestra, non necessariamente è utile.[31]

```
void glk_set_window(winid_t win);
```

Questa funzione seleziona lo stream della finestra come corrente. È esattamente equivalente a scrivere:

`glk_stream_set_current(glk_window_get_stream(win)).`

Vedere il Capitolo 5.

[31] Per esempio, mostrare testo su una finestra vuota non ha effetto.

4 Eventi

Come già descritto, tutte le attività del giocatore sono inviate al programma per mezzo della funzione glk_select(), sotto forma di eventi. È necessario scrivere almeno un ciclo per gli eventi per poterli gestire correttamente.

```
typedef struct event_struct {
    glui32 type;
    winid_t win;
    glui32 val1, val2;
} event_t;

void glk_select(event_t *event);
```

Con questa funzione il programma si arresta in attesa di un evento, per poi restituirlo nella struttura puntata dall'argomento. A differenza di molte altre funzioni Glk che prendono puntatori, l'argomento di glk_select() non può essere NULL.

Nella maggior parte dei casi, si ricevono solo gli eventi che sono stati richiesti. Esistono però alcuni eventi che possono arrivare in qualsiasi momento. Questo è il motivo per cui bisogna sempre chiamare glk_select() all'interno di un ciclo, e continuare a iterare fino alla ricezione dell'evento desiderato.

La struttura evento è autoesplicativa. type è il tipo di evento. La finestra da cui proviene l'evento, se significativa, è in win. I campi rimanenti contengono informazioni specifiche all'evento.

I tipi di evento sono elencati in Tabella 4.1.

Notare che evtype_None è zero e che gli altri valori sono positivi. I tipi di evento negativi (0x80000000–0xFFFFFFFF) sono riservati per eventi specifici all'implementazione della libreria.

Tabella 4.1. Tipologie di evento definite in Glk 0.7

Evento	Descrizione
evtype_None	Nessun evento. Questo valore non viene mai ritornato durante un'attesa.
evtype_Timer	Un evento che si ripete ad intervalli fissi
evtype_CharInput	Un evento di pressione tasto in una finestra.
evtype_LineInput	L'evento di input a linea completato in una finestra.
evtype_MouseInput	Un click del mouse in una finestra.
evtype_Arrange	Un evento che indica che la dimensione di alcune finestre è variata.
evtype_Redraw	Un evento che indica che le finestre grafiche devono essere ridisegnate.
evtype_Hyperlink	La selezione di un hyperlink in una finestra.

È anche possibile chiedere se è disponibile un evento, senza fermarsi ad attenderne uno in caso contrario.

```
void glk_select_poll(event_t *event);
```

Verifica se un evento generato internamente è disponibile. In tal caso, lo memorizza nella struttura puntata da event. In caso contrario, imposta il tipo dell'evento ritornato sarà evtype_None. In ogni caso ritorna quasi immediatamente.

La prima domanda da porsi ora è: quali sono gli eventi generati internamente? La funzione glk_select_poll() *non* ritorna eventi relativi all'interazione con il giocatore (generati quindi esternamente), come evtype_CharInput, evtype_LineInput o evtype_MouseInput. Il suo utilizzo è inteso nei casi in cui si vuole controllare la presenza di eventi durante la computazione ma non si è pronti alle richieste del giocatore. Per esempio, la funzione glk_select_poll() può essere utile per verificare la presenza di eventi evtype_Timer durante delle operazioni lunghe. Vedere la sezione 4.4.

Al momento, glk_select_poll() reagisce alla presenza di tre tipi di evento: evtype_Timer (se il timer è attivo e supportato), evtype_Arrange e, se il sonoro è supportato con le relative notifiche, evtype_SoundNotify. Vedere anche la sezione 4.9.

La seconda domanda è, cosa significa che glk_select_poll() ritorna *quasi immediatamente*? In alcune librerie Glk il testo inviato alle finestre è bufferizzato; non appare fisicamente fino a quando si passa il controllo al giocatore con glk_select(). Similmente fa glk_select_poll(), anche se non

Figura 4.1. Polling in attesa di eventi multipli

```
glk_select_poll(&ev);
while (ev.type != evtype_None) {
    /* gestire l'evento */
    glk_select_poll(&ev);
}
```

gestisce il "*Premi un tasto per proseguire*" a schermo pieno; quello fa parte dell'interazione con l'utente.

Similmente, in alcune piattaforme cooperative, `glk_select()` potrebbe cedere il processore ad altri processi; allo stesso modo potrebbe funzionare la chiamata `glk_select_poll()`, se necessario.

La morale è che non bisogna chiamare `glk_select_poll()` troppo spesso. Anzi, se non vengono effettuate molte operazioni fra una `glk_select()` e l'altra, e quindi l'intervallo fra una chiamata e l'altra è ridotto, non è neppure necessaria.[1]

Un momento adatto per impiegare la funzione `glk_select_poll()` è durante calcoli intensivi, per mantenere responsivo il sistema e cedere tempo ad altri processi. Un altro caso è durante la visualizzazione dei risultati intermedi; senza una chiamata a `glk_select_poll()` non necessariamente il giocatore vedrà lo schermo aggiornarsi fino alla successiva `glk_select()`.

Di solito la chiamata a `glk_select_poll()` non viene messa all'interno di un ciclo. Si usa generalmente per aggiornare lo schermo o verificare se ci sono eventi disponibili, non per aspettare un evento specifico. Un caso in cui può essere utile far ciò è durante l'utilizzo degli eventi di notifica audio. Se sono in riproduzione diversi canali audio potrebbe essere importante sapere quanti e quali sono completi in un dato momento. Un ciclo tipico per gestire l'attesa di eventi multipli è presentato in Figura 4.1.

Ma quando `glk_select_poll()` ritorna `evtype_None`, *non* bisogna richiamarla immediatamente. Sarebbe meglio fare dell'altro prima. Se si vuole attendere un evento senza avere altre operazioni da eseguire nel frattempo è meglio usare `glk_select()`, non un ciclo con `glk_select_poll()`.

❖ ❖ ❖

In garglk-mod *non* esiste una coda degli eventi propriamente detta. Esistono dei casi particolarmente esoterici per cui, in linea di principio, è possibile perdere un evento (in particolar modo riesposizioni o altre notifiche minori).

[1] Per esempio, in un interprete non è buona idea utilizzarla dopo ogni opcode.
È bene però chiamare `glk_tick()` spesso; una volta per ogni opcode in un interprete di VM. Vedere la sezione 1.4.

Anche se la possibilità tuttora esiste, non ho ancora riscontrato questo comportamento in nessuna situazione.

◆ ◆ ◆

4.1. Eventi di input a carattere

L'input a carattere è disponibile (e può essere richiesto) su finestre a buffer di testo e a griglia di testo. Esistono funzioni separate per richiedere input Latin-1 e Unicode; vedere la sezione 2.1.

```
void glk_request_char_event(winid_t win);
```

Richiede l'input di un carattere Latin-1 o di un tasto speciale. Una finestra non può avere in corso simultaneamente richieste di input a carattere e di linea. Similmente non può avere in corso richieste di input a carattere Latin-1 e Unicode allo stesso momento.

Non è legale chiamare glk_request_char_event() se la finestra ha già in corso una richiesta di input a carattere o a linea.

Unicode
Glk 0.7

```
void glk_request_char_event_uni(winid_t win);
```

Richiede l'input di un carattere Unicode o di un tasto speciale.

```
void glk_cancel_char_event(winid_t win);
```

Annulla una richiesta di input a carattere (sia di tipo Latin-1 che Unicode). Per comodità, è lecito chiamare glk_cancel_char_event() anche se non c'è input a carattere in corso sulla finestra. Glk in tal caso ignora la chiamata.

Quando una finestra ha attiva una richiesta per input a carattere e il giocatore preme un tasto in quella finestra, glk_select() ritorna un evento di tipo evtype_CharInput. Dopodiché la richiesta è completa e non più attiva. Bisogna chiamare nuovamente glk_request_char_event() (o la sua variante Unicode glk_request_char_event_uni()) per ottenere un nuovo tasto da quella finestra.

Nella struttura evento, win indica da quale finestra proviene il carattere. val1 indica il carattere premuto; questo potrà essere un codice carattere o un codice speciale (vedere la sezione 2.4). Se la richiesta originale era di tipo glk_request_char_event(), il valore di val1 sarà 0–255 oppure un codice speciale. In ogni caso il valore di val2 sarà 0.

4.2. Eventi di input a linea

È possibile richiedere input a linea dalle finestre a buffer e a griglia di testo. Esistono funzioni separate per richiede input Latin-1 e Unicode; vedere la sezione 2.1.

```
void glk_request_line_event(winid_t win,
    char *buf, glui32 maxlen, glui32 initlen);
```

Richiede l'input di una linea di caratteri Latin-1. Una finestra non può aver attive richieste di input a linea e a carattere simultaneamente. E nemmeno si può richiedere input di linea di entrambi i tipi (Latin-1 e Unicode). È quindi illegale chiamare glk_request_line_event() se la finestra ha già attiva una richiesta di input a linea o a carattere.

L'argomento buf è un puntatore ad uno spazio in memoria in cui memorizzare l'input (NULL non è valido). maxlen è la lunghezza di questo spazio, in byte; la libreria non accetterà più caratteri di quanti ne possa contenere. Se initlen non è zero, i primi initlen byte di buf saranno considerati come input preesistente, come se fossero appena stati digitati dal giocatore.[2]

Il contenuto del buffer è indefinito fino al completamento dell'input (con un evento di input a linea o a causa di glk_cancel_line_event()). La libreria è libera di utilizzare o meno il buffer durante la digitazione da parte del giocatore, finché l'input rimane attivo; è illegale cambiare il contenuto del buffer in questo periodo.

```
void glk_request_line_event_uni(winid_t win,
    glui32 *buf, glui32 maxlen, glui32 initlen);
```

Unicode
Glk 0.7

Richiede input di linea in caratteri Unicode. Il funzionamento è identico a quello di glk_request_line_event(), tranne per il fatto che il formato del buffer è di valori glui32 piuttosto che di caratteri, e che questi valori sono codepoint Unicode validi e non solo Latin-1.

Il risultato sarà in Unicode Normalization Form C. Il che significa, in sostanza, che i caratteri compositi saranno caratteri singoli ove possibile e non sequenze di caratteri base e combinatori. Vedere l'Annesso #15 dello standard Unicode per i dettagli.

```
void glk_cancel_line_event(winid_t win,
    event_t *event);
```

Annulla una richiesta attiva per input a linea (sia essa di tipo Latin-1 o Unicode). L'evento puntato dall'argomento event sarà valorizzato come se il

[2] Il giocatore potrà poi digitare in seguito o cancellarli o modificarli come al solito.

giocatore avesse premuto invio, e il testo digitato fino a quel punto inserito nel buffer. Se non interessa l'informazione è possibile passare NULL (anche se il buffer verrà comunque valorizzato).

Per comodità è legale chiamare glk_cancel_line_event() anche se non esistono richieste di input a linea attive sulla finestra. In questo caso verrà ritornato un evento di tipo evtype_None.

Se una finestra ha attiva una richiesta per input a linea e il giocatore preme invio in quella finestra (o compie qualsivoglia azione necessaria per confermare quel che ha digitato), la successiva chiamata a glk_select() ritornerà un evento di tipo evtype_LineInput. Dopodiché la richiesta è completa e non più attiva. Per ottenere un'altra linea di input da quella finestra è necessario chiamare un'altra volta glk_request_line_event() (o la sua variante Unicode glk_request_line_event_uni(), nel caso).

Nella struttura evento win indica da quale finestra proviene l'evento. val1 contiene il numero di caratteri digitati. val2 sarà comunque 0. I caratteri digitati saranno inseriti nel buffer specificati dalla chiamata che ha richiesto l'evento.[3]

4.3. Eventi del mouse

Su alcune piattaforme Glk è in grado di stabilire quando il mouse (o un simile puntatore) viene impiegato per selezionare un punto di una finestra. È possibile richiedere gli eventi del mouse solo nelle finestre grafiche o nelle griglie di testo.

```
void glk_request_mouse_event(winid_t win);
void glk_cancel_mouse_event(winid_t win);
```

Una finestra può avere attivi simultaneamente il mouse e l'input a carattere/linea.

Se il giocatore clicca in una finestra che ha attivo il mouse, glk_select() ritornerà un evento di tipo evtype_MouseInput. Come al solito, la richiesta verrà considerata completa e il mouse andrà riattivato per ottenere ulteriori eventi.

Nella struttura evento win indica da quale finestra proviene l'evento.

In una griglia di testo, i campi val1 e val2 sono le coordinate x e y del carattere su cui si è cliccato.[4] Il carattere in alto a sinistra è nella posizione $(0, 0)$.

In una finestra grafica, similmente, sono le coordinate x e y del pixel su cui si è cliccato. Ancora, il pixel in alto a sinistra è nella posizione $(0, 0)$.

[3] Non è previsto alcun terminatore per la stringa nel buffer.

[4] Quindi val1 è la colonna e val2 è la riga.

È possibile verificare se gli eventi del mouse sono supportati in una finestra per mezzo del selettore gestalt_MouseInput.

```
res = glk_gestalt(gestalt_MouseInput, windowtype);
```

Il valore ritornato sarà TRUE (1) se la finestra del tipo specificato supporta il mouse. Nel caso il risultato sia FALSE (0), è comunque legale chiamare glk_request_mouse_event() ma questa non avrà effetto e non si otterranno eventi mouse.[5]

❖ ❖ ❖

Gli eventi del mouse sono completamente gestiti in garglk-mod. Visto che però i collegamenti (hyperlink) ancora *non* sono completamente implementati, devo ancora decidere il comportamento da seguire nel caso siano *entrambi* richiesti.

◆ ◆ ◆

44. Eventi del timer

È possibile richiedere l'invio di un evento a intervalli regolari, indipendentemente da quel che fa il giocatore. A differenza degli eventi di input, gli eventi del timer sono ritornati da glk_select_poll(), oltre che da glk_select().

```
void glk_request_timer_events(glui32 millisecs);
```

È però possibile che la libreria non gestisca gli eventi del timer. Il selettore da verificare in questo caso è gestalt_Timer.

```
res = glk_gestalt(gestalt_Timer, 0);
```

Al solito, ritorna TRUE (1) se il timer è supportato, FALSE (0) altrimenti.

Inizialmente il timer è disattivato e non si ricevono eventi relativi ad esso. Se si chiama glk_request_timer_events(N) con N diverso da 0, si riceveranno in seguito eventi del timer ogni circa N millisecondi[6] (sempre

[5] La maggior parte degli idiomi basati su mouse definiscono funzioni standard per l'utilizzo del mouse nelle finestre di testo, tipicamente selezionare o copiare testo. È compito della libreria distinguerli dagli eventi mouse Glk. La libreria può scegliere di selezionare testo quando viene cliccato normalmente e generare eventi mouse quando viene CTRL-cliccato; oppure al contrario; oppure gestire il doppio click; o ancora un tasto distinto del mouse. Addirittura, potrebbe essere definito sulla tastiera il *tasto del mouse* che si riferisce alla posizione corrente del mouse quando viene premuto. Si potrebbero addirittura inventare altri sistemi esoterici di posizionamento. L'importante è sapere se l'utente può selezionare una posizione oppure no.

D'altra parte, siccome le piattaforme si comportano diversamente al riguardo, è necessario essere cauti nell'indicare al giocatore l'operazione da eseguire. Meglio non dire "cliccare", "doppio-cliccare" o "control-cliccare' in una finestra. Piuttosto, il termine preferito è "toccare la finestra" o un punto della finestra.

[6] Non è necessario dire che un millisecondo è un millesimo di secondo, vero?

che il timer sia supportato, in caso contrario la chiamata non ha effetto). Diversamente dagli eventi relativi a mouse e tastiera, gli eventi del timer si ripetono fino a quando non vengono disattivati. Non è quindi necessario richiederli ogni volta che se ne riceve uno. Per disattivare il timer, chiamare `glk_request_timer_events(0)`.

La regola è che durante una chiamata a `glk_select()` (e, visto che il timer è un evento generato internamente, a `glk_select_poll()`) se sono passati più di `N` millisecondi dall'ultimo evento del timer generato e se non c'è stato input da parte del giocatore (quest'ultima condizione vale solo per `glk_select()`) viene generato un evento di tipo `evtype_Timer` (`win`, `val1` e `val2` saranno tutti 0).

Gli eventi del timer non si accumulano. Anche aspettando `10N` millisecondi prima di chiamare `glk_select()`, non si riceveranno dieci eventi del timer. La libreria osserverà che sono passati più di `N` millisecondi e ritornerà immediatamente un evento del timer. Per ottenerne un altro ci vorranno comunque altri `N` millisecondi.

In sostanza, i tempi sono approssimati, e l'approssimazione cade dal lato del ritardo. Se sono disponibili sia eventi del timer sia eventi del giocatore, il giocatore ha la priorità.[7]

4.5. Eventi di ridimensionamento

Alcune piattaforme consentono al giocatore di ridimensionare la finestra di Glk. Questo naturalmente provoca una variazione della dimensione delle finestre. In questo caso, immediatamente *dopo* il ridimensionamento, `glk_select()` ritorna un evento di tipo `evtype_Arrange`. È possibile utilizzare questa notifica per ri-visualizzare i contenuti di una finestra grafica o di una griglia di testo.[8]

Nella struttura dell'evento `win` è `NULL` se tutte le finestre sono influenzate. Se solo alcune finestre sono state toccate, `win` si riferisce ad una finestra che le contiene tutte.[9] `val1` e `val2` sono sempre a `0`.

Un evento di riposizionamento è garantito quando il giocatore fa in modo che una qualunque finestra cambi dimensione, misurata nella sua metrica.[10]

[7] In questo modo il giocatore non può essere bloccato da eventi del timer troppo frequenti. D'altra parte, su macchine lente, il timer potrebbe essere bloccato da un giocatore smanettone. Glk non è un sistema operativo real-time.

[8] La visualizzazione dei buffer di testo è completamente a carico della libreria, quindi non bisogna preoccuparsene.

[9] Ma per essere sicuri si può ignorare `win` e ridisegnare ogni singola finestra grafica e griglia di testo.

[10] I cambi di dimensione generati dal programma (per esempio aprendo, chiudendo o ridimensionando una finestra) non generano eventi di questo tipo. In questo caso bisogna tener conto

Gli eventi di ridimensionamento, come gli eventi del timer, possono essere restituiti anche da `glk_select_poll()`. Possono, ma non necessariamente su tutte le piattaforme. Bisogna essere pronti a gestire questo evento anche nella gestione di `glk_select()`.[11]

4.6. Eventi di ridisegno

Sulle piattaforme che gestiscono finestre grafiche è possibile che il contenuto di una finestra grafica venga perduto, e quindi debba essere ridisegnato da capo. In questo caso `glk_select()` ritornerà un evento di tipo `evtype_Redraw`.

Nella struttura evento, `win` sarà `NULL` se tutte le finestre sono coinvolte. Se solo alcune finestre necessitano di essere ridisegnate, `win` si riferirà ad una finestra che le contiene tutte.[12] `val1` e `val2` saranno entrambe a `0`.

Le finestre da ridisegnare sono cancellate al loro colore di sfondo prima di inviare l'evento di ridisegno.

Come per gli eventi di ridimensionamento, anche quelli di ridisegno possono essere ritornati da `glk_select_poll()`. Ma anche in questo caso il comportamento varia a seconda della piattaforma. Vedere la sezione 4.5.

Per ulteriori informazioni sugli eventi di ridisegno e come influiscono sulle finestre grafiche vedere la sottosezione 3.5.5.

❖ ❖ ❖

garglk (così come garglk-mod) si basano su un frame buffer grande come tutta l'area utente, per questioni di compatibilità (e per avere un alpha blending decente). Per cui, in generale, niente eventi di ridisegno.

È quindi meglio testare le applicazioni su interpreti diversi, come sempre.

◆ ◆ ◆

4.7. Eventi di notifica audio

Su alcune piattaforme in grado di gestire il sonoro, è possibile richiedere di ricevere un evento di tipo `evtype_SoundNotify` quando termina la riproduzione audio. Vedere la sezione 8.3.

---

delle operazioni effettuate e ridisegnare le finestre influenzate.

Ci possono essere diversi tipi di azione che provocano un ridimensionamento. Per esempio, cambiando risoluzione dello schermo; oppure selezionando un font di dimensione diversa, dato che le finestre sono dimensionate in base alla dimensione del font.

[11] In alcune piattaforme il ridimensionamento è gestito come parte dell'interfaccia con il giocatore; su altre può essere generato da processi esterni, come un window manager.

[12] Per sicurezza e comodità è ovviamente possibile ignorare `win` e ridisegnare ogni finestra grafica.

❖ ❖ ❖

garglk *non* supporta le notifiche audio. garglk-mod, si.

◆ ◆ ◆

4.8. Eventi di hyperlink

Sulle piattaforme che gestiscono i collegamenti, è possibile richiedere di ricevere un evento di tipo `evtype_Hyperlink` quando il giocatore seleziona un collegamento. Vedere la sezione 9.2.

❖ ❖ ❖

Al momento niente hyperlink né in garglk né in garglk-mod (ammetto che, imbrogliando, pubblico il supporto per i collegamenti a livello di gestalt. Come se qualcuno lo controllasse!). garglk-mod è in grado di sottolineare il testo facente parte di un collegamento, ma poco più.

È interessante notare che, nonostante il selettore gestalt indichi che i link non sono disponibili, *City of Secrets* li richiede lo stesso.

Questi autori pigri che non seguono le regole! (gioco eccellente sotto tutti i punti di vista, a parte queste pignolerie)

◆ ◆ ◆

4.9. Altri eventi

Attualmente non sono definiti altri eventi in Glk. La costante "`evtype_None`" è fittizia e non è mai ritornata da `glk_select()`.

Ovviamente, in futuro potranno essere definiti nuovi tipi di evento.[13]

[13] Per esempio, nel caso in cui venissero aggiunte funzionalità video o di animazione, probabilmente verrebbe definita qualche tipo di notifica di completamento.

5 Stream

Ogni forma di visualizzazione e/o scrittura di caratteri in Glk è fatta attraverso uno stream. Ogni finestra ha uno stream di output associato con essa. È anche possibile scrivere file su disco; anche ogni file aperto è rappresentato da uno stream.

Esistono anche stream di input; vengono utilizzati per leggere file da disco. È possibile che uno stream sia sia in input che in output.[1]

È anche possibile creare uno stream che legge o scrive utilizzando un buffer in memoria.

Per finire, potrebbero esistere tipi particolari di stream su alcune piattaforme, creati ancor prima dell'inizio del programma.[2] Non è necessario preoccuparsi dell'origine di questi stream; si usano esattamente come gli altri. Per informazioni su come vengono in essere gli stream specifici alla piattaforma, vedere la sezione 10.1.

❖ ❖ ❖

Per l'autore di una comune Avventura Testuale gli stream sono di scarsa utilità. Vengono utilizzati implicitamente per visualizzare testo sulle finestre e gli stream necessari per salvare/ripristinare la situazione sono già gestiti automaticamente dalla libreria standard di Inform.

---

[1] La gestione dell'input del giocatore, d'altra parte, è fatta per mezzo di eventi, non stream. Questa, dal punto di vista teorico è una lieve ineleganza. In pratica, l'input del giocatore è lento e molte cose possono interromperlo, mentre l'input da file è immediato. Nel caso in cui venisse proposta un'estensione di rete per Glk, molto probabilmente utilizzerebbe eventi e non stream, dato che la comunicazione di rete non è immediata.

[2] Per esempio, un programma in esecuzione sotto Unix potrebbe avere accesso allo standard input come stream, anche se non esiste alcuna chiamata Glk dedicata. Su Mac, i dati in una risorsa Mac potrebbero essere disponibili attraverso un apposito tipo di stream.

Per qualche operazione particolare potrebbero essere utili gli stream in memoria (e per questo sono resi disponibili dalla libreria standard per mezzo della funzione `PrintAnyToArray` (sezione 14.3).

Di conseguenza, a meno di voler avere a che fare con i file esterni, la maggior parte di questo capitolo è di scarsa utilità; ma attenzione all'importantissima sezione sugli stili (sezione 5.5).

◆ ◆ ◆

Uno stream viene aperto con una particolare modalità:[3]

`filemode_Write` Uno stream in scrittura.

`filemode_Read` Uno stream in lettura.

`filemode_ReadWrite` Uno stream capace sia di lettura che di scrittura.

`filemode_WriteAppend` Uno stream in scrittura, ma in grado di aggiungere dati in coda alla destinazione, invece di sovrascriverla.

Per informazioni sull'apertura di nuovi stream, vedere la discussione di ogni specifico tipo di stream in la sottosezione 5.5.3. È sempre comunque possibile che l'apertura fallisca, nel qual caso la funzione di creazione ritornerà `NULL`.

Ogni stream ha associati due contatori di caratteri: il numero di caratteri scritti e rispettivamente letti dallo stream. Il conteggio in scrittura è avanzato esattamente di uno per ogni chiamata `glk_put_char()`; viene calcolato prima di ogni considerazione specifica alla piattaforma.[4] Il conteggio in lettura è esattamente di uno per ogni chiamata `glk_get_char_stream()` che ha avuto successo; in pratica per ogni carattere ritornato (escludendo la segnalazione di end of file).

Glk possiede la nozione di *stream (di output) corrente*. Se si scrive del testo senza specificare uno stream, finisce nello stream di output corrente. D'altra parte lo stream corrente può essere `NULL`, indicando che non ce n'è uno. È illegale mandare del testo allo stream `NULL`, o usare lo stream corrente quando non esiste.

Se lo stream che è attualmente il corrente viene chiuso, lo stream corrente diviene `NULL` (cessa di esistere).

[3] Le modalità di apertura sono derivate direttamente da stdio, dalla funzione `fopen()` (rispettivamente "`w`" per sola scrittura, "`r`" per sola lettura, "`r+`" in lettura scrittura e "`a`" in append).

[4] Per esempio, se il carattere di newline è convertito in linefeed e carriage return, lo stream avanza di un solo carattere; lo stesso vale se un carattere accentato o una legatura viene visualizzato come due o più caratteri.

```
void glk_stream_set_current(strid_t str);
```

Con questa funzione viene selezionato lo stream str (che deve essere in output) come corrente. È possibile rimuovere lo stream corrente settandolo a NULL.

```
strid_t glk_stream_get_current(void);
```

Ritorna lo stream corrente oppure NULL se non c'è.

❖ ❖ ❖

Può sembrare banale, ma è degno di nota: in Inform print utilizza sempre lo stream corrente!

◆ ◆ ◆

5.1. Come scrivere

```
void glk_put_char(unsigned char ch);
```

Manda un carattere allo stream corrente. Come tutte le funzioni di base, il carattere viene interpretato secondo la codifica Latin-1. Vedere la Capitolo 2.

```
void glk_put_string(char *ch);
```

Manda una stringa terminata da NULL sullo stream corrente. È esattamente equivalente a

```
for (ptr = s; *ptr; ptr++)
    glk_put_char(*ptr);
```

...anche se potenzialmente più efficiente.

```
void glk_put_buffer(char *buf, glui32 len);
```

Manda un blocco di carattere sullo stream corrente. In modo simile alla precedente è esattamente equivalente a

```
for (i = 0; i < len; i++)
    glk_put_char(buf[i]);
```

Allo stesso modo, potrebbe essere più efficiente.

```
void glk_put_char_stream(strid_t str,
    unsigned char ch);
void glk_put_string_stream(strid_t str,
    char *s);
void glk_put_buffer_stream(strid_t str,
    char *buf, glui32 len);
```

Sono le stesse funzioni, con la possibilità di specificare uno stream da utilizzare come destinazione. Al solito, NULL è illegale, come pure uno stream in sola lettura.

Unicode Glk 0.7

```
void glk_put_char_uni(glui32 ch);
```

Manda un carattere sullo stream corrente, ma in questo caso il carattere viene interpretato come codepoint Unicode. Vedere la Capitolo 2.

Unicode Glk 0.7

```
void glk_put_string_uni(glui32 *s);
```

Manda sullo stream corrente una stringa di caratteri Unicode. È equivalente ad una serie di chiamate a glk_put_char_uni(). La stringa termina su un valore 0.

Unicode Glk 0.7

```
void glk_put_buffer_uni(glui32 *s, glui32 len);
```

Manda un blocco di caratteri Unicode sullo stream corrente. Anche questa equivalente ad una serie di chiamate a glk_put_char_uni().

Unicode Glk 0.7

```
void glk_put_char_stream_uni(strid_t str,
    glui32 ch);
void glk_put_string_stream_uni(strid_t str,
    glui32 *s);
void glk_put_buffer_stream_uni(strid_t str,
    glui32 *buf, glui32 len);
```

Completano la serie delle funzioni di output su stream e sono, come è facile immaginare, le versioni Unicode su uno stream specificato.

5.2. Come leggere

```
glsi32 glk_get_char_stream(strid_t str);
```

La funzione legge un carattere dallo stream specificato (non esiste il concetto di *stream di input corrente*). `str` non può essere `NULL` né tanto meno uno stream di sola scrittura.

Il risultato sarà compreso fra 0 e 255. Come per tutte le funzioni di testo di base, Glk assume la codifica Latin-1. Vedere la Capitolo 2. Se lo stream è alla fine, il risultato sarà -1.[5]

Come caso speciale, se lo stream contiene caratteri Unicode (per esempio, se è stato creato con `glk_stream_open_file_uni()` nel caso di stream su disco o anche da `glk_stream_open_memory_uni()` per uno stream in memoria) i caratteri oltre 255 saranno ritornati come `0x3F` ("?").

```
glui32 glk_get_buffer_stream(strid_t str,
    char *buf, glui32 len);
```

Legge `len` caratteri dallo stream specificato, a meno che non si raggiunga prima la fine. Non viene messo nessun terminatore nel buffer. Ritorna il numero di caratteri che sono stati effettivamente letti.

```
glui32 glk_get_line_stream(strid_t str,
    char *buf, glui32 len);
```

Legge caratteri dallo stream specificato fino a che o `len`-1 caratteri sono stati letti o è stato letto un newline. Infine mette un terminatore `NUL` ("\0") in coda. Ritorna il numero di caratteri letti, incluso il newline (se presente) ma escludendo il terminatore `NUL`.

È generalmente più efficiente leggere blocchi di caratteri utilizzando le funzioni `glk_get_buffer_stream()` e `glk_get_line_stream()`, piuttosto che chiamare ripetutamente la primitiva `glk_get_char_stream()`.

```
glsi32 glk_get_char_stream_uni(strid_t str);
```

Unicode
Glk 0.7

Legge un carattere dallo stream. Il risultato sarà un codepoint Unicode compreso fra 0 e `0x7FFFFFFF`. Se lo stream è concluso, ritorna -1.

[5] Attenzione: i caratteri 128–255 *non* sono ritornati come numeri negativi.

Unicode Glk 0.7

```
glui32 glk_get_buffer_stream_uni(strid_t str,
    glui32 *buf, glui32 len);
```

Legge len Unicode caratteri dallo stream specificato, a meno che non venga prima raggiunta la fine dello stream. Non viene messo terminatore nel buffer. Ritorna il numero di caratteri Unicode effettivamente letti.

Unicode Glk 0.7

```
glui32 glk_get_line_stream_uni(strid_t str,
    glui32 *buf, glui32 len);
```

Legge caratteri Unicode dallo stream specificato fino a che o len−1 caratteri sono stati letti o è stato letto un newline. Infine mette un terminatore NUL (“\0”) in coda. Ritorna il numero di caratteri letti, incluso il newline (se presente) ma escludendo il terminatore NUL.

5.3. Chiudere gli stream

```
typedef struct stream_result_struct {
    glui32 readcount;
    glui32 writecount;
} stream_result_t;

void glk_stream_close(strid_t str,
    stream_result_t *result);
```

Chiude lo stream str. L’argomento result punta ad una struttura che viene valorizzata con i contatori finali dello stream. Se questi non interessano si può passare NULL come result.

Se str è lo stream corrente, lo stream corrente viene impostato a NULL.

Non è possibile chiudere direttamente lo stream di una finestra; è sufficiente chiamare glk_window_close() per chiuderla (e lo stream viene chiuso con essa). Vedere la sezione 3.2.

54. Posizione in uno stream

È possibile spostare la posizione del punto di lettura/scrittura in uno stream.

```
glui32 glk_stream_get_position(strid_t str);
```

Ritorna la posizione corrente sullo stream. Nel caso di stream in memoria o stream binari su file, è esattamente il numero di carattere letti o scritti dall'inizio dello stream (a meno che non sia già stata utilizzata precedentemente glk_stream_set_position()).

Per gli stream di testo su file, la definizione è ambigua dato che (per esempio) scrivere un byte su un file di testo può avanzare la posizione di più di un carattere nella codifica in uso. L'unica cosa certa è che la posizione aumenta mano a mano che si legge o scrive sul file.

C'è anche un'altra complicazione: per gli stream Latin-1 in memoria e su file, un carattere è un byte. Per gli stream Unicode (creati con le funzioni glk_stream_open_file_uni() e glk_stream_open_memory_uni()) un carattere è una word di 32 bit. Di conseguenza in un file binario Unicode, le posizioni sono multiple di quattro byte.[6]

```
void glk_stream_set_position(strid_t str,
    glsi32 pos, glui32 seekmode);
```

Sposta la posizione del punto di lettura/scrittura. La posizione è controllata da pos e il significato di pos è controllato da seekmode:

seekmode_Start pos caratteri a partire dall'inizio del file.

seekmode_Current pos caratteri dopo la posizione corrente (andando indietro se pos è negativo).

seekmode_End pos caratteri dopo la fine del file (pos in questo caso deve essere sempre zero o negativo, per rimanere all'interno del file).

❖ ❖ ❖

Per i programmatori C: le funzioni ANSI fseek() e ftell() funzionano *esattamente* allo stesso modo.

◆ ◆ ◆

È illegale specificare una posizione prima dell'inizio o dopo la fine del file.

Nei file binari la posizione è esatta: corrisponde al numero di caratteri letti o scritti. Nei file di testo, la corrispondenza può variare a causa della

[6] D'altra parte perché usare file binari Unicode? Per cosa potrebbero essere utili?

conversione dei newline e altre approssimazioni dovute al set di caratteri (vedere il Capitolo 5).

`glk_stream_set_position()` (così come `glk_stream_get_position()`) misura la posizione del puntatore nella codifica nativa della piattaforma, *dopo* tutte queste possibili conversioni.

In uno stream di testo, è meglio utilizzare il posizionamento solo verso l'inizio o la fine del file, o verso una posizione precedentemente determinata con `glk_stream_get_position()`.

Come sempre, negli stream Latin-1 i caratteri sono byte. Negli stream Unicode i caratteri sono word di 32 bit, di quattro byte ognuna.

5.5. Stili

È possibile mandare su uno stream comandi per cambiare lo stile del testo. Dopo un cambio di stile, il testo successivamente inviato sullo stream sarà visualizzato o scritto nel nuovo stile, a seconda dello stream in questione. Per lo stream di una finestra il testo apparirà in quello stile. Per uno stream in memoria i cambi di stile non hanno effetto alcuno. Per uno stream su file, se la macchina supporta file di testo con stile, gli stili potrebbero essere scritti sul file; ma molto più probabilmente le variazioni di stile non avranno effetto.

Gli stili sono mutuamente esclusivi. Un carattere ha esattamente uno stile e non un sottoinsieme degli stili possibili.[7]

Gli stili sono intesi per distinguere significato e uso, non per formattare. *Non* esiste alcuna definizione standard di come ciascuno stile debba apparire. L'apparenza è delegata alla libreria Glk, che utilizzerà un formato adatto per la piattaforma e le preferenze del giocatore.

Esistono attualmente undici stili definiti (Tabella 5.1). Altri potrebbero essere definiti in futuro.

❖ ❖ ❖

La scelta degli stili sembra alquanto arbitraria. Su Usenet la discussione al riguardo non è mai stata conclusa, a quanto pare.

Personalmente avrei definito gli stili *minimi* e un selettore gestalt per il numero di stili utente disponibili.

◆ ◆ ◆

Gli stili si possono distinguere su schermo per font, dimensione, colore, indentazione, giustificazione o altri attributi. Alcuni di questi attributi (in particolar modo giustificazione e indentazione) si applicano solo a interi

[7] Ogni stream e ogni finestra ha la sua propria nozione di *stile corrente*. Mandare un comando di cambio stile ad una finestra non ne influenza altri. A parte lo stream di eco di una finestra; vedere la sezione 3.6.

Tabella 5.1. Stili di testo disponibili in Glk 0.7

Stile	**Descrizione e utilizzo**
`Normal`	Lo stile del corpo del testo. Una nuova finestra o stream inizia sempre con `style_Normal` come stile corrente.
`Emphasized`	Testo messo in evidenza.
`Preformatted`	Testo con una particolare disposizione dei caratteri. Questo stile, a differenza degli altri, *ha* un formato standard; sarà sempre visualizzato con un font a spaziatura fissa, per motivi pratici. È spesso utile visualizzare mappe o diagrammi utilizzando la grafica a caratteri e questo è lo stile da utilizzare in queste circostanze.
`Header`	Testo che introduce una sezione. Adatto per il titolo di un gioco, oppure una sezione sostanziale, come un capitolo.
`Subheader`	Testo che introduce una sottosezione all'interno di una sezione. In una tipica Avventura Testuale, questo stile è adatto per il nome della stanza, quando il giocatore si guarda intorno.
`Alert`	Testo che avverte di una condizione pericolosa, o a cui il giocatore deve fare attenzione.
`Note`	Testo che notifica una condizione interessante. Adatto per notificare il giocatore di una variazione nel punteggio.
`BlockQuote`	Testo che forma una citazione o testo comunque estratto dal contesto.
`Input`	Testo digitato dal giocatore. Non è generalmente necessario utilizzare questo stile; la libreria lo utilizza per il testo digitato dal giocatore durante l'input a linea. Un caso in cui *è* appropriato l'utilizzo di questo stile è quando si simula l'input del giocatore, leggendo comandi da un file di testo.
`User1`	Questo stile non ha un significato predefinito. È il programmatore a definirne lo scopo e ad utilizzarlo come meglio crede.
`User2`	Un altro stile utilizzabile liberamente.

paragrafi. Se possibile è meglio applicare uno stile ad un intero paragrafo, chiamando glk_set_style() immediatamente dopo il newline e prima dell'inizio del testo.[8]

```
void glk_set_style(glui32 val);
```

Cambia lo stile dello stream corrente. val deve essere uno dei valori sopra elencati. In ogni caso, ogni valore è legale; se l'interprete non riconosce lo stile, lo deve trattare come style_Normal.[9]

```
void glk_set_style_stream(strid_t str, glui32 val);
```

Come sopra, ma relativo allo stream str, non allo stream corrente.

5.5.1. Suggerire il formato degli stili

Non ci sono garanzie di come possano apparire gli stili, ma è comunque possibile dare dei suggerimenti.

```
void glk_stylehint_set(glui32 wintype,
    glui32 styl, glui32 hint, glsi32 val);
void glk_stylehint_clear(glui32 wintype,
    glui32 styl, glui32 hint);
```

Queste funzioni mettono e tolgono suggerimenti (hint) al riguardo del formato di uno stile per un particolare tipo di finestra. È possibile anche usare la costante wintype_AllTypes come wintype per indicare che si vogliono impostare (o togliere) suggerimenti per tutti i tipi di finestra.[10]

Inizialmente non esistono suggerimenti al riguardo di qualsiasi tipo di finestra o di stile. Attenzione, poiché non avere un suggerimento non è come impostarlo a 0.

Queste funzioni *non* influenzano le finestre *preesistenti*. Solo le finestre create in seguito potranno tener conto dei suggerimenti assegnati. Se si vogliono impostare suggerimenti validi per tutte le finestre dell'applicazione bisogna utilizzare glk_stylehint_set() prima di creare la prima finestra.

[8] Per esempio, style_Header potrebbe essere testo centrato. Mostrando "*Welcome to Victim (a short interactive mystery)*" con solo la parola "*Victim*" in stile header, l'attributo di centratura verrebbe perso. Allo stesso modo, una citazione viene tipicamente indentata su entrambi i margini, ma l'indentazione è significativa solo quando applicata ad un intera linea o paragrafo. Per questo motivo, le citazioni devono comprendere paragrafi interi. Per contro, style_Emphasized non ha bisogno di essere utilizzato su interi paragrafi. È tipicamente usato per singole parole nel corpo del testo, per cui ci si può aspettare che si comporti in modo corretto; tipicamente sarà in italico o sottolineato, non centrato o indentato.

[9] In questo modo si possono definire nuovi stili in futuro senza toccare le vecchie librerie Glk.

[10] Per contro, non c'è una costante per suggerire su tutti gli stili di un singolo tipo di finestra.

Se si vogliono utilizzare suggerimenti diversi su finestre diversi, bisogna modificarli adeguatamente prima di creare ogni singola finestra.[11]

❖ ❖ ❖

L'effetto della *limitazione* degli stili fissati al momento della creazione è particolarmente visibile nel remake per glulx di *Photopia*. Ad ogni cambio scena le finestre vengono distrutte e ricreate; a parte questo il layout è stato fatto in un modo assurdo, che è visualizzabile correttamente solo con glulxe sotto Windows.

Decisamente un metodo da *non* imitare.

◆ ◆ ◆

D'altra parte i suggerimenti sono solo tali. La libreria può ignorarli o permettere al giocatore di decidere se accettarli o meno. Inoltre non è mai obbligatorio dare suggerimenti. Non è inoltre necessario suggerire che `style_Preformatted` sia a spaziatura fissa, o che `style_Emphasized` sia in grassetto o italico; esistono già dei default appropriati. I suggerimenti sono per le situazioni in cui si voglia *cambiare* il formato di uno stile rispetto a quello che possa essere normalmente. Il caso più appropriato è per configurare gli stili `style_User1` e `style_User2`.

Al momento sono definiti nove suggerimenti di stile. Altri potranno essere definiti in futuro.

`stylehint_Indentation` Quanto indentare le linee di testo in un dato stile. Può essere un numero negativo per spostare il testo verso l'esterno (a sinistra) piuttosto che verso l'interno (a destra). L'unità di misura non è specificata esattamente; indicativamente $+1$ è la più piccola indentazione possibile chiaramente visibile al giocatore.

`stylehint_ParaIndentation` Quanto indentare la prima linea di ogni paragrafo. Il valore dell'indentazione è in aggiunto, viene quindi sommato a quanto specificato da `stylehint_Indentation`. Anche in questo caso il valore può essere negativo ed è misurato nelle stesse unità utilizzate con `stylehint_Indentation`.

`stylehint_Justification` Il valore assegnato a questo suggerimento deve essere una delle costanti in Tabella 5.2.

`stylehint_Size` Di quanto aumentare o diminuire la dimensione del font. È un valore relativo; 0 indica di utilizzare la dimensione di font standard, numeri positivi richiedono caratteri più grandi e numeri negativi caratteri più piccoli. Anche in questo caso $+1$ è il più piccolo aumento di dimensione facilmente visibile.[12]

[11] In questo modo la vita per la libreria viene semplificata. Può stabilire tutto su una finestra nel momento in cui viene creata senza doversi preoccupare di eventuali successive modifiche.

[12] Questa quantità potrebbe anche non essere costante. $+1$ potrebbe incrementare un font di 8 punti a 9 punti, ma 16 punti potrebbero diventare 18.

Tabella 5.2. Costanti per l'allineamento del testo

`stylehint_just_LeftFlush` (allineato a sinistra);
`stylehint_just_LeftRight` (giustificato);
`stylehint_just_Centered` (centrato);
`stylehint_just_RightFlush` (allineato a destra).

`stylehint_Weight` Il valore deve essere 1 per un font pesante (grassetto), 0 per un carattere normale e -1 per un carattere leggero.

`stylehint_Oblique` Il valore deve essere 1 per un font obliquo (o italico) o 0 per un carattere normale (verticale).

`stylehint_Proportional` Il valore deve essere 1 per un font a spaziatura variabile o 0 per un font a spaziatura fissa.

`stylehint_TextColor` Il colore del testo, codificato su 32 bit: gli 8 bit più significativi devono essere zero, i successivi 8 sono il valore del rosso, seguiti da 8 bit per il verde e infine gli 8 meno significativi per il blu. I valori vanno da 0 a 255.[13]

`stylehint_BackColor` Il colore di sfondo del testo, codificato nello stesso modo di `stylehint_TextColor`.

`stylehint_ReverseColor` Il valore deve essere 0 per la visualizzazione normale (TextColor su BackColor) o uno per la visualizzazione inversa (BackColor su TextColor).[14]

❖ ❖ ❖

Il sistema dei suggerimenti (hint) per gli stili è, secondo me, dall'incompleto al completamente inaffidabile. Per cominciare le unità di misura sono completamente arbitrarie, alcuni parametri sono misteriosamente assenti (si può settare il margine sinistro, ma non quello destro, per esempio) ed è meglio non parlare dello stato di supporto dei colori e del reverse (paradossalmente, se un interprete supporta correttamente entrambi i suggerimenti e l'applicazione li specifica entrambi, il reverse cessa di avere effetto!).

Per questo motivo il sistema dei suggerimenti è *altamente* dipendente dall'implementazione Glk. Alcune implementazioni (fra cui Glk per Windows e garglk) sono passate all'utilizzo di *fogli di stile* esterni per sopperire a tutte queste limitazioni.

[13] Quindi `0x00000000` è nero, `0x00FFFFFF` è bianco e `0x00FF0000` è rosso brillante.

[14] Alcune librerie potrebbero gestire questo suggerimento ma non quelli relativi ai colori. Altre librerie potrebbero comportarsi in modo opposto; altre ancora supportarli entrambi o nessuno dei tre.

Il già citato remake di *Photopia* ha allegato in effetti il file `.ini` da utilizzare in congiunzione con glulxe...

◆ ◆ ◆

Anche in questo caso, passando un suggerimento ad una funzione Glk, ogni valore è lecito. Se l'interprete non lo riconosce lo deve ignorare.[15]

5.5.2. *Verificare il formato degli stili*

Se prima di creare una finestra è possibile suggerire il formato di ogni stile, dopo averla creata è possibile verificare come effettivamente appare. Queste funzioni non agiscono sui suggerimenti, bensì su come ogni stile appare al giocatore.

Anche se non è possibile modificare gli stili di una finestra dopo la creazione, la libreria potrebbe farlo. Una piattaforma potrebbe supportare un sistema di preferenze dinamiche, e cambiare la formattazione del testo durante l'esecuzione.[16]

```
glui32 glk_style_distinguish(winid_t win,
    glui32 styl1, glui32 styl2);
```

Confronta due stili e ritorna `TRUE` (1) se sono visibilmente distinguibili in una data finestra. In caso contrario ritorna `FALSE` (0). L'esatto significato è determinato dalla libreria.

```
glui32 glk_style_measure(winid_t win,
    glui32 styl, glui32 hint, glui32 *result);
```

Prova a determinare un attributo di uno stile in una data finestra. La libreria potrebbe non essere in grado di determinare l'attributo; in questo caso ritorna `FALSE` (0). Se ne è in grado ritorna `TRUE` (1) e mette il valore nella locazione puntata da `result`.[17]

Il significato del valore dipende dal tipo di suggerimento richiesto:

`stylehint_Indentation`, `stylehint_ParaIndentation` La dimensione del margine sinistro per il paragrafo (e di quanto indentare la prima linea). La metrica dipendi dalla piattaforma.[18]

[15] Come sempre, in questo modo si possono definire senza problemi in seguito altri suggerimenti di stile.

[16] Le modifiche che influenzano la dimensione delle finestre (come la dimensione del font) saranno segnalate da un evento `evtype_Arrange`. Ma altre variazioni meno appariscenti (come il colore del testo) non sono segnalate. Anche se si controlla il formato degli stili all'inizio del programma è bene tenere a mente che in seguito il giocatore potrebbe modificarli.

[17] Come al solito `result` può essere `NULL`, ma non ha comunque senso farlo.

[18] Molto probabilmente saranno caratteri o pixel.

stylehint_Justification Una delle costanti in tabella Tabella 5.2.

stylehint_Size La dimensione del font. Ancora, l'unità di misura dipende dalla piattaforma.[19]

stylehint_Weight 1 per un carattere pesante (grassetto), 0 per un carattere normale e −1 per un carattere leggero.

stylehint_Oblique 1 per un font obliquo (o italico) o 0 per un carattere normale (verticale).

stylehint_Proportional 1 per un font a spaziatura variabile o 0 per un font a spaziatura fissa.

stylehint_TextColor, stylehint_BackColor Un valore intero, codificato come descritto in precedenza (sottosezione 5.5.1).

stylehint_ReverseColor 0 per la visualizzazione normale, 1 per quella invertita.

5.5.3. I tipi di stream

Stream di finestra

Ogni finestra ha uno stream di output associato. Questo stream viene aperto automaticamente (con filemode_Write) quando viene creata la finestra. È accessibile con glk_window_get_stream().

Uno stream di finestra non può essere chiuso con glk_stream_close(). Viene chiuso automaticamente con la finestra con glk_window_close().

Su uno stream di finestra possono essere scritti solo caratteri stampabili (incluso il newline). Vedere la Capitolo 2.

Stream in memoria

È possibile aprire uno stream che legge o scrive in un'area di memoria.

```
strid_t glk_stream_open_memory(char *buf,
    glui32 buflen, glui32 fmode, glui32 rock);
```

fmode deve essere filemode_Read (sola lettura), filemode_Write (sola scrittura) o filemode_ReadWrite (lettura/scrittura). buf punta all'area di memoria a cui accederà lo stream. buflen è la lunghezza del buffer.

Durante la scrittura, se più di buflen caratteri sono mandati sullo stream, tutti quelli oltre la lunghezza sono scartati, per non uscire dal buffer (il

[19] Pixel, punti o anche 1 se la libreria usa solo font di una dimensione

totale dei caratteri dello stream viene comunque conteggiato correttamente. Si tratta del numero di caratteri scritti, non il numero di caratteri che sono entrati nel buffer).

Se `buf` è `NULL` o, ovviamente, se `buflen` è zero, *tutto* ciò che viene scritto viene scartato. Può essere utile se interessa solo il numero di caratteri totale.

Durante la lettura, se vengono letti più di `buflen` caratteri, lo stream segnala end of file ritornando -1. Se `buf` è `NULL` lo stream sarà sempre in condizione di end of file.

I dati sono scritti nel buffer esattamente come passati alle funzioni di output; le funzioni di input leggono i dati esattamente come sono in memoria. Non viene effettuata alcuna conversione specifica su di essi.[20]

I caratteri Unicode (con valori maggiori a 255) non possono essere scritti nel buffer. Nel caso vengono memorizzati come caratteri `0x3F` ("?").

Il contenuto del buffer è indefinito fino alla chiusura dello stream. La libreria è libera di accedervi nel modo che preferisce, un carattere alla volta oppure a blocchi. Anche durante la lettura è illegale modificare il contenuto del buffer mentre lo stream è aperto.

Unicode
Glk 0.7

```
strid_t glk_stream_open_memory_uni(glui32 *buf,
    glui32 buflen, glui32 fmode, glui32 rock);
```

Funziona nello stesso modo di `glk_stream_open_memory()` ma il buffer è un array di word di 32 bit, non di byte. In questo modo è possibile leggere o scrivere qualsiasi carattere Unicode. Il parametro `buflen` è il numero di word, non di byte.[21]

[20] Uno stream in memoria è quindi in sostanza sempre in modo binario.

[21] Se il buffer contiene il valore `0xFFFFFFFF` ed è aperto in lettura, non è possibile distinguere la situazione di end of file. Ma, d'altra parte, `0xFFFFFFFF` non è un carattere Unicode valido.

5.54. Stream su file

È possibile aprire uno stream che legge e/o scrive file su disco.

```
strid_t glk_stream_open_file(frefid_t fileref,
    glui32 fmode, glui32 rock);
```

fileref indica il file da aprire. fmode può essere filemode_Read (sola lettura), filemode_Write (sola scrittura), filemode_WriteAppend (scrittura in coda) o filemode_ReadWrite (lettura/scrittura).

Se fmode è filemode_Read il file deve essere già esistente; nelle altre modalità viene creato un file vuoto in caso contrario.

Se fmode è filemode_Write e il file esiste già viene troncato a lunghezza zero (e reso quindi vuoto). Se fmode è filemode_WriteAppend il punto di scrittura viene posizionato a fine file.

Il file può essere utilizzato in modalità testo o binaria; la modalità viene determinata dalla fileref. Altri simili attributi dipendenti dalla piattaforma sono determinati dalla fileref. Vedere il Capitolo 6.

Quando si scrive in modalità binaria i caratteri Unicode maggiori di 255 non possono essere scritti nel file. Nel caso, vengono memorizzati come caratteri 0x3F ("?"). In modalità testo i caratteri Unicode possono essere scritti esattamente, approssimati o abbreviati a seconda del supporto per i file di testo della piattaforma.

Unicode
Glk 0.7

```
strid_t glk_stream_open_file_uni(frefid_t fileref,
    glui32 fmode, glui32 rock);
```

Il funzionamento è come quello di glk_stream_open_file() ma in questo caso, in modalità binaria, i caratteri sono scritti e letti come valori a 32 bit (in formato big-endian). In questo modo è possibile scrivere e leggere qualunque carattere Unicode.

In modalità testo, il file viene scritto e letto in modo dipendente dalla piattaforma che potrebbe (oppure no) gestire tutti i caratteri Unicode. Un file di testo creato con glk_stream_open_file_uni() può anche avere lo stesso formato di un file di testo creato con glk_stream_open_file(); ma potrebbe utilizzare un formato specifico per Unicode.

5.6. Altre funzioni per gli stream

```
strid_t glk_stream_iterate(strid_t str,
    glui32 *rockptr);
```

Itera su tutti gli stream esistenti. Vedere la sottosezione 1.6.2.

```
glui32 glk_stream_get_rock(strid_t str);
```

Recupera il valore messo sotto al sasso di uno stream. Vedere la sottosezione 1.6.1.

6 Riferimenti a file

I file su disco vengono identificati per mezzo di riferimenti a file. Ogni fileref è un puntatore ad una struttura C opaca; vedere la sezione 1.6.

❖ ❖ ❖

L'utilizzo delle fileref è ancora più raro per l'autore di avventure testuali, ancora più degli stream.

Possono essere utili nel caso in cui si voglia mettere mano alle routine di sistema (salvataggio, ripristino e trascrizione) oppure per utilizzare file esterni dall'interno del gioco.

◆ ◆ ◆

Un riferimento a file contiene informazioni specifiche sul nome e sulla posizione di un file ed eventualmente il suo tipo, se la piattaforma possiede la nozione di tipo di file. Il riferimento include anche un flag che indica se il file è binario oppure di testo.[1]

Una fileref non necessariamente si riferisce ad un file esistente. È possibile creare un fileref per un file inesistente per poi aprirlo in scrittura al fine di creare un nuovo file.

Quando si crea un fileref bisogna sempre indicare per quale motivo verrà utilizzato. Questo motivo è una combinazione fra costanti per indicare il tipo di file e il suo modo (testo o binario). Questi valori sono impiegati durante la creazione e anche per filtrare la lista dei file durante la selezione di un file da caricare. Le costanti da utilizzare sono le seguenti:

`fileusage_SavedGame` Un file che contiene lo stato di gioco.

[1] In questo si differenzia dalla libreria di I/O standard del C, in cui si specifica il modo testo o binario solo durante l'apertura.

`fileusage_Transcript` Un file che raccoglie uno stream di testo dal gioco (spesso lo stream di eco da una finestra).

`fileusage_InputRecord` Un file per registrare l'input del giocatore.

`fileusage_Data` Un file di altro genere (preferenze, statistiche, dati arbitrari).

`fileusage_BinaryMode` Il contenuto del file è memorizzato esattamente come scritto e riletto nello stesso identico modo. Il file risultante potrebbe non essere mostrato correttamente sui visualizzatori di testo della piattaforme.

`fileusage_TextMode` Il contenuto del file viene convertito durante la scrittura in un formato di testo adatto alla piattaforma in uso. I newline potrebbero essere convertiti in linefeed con ritorno carrello; i caratteri Latin-1 convertiti nel set di caratteri locale. Quando si legge un file in modalità testo, le stesse conversioni vengono eseguite al contrario.[2]

In generale, si usa la modalità testo se al giocatore è consentito di rileggere il file con un editor di testo comune; il modo binario è invece più adatto quando i dati devono semplicemente essere riletti o, in generale, se devono essere preservati esattamente.

Il modo testo è appropriato per file di tipo `fileusage_Transcript`; nel caso di file aperto come `fileusage_SavedGame` (e, probabilmente anche per `fileusage_InputRecord`) la modalità binaria è sicuramente consigliata.

L'utilizzo del formato testo o binario in congiunzione con `fileusage_Data` dipende dall'uso che se ne intende fare.

6.1. I tipi di riferimento a file

Ci sono quattro funzioni distinte per creare una fileref, a seconda del modo in cui si intende specificarla. Come sempre, è possibile che la creazione della fileref fallisca e ritorni `NULL`.

[2] La conversione dei newline avviene in generale senza problemi; altre forme potrebbero essere soggette a limitazioni. Per esempio, scrivendo in modo testo un file e poi rileggendolo (sempre in modo testo) potrebbero venire alterati o persi i caratteri alti (nell'intervallo 128–255).

```
frefid_t glk_fileref_create_temp(glui32 usage,
    glui32 rock);
```

Crea un riferimento ad un file temporaneo. Sarà sempre un file nuovo (non esistente). Una volta creato, sarà in una posizione appropriata, dove non darà fastidio al giocatore.[3]

Un file temporaneo non deve essere utilizzato per memorizzare dati a lungo termine. Potrebbe essere cancellato automaticamente quando il programma esce, o in seguito, per esempio al riavvio della macchina. Non è necessario preoccuparsi della sua cancellazione: si presume sia gestita automaticamente.

```
frefid_t glk_fileref_create_by_prompt(glui32 usage,
    glui32 fmode, glui32 rock);
```

Crea un riferimento ad un file chiedendo al giocatore di specificarlo. La libreria potrebbe semplicemente chiedere al giocatore di digitare il nome, oppure utilizzare uno strumento adatto sulla piattaforma in uso (il formato della richiesta viene determinato dall'argomento `usage`).[4]

`fmode` deve essere uno dei seguenti valori:

`filemode_Read` Il file deve essere esistente; al giocatore sarà richiesto di scegliere fra file esistenti corrispondenti all'uso specificato.

`filemode_Write` Il file non deve esistere; se il giocatore ne specifica uno preesistente verrà avvisato della sostituzione.

`filemode_ReadWrite` Il file potrebbe già esistere; se il giocatore ne specifica uno preesistente verrà avvisato della sua possibile modifica.

`filemode_WriteAppend` Idem come `filemode_ReadWrite`.

In generale l'argomento `fmode` corrisponde all'`fmode` che verrà utilizzato per aprire il file.

[3] E questo è il motivo per cui non viene specificato alcun nome; il giocatore non ha bisogno di conoscerlo.

[4] È possibile che il prompt o lo strumento di scelta abbiano un'opzione per annullare la selezione. Se il giocatore decide di annullare, `glk_fileref_create_by_prompt()` ritornerà `NULL`. Un motivo in più per controllare il valore di ritorno prima di utilizzarlo.

```
frefid_t glk_fileref_create_by_name(glui32 usage,
    char *name, glui32 rock);
```

Crea un riferimento ad un file con un nome specifico. Il file sarà in una locazione relativa al programma e visibile al giocatore.[5]

Poiché i nomi dei file sono estremamente specifici alla piattaforma, è necessario usare glk_fileref_create_by_name() con attenzione. Sebbene sia legale passare qualunque stringa nell'argomento name, la libreria è libera di troncare, trasformare o modificare in generale questa stringa per renderla un nome di file adatto alla piattaforma in uso.[6]

Il metodo più sicuro è di passare una stringa di non più di 8 caratteri, consistente solo di lettere maiuscole e numeri, e una lettera come primo carattere. In questo modo si può essere ragionevolmente sicuri che il file risultante abbia il nome specificato (in qualche modo).

```
frefid_t glk_fileref_create_from_fileref(glui32 usage,
    frefid_t fref, glui32 rock);
```

Copia un riferimento a file preesistente, modificandone l'uso (la fileref originale non è modificata).

L'utilizzo di questa funzione potrebbe non essere così immediato. Cambiando il tipo della fileref, il nuovo riferimento potrebbe puntare ad un file differente dall'originale.[7] In questa situazione, se si aprono entrambi i file in scrittura, i risultati sono imprevedibili. È sicuro cambiare tipo ad una fileref solo se si riferisce ad un file non esistente.

Per contro, se si cambia la modalità di una fileref (modo testo o binario) ma si lascia invariato il tipo, la nuova fileref sicuramente punterà allo stesso file di quella originaria.

Ovviamente, scrivendo un file in modo testo e rileggendolo in modo binario (o viceversa) i risultati dipendono dalla piattaforma in uso.

[5] Il che, in genere, significa *nella stessa cartella del programma*.

[6] Per esempio, creando due fileref dai domi "File" e "FILE", esiste la possibilità che si riferiscano allo *stesso* file; la piattaforma potrebbe non distinguere le maiuscole nei nomi di file. Un altro esempio: in una piattaforma in cui il tipo di file è specificato da un'estensione del nome, la libreria aggiungerà un suffisso adeguato; un'estensione già presente nella stringa potrebbe essere sostituita. E in ogni caso, per esempio, il punto non è un carattere valido nei nomi di file sotto Acorn...

[7] Specialmente se la piattaforma usa qualche forma di suffisso per indicare il tipo di file.

6.2. Altre funzioni per i riferimenti a file

```
void glk_fileref_destroy(frefid_t fref);
```

Elimina un riferimento creato in precedenza. Il file su disco corrispondente *non* viene toccato; vengono solo recuperate le risorse allocate per il riferimento.

È perfettamente legale eliminare una fileref dopo averla utilizzata per aprire un file (e con il file ancora aperto). La fileref serve solo per l'operazione di apertura, in seguito non viene più utilizzata.

```
frefid_t glk_fileref_iterate(frefid_t fref,
    glui32 *rockptr);
```

Itera su tutte le fileref esistenti. Vedere la sottosezione 1.6.2.

```
glui32 glk_fileref_get_rock(frefid_t fref);
```

Recupera il valore sotto al sasso della fileref. Vedere la sottosezione 1.6.1.

```
void glk_fileref_delete_file(frefid_t fref);
```

Cancella il file a cui punta la fileref. La fileref non viene però eliminata.

```
glui32 glk_fileref_does_file_exist(frefid_t fref);
```

Ritorna `TRUE` (1) se la fileref punta ad un file esistente, e `FALSE` (0) in caso contrario.

7
Grafica

In accordo con i tempi moderni, Glk fornisce un minimo di supporto grafico. *Non* contiene però un toolkit grafico completo. Di questi già ce ne sono. Glk tenta il solito compromesso fra potenza, portabilità e facilità di implementazione: fornisce comandi per disporre immagini prefissate sullo schermo e inframezzate al testo.

Il supporto grafico è facoltativo in Glk; non tutte le librerie la supportano. Non dovrebbe essere una sorpresa.

❖ ❖ ❖

Ovviamente garglk-mod ha le routine grafiche implementate in maniera pressoché completa. L'unica funzionalità mancante sono le immagini a margine multiple (fra l'altro non è troppo chiaro quale sia il loro *corretto* funzionamento).

◆ ◆ ◆

7.1. Risorse grafiche

La maggior parte dei comandi grafici in Glk hanno a che fare con risorse di immagini. Il programma non deve curarsi di come queste immagini siano memorizzate. Tutto è una risorsa, e una risorsa è identificata da un numero intero. È possibile, per esempio, chiamare una funzione per visualizzare l'immagine numero 17. Il formato, il caricamento e la visualizzazione di quell'immagine è completamente a carico della libreria Glk per la piattaforma in questione.

Ovviamente è anche desiderabile avere un mezzo indipendente dalla piattaforma per immagazzinare immagini e sonoro. Blorb è il formato ufficiale di

Glk per contenere risorse. Una libreria Glk non è obbligata a gestire Blorb, ma è più probabile che lo supporti rispetto a qualunque altro formato.[1]

Al momento, è possibile inserire immagini solo nelle finestre grafiche e nei buffer di testo. Anzi, una libreria è libera di non implementare *entrambe* le funzionalità. Se si intende far uso di immagini è meglio utilizzare il selettore gestalt_DrawImage. Vedere la sezione 7.4.

```
glui32 glk_image_get_info(glui32 image,
    glui32 *width, glui32 *height);
```

Ottiene informazioni sulla risorsa immagine con l'identificatore specificato. Ritorna TRUE (1) se esiste una tale immagine, FALSE (0) in caso contrario. È anche possibile passare puntatori per le variabili width e height; se l'immagine esiste verranno valorizzate con la dimensione in pixel della stessa (è possibile utilizzare NULL se non interessano le informazioni sulla dimensione).[2]

```
glui32 glk_image_draw(winid_t win,
    glui32 image, glsi32 val1, glsi32 val2);
```

Disegna l'immagine specificata nella finestra win. La posizione dell'immagine è indicata da val1 e val2, il cui significato dipende dal tipo di finestra su cui si sta disegnando. Vedere la sezione 7.2 e la sezione 7.3.

Questa funziona ritorna un flag per indicare se l'immagine è stata disegnata oppure no.[3]

```
glui32 glk_image_draw_scaled(winid_t win,
    glui32 image, glsi32 val1, glsi32 val2,
    glui32 width, glui32 height);
```

Funziona in modo simile a glk_image_draw(), ma ridimensiona l'immagine alle dimensioni specificate (con glk_image_get_info() si possono ricavare le dimensioni originali).

[1] Anche se Glk non specifica l'esatto formato delle immagini, Blorb lo fa. Le immagini in un archivio Blorb devono essere file PNG o JPEG. Altri formati potranno essere aggiunti se in pratica si riveleranno desiderabili. Queste comunque sono questioni riguardanti le sole specifiche Blorb; per quanto riguarda la programmazione Glk, non cambia nulla.

[2] È bene utilizzare questa funzione per determinare la dimensione delle immagini mentre si disegna. Anche se si sa a priori quale *dovrebbe* essere la dimensione. Infatti le immagini potrebbero essere scalate passando su un'altra piattaforma, o per necessità di visualizzazione, o per preferenza del giocatore. In linea di principio le immagini saranno ridimensionate proporzionalmente, ma è comunque necessario utilizzare glk_image_get_info() per ottenere la dimensione definitiva.

[3] Un esito negativo può essere dovuto a varie ragioni. L'immagine potrebbe essere malformata o addirittura inesistente; potrebbe non esserci memoria sufficiente; o semplicemente quella finestra non supporta le immagini. Ovviamente questi non sono gli unici motivi.

Se `width` o `height` è zero, non viene disegnato nulla. Inoltre, essendo interi senza segno, non possono essere negativi. Se si passa un numero negativo, questo verrà interpretato come un grande intero positivo, con risultati quasi certamente catastrofici.

❖ ❖ ❖

Le routine di ridimensionamento di garglk, per quanto siano efficaci dal punto di vista del risultato, non sono esattamente dei mostri di velocità.

Fortunatamente tutte le immagini (comprese quelle sintetizzate scalando delle risorse) vengono tenute in una cache; allocando *sufficiente* memoria non si dovrebbero notare particolari rallentamenti.

È anche vero che il *PC medio* contemporaneo è tale da rendere a malapena percepibili i complessi calcoli sottostanti.

◆ ◆ ◆

7.2. Grafica nelle finestre grafiche

Una finestra grafica è un'area rettangolare di pixel, sulla quale è possibile disegnare delle immagini. Il contenuto è completamente sotto il controllo del programmatore. È possibile disegnare una quantità arbitraria di immagini, in qualunque posizione, anche sovrapposte. Se la finestra viene ridimensionate è però necessario ridisegnare tutto. Vedere la sottosezione 3.5.5.

Chiamando `glk_image_draw()` o `glk_image_draw_scaled()` su una finestra grafica, `val1` e `val2` sono interpretati come coordinate X e Y. L'immagine verrà disegnata con il suo angolo in alto a sinistra in questa posizione.

È lecito far si che una parte dell'immagine cada al di fuori della finestra; la parte in eccesso non viene disegnata. Visto che gli argomenti hanno un segno, è anche possibile disegnare un'immagine che cade oltre il bordo superiore e/o sinistro della finestra.

❖ ❖ ❖

In altre parole: è disponibile il clipping delle risorse grafiche. Ed è anche questo il motivo per cui le coordinate sono interi *con il segno*.

In questo modo è facilmente realizzabile, per esempio, una mappa scorrevole centrata sulla posizione corrente o altre cose del genere.

◆ ◆ ◆

Esistono anche altri comandi specifici per le finestre grafiche:

```
void glk_window_set_background_color(winid_t win,
    glui32 color);
```

Modifica il colore di sfondo della finestra. Quello che è attualmente visualizzato *non* viene alterato; il colore di sfondo influisce solo sulle successive pulizie e ridimensionamenti. Lo sfondo iniziale di ogni finestra è bianco.[4]

❖ ❖ ❖

Ripeto qui quel che è riportato a piè di pagina: il colore di sfondo si setta con questa funzione *solo per le finestre grafiche*. In teoria sarebbe illegale utilizzarla con buffer o griglie di testo. Ma tantissimi giochi lo fanno, dato che funziona con glulxe sotto Windows.

Il metodo corretto sarebbe di settare il colore di sfondo dello stile Normal, ma (dato che comunque non ci sono ambiguità) garglk-mod accetta *anche* questa modalità, per compatibilità.

◆ ◆ ◆

I colori sono codificati come valori a 32 bit: gli 8 bit più significativi devono essere zero, i successivi 8 bit indicano il valore rosso, i successivi 8 il valore verde e gli 8 bit meno significativi sono per il valore blu. I tre valori sono nell'intervallo 0--255.[5]

```
void glk_window_fill_rect(winid_t win,
    glui32 color, glsi32 left, glsi32 top,
    glui32 width, glui32 height);
```

Riempie il rettangolo specificato con un dato colore. È legittimo per parte del rettangolo di cadere al di fuori della finestra. Se `width` o `height` sono 0 non viene disegnato nulla.

```
void glk_window_erase_rect(winid_t win,
    glsi32 left, glsi32 top,
    glui32 width, glui32 height);
```

Riempie il rettangolo specificato con il colore di sfondo della finestra.

È possibile riempire l'intera finestra con il colore di sfondo con la funzione `glk_window_clear()`.

Le finestre grafiche non hanno un set di comandi molto esteso per disegnare oggetti, né tanto meno, permettono di disegnare testo. Queste funzionalità

[4] Questa funzione deve essere utilizzata solo con le finestre grafiche. Per impostare i colori di sfondo per le finestre di testo bisogna usare gli stili suggerendo i colori; vedere la sezione 5.5.

[5] Per cui 0 è nero, `0x00FFFFFF` è bianco e `0x00FF0000` è rosso brillante.

potrebbero essere presenti in una futura estensione per Glk. Per il momento sembra ragionevole limitarsi ad una singola primitiva, il disegno di un'immagine. Oltre alla possibilità di riempire un rettangolo con un colore; d'altra parte, la libreria deve già essere in grado di riempire lo sfondo della, quindi non è una grande estensione da implementare.

7.3. Grafica nei buffer di testo

Un buffer di testo è collegato ad uno stream di testo. È possibile disegnare immagini all'interno di questo testo. Se si è familiari con l'HTML, il modello è il medesimo. È possibile indicare l'allineamento di ogni immagine. Lo scorrimento, ridimensionamento e la riformattazione del testo all'interno delle finestre a buffer di testo sono completamente a carico della libreria.

Quando si usa `glk_image_draw()` o `glk_image_draw_scaled()` su un buffer di testo `val1` indica l'allineamento dell'immagine. L'argomento `val2` attualmente non viene utilizzato, ma deve essere comunque zero.

imagealign_InlineUp L'immagine appare nel punto corrente del testo, sporgente verso l'alto. In sostanza, il bordo inferiore dell'immagine è allineato con la linea base del testo.

imagealign_InlineDown L'immagine appare nel punto corrente del testo, sporgente verso il basso, con il margine superiore allineato alla cima della riga di testo.

imagealign_InlineCenter L'immagine appare nel punto corrente del testo, centrata sulla linea base del testo. Se l'immagine è più alta della linea di testo sporgerà in alto e in basso in ugual misura.

imagealign_MarginLeft L'immagine viene messa sul margine sinistro. Il testo a seguire sarà alla destra dell'immagine e scorrerà attorno ad essa, rimanendo indentato per le linee necessarie a superare l'immagine.

imagealign_MarginRight L'immagine viene messa sul margine destro, con il testo a seguire che scorre alla sua sinistra.

I due allineamenti a margine richiedono qualche cautela. Per consentire il posizionamento in maniera corretta, le immagini a margine *devono* essere inserite all'inizio di un paragrafo. Per essere chiari, è lecito chiamare `glk_image_draw()` (con uno di questi due allineamenti) in una finestra solo se si è appena dato newline sullo stream della finestra, o se la finestra è completamente vuota. Se si inserisce un'immagine allineata a margine in una linea che contiene già del testo, non apparirà nessuna immagine.

Le immagini in linea sono considerate come *testo* ai fini di questa regola.

È possibile avere immagini su entrambi i margini simultaneamente.

È anche legale avere più di una immagine sullo *stesso* margine. Dato che però è difficile predire come verrà disposto il testo in questo caso alcune librerie potrebbero non interpretare correttamente questa richiesta.

❖ ❖ ❖

E, in effetti, garglk-mod *non* gestisce due immagini sullo stesso margine simultaneamente!

Nella sottosezione 14.13.1 sono riportate delle immagini che chiariscono le modalità di allineamento disponibili.

◆ ◆ ◆

È anche possibile richiedere di *interrompere* il flusso di testo fino alla fine delle immagini attualmente sui margini. Dato che le linee di testo possono essere di dimensione diverso, non è possibile contare le linee. Si usa pertanto questa funzione:

```
void glk_window_flow_break(winid_t win);
```

Se il testo sta scorrendo intorno a delle immagini a margine, il risultato è simile ad una serie di newline fino alla fine delle immagini (se c'è più di una immagine a margine, fino alla fine di tutte le immagini). Se il punto corrente *non* è a fianco di una immagine, la chiamata non ha effetto.

Quando un buffer di testo viene ridimensionato, l'interruzione si comporta in maniera intelligente; può essere attivata o disattivata secondo necessità. Deve essere considerata come un marcatore invisibile nel testo che inserisce automaticamente newline per formattare correttamente il testo.

Nelle finestre che non sono buffer di testo, `glk_window_flow_break()` non ha effetto.

❖ ❖ ❖

O almeno *dovrebbe* comportarsi in maniera intelligente; il codice di garglk-mod al riguardo è relativamente giovane e anche se *sembra* comportarsi correttamente è una probabile fonte di funzionamento anomalo.

I bug report con esempi sono ovviamente sempre bene accetti!

Nel frattempo, se ancora non è chiaro il funzionamento, nella sezione 7.3 è riportata una immagine esplicativa.

◆ ◆ ◆

Tabella 7.1. Le funzioni del pacchetto grafico di Glk

```
glk_image_draw()
glk_image_draw_scaled()
glk_image_get_info()
glk_window_erase_rect()
glk_window_fill_rect()
glk_window_set_background_color()
glk_window_flow_break()
```

7.4. Verificare le funzionalità grafiche

Prima di chiamare le funzioni grafiche Glk, è utile l'uso dei seguenti selettori.

```
glui32 res;
res = glk_gestalt(gestalt_Graphics, 0);
```

Ritorna 1 se le funzioni relative alla grafica sono presenti. Le funzioni relative sono riportate in Tabella 7.1. Anche la possibilità di creare finestre grafiche è controllata da questo selettore.

Se il selettore ritorna 0, non bisogna chiamare queste funzioni. Potrebbero non avere effetto o anche causare un errore. Se si tenta di creare una finestra grafica si ottiene NULL.

Come per altre estensioni, esiste una complicazione per i programmi scritti in C. Una libreria che non supporta le funzioni grafiche potrebbe non averle neppure dichiarate.

Per evitare questo genere di problemi è possibile controllare la presenza del simbolo di preprocessore GLK_MODULE_IMAGE. Se è definito, lo sono anche le funzioni e le costanti descritte in questa sezione. In caso contrario, no.

```
glui32 res;
res = glk_gestalt(gestalt_DrawImage, windowtype);
```

Ritorna 1 se è possibile disegnare immagini nelle finestre di un dato tipo. Se il risultato è 0, glk_image_draw() fallirà e ritornerà FALSE (0). È importante testare separatamente wintype_Graphics e wintype_TextBuffer dato che le librerie possono implementarle indipendentemente.

❖ ❖ ❖

Se si, ehm, *tralascia* il problema della dimensione dei font le immagini in una griglia di testo sarebbero veramente utili, secondo me (non sarebbe comoda la grafica nella finestra di stato, per esempio?). Vedremo se si riuscirà a decidere qualcosa per le prossime revisioni.

◆ ◆ ◆

```
glui32 res;
res = glk_gestalt(gestalt_GraphicsTransparency, 0);
```

Ritorna 1 se le immagini con il canale alpha sono effettivamente disegnate con l'appropriato grado di trasparenza. Se il risultato è 0, il canale alpha è ignorato e le aree completamente trasparenti sono gestite in un modo dipendente dall'implementazione.[6]

❖ ❖ ❖

Il supporto di garglk per il canale alpha è completo. Viene gestito completamente l'alpha blending su 8 bit.

◆ ◆ ◆

[6] Il formato JPEG non è in grado di gestire la trasparenza (non può avere il canale alpha); il formato PNG si.

8

Audio

Come per la grafica, esiste anche il sonoro. Il suono però, ovviamente, non appare nelle finestre. Per riprodurre un suono con Glk, bisogna per prima cosa creare un canale audio per contenerlo. Si tratta di una classe indipendente di oggetti opachi; come sempre ci sono funzioni per creare, distruggere, iterare e prendere il valore sotto ai sassi dei canali, proprio come esistono per finestre, stream e fileref.

Un canale è in grado di riprodurre esattamente un suono alla volta. Se si vogliono più suoni in simultanea, ci vogliono più canali. Per contro, un singolo suono può essere riprodotto su più canali, anche simultaneamente o in maniera sovrapposta.

Come per la grafica, il sonoro è una funzionalità facoltativa in Glk.

8.1. *Risorse audio*

Come le immagini, anche i suoni sono memorizzati sotto forma di risorse e, allo stesso modo, il programma non si deve preoccupare del loro formato o della loro posizione; anche le risorse audio sono identificate da un numero intero.

In teoria, una risorsa può contenere qualunque tipo di dati sonori, di qualunque lunghezza. Una risorsa potrebbe essere anche infinitamente lunga.[1] Una risorsa può contenere anche due o più canali audio (audio

[1] La *lunghezza infinita* potrebbe essere rappresentata da una forma di codifica audio in grado di rappresentare un ciclo infinito, ma questi sono dettagli che non riguardano Glk.

stereo) ma questi non vanno confusi con i canali audio Glk. Un singolo canale Glk è sufficiente per una risorsa audio, anche se è stereofonica.[2]

8.2. Creare e distruggere canali audio

```
schanid_t glk_schannel_create(glui32 rock);
```

Crea un canale audio, come è possibile immaginare. Come sempre, vale l'avvertimento per cui il risultato può essere NULL se per qualche motivo non è possibile creare un nuovo canale.

❖ ❖ ❖

garglk-mod per il supporto audio si appoggia a SDL-mixer e MikMod. Ne consegue, in linea di principio, il supporto per un canale *musicale* per audio MOD e almeno 8 canali per suono campionato.

Inoltre, come bonus, si ottiene automaticamente il supporto per i formati audio WAV e MP3 (che non sono però accettati dallo standard Blorb).

◆ ◆ ◆

```
void glk_schannel_destroy(schanid_t chan);
```

Elimina il canale audio. Se il canale è in riproduzione viene arrestato immediatamente (senza alcun evento di notifica).

8.3. Riprodurre suoni

```
glui32 glk_schannel_play(schanid_t chan, glui32 snd);
```

Inizia la riproduzione del suono specificato sul canale. Se il canale è già impegnato (anche se dalla stessa risorsa) il suono precedente viene prima arrestato (senza alcun evento di notifica).

Il valore di ritorno è 1 se il suono è effettivamente in riproduzione, o 0 se ci sono stati dei problemi.[3]

[2] Anche in questo caso, il formato ufficiale è Blorb. I tipi di risorsa audio ammessi da Blorb sono quelli codificati in formato AIFF, MOD e OGG. Vedere le specifiche Blorb per i dettagli.

[3] Il problema più ovvio è l'irreperibilità della risorsa audio indicata. Ma potrebbero esistere altre limitazioni; per esempio, tipicamente, una libreria non è in grado di riprodurre due MOD simultaneamente. In un caso del genere la seconda richiesta fallirebbe, se la prima fosse ancora attiva.

```
glui32 glk_schannel_play_ext(schanid_t chan,
    glui32 snd, glui32 repeats, glui32 notify);
```

Ha la stessa funzione di glk_schannel_play() ma permette di specificare opzioni aggiuntive. glk_schannel_play(chan, snd) è esattamente equivalente a glk_schannel_play_ext(chan, snd, 1, 0).

Il valore repeats è il numero di volte per cui il suono deve essere ripetuto. Un valore di -1 (ovvero 0xFFFFFFFF) indica che il suono deve essere ripetuto indefinitamente. Un valore di ripetizione di 0 indica che il suono non deve essere riprodotto; non succede nulla, anche se un eventuale suono precedentemente sul canale viene interrotto e la funzione ritorna comunque 1.

Il parametro notify, se diverso da zero, richiede un evento di notifica audio. Se richiesto, nel momento in cui il canale termina la riproduzione, viene emesso un evento di tipo evtype_SoundNotify. Nell'evento win sarà NULL, val1 sarà l'id della risorsa audio e val2 sarà il valore passato come notify.

Se viene richiesta la notifica e il valore di ripetizione è maggiore di uno, l'evento verrà emesso solo dopo l'*ultima* ripetizione. Se repeats è 0 o -1, non si otterrà mai la notifica. Anche nel caso in cui il canale venisse interrotto o cancellato non viene generato alcun evento di notifica.

Dato che non tutte le librerie sono in grado di fornire le notifiche è necessario utilizzare il selettore gestalt_SoundNotify prima di farvi affidamento; vedere la sezione 8.5.

```
void glk_schannel_stop(schanid_t chan);
```

Arresta il suono in esecuzione sul canale. Non viene emesso alcun evento di notifica, anche se richiesto. Se nessun suono è in esecuzione, non ha alcun effetto.

```
void glk_schannel_set_volume(schanid_t chan,
    glui32 vol);
```

Imposta il volume del canale audio. Quando viene inizialmente creato, il canale ha il volume al massimo, rappresentato dal valore 0x10000. Metà volume è 0x8000, tre quarti a 0xC000 e così via. Un volume di 0 ha come effetto il silenzio, anche se il canale è a tutti gli effetti in riproduzione.

È possibile chiamare questa funzione fra i suoni o anche mentre il canale è in riproduzione. L'effetto è immediato.

È possibile forzare il volume oltre il massimo specificando un valore superiore a 0x10000. Questo operazione non è comunque raccomandata; la libreria potrebbe non essere in grado di aumentare ulteriormente il volume o

il l'audio potrebbe risultare distorto. È sempre possibile diminuire il volume quando è appropriato, comunque.

Non tutte le librerie sono in grado di regolare il volume. Il selettore preposto è gestalt_SoundVolume; vedere la sezione 8.5.

```
void glk_sound_load_hint(glui32 snd, glui32 flag);
```

Permette di anticipare alla libreria se un dato suono sarà presto necessario oppure no. Se flag è diverso da zero, la libreria può pre-caricare la risorsa o prepararsi in modo che glk_schannel_play() sia più veloce. Se flag è zero, la libreria può liberare memoria o altre risorse associate con la risorsa. Chiamare questa funzione è sempre facoltativo e non ha comunque effetto sul risultato (se non una maggiore efficienza).

84. Altre funzioni per i canali audio

```
schanid_t glk_schannel_iterate(schanid_t chan,
    glui32 *rockptr);
```

Permette di iterare su tutti i canali audio aperti. Vedere la sottosezione 1.6.2.

Come già indicato, l'ordine in cui vengono ritornati i canali è arbitrario.

```
glui32 glk_schannel_get_rock(schanid_t chan);
```

Ritorna il valore tenuto sotto al sasso del canale. Vedere la sottosezione 1.6.1

8.5. Verificare le funzionalità audio

Prima di utilizzare le funzioni audio Glk, bisogna verificare i seguenti selettori.

```
glui32 res;
res = glk_gestalt(gestalt_Sound, 0);
```

Se il risultato è 1, le funzioni audio sono disponibili (sono elencate in Tabella 8.1).

Se il selettore ritorna 0 non bisogna chiamare queste funzioni. Potrebbero non avere effetto o causare errori.

Come per gli altri moduli facoltativi, la scrittura di un programma in C ha la problematica aggiuntiva relativa alla presenza delle definizioni stesse

Tabella 8.1. Funzioni audio

glk_schannel_create()
glk_schannel_destroy()
glk_schannel_iterate()
glk_schannel_get_rock()
glk_schannel_play()
glk_schannel_play_ext()
glk_schannel_stop()
glk_schannel_set_volume()
glk_sound_load_hint()

di queste funzioni. Una libreria che supporti le funzioni audio definisce il simbolo del preprocessore GLK_MODULE_SOUND.

Se questo simbolo è definito, lo sono anche tutte le funzioni e le costanti in questo capitolo. In caso contrario, no.

```
glui32 res;
res = glk_gestalt(gestalt_SoundMusic, 0);
```

L'esito sarà 1 se la libreria è in grado di riprodurre risorse di tipo musicale.[4] Se ritorna 0, sono utilizzabili solo le risorse contenenti audio campionato.

```
glui32 res;
res = glk_gestalt(gestalt_SoundVolume, 0);
```

Se la funzione glk_schannel_set_volume() funziona, il selettore ritornerà 1. In caso contrario, non ha effetto.

```
glui32 res;
res = glk_gestalt(gestalt_SoundNotify, 0);
```

Se il selettore ritorna 1 la libreria è in grado di generare eventi di notifica audio. Se ritorna 0, non si otterranno mai eventi di questo tipo.

❖ ❖ ❖

garglk-mod supporta tutte le funzionalità audio. La libreria garglk originale invece non gestisce gli eventi di notifica audio.

◆ ◆ ◆

[4] Con *risorse di tipo musicale* si intendono risorse di tipo MOD, ovvero l'unico formato musicale supportato da Blorb al momento. Questo selettore è giustificato dal fatto che, attualmente, alcune librerie sono in grado di gestire AIFF ma non il formato MOD.

9 Collegamenti

Alcuni giochi potrebbero voler evidenziare il testo nelle finestre con dei collegamenti (hyperlink) selezionabili dal giocatore, molto probabilmente con un click del mouse. Glk permette di farlo con un metodo simile a quello utilizzato per gli stili di testo.

I collegamenti sono una funzionalità facoltativa di Glk.

❖ ❖ ❖

I collegamenti infatti sono l'*ultima* cosa da implementare in garglk-mod per avere tutte le funzionalità di Glk 0.6.1.

Al momento sono visualizzati ma non sono selezionabili. È anche vero che non ci sono molti giochi in circolazione che ne fanno utilizzo.

◆ ◆ ◆

9.1. Creare collegamenti

```
void glk_set_hyperlink(glui32 linkval);
void glk_set_hyperlink_stream(strid_t str,
    glui32 linkval);
```

Queste funzioni permettono si settare il *valore di collegamento* sullo stream di output corrente o sullo stream specificato, rispettivamente. Il valore di collegamento è un qualunque intero diverso da zero; zero indica *nessun collegamento*. Il testo a seguire viene considerato come corpo del collegamento, che rimane attivo fino al successivo cambiamento (eventualmente a zero, per terminarlo).

È alquanto inutile cambiare il valore di collegamento su uno stream due volte senza inserire del testo o delle immagini fra le chiamate. Il risultato sarebbe un collegamento di lunghezza zero, che il giocatore probabilmente non sarebbe in grado di vedere né selezionare; la libreria può, anzi, ottimizzare questi collegamenti nulli rimuovendoli del tutto.

Allo stesso modo, settare il valore di collegamento di uno stream al valore che ha già non ha alcun effetto.

❖ ❖ ❖

I valori di collegamento funzionano in pratica come un cambio di stile. Un valore zero termina il collegamento corrente, un valore diverso da zero ne inizia uno nuovo.

L'utilizzo con Glk è piuttosto semplice ma come associare il valore del collegamento prescelto dall'utente con il comportamento richiesto è un'altra storia...

◆ ◆ ◆

Se la libreria è in grado di visualizzare immagini nei buffer di testo, queste prendono il valore di collegamento di quando sono state inserite, proprio come fa il testo. Il giocatore può selezionare un'immagine in un collegamento esattamente come se fosse testo (anche le immagini a margine funzionano come collegamenti e, dato che non sono direttamente adiacenti al testo corrispondente, possono generare situazioni particolari).

La libreria tenterà di visualizzare i collegamenti in modo da metterli in evidenza (e lo farà indipendentemente dal fatto che sia stato richiesto l'input da hyperlink nella finestra). Naturalmente, il formato più probabile sarà blu sottolineato. Le immagini facenti parti di un collegamento potrebbero però essere indistinguibili; è consigliabile utilizzare immagini specifiche allo scopo.

9.2. Accettare eventi di hyperlink

```
void glk_request_hyperlink_event(winid_t win);
void glk_cancel_hyperlink_event(winid_t win);
```

Queste chiamate funzionano come tutte le altre utilizzate per richiedere eventi. Una richiesta in corso rimane attiva fino alla selezione di un collegamento da parte del giocatore o fino all'annullamento della richiesta.

Una finestra può avere attivo l'input da hyperlink indipendentemente da quello di mouse o a linea/carattere. Tuttavia, se sono attivi simultaneamente hyperlink e mouse, il giocatore potrebbe essere confuso e la libreria potrebbe non essere in grado di differenziare adeguatamente le due tipologie di evento. Di conseguenza è meglio evitare questa combinazione.

Quando viene selezionato un collegamento in una finestra che ha una richiesta attiva, `glk_select()` ritorna un evento di tipo `evtype_Hyperlink`.

Tabella 9.1. Funzioni relative ai collegamenti

glk_set_hyperlink()
glk_set_hyperlink_stream()
glk_request_hyperlink_event()
glk_cancel_hyperlink_event()

Nella struttura evento, win indica la finestra di provenienza, mentre val1 conterrà il valore del collegamento (ovviamente, diverso da zero).

Se non è attiva una richiesta di input da hyperlink, la libreria ignora i tentativi di selezionare un collegamento. Come al solito, nessun evento di tipo evtype_Hyperlink viene generato senza che sia stato richiesto.

9.3. Verificare le funzionalità degli hyperlink

Come sempre, prima di chiamare le funzioni relative, è necessario verificarne la disponibilità con i seguenti selettori.

```
glui32 res;
res = glk_gestalt(gestalt_Hyperlinks, 0);
```

Il risultato è TRUE (1) se le funzioni per la gestione dei collegamenti sono disponibili (Tabella 9.1).

Se il selettore ritorna FALSE (0) non bisogna provare a chiamare queste funzioni. Potrebbero non aver effetto o anche causare un errore.

Per sapere se i collegamenti sono supportati in un dato tipo di finestra si usa il selettore gestalt_HyperlinkInput.

```
glui32 res;
res = glk_gestalt(gestalt_HyperlinkInput, windowtype);
```

Il valore ritornato è TRUE (1) se una finestra del tipo specificato supporta i collegamenti. Se il risultato è FALSE (0), è comunque legale chiamare le funzioni relative, ma senza alcun effetto e comunque non si riceveranno mai gli eventi richiesti.

Per quanto riguarda la programmazione in C, è possibile testare l'esistenza del simbolo di preprocessore GLK_MODULE_HYPERLINKS per sapere se le funzioni relative sono disponibili.

10

Porting e particolari oscuri

La vita non è perfetta, e neppure i nostri giocattoli lo sono. In un mondo di giocattoli perfetti, un programma Glk potrebbe essere compilato ed eseguito con qualunque libreria Glk senza alcun intervento umano. Ma indovinate un po'...

❖ ❖ ❖

Autori Inform, se ci tenete alla vostra salute mentale saltate questo capitolo e i due successivi. Non c'è niente di utile per voi qui dentro.

Sappiate solo che le librerie Glk garglk non esportano all'avvio nessuna risorsa aggiuntiva.

E che, anche se in effetti ci sono estensioni private di Glk, queste non sono disponibili per il vostro programma.

◆ ◆ ◆

10.1. Opzioni di avvio

Una delle grandi aree grigie è costituita dall'avvio, dai file di inizializzazione e da altre opzioni del programma. Sarebbe comodo poter assumere che tutti i programmi C funzionino con il modello argc/argv, ovvero che tutte le informazioni di cui hanno bisogno provengano da un array di stringhe all'avvio. Talvolta ciò si verifica. Ma in un sistema a GUI i file spesso si aprono cliccando, le opzioni si impostano in finestre di dialogo e così via; e tutto questo non avviene necessariamente all'inizio di `main()`.

Di conseguenza, Glk non prova a passare una lista argc/argv alla funzione `glk_main()`. E nemmeno fornisce delle API portabili per file di avvio e opzioni.[1]

[1] Implementare correttamente tutto ciò richiederebbe l'interpretazione di argomenti di linea di

I file di avvio e le opzioni sono gestiti quindi in una maniera *completamente dipendente dalla piattaforma*. L'autore del programma Glk deve descrivere in che modo il programma si deve comportare. Mano a mano che il programma viene adattato (portato) su altre piattaforme, sarà necessario decidere come implementare questo comportamento sulla piattaforma in oggetto. La libreria metterà opzioni e file all'interno di variabili globali, dove la `glk_main()` potrà recuperarle.

È ragionevole separare e modularizzare questo codice, e chiamarlo *codice di avvio*. Ma il codice di avvio non è necessariamente composto da una singola funzione e certamente non avrà argomenti semplici come una lista argc/argv. È quindi possibile considerare una divisione del codice di avvio in due parti: una non-portabile e quindi facente strettamente parte del codice di avvio, l'altra pulita e completamente Glk-portabile, quindi presente all'inizio della funzione `glk_main()`.

In fin dei conti non è poi tanto complicato quanto sembra. Molti programmi, e la maggior parte di quelli dedicati all'IF, utilizzano uno di questi due modelli:

Il modello semplice Non ci sono file di avvio. Il programma semplicemente entra in esecuzione quando lanciato.

Il modello a file Il programma entra in esecuzione e ottiene un singolo file di un particolare tipo. Su sistemi a linea di comando, è un nome di file su linea di comando. Su un sistema GUI sarà generalmente un evento specifico della piattaforma contenente un riferimento al file.

Ogni libreria Glk sarà in grado di supportare questi due semplici modelli, probabilmente con l'ausilio di alcune opzioni di compilazione. I dettagli ovviamente varieranno.[2]

Sono sicuramente possibili modelli più complessi. Si potrebbero accettare file attraverso eventi della GUI in qualunque momento, e non solamente

comando di vario tipo, oltre all'implementazione di dialog box adatte allo scopo. Sicuramente è oltre al livello di complessità che ci si è prefissati per Glk.

[2] In particolare, la libreria Mac Glk ha due possibili comportamenti quando compilata con il modello a file. Se il giocatore seleziona un file, la libreria chiama immediatamente `glk_main()`. Se invece seleziona l'applicazione, la libreria permette al giocatore di attendere, magari per modificare alcune preferenza; la funzione `glk_main()` viene chiamata solo quando viene selezionato un file attraverso una finestra di dialogo.

Se la vita fosse così semplice, basterebbe aggiungere questi due modelli alle API Glk in qualche modo. Sfortunatamente non sarebbe sufficiente. Considerando AGT: il *file di gioco* è formato in realtà da una decina di file separati, con nomi correlati, nella stessa cartella. Glk non contiene chiamate per effettuare con precisione manipolazioni su nomi di file e di percorso; sarebbe una cosa troppo complessa da gestire. Per questo motivo, queste sono situazioni da gestire in modo non portabile.

all'avvio. Si potrebbe allo scopo definire un nuovo tipo di evento Glk, e far si che la libreria invii un tale evento quando la piattaforma rileva un click significativo. Bisognerebbe anche però decidere come gestire la situazione in una libreria Glk a linea di comando.

Opzioni e preferenze sono un problema a parte. In generale, una libreria su linea di comando li accetterebbe come argomenti sulla linea di comando, mentre una libreria su GUI li gestirebbe con una finestra di dialogo. Idealmente, sarebbe necessaria una descrizione per entrambi i casi: elencare le opzioni a linea di comando e magari come disporle in una finestra di dialogo.[3]

Inoltre, la libreria Glk probabilmente avrà qualche opzione propria; per esempio per gli stili di testo e così via. Una libreria su linea di comando avrà probabilmente qualche API al fine di estrarre le proprie opzioni e passarne il resto al codice di avvio.

10.2. Al di fuori delle API di Glk

I problemi di portabilità non sono limitati all'avvio del programma. Esistono anche altri servizi del sistema operativo che non sono rappresentati da Glk. Le librerie ANSI C, così familiari da sembrare universali, non sono necessariamente presenti. Le piattaforme quali PalmOS sono particolarmente famose per non utilizzare librerie ANSI.

10.2.1. Gestione della memoria

Tutti usano `malloc()`, `realloc()` e `free()`. Tuttavia alcune piattaforme posseggono sistemi di gestione della memoria nativi che potrebbero essere più adeguati per il programma.

Il sistema usato da `malloc()` è semplice; probabilmente è implementabile come strato al di sopra di qualunque API nativa sia disponibile. Di conseguenza non è necessario preoccuparsi più di tanto, ma non farebbe comunque male raggruppare le chiamate a `malloc()` e compagnia in un'unica sezione di codice, per facilitare il lavoro nel caso sia necessario adattarle.

10.2.2. Manipolazione di stringhe

Il problema con le stringhe è che l'insieme delle funzioni disponibili varia alquanto fra una piattaforma e l'altra. Per esempio, fra le funzioni *standard* ci sono `bcopy()`, `memcpy()` e `memmove()`; `stricmp()` e `strcasecmp()`; `strchr()` e `index()`; e la lista va avanti. Nonostante tutto, sotto PalmOS nessuna di

[3] Non sarebbe molto complicato. Chissà...

queste funzioni è disponibile. Per la massima sicurezza sarebbe necessario implementare personalmente le funzioni richieste.[4]

D'altra parte la massima sicurezza è anche una massima scocciatura, e probabilmente pure inefficiente (le funzioni stringa sono generalmente ottimizzate a livello di libreria standard). Ma questi sono normali problemi di porting.[5]

10.2.3. Gestione dei file

Questo è il vero problema, dato che Glk fornisce un insieme limitato di funzioni per stream e file. E nonostante tutto ci sono tutte quelle meravigliose chiamate ANSI, con tutte quelle comodità: `ungetc()`, `fgetc()`, la formattazione di `fprintf()`... per non parlare della manipolazione diretta dei pathname. Perché usare le funzioni Glk?

Il fatto è che la libreria stdio non è sempre la scelta migliore. PalmOS e Newton, molto semplicemente, non hanno stdio. Il Mac ha una libreria stdio, ma è un extra molto ingombrate e non tutti vorrebbero utilizzarla.

C'è anche il problema delle chiamate interne a Glk. Le finestre vengono gestite con degli stream Glk.[6]

Come sempre, è una questione di giudizio. Se si tratta di codice preesistente e usa molte particolarità di stdio quali `ungetc()`, potrebbe non valer la pena di cambiare tutto per utilizzare le API Glk. Ma cominciando da zero, l'utilizzo di Glk è probabilmente più pulito.

10.24. Estensioni private a Glk

Qualche volta (raramente, si spera) ci sono cose che bisogna proprio fare.

La gestione dei pathname è un caso possibile. Creare o cancellare directory. Nuovi eventi Glk causati dall'interfaccia. Controllo sui menù a tendina.

Come per il codice di avvio, bisogna decidere quel che si vuole e chiedere che ne venga fatto il porting. Come detto prima, normale amministrazione.

Se un'estensione o una nuova funzione è così utile che la implementano tutti, potrò considerare [Andrew Plotkin] di aggiungerla alle API Glk (come funzionalità aggiuntiva, con un selettore gestalt e tutto il resto). Sono flessibile. Ovviamente in modo moralmente corretto.

---

[4] Il programma `model.c` per esempio implementa le proprie `str_eq()` e `str_len()`.

[5] Incidentalmente, la prossima persona che `#define`isce `memmove()` come `memcpy()` quando non è disponibile la vera `memmove()` si merita di essere schiaffeggiata con un salmone di gomma foderato di piombo.

[6] Sarebbe stato bello poter utilizzare stdio per le finestre, ma in generale non è possibile.

10.3. Glk e la macchina virtuale

La maggior parte dei giochi di IF sono costruiti attorno ad una virtual machine, come la Z-machine V5–8 (con la sola eccezione del testo colorato).

10.3.1. Glk come API nativa

Un altro approccio è quello di utilizzare direttamente Glk come sistema di I/O per la virtual machine e di esporla direttamente all'autore. La macchina virtuale Glulx funziona esattamente in questo modo. È indubbiamente più potente, difatti tutte le funzionalità di Glk diventano accessibili, non solo un sottoinsieme. Poiché Glk è progettata per essere facilmente espandibile e verranno definite nuove funzionalità (opzionali) nel tempo, in questo modo la VM viene automaticamente estesa senza troppe modifiche.[7][8]

Come esportare Glk è un problema comunque non banale, dato che Glk ha molte funzioni e oltremodo ne verranno aggiunte altre col tempo.

In una VM con un numero limitato di opcode è probabilmente meglio allocare un singolo opcode "Glk", con un numero variabile di argomenti, il primo dei quali è un selettore di funzione (Glulx funziona in questo modo). È meglio prevedere almeno 16 bit per il selettore; potrebbero esistere più di 256 chiamate Glk un giorno (per una lista dei selettori standard, vedere la sezione 11.6).

Se la VM ha un grande range a disposizioen per gli opcode, allora è possibile riservare un blocco di 65536 opcode per Glk.

Potrebbe anche essere possibile estendere il meccanismo di chiamata a funzione in qualche modo per includere le funzioni Glk.

In ogni caso, le API devono essere presentate all'autore in qualunque linguaggio venga compilato per la VM. Idealmente, potrebbero essere una serie di chiamate a funzione.[9]

[7] A dire il vero, Glk è stata progettata per essere utilizzata specialmente in questo modo. Questo è il motivo per cui tutti gli argomenti sono puntatori o interi di 32 bit, e anche il motivo per cui tutte le strutture pubbliche sono effettivamente array di interi. È anche il motivo per cui esistono le funzioni di iterazione, dato che l'intero spazio di memoria potrebbe essere alterato da un comando di *undo* o *restore*. Inoltre, questo è anche il motivo per cui Glk fornisce accesso ai file. In questo modo una VM può esporre Glk come unica interfaccia sia per lo schermo che per il filesystem. In questo modo la VM diventa più semplice, modulare e non legata al sistema operativo; nulla di cui ci si possa lamentare, anzi!

[8] La Società dei Pignoli del C fa notare che le strutture nelle API Glk non sono *realmente* array di 32 bit. Una struct, anche se composta solo da interi di 32 bit, può comunque essere allineata dal compilatore nel modo che preferisce. Il problema viene comunque risolto dallo strato di dispatch.

[9] Ma non necessariamente. Il compilatore Inform, per esempio, è in grado di accettare codice assembly in linea al codice Inform. È quasi conveniente come permettere all'autore di usare un opcode come chiamata a funzione.

Esiste anche la complicazione dovuta alle nuove chiamate aggiunte a Glk. Non dovrebbe comunque essere un grande problema. Il compilatore sta comunque mappando le chiamate Glk con delle funzioni o degli opcode, quindi probabilmente basta aggiornare una lista da qualche parte all'interno del compilatore per attivare le nuove funzioni.

Alternativamente, se il compilatore è in grado di definire nuovi opcode, anche questo non è necessario.[10]

È anche possibile fornire un'interfaccia completamente dinamica a Glk. Questo è compito dello strato di dispatch, che non fa parte di Glk; piuttosto, vi sta sopra. Vedere il Capitolo 11.

[10] Il compilatore Inform è progettato in questo modo; l'autore del gioco può definire ed utilizzare nuovi op-code. Quindi se viene aggiunta una nuova chiamata a Glk (ed è stata implementata nell'interprete con un selettore noto) può essere utilizzata immediatamente in Inform, senza dover aggiornare il compilatore.

11

Lo strato di dispatch

Il materiale descritto in questo capitolo non fa parte delle API Glk. È uno strato esterno, al di sopra di Glk, che permette ad un programma di invocare Glk dinamicamente, determinando interfacce e funzionalità disponibili a runtime.

Ciò è particolarmente utile per macchine virtuali e altri sistemi con runtime che vogliano utilizzare Glk senza essere vincolati ad una particolare versione delle API Glk. In altre parole è possibile esportare l'intera libreria Glk, senza una lista predefinita delle funzioni Glk. In questo modo, se viene rilasciata una nuova versione di Glk le nuove funzioni saranno immediatamente disponibili senza lavoro aggiuntivo.

Se si scrive un programma in C che usa Glk, si può ignorare completamente questa sezione. Se si sta scrivendo una VM che utilizza Glk, può essere utile leggerlo. Se si sta implementando una libreria Glk, è necessario leggerlo (esistono alcune interfacce interne che devono essere implementate per far funzionare correttamente lo strato di dispatch).

❖ ❖ ❖

Se siete autori Inform/glulx, anche in questo capitolo non troverete nulla di utile per voi.

Eccetto forse la tabella dei selettori in coda al capitolo, nel caso foste così masochisti da non utilizzare infglk (Capitolo 13) o una libreria ad alto livello come sgw (Capitolo 17).

◆ ◆ ◆

11.1. Come funziona

Lo strato di dispatch è implementato in un sorgente C, gi_dispa.c, e nel corrispondente header gi_dispa.h. Il codice è indipendente dalla piattaforma: identico in ogni libreria, proprio come glk.h. Ogni implementatore della libreria dovrebbe utilizzare i due file (scaricabili dal sito web Glk) e compilarli senza modifiche nella libreria.

Sostanzialmente, il codice è esterno a Glk; funziona chiamando le API Glk documentate, non funzioni interne alla libreria. In questo modo gi_dispa.c può essere indipendente dalla piattaforma. Sfortunatamente questa divisione non è perfetta. Esistono alcune funzioni extra, non facenti parte delle API Glk, che la libreria deve implementare; gi_dispa.c (e nessun altro) le utilizza. Sono comunque funzioni semplici e non dovrebbero complicare la vita agli implementatori di librerie.

Lo strato di dispatch fornisce quindi le *sue* API. Al cuore di tutto sta la funzione gidispatch_call() che permette di chiamare *qualunque* funzione Glk (specificata da un selettore) e passare una lista di argomenti in modo standardizzato. La funzione gidispatch_prototype() fornisce il formato di questa lista per ogni funzione. Altre funzioni ausiliarie consentono di enumerare funzioni e costanti delle API Glk.

11.2. Interrogare l'interfaccia

Ecco le funzioni ausiliare che consentono di enumerare classi, funzioni e costanti.

```
glui32 gidispatch_count_classes(void);
```

Ritorna in numero di classi di oggetti opachi utilizzati dalla libreria. Questo valore è utile se è necessario tener traccia degli oggetti opachi che vengono creati; vedere la sottosezione 11.5.1.

Nelle API Glk 0.7.0 sono definite quattro classi: finestre, stream, fileref e canali audio (numerate rispettivamente 0, 1, 2 e 3).

```
glui32 gidispatch_count_intconst(void);
```

Ritorna il numero di costanti intere esportate dalla libreria.

```
typedef struct gidispatch_intconst_struct {
    char *name;
    glui32 val;
} gidispatch_intconst_t;

gidispatch_intconst_t *
  gidispatch_get_intconst(glui32 index);
```

Ritorna una struttura che descrive una costante intera esportata dalla libreria. Le costanti sono, approssimativamente, tutte quelle definite nel file `glk.h`. `index` può assumere valori da 0 a $N-1$, dove N è il valore ritornato da `gidispatch_count_intconst()`.

La struttura contiene semplicemente una stringa e il valore ad essa associato. La stringa è una rappresentazione del nome del valore, e può essere riesportata a chiunque possa essere interessato alle costanti Glk.

```
glui32 gidispatch_count_functions(void);
```

Ritorna il numero di funzioni esportate dalla libreria.

```
typedef struct gidispatch_function_struct {
    glui32 id;
    void *fnptr;
    char *name;
} gidispatch_function_t;

gidispatch_function_t *
  gidispatch_get_function(glui32 index);
```

Ritorna una struttura che descrive una funzione Glk. `index` varia da 0 a $N-1$, dove N è il valore ritornato da `gidispatch_count_functions()`.

Il campo `id` è un selettore, una costante numerica utilizzata per riferirsi alla funzione in questione. `name` è il nome della funzione, come nel file `glk.h` ma senza il prefisso `glk_`. E `fnptr` è l'indirizzo della funzione.[1]

```
gidispatch_function_t *
  gidispatch_get_function_by_id(glui32 id);
```

Ritorna informazioni sulla funzione Glk che ha selettore `id`. Se non esiste una simile funzione, ritorna `NULL`.

[1] L'indirizzo è stato incluso in quanto potrebbe essere utile, ma il suo uso *non* è consigliato. Per chiamare una funzione Glk arbitraria è meglio usare `gidispatch_call()`. Vedere la sezione 11.6 per l'elenco dei selettori. Vedere la sezione 11.3 per ulteriori informazioni su come chiamare le funzioni Glk via selettore.

11.3. Effettuare chiamate

```
typedef union gluniversal_union {
    glui32 uint;
    glsi32 sint;
    void *opaqueref;
    unsigned char uch;
    signed char sch;
    char ch;
    char *charstr;
    void *array;
    glui32 ptrflag;
} gluniversal_t;

void gidispatch_call(glui32 funcnum,
    glui32 numargs, gluniversal_t *arglist);
```

funcnum è il numero di funzione da chiamare; vedere la sezione 11.6. arglist è la lista dei parametri, mentre numargs è la lunghezza della lista.

Gli argomenti sono memorizzati come oggetti di tipo gluniversal_t, una union in grado di contenere qualunque tipo possa essere passato ad una funzione Glk.

11.3.1. Tipi di base

Gli argomenti numerici sono passati in maniera ovvia: un argomento per ogni gluniversal_t, con il campo uint o sint al valore desiderato. Anche caratteri e stringhe sono passate in questo modo; i caratteri nei campi uch, sch o ch (a seconda del tipo, segnato o meno) e le stringhe nel campo charstr. Gli oggetti opachi (finestre, stream, ecc.) sono passati nel campo opaqueref (che essendo void* può contenere qualunque tipo di puntatore).

Ciò che complica la vita sono i puntatori (quelli che non rappresentano stringhe), gli array e le strutture. Così come i valori di ritorno.

11.3.2. Riferimenti

Un riferimento ad un tipo numerico o un riferimento ad oggetto (come ad esempio glui32* o winid_t*) impegna *uno o due* elementi nella lista. Il primo serve ad indicare se l'argomento è NULL oppure no. Il campo ptrflag di questo gluniversal_t deve essere FALSE (0) se il riferimento è NULL, TRUE (1) altrimenti. Se è FALSE (0), non è necessario indicare altro; non bisogna utilizzare un altro gluniversal_t per indicare esplicitamente il riferimento

NULL. Se il flag è TRUE (1), bisogna invece aggiungere un gluniversal_t per il tipo base del riferimento.

Considerando per esempio una funzione ipotetica, con selettore 0xABCD:

```
void glk_glomp(glui32 num, winid_t win,
    glui32 *numref, strid_t *strref);
```

...e la chiamata

```
glui32 value;
winid_t mainwin;
strid_t gamefile;
glk_glomp(5, mainwin, &value, &gamefile);
```

Per effettuare lo stesso utilizzando gidispatch_call() si userà:

```
gluniversal_t arglist[6];
arglist[0].uint = 5;
arglist[1].opaqueref = mainwin;
arglist[2].ptrflag = TRUE;
arglist[3].uint = value;
arglist[4].ptrflag = TRUE;
arglist[5].opaqueref = gamefile;
gidispatch_call(0xABCD, 6, arglist);
value = arglist[3].uint;
gamefile = arglist[5].opaqueref;
```

Notare come il valore dei parametri per riferimento viene travasato nell'arglist. Ovviamente, se la funzione glk_glomp() utilizzasse questi valori solo come riferimenti in ingresso o in uscita, non sarebbe necessario effettuare entrambe le copie.

Altri esempi:

```
glk_glomp(7, mainwin, NULL, NULL);
```

diventa

```
gluniversal_t arglist[4];
arglist[0].uint = 7;
arglist[1].opaqueref = mainwin;
arglist[2].ptrflag = FALSE;
arglist[3].ptrflag = FALSE;
gidispatch_call(0xABCD, 4, arglist);
```

```
glk_glomp(13, NULL, NULL, &gamefile);
```

diventa

```
gluniversal_t arglist[5];
arglist[0].uint = 13;
arglist[1].opaqueref = NULL;
arglist[2].ptrflag = FALSE;
arglist[3].ptrflag = TRUE;
arglist[4].opaqueref = gamefile;
gidispatch_call(0xABCD, 5, arglist);
gamefile = arglist[4].opaqueref;

glk_glomp(17, NULL, &value, NULL);
```

diventa

```
gluniversal_t arglist[5];
arglist[0].uint = 17;
arglist[1].opaqueref = NULL;
arglist[2].ptrflag = TRUE;
arglist[3].uint = value;
arglist[4].ptrflag = FALSE;
gidispatch_call(0xABCD, 5, arglist);
value = arglist[3].uint;
```

Come si può notare la lunghezza di arglist dipende da quanti riferimenti sono NULL.

11.3.3. Strutture

Un puntatore a struct è rappresentato da un singolo ptrflag, seguito eventualmente da una sequenza di gluniversal_t (uno per ogni campo della struct. Anche in questo caso, se il puntatore alla struct è non-NULL, il ptrflag sarà TRUE (1) e seguito dai valori; in caso contrario il ptrflag sarà FALSE (0) (indicante quindi NULL) e solitario.

Per esempio, la funzione glk_select() può essere chiamata nel modo seguente:

```
event_t ev;
gluniversal_t arglist[5];
arglist[0].ptrflag = TRUE;
gidispatch_call(0x00C0, 5, arglist);
ev.type = arglist[1].uint;
ev.win = arglist[2].opaqueref;
ev.val1 = arglist[3].uint;
ev.val2 = arglist[4].uint;
```

Dato che la struttura passata è solo in uscita (i valori all'ingresso sono ignorati), non è stato necessario valorizzare `arglist [1...4]` prima di chiamare la funzione con `gidispatch_call()`.[2]

11.34. Array

Nelle API Glk, un array viene sempre fatto seguire da un argomento numerico contenente la lunghezza dell'array. Questi due parametri vengono trattati come un singolo argomento logico, rappresentato da *uno o tre* oggetti `gluniversal_t`. Il primo è un `ptrflag`, indicante se l'argomento è `NULL` oppure no. Il secondo è un puntatore, nel campo `array`. Il terzo è la lunghezza dell'array, nel campo `uint`. Come sempre, se il `ptrflag` è `FALSE` (0), i due seguenti devono essere omessi.

La funzione `glk_put_buffer()` viene quindi chiamata come segue:

```
char buf[64];
glui32 len = 64;
glk_put_buffer(buf, len);
```

diventa

```
gluniversal_t arglist[3];
arglist[0].ptrflag = TRUE;
arglist[1].array = buf;
arglist[2].uint = len;
gidispatch_call(0x0084, 3, arglist);
```

Visto che viene passato un array di `char` a `gidispatch_call()` i contenuti saranno letti direttamente da quello; non vi è alcun bisogno di copiare i dati nella arglist, come per un tipo di base.

Se si sta implementando una VM la cui rappresentazione di un array di caratteri è più complessa, è necessario del lavoro ulteriore. È necessario, prima di tutto, allocare un array di `char`, copiare i caratteri in esso, effettuare la chiamata e quindi liberare la memoria.[3]

11.3.5. Valori di ritorno

Il valore di ritorno non ha alcun trattamento particolare. È semplicemente un passaggio per riferimento (che non può essere `NULL`). Nella arglist appare in coda, dopo tutti gli altri argomenti della funzione.

---

[2] `glk_select(NULL)` potrebbe essere effettuata settando a `FALSE` (0) `arglist [0]. ptrflag` e utilizzando una arglist di un solo elemento. Ma è illegale passare `NULL` a `glk_select()`, quindi non si può.

[3] `glk_put_buffer()` non modifica l'array, quindi non è necessario ricopiarli in uscita, in questo caso.

Per esempio, la funzione glk_window_get_rock() può essere chiamata in questo modo:

```
glui32 rock;
winid_t win;
rock = glk_window_get_rock(win);
```

diventa

```
gluniversal_t arglist[3];
arglist[0].opaqueref = win;
arglist[1].ptrflag = TRUE;
gidispatch_call(0x0021, 3, arglist);
rock = arglist[2].uint;
```

114. Ottenere i prototipi di funzione

Esistono molti modi possibili per costruire una arglist, ed è illegale chiamare gidispatch_call() con un array non adatto alla funzione richiesta. Inoltre, alcuni riferimenti sono in ingresso, altri sono in uscita e altri ancora in entrambi i sensi. Come sapere in che modo costruire la lista degli argomenti?

Una possibilità sarebbe quella di riconoscere ogni selettore a funzione e preparare la lista adeguatamente. Ma in questo modo sarebbe necessario scrivere codice speciale per ogni funzione Glk; il che è esattamente quello che *non* si vuole fare.

Per risolvere il problema, si chiama gidispatch_prototype().

```
char *gidispatch_prototype(glui32 funcnum);
```

Il valore ritornato è una stringa che rappresenta la lista degli argomenti richiesta dalla funzione. Se non esiste una tale funzione, il risultato è NULL.

La stringa di prototipo per la funzione immaginaria glk_glomp() descritta sopra sarebbe "4IuQa&Iu&Qb:". Il '4' è il numero di argomenti (incluso il valore di ritorno, se c'è; in questo caso no). "Iu" indica un intero senza segno; "Qa" è un oggetto opaco di class 0 (una finestra). "&Iu" è un *riferimento* ad un intero senza segno, mentre "&Qb" è un riferimento ad uno stream (classe 1). I due punti terminano la lista degli argomenti; sarebbe seguito dal valore di ritorno, se ci fosse.

Il numero iniziale ("4" in questo caso) è il numero di argomenti logici, non il numero di oggetti gluniversal_t passati a gidispatch_call(). Come visto sopra, la funzione glk_glomp() utilizzerebbe da quattro a sei oggetti.

I codici per i tipi di base sono:

Iu, Is Interi a 32 bit, senza segno e segnati, rispettivamente.

Cn, Cu, Cs `char`, `unsigned char` e `signed char`.[4]

S Una stringa C (un array di caratteri terminato da `NUL`). In Glk le stringhe sono sempre in sola lettura ed utilizzate immediatamente; la libreria non mantiene riferimenti ad esse dopo il termine della chiamata. Una funzione Glk che intenda utilizzare un array di caratteri in scrittura utilizzerà piuttosto un array ("#C"), non una stringa ("S").

U Un array di interi a 32 bit terminato da 0. Il concetto è quello del tipo equivalente a "S" ma per i caratteri Unicode. Per questo motivo valgono le stesse considerazioni: le stringhe "U" sono utilizzate immediatamente, e in sola lettura. Un array scrivibile di caratteri Unicode sarà dichiarato come "#Iu", piuttosto.

F Un valore in virgola mobile. Attualmente Glk non utilizza questo tipo di dati, ma intanto il codice è stato definito.

Qa, Qb, Qc... Un oggetto opaco. La seconda lettera determina la classe richiesta.[5] Si può ottenere il numero di classi correntemente definite (cioè 4) con `gidispatch_count_classes()`; vedere la sezione 11.2.

Ogni codice di tipo può essere prefissato da uno o più dei seguenti caratteri (l'ordine non è significativo):

& Un riferimento al tipo; ovvero una variabile passata per indirizzo. Il riferimento è utilizzato sia in entrata che in uscita, quindi è necessario valorizzarlo prima della chiamata a `gidispatch_call()` e copiare il risultato in seguito.

< Un riferimento utilizzato solo in uscita. Il valore iniziale viene ignorato, è necessario solo copiare il valore al ritorno.

> Un riferimento utilizzato solo in ingresso.[6]

+ In combinazione con "&", "<" o ">", indica che è obbligatorio un riferimento valido; `NULL` non può essere accettato.[7]

: Separa gli argomenti dal valore di ritorno o termina la stringa se non c'è valore di ritorno. Dato che i valori di ritorno sono sempre obbligatori in uscita, può essere considerato equivalente a "<+". I due punti non sono mai combinati con altri prefissi.

[4] A seconda della piattaforma `Cn` sarà equivalente a `Cu` o a `Cs`. Per questo motivo Glk non lo utilizza, ma è incluso solo per completezza.

[5] Se Glk verrà espansa a più di 26 classi ci si inventerà qualcosa.

[6] Per i tipi semplici non è utile, ma viene utilizzato per strutture e array.

[7] Attenzione, anche se il `ptrflag` in questo caso sarà sempre `TRUE` (1), non può essere omesso.

[...] In combinazione con "&", "<" o ">", indica un riferimento a struttura. Fra le parentesi è indicata una stringa di argomenti completa, completa di numero di argomenti.[8] Attualmente le strutture in Glk contengono solo tipi di base.

In combinazione con "&", "<" o ">", indica un riferimento ad array. Come descritto sopra, può essere rappresentato da fino a tre oggetti in arglist: `ptrflag`, puntatore e lunghezza intera. Vedere la sottosezione 11.3.4 indica un riferimento ad array.

! In combinazione con "#", indica che l'array viene *trattenuto* dalla libreria. La libreria manterrà un riferimento all'array, e i suoi contenuti sono indefiniti fino a comunicazione successiva. Non è possibile usare i contenuti dell'array dopo la chiamata, neppure per array di tipo "&#!" o "<#!".[9]

11.5. Funzioni che la libreria deve fornire

In un mondo ideale, i tre strati (programma, dispatch e libreria Glk) sarebbero completamente modulari; ogni strato si riferirebbe solo ai sottostanti. Sfortunatamente esistono situazioni in cui la libreria deve notificare qualcosa al programma. Ancora peggio, queste situazioni sono rilevanti solo per i programmi che utilizzano lo strato di dispatch, e nemmeno a tutti.

Dato che il C non è adatto al concetto di *chiamata ad una funzione che potrebbe non esistere*, Glk utilizza puntatori a funzioni callback. Il programma può passare le callback alla libreria; se lo fa, la libreria le utilizza, in caso contrario, non ci prova nemmeno.

Queste callback sono facoltative, nel senso che *il programma* può dichiararle oppure no. Per contro, *ogni* libreria che vuole essere compatibile con lo strato di dispatch deve *permettere* al programma di impostarle; la scelta è del programma, non della libreria. La libreria per far questo implementa le funzioni di registry; a queste funzioni il programma passa le sue callback.[10]

[8] Per esempio, il prototipo di `glk_select()` è "1<+[4IuQaIuIu]:": un riferimento, in uscita e non-NULL, ad una struttura contenente quattro argomenti.

[9] Per esempio, `glk_stream_open_memory()` trattiene l'array passato e lo rilascia quando lo stream viene chiuso. La libreria può notificare automaticamente quando gli array vengono trattenuti o rilasciati; vedere la sottosezione 11.5.2.

[10] Anche se queste callback e le funzioni relative sono dichiarate in `gi_dispa.h`, non sono definite in `gi_dispa.c`. Lo strato di dispatch non fa altro che coordinarle. Il programma definisce le funzioni di callback; la libreria le chiama.

11.5.1. Il registro degli oggetti opachi

Le API Glk utilizzano puntatori per riferirsi agli oggetti opachi; ma una VM forse non è in grado di tener traccia di tutti questi puntatori a oggetti in memoria. È quindi utile tener traccia degli oggetti opachi.[11]

Per fare questo, una libreria Glk deve implementare la seguente funzione:

```
typedef union glk_objrock_union {
    glui32 num;
    void *ptr;
} gidispatch_rock_t;

void gidispatch_set_object_registry(
  gidispatch_rock_t (*reg)(void *obj, glui32 objclass),
  void (*unreg)(void *obj, glui32 objclass,
    gidispatch_rock_t objrock));
```

Il programma chiama questa funzione all'inizio (prima di cominciare ad eseguire codice utente). Vengono passati due puntatori a funzione, corrispondenti ai seguenti prototipi:

```
gidispatch_rock_t my_vm_reg_object(void *obj,
    glui32 objclass);
void my_vm_unreg_object(void *obj,
    glui32 objclass, gidispatch_rock_t objrock);
```

Quando la libreria Glk crea un oggetto chiama `my_vm_reg_object()`. Ad essa verranno passati il puntatore all'oggetto e il numero di classe (da 0 a $N-1$, dove N è il valore ritornato da `gidispatch_count_classes()`).

È possibile ritornare qualunque valore nell'oggetto `gidispatch_rock_t`; la libreria lo terrà da conto all'interno dell'oggetto.[12]

La funzione `my_vm_unreg_object()` viene invece chiamata quando la libreria distrugge un oggetto. I valori passati sono il puntatore all'oggetto, in numero di classe e il valore sotto l'*altro* sasso.

[11] Per esempio, utilizzando una hash table per ogni classe di oggetto opaco, mappando identificatori interi sui puntatori.

[12] Questo valore è messo sotto *un altro* sasso. Non ha nessuna relazione con il tradizionale valore `glui32` dichiarato durante la creazione.

Questo valore è accessibile in qualunque momento. La libreria fornisce la seguente funzione:

```
gidispatch_rock_t gidispatch_get_objrock(void *obj,
    glui32 objclass);
```

Con l'ausilio di questa funzione e delle callback è possibile mantenere (per esempio) una hash table e convertire rapidamente le chiavi della tabella in puntatori Glk (e viceversa). Un sistema più sofisticato (come Java) potrebbe creare un oggetto della VM per ogni oggetto Glk, permettendo una manipolazione intelligente degli oggetti Glk.

Esiste un particolare significativo: è possibile che alcuni oggetti Glk siano già stati creati nel momento in cui viene chiamata glk_main().[13] È quindi possibile che durante la chiamata a gidispatch_set_object_registry() venga *immediatamente* chiamata la funzione my_vm_reg_object() registrata, per notificare degli oggetti preesistenti. È necessario essere preparati a questa eventualità.[14]

11.5.2. Il registro degli array trattenuti

Alcune funzioni Glk accettano un array e se lo tengono da parte. La memoria viene *posseduta* dalla libreria fino al momento in cui qualche chiamata la rilascia. Mentre la libreria ha il possesso di un array, il programma non deve leggere, scrivere, spostare o de-allocare la memoria associata. Quando la libreria lo rilascia, è nella sua forma definitiva e quindi vi si può fare quel che si vuole.

Per tener traccia di ciò la libreria deve implementare la seguente funzione:

```
void gidispatch_set_retained_registry(
    gidispatch_rock_t (*reg)(void *array,
        glui32 len, char *typecode),
    void (*unreg)(void *array, glui32 len,
        char *typecode, gidispatch_rock_t objrock));
```

[13] Per esempio, MacGlk può aver già aperto uno stream con il file selezionato dall'utente; questo avviene prima di glk_main().

[14] Per esempio, creando le hash table *prima* di registrare le funzioni di callback.

Anche in questo caso vengono specificati due puntatori a funzione:

```
gidispatch_rock_t my_vm_reg_array(void *array,
    glui32 len, char *typecode);
void my_vm_unreg_array(void *array, glui32 len,
    char *typecode, gidispatch_rock_t objrock);
```

Nel momento in cui una funzione prende possesso di un array (lo trattiene) chiama my_vm_reg_array(). Questo può succedere solo per gli array con il prefisso "#!".[15] La libreria passa l'array e la sua lunghezza, come specificate nell'array di gluniversal_t. Passa anche la stringa che descrive il tipo di argomento.[16]

È possibile restituire qualunque valore come gidispatch_rock_t; la libreria lo tiene da conto assieme all'array.

Quando una funzione Glk rilascia un array trattenuto chiama la funzione my_vm_unreg_array(). Durante questa seconda chiamata vengono passati gli stessi parametri di prima, con in più il valore sotto al secondo sasso ritornato precedentemente da my_vm_reg_array().

Con queste due callback è possibile tener traccia degli array trattenuti all'interno della libreria. In questo modo è possibile, per esempio, copiare i dati in strutture proprietarie, o tenere bloccata della memoria rilocabile o impedire che venga de-allocata per sbaglio.

11.6. La tabella dei selettori

Questi valori, e tutti i valori che saranno impiegati nelle future chiamate Glk, sono interi nell'intervallo 1–65535. I valori non sono sequenziali; possono essere divisi in gruppi, approssimativamente per categoria. Zero non è un selettore valido, quindi può essere utilizzato come NULL, se necessario.

Tabella 11.1.: Selettori di dispatch per Glk 0.7

Selettore	Funzione
0x0001	glk_exit
0x0002	glk_set_interrupt_handler
0x0003	glk_tick

Continua...

[15] Ma non sempre. Solo una chiamata a my_vm_reg_array() lo può confermare.

[16] Al momento le uniche quattro chiamate che trattengono array sono le due funzioni glk_request_line_event(), glk_stream_open_memory(), e le corrispondenti due funzioni Unicode glk_request_line_event_uni() e gli_stream_open_memory_uni() introdotte in Glk 0.7. Le prime due usano array di caratteri, con tipo "&+#!Cn". Le altre due invece array di glui32, con tipo "&+#!Iu".

Selettore	Funzione
0x0004	glk_gestalt
0x0005	glk_gestalt_ext
0x0020	glk_window_iterate
0x0021	glk_window_get_rock
0x0022	glk_window_get_root
0x0023	glk_window_open
0x0024	glk_window_close
0x0025	glk_window_get_size
0x0026	glk_window_set_arrangement
0x0027	glk_window_get_arrangement
0x0028	glk_window_get_type
0x0029	glk_window_get_parent
0x002A	glk_window_clear
0x002B	glk_window_move_cursor
0x002C	glk_window_get_stream
0x002D	glk_window_set_echo_stream
0x002E	glk_window_get_echo_stream
0x002F	glk_set_window
0x0030	glk_window_get_sibling
0x0040	glk_stream_iterate
0x0041	glk_stream_get_rock
0x0042	glk_stream_open_file
0x0043	glk_stream_open_memory
0x0044	glk_stream_close
0x0045	glk_stream_set_position
0x0046	glk_stream_get_position
0x0047	glk_stream_set_current
0x0048	glk_stream_get_current
0x0060	glk_fileref_create_temp
0x0061	glk_fileref_create_by_name
0x0062	glk_fileref_create_by_prompt
0x0063	glk_fileref_destroy
0x0064	glk_fileref_iterate
0x0065	glk_fileref_get_rock
0x0066	glk_fileref_delete_file
0x0067	glk_fileref_does_file_exist
0x0068	glk_fileref_create_from_fileref
0x0080	glk_put_char
0x0081	glk_put_char_stream
0x0082	glk_put_string

Continua...

Selettore	**Funzione**
0x0083	glk_put_string_stream
0x0084	glk_put_buffer
0x0085	glk_put_buffer_stream
0x0086	glk_set_style
0x0087	glk_set_style_stream
0x0090	glk_get_char_stream
0x0091	glk_get_line_stream
0x0092	glk_get_buffer_stream
0x00A0	glk_char_to_lower
0x00A1	glk_char_to_upper
0x00B0	glk_stylehint_set
0x00B1	glk_stylehint_clear
0x00B2	glk_style_distinguish
0x00B3	glk_style_measure
0x00C0	glk_select
0x00C1	glk_select_poll
0x00D0	glk_request_line_event
0x00D1	glk_cancel_line_event
0x00D2	glk_request_char_event
0x00D3	glk_cancel_char_event
0x00D4	glk_request_mouse_event
0x00D5	glk_cancel_mouse_event
0x00D6	glk_request_timer_events
0x00E0	glk_image_get_info
0x00E1	glk_image_draw
0x00E2	glk_image_draw_scaled
0x00E8	glk_window_flow_break
0x00E9	glk_window_erase_rect
0x00EA	glk_window_fill_rect
0x00EB	glk_window_set_background_color
0x00F0	glk_schannel_iterate
0x00F1	glk_schannel_get_rock
0x00F2	glk_schannel_create
0x00F3	glk_schannel_destroy
0x00F8	glk_schannel_play
0x00F9	glk_schannel_play_ext
0x00FA	glk_schannel_stop
0x00FB	glk_schannel_set_volume
0x00FC	glk_sound_load_hint
0x0100	glk_set_hyperlink

Continua...

Selettore	Funzione
0x0101	glk_set_hyperlink_stream
0x0102	glk_request_hyperlink_event
0x0103	glk_cancel_hyperlink_event
0x0120	glk_buffer_to_lower_case_uni
0x0121	glk_buffer_to_upper_case_uni
0x0122	glk_buffer_to_title_case_uni
0x0128	glk_put_char_uni
0x0129	glk_put_string_uni
0x012A	glk_put_buffer_uni
0x012B	glk_put_char_stream_uni
0x012C	glk_put_string_stream_uni
0x012D	glk_put_buffer_stream_uni
0x0130	glk_get_char_stream_uni
0x0131	glk_get_buffer_stream_uni
0x0132	glk_get_line_stream_uni
0x0138	glk_stream_open_file_uni
0x0139	glk_stream_open_memory_uni
0x0140	glk_request_char_event_uni
0x0141	glk_request_line_event_uni

La funzione `glk_main()` non ha un selettore, dato che è fornita dal programma e non dalla libreria.

Non esiste un modo per utilizzare i selettori direttamente con le API Glk.[17] Sono definiti solo per essere utilizzati con lo strato di dispatch.

[17] Versioni precedenti di Glk avevano selettori gestalt per la conversione, ma sono stati rimossi.

12

Lo strato Blorb

Il materiale di questo capitolo non fa parte delle API Glk. È uno strato aggiuntivo che permette alla libreria di caricare risorse (immagini e suoni) da un file specificato dal programma. Il formato Blorb è uno standard per archiviare risorse nell'IF.

Le specifiche Glk non richiedono che le risorse siano prelevate da un file Blorb. Indicano solo che la libreria sa come caricarle ed utilizzarle quando sono richieste. D'altra parte Blorb è il formato raccomandato per fornire portabilmente queste risorse. La maggior parte delle librerie Glk forniranno supporto Blorb con le funzioni definite in questo capitolo.

Per le specifiche Blorb e gli strumenti per la manipolazione dei file, vedere il sito:

http://www.eblong.com/zarf/blorb/

❖ ❖ ❖

Per gli autori di avventure, non è tanto utile sapere come l'interprete accede ai file Blorb, quanto il *come crearne*.

A seconda della piattaforma in uso esistono diversi strumenti: il Capitolo 16 tratta i più comuni (non che ce ne siano tanti).

Durante lo sviluppo o in circostanze particolari (per esempio quando qualche risorsa *proprio non ne vuol saperne di funzionare* e si sospetta del file Blorb) il meccanismo di caricamento di riserva è molto simile a quello suggerito: viene costruito un nome di file con "PIC" o "SND" come primi caratteri (rispettivamente per risorse grafiche o audio) e il numero della risorsa in decimale, *senza estensione*.

Il formato della risorsa viene determinato automaticamente in base al contenuto.

◆ ◆ ◆

12.1. Come funziona

Come per lo strato di dispatch, anche il supporto per Blorb è implementato in un sorgente separato, gi_blorb.c e nel suo header gi_blorb.h. Il codice è (per la maggior parte) indipendente dalla piattaforma. Ogni libreria che intente supportare Blorb dovrebbe includere questi file (scaricabili dal sito sopraindicato).

La maggior parte delle funzioni definite in gi_blorb.h sono destinate all'uso della libreria. Durante la scrittura di un programma Glk possono essere ignorate, a parte giblorb_set_resource_map(); vedere sezione 12.2. Se si sta implementando una libreria Glk, è possibile utilizzare queste API per trovare e caricare i dati relativi alle risorse.

12.2. Cosa fa il programma

Se si vuol fare in modo che il programma carichi le risorse da un file Blorb è necessario trovare ed aprire quel file nel codice di avvio (vedere la sezione 10.1). Ciascuna piattaforma avrà funzioni appropriate disponibili per trovare i dati relativi. È importante aprire il file in modo binario.

Una volta che si è ottenuto uno stream Glk collegato al file, basta passarlo a giblorb_set_resource_map().

```
giblorb_err_t giblorb_set_resource_map(strid_t file);
```

Questa funziona indica alla libreria che il file è la sorgente Blorb di tutte le risorse. Ogni volta che il programma richiede un'immagine o un suono, la libreria cercherà all'interno di questo file la risorsa richiesta.

Non bisogna chiudere lo stream dopo aver chiamato questa funzione. La libreria è responsabile della sua chiusura.

Se non si utilizza la funzione giblorb_set_resource_map() nel codice di avvio, o se fallisce, la libreria deve arrangiarsi per trovare le risorse. Alcune librerie potrebbero provare a caricare risorse da file individuali, per esempio PIC1, PIC2, PIC3 e così via (nelle specifiche Blorb è dettagliato questo approccio). Altre librerie potrebbero non avere altri meccanismi di caricamento a disposizione; non saranno quindi disponibili le risorse se non attraverso un archivio Blorb.

12.3. Cosa fa la libreria

Ogni libreria deve implementare giblorb_set_resource_map() per supportare Blorb. In generale, questa funzione carica la mappa del file e se la tiene

da qualche parte. Potrebbe anche tenersi da parte lo stream, per leggere i dati direttamente.

giblorb_set_resource_map() ritornerà giblorb_err_None (0) in caso di successo, o un codice di errore appropriato in caso contrario. Vedere la sezione 12.5.

La libreria deve anche integrare il file gi_blorb.c. La maggior parte di questo file non dovrebbe presentare problemi sulla maggior parte delle piattaforme. Ha però necessità di allocare memoria. Nella sua forma originale gi_blorb.c utilizza le funzioni ANSI malloc(), realloc() e free(). Se questo dà problemi, le funzioni coinvolte sono isolate verso la fine del file.

124. Cosa fa lo strato Blorb

Queste funzioni sono implementate in gi_blorb.c. Faranno parte della libreria, ma saranno identiche in ogni piattaforma. In generale, solo la libreria avrà necessità di chiamarle. Il programma Glk deve lasciare alla libreria il compito di gestire le risorse.

```
giblorb_err_t giblorb_create_map(strid_t file,
    giblorb_map_t **newmap);
```

Legge i dati Blorb da uno stream Glk. Non carica ogni risorsa; piuttosto prepara una mappa in memoria per facilitare l'accesso alle risorse. In newmap viene inserito un puntatore alla mappa in memoria. Il puntatore è opaco; viene solo utilizzato dalle funzioni dello strato Blorb.

```
giblorb_err_t giblorb_destroy_map(giblorb_map_t *map);
```

De-alloca la mappa e tutta la memoria associata. Lo stream originale *non* viene chiuso.

```
#define giblorb_method_DontLoad (0)
#define giblorb_method_Memory (1)
#define giblorb_method_FilePos (2)
typedef struct giblorb_result_struct {
    glui32 chunknum;
    union {
        void *ptr;
        glui32 startpos;
    } data;
    glui32 length;
    glui32 chunktype;
} giblorb_result_t;

giblorb_err_t
  giblorb_load_chunk_by_type(giblorb_map_t *map,
    glui32 method, giblorb_result_t *res,
    glui32 chunktype, glui32 count);
```

Carica un chunk del tipo specificato. Il parametro count distingue fra chunk dello stesso tipo. Se count è zero viene caricato il primo chunk di quel tipo, e così via.

Per caricare un chunk di tipo IFF FORM (come AIFF) è necessario passare il tipo di form, non FORM.[1]

I dati ritornati sono messi in res, a seconda di method. Il campo chunknum viene impostato al numero del chunk (questo valore può essere utilizzato con giblorb_load_chunk_by_number() o giblorb_unload_chunk()). Il campo length conterrà la lunghezza del chunk in byte. Il campo chunktype sarà il tipo del chunk, ovviamente quello richiesto.

Se viene specificato giblorb_method_DontLoad non vengono caricati dati. È utile per sapere se un chunk esiste o il suo numero e lunghezza.

Con giblorb_method_FilePos, data.startpos viene valorizzato con la posizione del chunk all'interno del file. Sarà quindi in seguito possibile usare la funzione glk_stream_set_position() per leggere i dati direttamente dallo stream.

Se invece viene specificato giblorb_method_Memory, data.ptr punterà ad un blocco di memoria contenente i dati del chunk. La proprietà di questo blocco rimane della mappa. Se si carica il chunk più di una volta con questo metodo, lo strato Blorb è sufficientemente intelligente da tenerne in memoria

[1] Viene comunque introdotta una leggera ambiguità: non è possibile distinguere fra un chunk FORM AIFF e un chunk AIFF non di tipo FORM. D'altra parte il secondo caso è quasi certamente un errore.

solo una copia. *Non* bisogna de-allocare questa memoria; la funzione da utilizzare è giblorb_unload_chunk().

```
giblorb_err_t
  giblorb_load_chunk_by_number(giblorb_map_t *map,
    glui32 method, giblorb_result_t *res,
    glui32 chunknum);
```

È simile a giblorb_load_chunk_by_type() ma carica il chunk in base al suo numero. Il tipo del chunk sarà disponibile nel campo chunktype del risultato. Il numero del chunk è disponibile nel campo chunknum dopo aver caricato il chunk in altro modo.

```
giblorb_err_t giblorb_unload_chunk(giblorb_map_t *map,
    glui32 chunknum);
```

Libera la memoria allocata da giblorb_method_Memory. Se il chunk specificato non è mai stato caricato in memoria, non ha effetto.

```
giblorb_err_t giblorb_load_resource(giblorb_map_t *map,
    glui32 method, giblorb_result_t *res,
    glui32 usage, glui32 resnum);
```

Carica una risorsa in base al suo utilizzo e numero di risorsa. Al momento sono definiti giblorb_ID_Pict (immagini), giblorb_ID_Snd (suoni) e giblorb_ID_Exec (codice eseguibile). I tipi di dati ammessi per ciascun utilizzo sono dettagliati nelle specifiche Blorb.

Attenzione, il numero di risorsa non è il numero di chunk. Il numero di risorsa è il numero specificato dal programma Glk per l'immagine o il suono.

giblorb_load_resource() determina il numero di chunk di una risorsa; il risultato è in res.chunknum.

```
giblorb_err_t giblorb_count_resources(
    giblorb_map_t *map, glui32 usage,
    glui32 *num, glui32 *min,
    glui32 *max);
```

Conta il numero di chunk con l'utilizzo specificato (immagine, suono o eseguibile). Il numero totale di chunk con quell'usage è messo in num. Il più piccolo e il più grande numero di risorsa corrispondenti sono messi rispettivamente in min e max. Se un'informazione non è di interesse è possibile passare puntatori NULL.

12.5. Errori Blorb

Tutte le funzioni dello strato Blorb (anche `giblorb_set_resource_map()`) ritornano uno dei seguenti codici di stato.

giblorb_err_None (che vale zero). Nessun errore.

giblorb_err_CompileTime Qualcosa è andato storto nella compilazione dello strato Blorb.

giblorb_err_Alloc Non è stato possibile allocare memoria.

giblorb_err_Read Non è stato possibile leggere dal file.

giblorb_err_NotAMap Il parametro `map` non è valido.

giblorb_err_Format Il file Blorb è corrotto o non valido.

giblorb_err_NotFound I dati richiesti non sono stati trovati.

Parte II.

Glk e Inform/glulx

13 infglk

Per rendere perlomeno utilizzabile Glk in Inform esiste una libreria di comodo che permette di chiamare tutte le funzioni Glk per nome, anziché per numero (l'alternativa è usare l'opcode `glk` con valori numerici, cosa che viene fatta in tutta la libreria standard).

Attenzione però: infglk è solo un sottile strato di adattamento fra Inform e Glk. Non fa né più né meno che adattare la chiamata della funzione. In compenso, la documentazione delle funzioni C (le funzioni `glk_` descritte nella Parte I) è completamente applicabile alle funzioni infglk.

13.1. Come usare infglk

L'utilizzo è banale: basta scaricare il pacchetto e mettere nel path di Inform il file `infglk.h`. Con il file viene anche distribuito uno script che permette di rigenerarlo in modo automatico da `glk.h`. L'utilizzo di questo script *non* è banale e personalmente lo sconsiglio (dato che comunque attualmente infglk è allineato all'ultima release di Glk).

È possibile utilizzare senza alcun problema l'ultima release del file anche con interpreti che non supportano le funzioni della versione 0.7 di Glk: per come è progettata la libreria, finché una funzione non viene chiamata non può dare problemi (è quindi bene verificare che sia disponibile *prima* di chiamarla; vedere la sezione 1.7).

Il file `infglk.h` deve essere incluso come una qualunque estensione della libreria, senza precauzioni particolari (ovviamente dopo il parser e prima della grammatica, come di consueto).

13.2. Passare i parametri

Anche se infglk permette di chiamare a basso livello ogni funzione Glk (e con gli stessi parametri), è necessario prestare cautela nel passaggio dei valori, specialmente quelli per riferimento.

Sotto Inform/glulx *tutto* è un intero a 32 bit. Quindi stringhe, oggetti, array e quant'altro in fin dei conti sono visti come numeri interi. Per la precisione come *l'indirizzo* dell'oggetto corrispondente.

In particolar modo, nel caso di un array è l'indirizzo del primo elemento dell'array. Dato che gli elementi di un array sono di quattro byte (per un array di word, quelli dichiarati con -->; gli array di caratteri, di tipo -> hanno elementi di un singolo byte) per accedere all'elemento successivo occorre aggiungere 4 (il valore di WORDSIZE).

Per esempio, nell'array *ufficiale* gg_arguments gli indirizzi degli elementi sono (l'array può contenere 8 word):

```
gg_arguments              ! gg_arguments-->0
gg_arguments+4            ! gg_arguments-->1
gg_arguments+8            ! gg_arguments-->2
! eccetera
gg_arguments+WORDSIZE*i   ! gg_arguments-->i
```

In questo modo è possibile passare i parametri *per riferimento*. Rubando un esempio dalla documentazione:

```
glk_window_get_size(gg_mygraph_win,
    gg_arguments, gg_arguments+4);

mygraph_width  = gg_arguments-->0;
mygraph_height = gg_arguments-->1;
```

Qui l'array gg_arguments viene utilizzato per ricevere i valori di ritorno della funzione.

Come già accennato, quando Glk si aspetta un array di caratteri (ma non una stringa) si usa un array di tipo ->.

Per convenienza (e leggibilità!) sono definite come costanti Inform anche tutte le costanti Glk (selettori gestalt, tipi di evento, e altro).

I colori codificati in esadecimale sono facilmente inseribili con la sintassi $00123456. Altri esempi rubati dal manuale per chiarire questi ultimi due punti:

```
! Questo specifica bianco su nero
glk_stylehint_set(wintype_TextBuffer,
    style_Normal, stylehint_BackColor, $000000);
glk_stylehint_set(wintype_TextBuffer,
```

```
    style_Normal, stylehint_TextColor, $ffffff);

! Questo invece proporzionale allineato a sinistra
glk_stylehint_set(wintype_TextBuffer,
    style_Note, stylehint_Proportional, 1);
glk_stylehint_set(wintype_TextBuffer, style_Note,
    stylehint_just_FlushLeft, 0);
```

13.3. Stringhe e array

Alcune funzioni Glk utilizzano array (di byte o di word). Per questi, come si è già visto, basta utilizzare un normale array di Inform (di tipo "->" o "-->", a seconda della dimensione dell'elemento).

Per i più raffinati, Inform 6.30 mette a disposizione un *terzo* tipo di array, dichiarato con `buffer`. È un array di byte prefissato da una word che ne indica la lunghezza. Nulla di spettacolare, rende solo più comodo l'utilizzo in certe circostanze (la libreria standard, per esempio, lo utilizza per l'input del parser).

Quando invece le specifiche Glk si attendono una *stringa* (e non un array di caratteri, riconoscibile dal fatto che è *sempre* seguito dalla sua lunghezza) l'interfaccia glulx impone il passaggio di una *stringa glulx non codificata*.

La versione corta della storia è che deve essere una stringa formata da soli caratteri, non compressa, e marchiata come tale.[1]

Le normali stringhe fra apici (`"sono una stringa"`) sono stringhe glulx ma probabilmente compresse dal compilatore. Non sono quindi adatte all'uso con Glk.

Come deve essere formata quindi una stringa?

Per prepararsi una stringa da passare ad una funzione Glk che si attende una stringa è necessario appoggiarsi su un array di caratteri (il che è comunque la prassi per tutte le operazioni su stringa in Inform). Questo array deve contenere, nell'ordine. a partire dall'elemento `->0`:

- Il marcatore che identifica la stringa come tale. Il numero magico è `0xE0`. I programmatori d'avanguardia che volessero tentare l'utilizzo delle varianti Unicode sappiano fin da subito che in tal caso l'identificativo da utilizzare è `0xE2`.

[1] La versione lunga della storia impone la spiegazione delle tabelle di codifica, dei nodi di compressione e di altre cose che il compilatore fa in modo automatico. Basti sapere che *non si può* passare una normale stringa fra virgolette.

- I caratteri che formano la stringa. Con codifica Latin-1 (il che non dovrebbe causare problemi visto che *tutto* in Inform/glulx è codificato in Latin-1; ma è meglio precisare).
- Un terminatore, il carattere NUL. Ovvero uno zero. Ovviamente in questo modo un NUL non potrà mai far parte della stringa, ma poco male (come carattere non esiste).

Per semplificare la manovra suddetta, la libreria standard di Inform fornisce una funzione, ChangeAnyToCString. Nonostante il nome della funzione la stringa preparata è perfettamente compatibile con le chiamate Glk. Attenzione però, il buffer di destinazione è limitato a 64 caratteri! Per ulteriori informazioni vedere la sezione 14.3.

134. Attenzione allo zero

Un numero speciale, molto importante, è il NULL (indica *nessun oggetto*, in generale; o anche un problema nella creazione di un oggetto). Sfortunatamente Inform ha un concetto diverso di NULL; il suo valore in questo contesto è -1. Nel file infglk.h viene definito GLK_NULL come 0. È importante non confonderli! (l'uso di NULL$+1$ non mi sembra allettante, anche se lecito)

L'ultima particolarità (più di comodo che altro): grazie alla sintassi di Inform, è possibile omettere gli ultimi argomenti se sono zero (ovvero, in sostituzione a GLK_NULL).

13.5. Prontuario delle funzioni Glk

In questa sezione sono elencate tutte le funzioni che possono essere di qualche uso in Inform/glulx. Sono quindi state omesse quelle funzioni *interne* di uso limitato per l'autore (sono utilizzate solo dall'interprete, in sostanza). Per una descrizione dettagliata, consultare la relativa sezione nelle specifiche principali (quelle relative al C, dato che la chiamata è sostanzialmente *identica* come già descritto).

13.5.1. Funzioni di uso generico

```
glk_gestalt(sel, val);
glk_gestalt(sel, val, arr, arrref);
```

Determina le funzionalità disponibili sull'implementazione in uso della libreria Glk (sezione 1.7).

```
glk_char_to_lower(ch);
```

Trasforma un carattere nel corrispondente minuscolo (sezione 2.5).

```
glk_char_to_upper(ch);
```

Trasforma un carattere nel corrispondente maiuscolo (sezione 2.5).

```
glk_buffer_to_lower_case_uni(buf, len, numchars);
```

Trasforma una stringa Unicode in minuscolo (sezione 2.5).

```
glk_buffer_to_upper_case_uni(buf, len, numchars);
```

Trasforma una stringa Unicode in maiuscolo (sezione 2.5).

```
glk_buffer_to_title_case_uni(buf, len, numchars,
    lowerrest);
```

Trasforma in maiuscolo il primo carattere di una stringa Unicode, eventualmente forzando in minuscolo tutti i caratteri seguenti (sezione 2.5).

13.5.2. *Funzioni relative alle finestre*

```
glk_window_get_root();
```

Restituisce la finestra radice, se esistente (sezione 3.7).

```
glk_window_open(split, method, size, wintype, rock);
```

Apre una nuova finestra (eventualmente dividendone un'altra) (sezione 3.2).

```
glk_window_close(win, result);
```

Chiude una finestra (sezione 3.2).

```
glk_window_get_size(win, widthptr, heightptr);
```

Ricava le dimensioni effettive di una finestra (sezione 3.3).

```
glk_window_set_arrangement(win, method, size, keywin);
```

Cambia le dimensioni di una finestra (sezione 3.3).

```
glk_window_set_arrangement(win, method,
    size, keywin);
```

Cambia le dimensioni di una finestra (sezione 3.3).

```
glk_window_get_arrangement(win, methodptr,
    sizeptr, keywinptr);
```

Cambia le dimensioni di una finestra (sezione 3.3).

```
glk_window_iterate(win, rockptr);
```

Itera su tutte le finestre aperte (sezione 3.7).

```
glk_window_get_rock(win);
```

Ritorna il valore sotto il sasso di una finestra (sezione 3.7).

```
glk_window_get_type(win);
```

Ritorna il tipo di una finestra (sezione 3.7).

```
glk_window_get_parent(win);
```

Ritorna la finestra padre di una finestra (sezione 3.7).

```
glk_window_clear(win);
```

Ripulisce la finestra dal suo contenuto (sezione 3.7).

```
glk_window_move_cursor(win, xpos, ypos);
```

Sposta la posizione del cursore in una griglia di testo (sottosezione 3.5.4).

```
glk_window_get_stream(win);
```

Ottiene lo stream associato ad una finestra (sezione 3.7).

```
glk_window_set_echo_stream(win, str);
```

Imposta lo stream di eco di una finestra (sezione 3.6).

```
glk_window_get_echo_stream(win, str);
```

Ottiene lo stream di eco associato ad una finestra (sezione 3.6).

```
glk_set_window(win);
```

Seleziona lo stream della finestra come corrente (sezione 3.7).

13.5.3. Funzioni relative agli stream

```
glk_stream_open_file(fileref, fmode, rock);
```

Apre uno stream da file identificato da una fileref (sottosezione 5.5.4).

```
glk_stream_open_file_uni(fileref, fmode, rock);
```

Apre uno stream da file identificato da una fileref in modalità Unicode (sottosezione 5.5.4).

```
glk_stream_open_memory(buf, buflen, fmode, rock);
```

Apre uno stream in memoria (sottosottosezione 5.5.3).

```
glk_stream_open_memory_uni(buf, buflen, fmode, rock);
```

Apre uno stream in memoria in modalità Unicode (sottosottosezione 5.5.3).

```
glk_stream_close(str, result);
```

Chiude uno stream (sezione 5.3).

```
glk_stream_iterate(str, rockptr);
```

Itera sugli stream aperti (sezione 5.6).

```
glk_stream_get_rock(str);
```

Ritorna il valore sotto il sasso di uno stream (sezione 5.6).

```
glk_stream_set_position(str, pos, seekmode);
```

Sposta la posizione del puntatore di lettura/scrittura dello stream (sezione 5.4).

```
glk_stream_get_position(str);
```

Ritorna la posizione del puntatore di lettura/scrittura dello stream (sezione 5.4).

```
glk_stream_set_current(str);
```

Seleziona lo stream corrente (Capitolo 5).

```
glk_stream_get_current();
```

Ritorna lo stream corrente (Capitolo 5).

```
glk_put_char(ch);
```

Scrive un carattere sullo stream corrente (sezione 5.1).

```
glk_put_char_uni(ch);
```

Scrive un carattere Unicode sullo stream corrente (sezione 5.1).

```
glk_put_char_stream(str, ch);
```

Scrive un carattere su uno stream (sezione 5.1).

```
glk_put_char_stream_uni(str, ch);
```

Scrive un carattere Unicode su uno stream (sezione 5.1).

```
glk_put_string(s);
```

Scrive una stringa sullo stream corrente (sezione 5.1).

```
glk_put_string_uni(s);
```

Scrive una stringa Unicode sullo stream corrente (sezione 5.1).

```
glk_put_string_stream(str, s);
```

Scrive una stringa su uno stream (sezione 5.1).

```
glk_put_string_stream_uni(str, s);
```

Scrive una stringa Unicode su uno stream (sezione 5.1).

```
glk_put_buffer(buf, len);
```

Scrive un array di caratteri sullo stream di default (sezione 5.1).

```
glk_put_buffer_uni(buf, len);
```

Scrive un array di caratteri Unicode sullo stream di default (sezione 5.1).

```
glk_put_buffer_stream(str, buf, len);
```

Scrive un array di caratteri su uno stream (sezione 5.1).

```
glk_put_buffer_stream_uni(str, buf, len);
```

Scrive un array di caratteri Unicode su uno stream (sezione 5.1).

```
glk_set_style(styl);
```

Seleziona lo stile dello stream corrente (sezione 5.5).

```
glk_set_style_stream(str, styl);
```

Seleziona lo stile di uno stream (sezione 5.5).

```
glk_get_char_stream(str);
```

Legge un carattere da uno stream (sezione 5.2).

```
glk_get_char_stream_uni(str);
```

Legge un carattere Unicode da uno stream (sezione 5.2).

```
glk_get_line_stream(str, buf, len);
```

Legge una riga (fino al newline) da uno stream (sezione 5.2).

```
glk_get_line_stream_uni(str, buf, len);
```

Legge una riga Unicode da uno stream (sezione 5.2).

```
glk_get_buffer_stream(str, buf, len);
```

Legge un array di caratteri da uno stream (sezione 5.2).

```
glk_get_buffer_stream_uni(str, buf, len);
```

Legge un array di caratteri Unicode da uno stream (sezione 5.2).

```
glk_stylehint_set(wintype, styl, hint, val);
```

Imposta un suggerimento per uno stile (sottosezione 5.5.1).

```
glk_stylehint_clear(wintype, styl, hint);
```

Rimuove un suggerimento per uno stile (sottosezione 5.5.1).

```
glk_style_distinguish(win, styl1, styl2);
```

Determina se due stili hanno aspetto differente (sottosezione 5.5.2).

```
glk_style_measure(win, styl, hint, result);
```

Determina l'apparenza di uno stile (sottosezione 5.5.2).

```
glk_fileref_create_temp(usage, rock);
```

Crea una fileref per un file temporaneo (sezione 6.1).

```
glk_fileref_create_by_name(usage, name, rock);
```

Crea una fileref con il nome specificato (sezione 6.1).

```
glk_fileref_create_by_prompt(usage, fmode, rock);
```

Crea una fileref chiedendo il nome al giocatore (sezione 6.1).

```
glk_fileref_create_from_fileref(usage, fref, rock);
```

Crea una fileref duplicandone un'altra (sezione 6.1).

```
glk_fileref_destroy(fref);
```

Distrugge una fileref (sezione 6.2).

```
glk_fileref_iterate(fref, rockptr);
```

Itera su tutte le fileref presenti (sezione 6.2).

```
glk_fileref_get_rock(fref);
```

Ritorna il valore sotto il sasso di una fileref (sezione 6.2).

```
glk_fileref_delete_file(fref);
```

Cancella il file identificato da una fileref (sezione 6.2).

```
glk_fileref_does_file_exist(fref);
```

Determina se il file specificato da una fileref esiste (sezione 6.2).

13.54. Funzioni relative agli eventi

```
glk_select(event);
```

Aspetta un evento (Capitolo 4).

```
glk_select_poll(event);
```

Verifica la presenza di un evento generato internamente (Capitolo 4).

```
glk_request_timer_events(millisecs);
```

Richiede la generazione di eventi periodici (sezione 4.4).

```
glk_request_line_event(win, buf, maxlen, initlen);
```

Richiede input di linea in una finestra (sezione 4.2).

```
glk_request_line_event_uni(win, buf, maxlen, initlen);
```

Richiede input di linea Unicode in una finestra (sezione 4.2).

```
glk_request_char_event(win);
```

Richiede input a carattere in una finestra (sezione 4.1).

```
glk_request_char_event_uni(win);
```

Richiede input a carattere Unicode in una finestra (sezione 4.1).

```
glk_request_mouse_event(win);
```

Richiede eventi del mouse in una finestra (sezione 4.3).

```
glk_cancel_line_event(win, event);
```

Annulla la richiesta di input a linea in una finestra (sezione 4.2).

```
glk_request_char_event(win);
```

Annulla la richiesta di input a carattere in una finestra (sezione 4.1).

```
glk_request_mouse_event(win);
```

Annulla la richiesta di eventi del mouse in una finestra (sezione 4.3).

13.5.5. Funzioni grafiche

```
glk_image_draw(win, image, val1, val2);
```

Disegna un'immagine in una finestra (sezione 7.1).

```
glk_image_draw_scaled(win, image, val1, val2,
    width, height);
```

Disegna un'immagine ridimensionata in una finestra (sezione 7.1).

```
glk_image_get_info(image, widthptr, heightptr);
```

Ritorna le dimensioni di un'immagine (sezione 7.1).

```
glk_window_flow_break(win);
```

Avanza la posizione del testo fino alla fine delle immagini a margine (sezione 7.1).

```
glk_window_erase_rect(win, left, top, width, height);
```

Cancella un rettangolo in una finestra grafica (sezione 7.2).

```
glk_window_fill_rect(win, color,
    left, top, width, height);
```

Riempie un rettangolo in una finestra grafica (sezione 7.2).

```
glk_window_set_background_color(win, color);
```

Imposta il colore di sfondo di una finestra grafica (sezione 7.2).

13.5.6. Funzioni audio

```
glk_schannel_create(rock);
```

Crea un canale audio (sezione 8.2).

```
glk_schannel_destroy(chan);
```

Chiude un canale audio (sezione 8.2).

```
glk_schannel_iterate(chan, rockptr);
```

Itera sui canali audio aperti (sezione 8.4).

```
glk_schannel_get_rock(chan);
```

Ritorna il valore sotto al sasso di un canale audio (sezione 8.4).

```
glk_schannel_play(chan, snd);
```

Avvia la riproduzione di una risorsa audio (sezione 8.3).

```
glk_schannel_play_ext(chan, snd, repeats, notify);
```

Avvia la riproduzione di una risorsa audio con opzioni aggiuntive (sezione 8.3).

```
glk_schannel_stop(chan);
```

Arresta la riproduzione su un canale audio (sezione 8.3).

```
glk_schannel_set_volume(chan, vol);
```

Imposta il volume di un canale audio (sezione 8.3).

```
glk_schannel_load_hint(snd, flag);
```

Pre-carica una risorsa audio (sezione 8.3).

13.5.7. Funzioni relative ai collegamenti

```
glk_set_hyperlink(linkval);
```

Setta il valore di collegamento dello stream corrente (sezione 9.1).

```
glk_set_hyperlink_stream(str, linkval);
```

Setta il valore di collegamento di uno stream (sezione 9.1).

```
glk_request_hyperlink_event(win);
```

Richiede eventi di collegamento su una finestra (sezione 9.2).

```
glk_cancel_hyperlink_event(win);
```

Annulla la richiesti di eventi di collegamento su una finestra (sezione 9.2).

14

La libreria standard e Glk

14.1. Quel che è nuovo e quel che manca

Inform (e la libreria standard) si comportano in maniera *quasi* uguale quando utilizzati per compilare codice glulx piuttosto che Z-code. Il *quasi* implica che alcune cose sono nuove, altre non ci sono più e altre ancora sono variate (leggermente, per fortuna). Queste sono problematiche relative a glulx, non a Glk, ma visto che vanno a braccetto un ripasso non può che far bene. In particolare, in nessun ordine preciso:

- La Z-machine è a 8/16 bit; glulx è a 32 bit. A parte l'evidente miglioria nei numeri rappresentabili (da 65536 a, ehm, 4 miliardi e rotti) anche la dimensione della word cambia.
- Anche se gli array --> contengono interi a 32 bit, gli array -> continuano ad essere con valori di soli 8 bit. Come questo si concilierà con l'Unicode sarà da vedere.[1]
- Per lo stesso motivo l'operatore .# utilizzato su una proprietà necessità di particolare attenzione: se prima si divideva per 2 per ottenere il numero di elementi, ora bisogna dividere per 4 (l'operatore .# restituisce il numero di *byte*). Ancora meglio, è definita una costante WORDSIZE che assume il valore corretto a seconda del tipo di compilazione.
- L'assembler della Z-machine non funziona (ovviamente). L'assembler glulx è completamente diverso, oltretutto.

[1] Ci sono patch per Inform per il supporto Unicode completo, ma non ho ancora indagato su di esse.

- Il modello di I/O è completamente diverso (e qui entra in gioco Glk). In particolare gli statement `style` e `read` non esistono proprio.
- Per comodità Graham Nelson ha lasciato disponibili `font on` e `font off` che passano rispettivamente agli stili Preformatted e Normal. Ma tanto vale usare *correttamente* gli stili di Glk.
- Esiste un nuovo modo per passare parametri alle funzioni. Ma se non vi interessa metter mano all'assembler glulx è meglio non pensarci.
- Le variabili di stampa (quelle cose strane che nessuno usa mai, `@00` fino a `@31`) sono raddoppiate, quindi sono disponibili fino a `@63`. Ma non solo, ci sono anche altre novità (vedere la sezione 14.2).
- Esistono funzioni nuove dedicate a Glk, e altre di uso generale. In glulx 0.3 sembra sia addirittura disponibile qualche forma di allocazione dinamica della memoria.
- In particolare, *tutto* l'accesso a Glk è gestito con un solo opcode, `@glk` o il suo equivalente funzionale `glk`, per i casi più semplici (infglk permette di non doversi curare di questa funzione);
- I formati di stringhe e oggetti in memoria e i loro riferimenti sono cambiati completamente. Così come il meccanismo di *save/restore*, *restart* e *undo*. In particolare gli oggetti non sono più numerati consecutivamente. Ma sono questioni squisitamente tecniche relative a glulx di nessuna importanza per i normali autori. Ai fini dell'utilizzo di Glk *tutti* gli oggetti in glulx sono rappresentati da interi a 32 bit. In memoria il primo byte rappresenta il tipo di oggetto (Tabella 14.1). Le funzioni standard `metaclass` e `ZRegion` tengono conto automaticamente di questo fatto, e ritornano i valori tradizionali.
- Anche alcune routine del parser si appoggiano per motivi di efficienza a opcode speciali di glulx (in particolar modo l'accesso al dizionario).

14.2. Migliorie alle stringhe di stampa

Questo con Glk non c'entra proprio nulla... ma tant'è!

Come è noto (o almeno spero, visto che si tratta di una funzionalità impiegata di rado), in Inform è possibile scrivere

```
print "Qualunque␣cosa␣sia␣@03";
```

Tabella 14.1. Byte identificativi degli oggetti standard glulx

ID (esadecimale)	Tipo di oggetto
00	Nessun oggetto
01–3F	Disponibili per il programmatore
40–5F	*Riservato alla libreria Inform*
60	Parole del dizionario
61–6F	*Riservato alla libreria Inform*
70	Classi e oggetti
71–7F	*Riservato alla libreria Inform*
80–BF	*Riservato alla* VM
C0	Funzioni con argomenti su stack
C1	Funzioni con argomenti locali
C2–DF	*Riservato per futuri tipi di funzione*
E0	Stringhe terminate da NUL
E1	Stringhe compresse
E2	Stringhe Unicode
E3–FF	*Riservato per futuri tipi di stringa*

...e durante l'esecuzione il contenuto della variabile stringa @03 viene mostrato (le variabili stringa sono completamente distinte dalle comuni variabili, locali o globali che siano; le variabili stringa sono sempre globali).

Il vantaggio è che in qualunque momento è possibile utilizzare il comando

```
string 3 "quell'oggetto";
```

...per modificare il testo in *tutte* le espansioni di @03. Nella Z-machine[2] sono disponibili 32 di queste stringhe.

Sotto glulx (visto che tutto è più grande), la disponibilità aumenta. Ora ci sono 64 stringhe a disposizione. Ma non solo.

Per prima cosa è possibile espandere una stringa dentro un'altra stringa:

```
string 1 "schifoso";
string 3 "quell'oggetto␣@01";
print "Qualunque␣cosa␣sia␣@03";
```

Anche se comunque è vietato creare una ricorsione infinita, direttamente o indirettamente (quindi niente cose del tipo)

```
string 1 "io␣sono␣@01" !!! SBAGLIATO
```

Poi, la grande novità, è possibile usare *una funzione* al posto della stringa. Al posto di visualizzare la stringa, Inform chiama la funzione specificata.

[2] Non proprio nella Z-machine. Le variabili stringa sono implementate da Inform, non nella Z-machine in se.

Al suo interno la funzione può fare sostanzialmente quel che vuole. La cosa ovvia è stampare il testo sostitutivo, ma possono esserci anche altri utilizzi più... contorti (per esempio una funzione che tiene traccia di quante volte il giocatore legge *qualcosa* e il *qualcosa* è disperso in molti punti; certo *esistono* modi migliori, ma quasi tutti gli utilizzi delle variabili stringa (se non tutti) sono sostituibili da codice scritto ad hoc).

Alla funzione non vengono passato parametri. Inoltre il valore di ritorno viene ignorato.

14.3. Migliorie a print_to_array

Anche questa funzione con Glk non ha una grande attinenza. Può però essere una valida sostituta di uno stream in memoria (vedere la sottosottosezione 5.5.3) nei casi più semplici.

Il metodo `print_to_array` di Inform travasa una stringa in un array. Il Designer Manual spiega tutto nei dettagli.

Cosa cambia: intanto la dimensione della stringa viene messa nei primi *quattro* byte dell'array (e non solo nei primi due), per gli ovvi motivi. Inoltre, sotto glulx, è possibile specificare la dimensione *complessiva* dell'array.

In sostanza, chiamando

```
len = str.print_to_array(arr, 80);
```

non saranno inseriti più di 76 caratteri. Il secondo elemento deve essere necessariamente ≥ 4. Nonostante il limite di memorizzazione, il valore ritornato e quello messo nei primi 4 byte dell'array indicano il numero *totale* dei caratteri, non solo di quelli memorizzati.[3]

La documentazione cita l'idioma (un po' stupido, a dire il vero)

```
len = str.print_to_array(arr, 4);
```

... per ricavare la lunghezza di una stringa utilizzando un array di 4 byte.

[3] Questo è *esattamente* lo stesso funzionamento di uno stream in memoria. E della ridirezione dell'I/O di glulx, se può interessare

144. Stampare qualunque cosa

Simile a print_to_array è la funzione PrintAnything (un'altra novità che non c'entra nulla con Glk, almeno in apparenza...):

```
PrintAnything();                    ! (1)
PrintAnything(0);                   ! (2)
PrintAnything("string");            ! (3)
PrintAnything('word');              ! (4)
PrintAnything(obj);                 ! (5)
PrintAnything(obj, prop);           ! (6)
PrintAnything(obj, prop, args...);  ! (7)
PrintAnything(func);                ! (8)
PrintAnything(func, args...);       ! (9)
```

Tutte le forme di questa funzione permettono di visualizzare qualunque cosa sia presente all'interno della macchina virtuale.

Le forme (1) e (2) sono per uniformità. Non stampano nulla.

La forma (3) stampa una stringa, la (4) una parola di dizionario (come sulla Z-machine la codifica interna è diversa).

Le forme (5), (6) e (7) permettono di visualizzare il valore di una proprietà di un oggetto (viene usata la name se non specificato altrimenti, eventuali altri parametri sono passati alla proprietà). Per finire, le forme (8) e (9) semplicemente richiamano una funzione (passando eventualmente argomenti). Il valore di ritorno *non* viene utilizzato, le funzioni sono responsabili della stampa.

Dato che sotto glulx *tutto* è un intero, per mostrare *davvero un intero* bisogna utilizzare una funzione di comodo.

```
DecimalNumber(num);
```

Semplicemente stampa il numero. Può quindi essere applicata la forma (9) della chiamata per ottenere l'effetto desiderato:

```
PrintAnything(DecimalNumber, num);
```

In modo simile a print_to_array c'è la possibilità di ridirigere il risultato su un array.

```
PrintAnyToArray(arry, arraylen, obj, ...);
```

A differenza di quella funzione l'array conterrà *solo* i caratteri (e non la lunghezza risultante). Il risultato viene troncato alla lunghezza massima specificata. Il valore ritornato è la lunghezza complessiva (indipendentemente dal fatto che la stringa sia stata troncata).

È legale innestare le chiamate a PrintAnyToArray. È legale anche avere array a zero (solo se anche arraylen è zero). In questo caso viene solo calcolata la lunghezza della stringa risultante.

Incidentalmente, la funzione è implementata con l'ausilio di uno stream in memoria. Ed è anche un ottimo esempio di quanto sia da masochisti utilizzare Glk (e le funzioni vararg) senza l'ausilio di un qualche wrapper (come infglk).

Esiste anche, definita nella libreria standard ma misteriosamente inutilizzata né tanto meno documentata, una funzione ChangeAnyToCString e un array AnyToStrArr:

```
Constant GG_ANYTOSTRING_LEN 66;
Array AnyToStrArr -> GG_ANYTOSTRING_LEN+1;

[ ChangeAnyToCString _vararg_count ix len;
! ...
```

Chiama PrintAnyToArray per creare una stringa NUL-terminata, prefissata da 0xE0 (è uno dei tipi di stringa predefiniti in glulx). La capacità è ovviamente di 64 caratteri (66 meno il tag e il NUL finale).

L'utilizzo principale di questa funzione è la preparazione di stringhe da passare a Glk: Glk fa distinzione fra array di caratteri e stringhe; le stringhe sono prefissate da un codice identificativo (0xE0 nel caso più semplice) e terminate da un carattere NUL (come le stringhe in C, da cui il nome della funzione). Il Capitolo 13 da' spiegazioni aggiuntive al riguardo.

14.5. Le variabili globali dedicate a Glk

La libreria standard per la gestione di Glk utilizza alcune variabili globali. Di queste, la maggior parte sono di interesse per l'autore che intenda personalizzare il comportamento del gioco.

gg_event è una specie di struttura. I quattro elementi (word, quindi indicizzata con -->) di questo array contengono i valori della struttura evento Glk, e, per la precisione, i valori relativi all'ultimo evento ritornato da una select (la libreria standard non effettua mai poll).

Ha lo stesso formato del primo parametro passato a HandleGlkEvent, mostrato in Tabella 14.3 (anzi, questa variabile *è passata* come primo parametro). In ogni caso è un buon posto per appoggiarsi nel caso si debbano fare poll o select nel proprio codice.

gg_arguments è il buffer "ufficiale" da passare alle funzioni Glk. Ha una capacità di 8 word (sul compilatore Inform 6.30 viene usata la nuova dichiarazione buffer, ma il succo è lo stesso—un array buffer ha nei primi quattro

byte (quindi in `buf-->0`) la lunghezza, e al seguito i singoli byte—è solo una dichiarazione di comodo).

Il contenuto dell'array riflette quello per cui è stato usato l'ultima volta: in sostanza, è solo una variabile di appoggio e non si sa mai quel che rimane al suo interno. Viene però usato anche con infglk (Capitolo 13), quindi è bene sapere che c'è.

Esiste poi tutta una serie di variabili per contenere i riferimenti agli oggetti opachi (solo finestre, stream e fileref: la libreria standard non ha supporto audio). Di queste alcune sono vitali per la personalizzazione (vedere la sezione 14.6), altre invece sono solo di servizio (per esempio il fileref per il file di script, gli stream di record/replay e lo stream di save/restore).

L'ultima variabile globale che può essere di un certo interesse (almeno per quel che riguarda Glk) è `gg_statuswin_size`. Questa variabile deve essere settata durante la prima chiamata a `InitGlkWindow` (vedere sezione 14.9) per richiedere un certo numero di righe nella barra di status con il layout standard (la finestra di stato è una griglia di testo, come nella Z-machine).

14.6. Sotto ai sassi

L'intera filosofia di Glk sotto Inform è permeata dal concetto di *mettere numeri sotto ai sassi*. Anzi, con lo strato di dispatch qualche oggetto ha *due* sassi a disposizione (non proprio, ma è per rendere l'idea).

Ma, in sostanza, a cosa servono?

Il problema è di ordine tecnico/pratico...

In due parole, sotto Inform/glulx servono per continuare a riconoscere le finestre (principalmente, ma il discorso vale anche per le altre classi) quando si ricarica la partita!

Da dove viene il problema? Il riferimento all'oggetto creato dalle funzioni nella stessa famiglia di `glk_window_open()` e compagnia è, sostanzialmente, un indirizzo in memoria della struttura Glk corrispondente. Per poterlo utilizzare in seguito questo riferimento sarà salvato, molto probabilmente in una variabile globale (o una proprietà).

Cosa succede se a questo punto il gioco viene ripristinato ad uno stato precedente (con una restore o una undo, per esempio)? Lo stato di glulx (quello del gioco) viene ripristinato, ma lo stato di Glk *rimane immutato*.

Quello che il gioco è costretto a fare, a questo punto, è enumerare tutte le risorse Glk che si ritrova (con le funzioni iterate, vedere la sottosezione 1.6.2) e cercare di rimettere ordine alla situazione.

È a questo punto che entrano in gioco i valori nascosti sotto ai sassi... dato che le funzioni di enumerazione *non danno* alcuna garanzia sull'ordine degli oggetti ritornati, la libreria standard di Inform prende la saggia precauzione

di *etichettare* ogni singolo oggetto in modo da poterlo riconoscere in seguito, dopo l'attacco di amnesia causato da un ripristino. Questa *etichetta* viene, per l'appunto, messa sotto il sasso dell'oggetto.

Per essere precisi, la libreria standard (in `parserm.h`, dove avvengono tutte le cose *veramente* interessanti della libreria) utilizza le seguenti costanti (per il layout di default, quello simile alla Z-machine):

`GG_MAINWIN_ROCK` Per la finestra di testo principale (un buffer di testo).

`GG_STATUSWIN_ROCK` Per la finestra di stato (una griglia di testo).

`GG_QUOTEWIN_ROCK` La finestra per emulare il vecchio op-code `box` (viene usata nella sola routine `Box___Routine` chiamata direttamente dal veener[4] di Inform; in sostanza è una funzione super-interna di Inform!)

`GG_SAVESTR_ROCK` Lo stream per salvare/ripristinare il gioco (usato nella verblib negli ovvi verbi).

`GG_SCRIPTSTR_ROCK` Lo stream per la trascrizione del gioco (quindi verbi ScriptOn e ScriptOff).

`GG_SCRIPTFREF_ROCK` Il fileref del file di trascrizione (per poterlo riaprire senza chiedere il nome un'altra volta, verbo ScriptOn).

`GG_COMMANDWSTR_ROCK` Uno stream scritto dal debugger per salvare i comandi digitati (verbo CommandsOn).

`GG_COMMANDRSTR_ROCK` Uno stream letto dal debugger durante il replay (verbo CommandsRead).

Ora che è ben chiaro a cosa servono i sassi sotto Inform/glulx... possiamo dimenticarcene! Almeno per il momento, per quello che riguarda il funzionamento con il layout di default.

Quando si cominciano a creare oggetti Glk personali è bene scegliere il valore da mettere sotto ai sassi in modo appropriato, per evitare di andare in conflitto con gli oggetti predefiniti. Nonostante questo valore possa essere un intero *qualsiasi* (di 32 bit, cioè circa più o meno due miliardi), esiste una convenzione mostrata in Tabella 14.2. Anche se non è obbligatorio, può essere un buon punto di partenza per decidere il valore da utilizzare (non viene detto nulla a riguardo dei canali audio, dato che la libreria non ne utilizza; per estensione il range 510–599 sembra appropriato).

[4] Chiamasi veener un insieme di routine di supporto presenti all'interno di Inform ed emesse in *ogni* file compilato. Si occupano principalmente del modello ad oggetti e di funzioni di utilità varia, ma *non* sono accessibili dal codice utente, se non indirettamente

Tabella 14.2. Valori convenzionali da mettere sotto ai sassi in Inform

Valore	Utilizzo
201	Finestra principale (GG_MAINWIN_ROCK)
202	Finestra di stato (GG_STATUSWIN_ROCK)
203	Finestra per citazioni (box) (GG_QUOTEWIN_ROCK)
204–209	*Riservato per la libreria*
210–299	Finestre personalizzate
301	Stream di save/restore (GG_SAVESTR_ROCK)
302	Stream di trascrizione (GG_SCRIPTSTR_ROCK)
303	Stream di record (GG_COMMANDWSTR_ROCK)
304	Stream di replay (GG_COMMANDRSTR_ROCK
305–309	*Riservato per la libreria*
310–399	Stream personalizzati
401	Fileref di trascrizione (GG_SCRIPTFREF_ROCK)
402–409	*Riservato per la libreria*
410–499	Fileref personalizzati
510–599	Canali audio (per estensione)

Le variabili della libreria standard dove vengono salvati i riferimenti agli oggetti (i _ROCK sono costanti!) sono le corrispondenti variabili globali gg_:

gg_mainwin, è la più importante in assoluto, gg_statuswin è interessante ai fini della personalizzazione dell'interfaccia (la linea di status è una delle modifiche più semplici e risale alla Z-machine), gg_quotewin, gg_scriptfref, gg_scriptstr, gg_savestr, gg_commandstr sono gli *ovvi* riferimenti.

Di queste l'unica *veramente* essenziale per la libreria è gg_mainwin. Se questa variabile viene trovata a 0 la libreria termina l'esecuzione con un quit. Ovviamente, l'unica circostanza in cui può accadere è attraverso l'implementazione di InitGlkWindow (vedere la sezione 14.9) per un layout di finestre personalizzato.

Tutto il lavoro di *riassociazione* delle variabili agli oggetti corrispondenti (semplicemente guardando sotto a tutti i sassi che trova!) viene effettuato dalla libreria dentro a GGRecoverObjects. Tale funzione è chiamata, prevedibilmente, durante l'inizializzazione/*restart* (il programma non è in grado di distinguerle facilmente!), l'*undo* e il *restore* dello stato di gioco.

La parte più interessante di questa routine per l'autore è la chiamata alla funzione IdentifyGlkObject. Questa funzione *non* fa parte della libreria di Inform, anzi è proprio uno stub[5] Il suo scopo è di consentire all'autore di

[5] Per ricordare: uno stub è una funzione dichiarata (con Stub per l'appunto) ma non definita. Se non viene definita in seguito la chiamata non ha effetto.

dire la sua durante il processo di identificazione degli oggetti Glk, durante l'esecuzione di GGRecoverObjects.

La funzione IdentifyGlkObject (se definita dall'autore) viene chiamate in tre fasi distinte:

- Per prima cosa viene chiamata con un solo parametro, 0. Questa chiamata segnala l'inizio delle operazioni di enumerazione degli oggetti Glk (fase 0).

```
IdentifyGlkObject(0);
```

- Durante l'enumerazione propriamente detta, viene chiamata con quattro parametri.

```
IdentifyGlkObject(1, type, id, rock);
```

  La costante "1" identifica questa modalità di operazione. Il parametro type indica la classe dell'oggetto (0 per una finestra, 1 per uno stream, 2 per un fileref); il campo id è il riferimento propriamente detto mentre rock è il valore che consentirà di identificare l'oggetto.

  Un lettore attento avrà sicuramente notato che *non* vi è traccia della quarta classe di oggetti, i canali audio. Molto semplicemente la libreria standard non li gestisce. Ma è un problema facilmente risolvibile nella terza fase...

- Alla fine, immediatamente prima di ritornare, GGRecoverObjects chiama un'ultima volta la routine IdentifyGlkObject:

```
IdentifyGlkObject(2);
```

  Il "2" segnala che è l'ultima chiamata prima della conclusione dell'enumerazione (fase 2). *Questo* è un ottimo momento per enumerare i canali audio o per fare qualsivoglia pulizia o reinizializzazione necessaria.

14.7. Iterare sugli oggetti

Il grosso del lavoro riguardante l'iterazione è già stato trattato nella sezione 14.6.

Esiste però un'altra routine degna di menzione, potenzialmente interessante in fase di sviluppo e debug: GlkListSub (in verblibm.h). Molto semplicemente elenca nella finestra principale tutte le risorse Glk attive. Fatto curioso: a differenza della indispensabile GGRecoverObjects, vengono elencati anche i canali audio.

È anche un ottimo esempio per capire come funzionano le funzioni di iterazione.

14.8. Verificare le funzionalità di Glk

Come già più volte specificato, le librerie Glk non sono tutte uguali. Alcune potrebbero supportare la grafica, altre no, altre l'audio campionato non la musica, per non parlare delle nuove funzionalità quali Unicode e hyperlink. Ma il gioco *dovrebbe* poter funzionare correttamente su *qualunque* interprete.

Certo, in qualche caso si potrebbe avere la tentazione di specificare "*questo gioco funziona solo con la libreria Glk* tal-dei-tali". Legittimo, ma attenzione: si rischia di perdere del pubblico (non sembra una nuova *guerra dei browser*? Esattamente quello che intendevo).

Il sistema gestalt (per sapere quel che è disponibile e quel che non lo è) è ampiamente descritto nella sezione 1.7 e nelle sezioni relative alle funzionalità facoltative. Ma un esempio non fa male, preso in questo caso da *Gull*.

> Immaginiamo un gioco ambientato nel Far West, e quando il protagonista arriva in città nota un cartello di legno vecchio e tempestato di proiettili: "*Dead Dog, Kansas, Popolazione 213*".
>
> Il giocatore inizialmente vede solo un cartello, per sapere quel che vi è scritto lo deve esaminare più da vicino, oppure leggerlo; nella stanza è solo menzionato "*C'è un cartello qui.*"

Ora, *casualmente*, immaginiamo anche di avere una bellissima immagine rappresentante il cartello, in legno, coi buchi, eccetera... l'idea sarebbe di mostrarla al giocatore al momento giusto. *Ma* non tutti gli interpreti supportano la grafica; magari il cartello è vitale alla soluzione del gioco! Quindi è buona norma fornire almeno una descrizione sostitutiva da mostrare in caso di necessità. Per decidere, la funzione glk_gestalt è proprio quel che serve:

```
Object cartello "cartello␣rovinato"
    with name 'rovinato' 'cartello',
    description [;
        if (glk_gestalt(gestalt_Graphics, 0)) {
            ! Qui mostreremo l'immagine
        } else
"Sul␣cartello␣c'e'␣scritto␣~DEAD␣DOG,␣KANSAS,␣POP.␣213~
␣O␣magari␣intendeva␣2130␣o␣21300␣---␣qualcuno␣ha␣rotto
␣l'angolo␣del␣cartello␣a␣colpi␣di␣fucile.";
    ],
    has static;
```

14.9. Creare le finestre iniziali

Finché si intende utilizzare il layout standard delle finestre (una finestra di stato e una della storia, con eventualmente una aggiuntiva per le citazioni) la gestione delle finestre (in particolare la divisione iniziale) è delegata completamente alla libreria standard.

Se si vuole personalizzare il layout è necessario implementare la funzione InitGlkWindow (uno stub dichiarato nella libreria).

`InitGlkWindow(winrock)`

Questa funzione viene chiamata per la prima volta dalla libreria prima di Initialise . Durante la prima chiamata winrock viene passato a zero.

In questa situazione il compito della funzione è di creare il layout di finestre desiderato.

La documentazione consiglia di settare gg_statuswin_size al valore desiderato durante questa fase.

Questa variabile (di default vale 1) viene utilizzata dalla gestione di default degli eventi per mantenere la griglia di testo utilizzata per la finestra di stato dell'altezza desiderata. La libreria non la modifica mai di propria iniziativa.

Attenzione: c'è la possibilità per cui le finestre siano già esistenti! Come, per esempio, in caso di restart (vedere il discorso sull'enumerazione degli oggetti nella sezione 14.6. Basta tenerne conto (al limite chiudere e riaprire tutto!).

È assolutamente necessario creare la finestra gg_mainwin: in caso contrario la libreria esegue un quit. Settando gg_statuswin si può usufruire del trattamento automatico da parte della libreria.

Il valore di ritorno della prima chiamata deve essere 0 per proseguire con la creazione delle finestre di default. Se viene ritornato 1 la libreria non effettua altre operazioni. In caso contrario, se le finestre di base (gg_statuswin e gg_mainwin) sono già state create, non vengono comunque toccate.

Se si vuole sfruttare il codice di creazione standard delle finestre, immediatamente prima dell'apertura di una delle finestre standard (principale, di stato o di citazione) la libreria chiama InitGlkWindow con il valore sotto al sasso corrispondente (GG_MAINWIN_ROCK durante la creazione della finestra principale, GG_STATUSWIN_ROCK per quella di stato o GG_QUOTEWIN_ROCK durante una citazione). Questo può essere utile per configurare gli stili della finestra. Anche in questo caso, ritornando 1 si evita qualunque altra operazione al riguardo da parte della libreria.

Nella configurazione di default la finestra principale e quella per le citazioni sono buffer di testo (la finestra per le citazioni viene creata solo in caso di

necessità, con lo statement `box`. La finestra di stato è invece una griglia di testo. Niente finestre grafiche nel layout standard![6]

Incidentalmente, se per qualche motivo non fosse possibile creare la finestra per le citazioni, viene comunque visualizzato il testo nella finestra principale.

Alla fine dell'inizializzazione delle finestre (prima di completare il resto dell'inizializzazione della libreria e passare a `Initialise`) viene chiamata per un'ultima volta `InitGlkWindow` con parametro 1. Durante questa fase si possono *fare le pulizie* e completare qualunque cosa fosse rimasta in sospeso.

La finestra per le citazioni *non* viene aperta durante la fase iniziale, ma solo quando richiesto (ovvero lo statement `box`). Se non viene utilizzata questa istruzione non si vedrà mai una chiamata con `GG_QUOTEWIN_ROCK`.

Durante la creazione della finestra per le citazioni si può trovare nella variabile `gg_arguments-->0` il numero di righe richieste, nel caso si desideri crearla all'interno di `InitGlkWindow`.

Come esempio di creazione di una finestra *fuori dalla norma*, ecco come creare una finestra grafica (clamorosamente copiato da *Glk for Dunces*).

```
Constant GG_MYGRAPHWIN_ROCK 210;
Global gg_mygraphwin = 0;
```

Una costante per identificare la finestra (il valore sotto il sasso) e una variabile per contenere il riferimento alla finestra.

```
gg_mygraphwin = glk_window_open(gg_mainwin,
    winmethod_Above+winmethod_Proportional,
    30, wintype_Graphics, GG_MYGRAPHWIN_ROCK);
```

Il significato della chiamata è: dividere orizzontalmente la finestra principale (`gg_mainwin`) in modo da ricavare una nuova finestra grafica nel 30% superiore della finestra.[7] Sotto il sasso ci finisce `210`.

Ovviamente la finestra principale deve già esistere. Inoltre devono essere supportate le finestre grafiche (gestalt può confermare il fatto). Ma qualunque cosa accada alla fine `gg_mygraphwin` conterrà zero (niente finestra grafica, in tal caso) oppure un riferimento alla nuova finestra da passare a successive chiamate Glk.

Un altro caso probabilmente interessante: ammettiamo che le nostre immagini grafiche siano di 240 pixel. Le finestre grafiche sono misurate in pixel, quindi in questo caso conviene una divisione non proporzionale.

```
gg_mygraphwin = glk_window_open(gg_mainwin,
```

[6] E neppure finestre vuote, se è per quello.

[7] Le costanti `winmethod` possono essere sommate senza problemi, anche se l'operazione corretta sarebbe un *or* logico.

```
    winmethod_Left+winmethod_Fixed,
    240, wintype_Graphics, GG_MYGRAPHWIN_ROCK);
```

In questo caso la finestra viene ricavata sul lato *sinistro* della finestra principale (per 240 pixel). Se si fosse utilizzato `winmethod_Above`, come nel caso precedente, la finestra sarebbe stata *alta* 240 pixel.

❖ ❖ ❖

Si può forzare solo *una* dimensione di una finestra per volta. Per specificarle entrambe bisogna aprire *due* finestre, dividendo in modo appropriato: una volta in orizzontale e una in verticale.

Anche *l'ordine* in cui vengono aperte le finestre è importante (per determinare la finestra chiave e la gerarchia di dimensionamento, vedere la sezione 3.2).

Inoltre non necessariamente lo spazio finale è quello richiesto.

`glk_window_get_size` permette di stabilire la dimensione effettiva ottenuta.

◆ ◆ ◆

Non è tutto qui il lavoro da fare: per gestire correttamente lo stato di gioco sarà anche necessario implementare `IdentifyGlkObject`, per riassociare correttamente i riferimenti in caso di undo/restore.

14.10. Gli stili di testo

La libreria standard di Inform utilizza gli stili in modo... standard. Per la precisione:

Input Per l'input emulato da file di script;

Header Titolo dell'avventura (`Banner`);

Subheader Nome della locazione corrente, intestazione del menù `DoMenu`;

Alert Messaggi di fine gioco: hai vinto, sei morto, `DeathMessage`;

Note Notifiche di cambiamento del punteggio (`NotifyTheScore`);

BlockQuote Finestra di citazione (statement `box`);

Preformatted Testo a spaziatura fissa (statement `font on`).

Ovviamente l'autore ha completa libertà al riguardo degli stili, in particolar modo gli stili User1 e User2 che di default non vengono impiegati.

La configurazione dell'aspetto degli stili deve essere fatta *prima* della creazione di una finestra (sottosezione 5.5.1). Il punto ideale è `InitGlkWindow` (la prima chiamata) o addirittura un file di configurazione esterno (glulxe e garglk entrambi supportano questa possibilità, con maggiore flessibilità rispetto ai suggerimenti di stile, anche se limitati allo stesso aspetto per ogni

Figura 14.1. Cambiare stile con Inform sulla Z-machine e su glulx

```
-- Questo funziona sulla Z-machine, non su glulx
print "Come ho detto, la violenza ";
style bold;
print "non";
style roman;
print " e' la risposta in questo caso."

-- L'equivalente per glulx e' questo
print "Come ho detto, la violenza ";
glk_set_style(style_Emphasized);
print "non";
glk_set_style(style_Normal);
print " e' la risposta in questo caso.";
```

Figura 14.2. Funzione di cambio stile di comodo per la stampa formattata

```
[ b text;
    glk_set_style(style_Emphasized);
    print (string) text;
    glk_set_style(style_Normal);
];
```

finestra). L'impostazione dei suggerimenti di stile ha effetto quindi solo sulla *successiva* finestra creata.

Per utilizzare uno stile diverso da quello Normal, basta utilizzare la funzione glk_set_style(), in modo paragonabile all'istruzione Style disponibile sulla Z-machine (Figura 14.1).

Attenzione: a differenza degli stili della Z-machine, gli stili Glk sono mutuamente esclusivi. Così mentre è possibile richiedere uno stile grassetto/italico nella Z-machine, Glk è limitato ad *un solo* stile per volta.

Per semplificare la programmazione è possibile definire delle routine di comodo per semplificare la formattazione (Figura 14.2). La stessa tecnica è già ampiamente in uso per la programmazione destinata alla Z-machine.

In questo modo si può semplicemente scrivere

```
print "Come ho detto, la violenza ",
    (b) "non",
    " e' la risposta in questo caso.";
```

Figura 14.3. Eliminare la finestra di stato

```
[ InitGlkWindow winrock;
    switch (winrock) {
    ! Solo per questa finestra, non facciamo nulla
    GG_STATUSWIN_ROCK:
        rtrue;
    }
    ! Se non si ritorna false non viene creata
    ! neppure la finestra principale!
    rfalse;
];
```

14.11. Finestre di stato personalizzate

La personalizzazione della linea di stato (linea per default, ma è più corretto chiamarla finestra di stato, visto che può essere alta più di una linea) è una delle operazioni classiche in un programma Inform.

La prassi comune è una `Replace DrawStatusLine;` a inizio programma, seguita poi da una definizione alternativa della funzione stessa. La nuova routine è generalmente infarcita di codice assembly per creare il layout personalizzato.

Con Inform/glulx il meccanismo di base è lo stesso, ma a differenza dell'assembly si utilizzano chiamate a Glk. Ovviamente anche alcuni dettagli sono differenti.

Per cominciare la finestra di stato viene *creata* in punti differenti. Mentre sulla Z-machine l'opcode `@split_window` doveva essere richiamato durante la `DrawStatusLine`, la finestra Glk corrispondente viene creata all'inizio, nella routine `GGInitialise`. Come già descritto nella sezione 14.9 è possibile intercettare la chiamata definendo la funzione InitGlkWindow.

Il caso più semplice (l'eliminazione della finestra di stato) si gestisce *non creando* la finestra (Figura 14.3). Le routine della libreria sono sufficientemente intelligenti da non tentare di utilizzare una finestra inesistente.

Se invece si vuole creare una finestra di stato più alta di una linea (una vera *finestra*), nella libreria standard è già presente il codice per farlo senza grossi problemi (Figura 14.4). Se non fosse ancora chiaro `rfalse` in questa funzione significa *procedi con il trattamento di default*.

Una volta che si è in possesso di una finestra di stato della dimensione desiderata, la personalizzazione è persino più semplice che sulla Z-machine. Niente istruzioni assembly, specialmente se si usa infglk.

Figura 14.4. Richiedere una finestra di stato più alta

```
[ InitGlkWindow winrock;
    switch (winrock) {
    ! Settiamo l'altezza richiesta
    GG_STATUSWIN_ROCK:
        gg_statuswin_size = 2; ! Per esempio, o più
        ! Però questa volta ritorniamo false, in modo
        ! da procedere normalmente con la creazione
        rfalse;
    }
    ! Se non si ritorna false non viene creata
    ! neppure la finestra principale!
    rfalse;
];
```

È *estremamente* consigliabile cominciare la nuova DrawStatusLine con il codice standard in Figura 14.5. In questo modo si prevengono malfunzionamenti in casi relativamente infrequenti e difficili da testare.

width e height non sono parametri, ma variabili globali. Come descritto nella sezione 13.2 viene utilizzato l'array di appoggio per ottenere le dimensioni effettive della finestra di stato. Attenzione, il numero di righe disponibili potrebbe essere *inferiore* a quello richiesto!

A questo punto si può scrivere tranquillamente nella griglia di testo:

```
glk_window_move_cursor(gg_statuswin, 3, 0);
print "Ora: ";
```

Un piccolo ma importante particolare: le coordinate in una griglia di testo cominciano da $(0, 0)$, mentre nella Z-machine l'origine è a $(1, 1)$. Quindi la scritta sarà a partire del quarto carattere della prima riga.

Il cursore avanza automaticamente ma attenzione ai caratteri estesi: una particolare libreria è libera di rappresentare, per esempio, i caratteri accentati con *due* caratteri consecutivi (sezione 2.2, in particolare il commento a gestalt_CharOutput).

Con l'ausilio delle variabili width e height si possono realizzare tutti gli allineamenti voluti, conoscendo la lunghezza della stringa (le tecniche sono *le stesse* impiegate nella programmazione per Z-machine).

Ovviamente è anche possibile sfruttare stili di testo distinti: come sempre vale la regola:

> "per ogni carattere uno ed un solo stile".

Figura 14.5. Scheletro per una DrawStatusLine personalizzata

```
[ DrawStatusLine width height;
    ! Se non c'e' finestra di stato, nulla da fare
    if (gg_statuswin == 0)
        return;

    ! Se il giocatore non e' da nessuna parte,
    ! nulla da fare (caso anomalo ma possibile)
    if (location == nothing ||
        parent(player) == nothing)
        return;

    ! Selezioniamo la finestra per la print
    glk_set_window(gg_statuswin);

    ! Stabiliamo quanto spazio abbiamo a disposizione
    glk_window_get_size(gg_statuswin,
        gg_arguments, gg_arguments+4);
    width = gg_arguments-->0;
    height = gg_arguments-->1;

    ! Puliamo la lavagna
    glk_window_clear(gg_statuswin);

    ! ...
    ! Qui scriveremo nella finestra
    ! ...

    ! Ripristiniamo l'output nella finestra principale
    glk_set_window(gg_mainwin);
];
```

14.12. La gestione degli eventi

Come già accennato nel commento alla sezione 1.1, un programma Inform/-glulx non segue *esattamente* la struttura consigliata dalle specifiche (un ciclo degli eventi e tutto che vi ruota intorno).

Molto probabilmente questa differente organizzazione deriva dal target originale di Inform: la Z-machine. Sebbene avesse diverse innovazioni per i suoi tempi (come la memoria paginata e lo stesso concetto di macchina virtuale), il modello di controlla della Z-machine è classico: *il programma comanda l'I/O*.[8]

Il modello classico è quindi il seguente: la Z-machine esegue del codice, richiede l'I/O e poi *si ferma*. Lo stesso timed input (inventato per *Border Cross*, se non ricordo male) altro non è che un rudimentale timeout applicato all'opcode `read` (che legge una riga di input dal giocatore).

Per questo motivo, a differenza di un *tipico* programma Glk, Inform utilizza la pompa degli eventi di Glk solo in una circostanza definita: durante l'accesso alla tastiera. In particolare nella routine `KeyCharPrimitive` (usata come sostituzione della vecchia `@read_char`, principalmente per i menù), `KeyDelay` (una new entry di Inform 6.3, attende un tasto con un timeout), e `KeyboardPrimitive` (legge una riga di input, coprendo quindi il 99% dell'input di una Avventura Testuale).

La cosa importante è che, qualunque cosa venga fatta, fino a che non viene eseguita (direttamente o indirettamente) una di queste tre routine, il giocatore *non vede* alcun aggiornamento dello schermo (a meno di chiamate esplicite per poll o select di nuovi eventi).

Un'altra peculiarità del ciclo degli eventi di Inform/glulx (ovvero, dei *tre* cicli) è la possibilità di definire una funzione per gestire gli eventi, in seguito alla gestione di default che ne fa la libreria (la gestione di default viene *sempre* eseguita; se la si vuole eliminare bisogna utilizzare Replace o modificare la libreria standard, in generale).

```
HandleGlkEvent(ev, context, abortres)
```

Questa funzione (se fornita dall'autore, in quanto Stub) permette di gestire ogni singolo evento che viene passato alla libreria da Glk. È quindi un punto di passaggio obbligato per, ad esempio, gestione del mouse, dei link, delle notifiche audio e altro.

Il primo parametro `ev` è un array, ma in realtà è una struttura (Inform non ha il concetto di struct come il C). I tre elementi `ev-->0`...`ev-->3` so-

[8] Al contrario, oggigiorno, la programmazione ad eventi ha rivoltato come un guanto la struttura per cui *l'I/O guida il programma*.

Tabella 14.3. Elementi dell'array `ev` passato a `HandleGlkEvent`

Elemento	Valore della struttura evento
`ev-->0`	Tipo dell'evento (Capitolo 4)
`ev-->1`	Finestra che ha generato l'evento (0 se non significativa)
`ev-->2`	`val1`, dipende dal tipo evento
`ev-->3`	`val2`, dipende dal tipo evento

no corrispondenti ai campi della struttura evento (vedere il Capitolo 4): rispettivamente `type`, `win`, `val1` e `val2` (Tabella 14.3).

Il secondo parametro, `context`, indica che tipo di input si attendeva la libreria in quel momento: 0 se in attesa di input a linea (`KeyboardPrimitive`) oppure 1 se in attesa di input a carattere (`KeyCharPrimitive` o `KeyDelay`). Può essere utile, in particolare, per annullare la richiesta di evento in corso.

Il terzo parametro è significativo solo in un contesto di input a linea, ed è il buffer di ingresso della libreria (*non* il buffer di ingresso di Glk!). Può essere utilizzato per forzare la linea di input (per esempio in risposta a click del mouse), in congiunzione con un valore di ritorno a 2.

Il valore di ritorno di `HandleGlkEvent` è significativo solo per due valori: se 2, l'input a linea viene annullato e l'input del giocatore viene forzato a quanto inserito in `abortres`. Se a -1 viene riproposto l'input al giocatore anche in caso di conferma dello stesso (anche se non si capisce a cosa possa servire). In questo ultimo caso è prima necessario ri-richiedere l'input (visto che l'evento è già stato completo).

Per la precisione: in un contesto 1 (input a carattere) si annulla per sicurezza l'input a carattere (con `glk_cancel_key_event()`). Per finire si setta `abortres-->0` al carattere che si vuole forzare e si ritorna 2.

In un contesto 0 (input a linea) si deve prima annullare l'input a linea (con `glk_cancel_line_event()`) eventualmente recuperando quel che il giocatore ha digitato. È possibile poi mettere la lunghezza della stringa da forzare in `abortres-->0` (non più di 256 caratteri). Nei byte successivi va poi messa la stringa da passare al parser (eventualmente ottenuta elaborando quel che stava digitando il giocatore). Per finire, si ritorna 2.

Un esempio su come forzare un comando ("*inventario*" nel caso in questione) può essere visto in Figura 14.6.

Attenzione: la cosa interessante è che in questo caso `newcmd` è una stringa costante (quindi identificata dal proprio indirizzo, in glulx). La chiamata a `PrintAnyToArray` (con i parametri indicati) permette di trasformarla nell'array di caratteri su cui poi lavorerà il parser per eseguire il comando (`WORDSIZE` sarà sempre uguale a 4 sotto glulx; la prima word del buffer è riservata al numero di caratteri che seguono).

Figura 14.6. Forzare un comando in seguito ad un evento

```
[ HandleGlkEvent ev context abortres
    newcmd cmdlen; ! Variabili locali
    ! Condizioni per determinare quando eseguire:
    ! tipicamente un click del mouse o un collegamento
    if (...) {
        ! Per prima cosa annulliamo l'input in corso
        glk_cancel_line_event(gg_mainwin, 0);
        ! Poi prepariamo il nuovo comando nel buffer
        newcmd = "inventario";
        cmdlen = PrintAnyToArray(abortres+WORDSIZE,
            INPUT_BUFFER_LEN-WORDSIZE, newcmd);
        abortres-->0 = cmdlen;
        ! Indichiamo quel che stiamo facendo nello
        ! stile Input
        glk_set_style(style_Input);
        print "(inventario)";
        glk_set_style(style_Normal);
        new_line;
        ! Per finire segnaliamo alla libreria che
        ! c'e' un comando da eseguire
        return 2;
    }
];
```

La gestione in dettaglio delle singole tipologie di evento viene trattata nel Capitolo 4.

Per un esempio dettagliato su come gestire gli eventi di ridisegno in una finestra grafica vedere la sezione 14.14.

14.13. Mostrare immagini

Se si vogliono mostrare immagini (sempre ammesso che la libreria in uso supporti visualizzazione di immagini) si hanno a disposizione due possibilità:

Immagini inframezzate al testo o su righe a se stanti, o in linea con il testo o anche a margine. Appaiono in un buffer di testo, tipicamente nella finestra principale.

Immagini in una finestra dedicata appaiono in una finestra grafica (che deve essere aperta esplicitamente, il layout di default non gestisce finestre grafiche). In questo caso il posizionamento è assoluto, in pixel, all'interno della finestra.

Questi due metodi di visualizzazione delle immagini sono *entrambi* facoltativi e devono quindi essere verificati indipendentemente con il selettore `gestalt_DrawImage` (sezione 7.4).

Ortogonalmente, lo standard Blorb mette a disposizione *due formati* di immagine: le immagini di tipo JPEG adatte per fotografie e con maggiore compressione e quelle in formato PNG più adatte per disegni. Sono facilmente reperibili informazioni su entrambi i tipi di formati.

Se la libreria lo supporta, il formato PNG permette di gestire la trasparenza (il canale alpha) (sezione 7.4).

14.13.1. Grafica in una finestra di testo

Mostrare un'immagine in un buffer di testo (come per esempio nella finestra principale `gg_mainwin`) è il caso più semplice. Basta decidere *quale* immagine visualizzare e dove metterla in relazione al testo.

Le immagini per Glk sono identificate dal numero di risorsa nel file Blorb. I tool per costruire i file Blorb generalmente sono in grado di generare un file include con la dichiarazione delle costanti corrispondenti alle risorse richieste.

Per quanto riguarda il *dove* esistono cinque tipi possibili di allineamento, specificato nel terzo parametro di `glk_image_draw` quando si utilizza un buffer di testo. Tre di questi allineamenti mettono l'immagine esattamente nel punto di inserimento (Figura 14.7, Figura 14.8, Figura 14.9).[9]

Le due modalità rimanenti permettono di mettere le immagini a margine, ma con una limitazione: possono essere inserite solo a inizio paragrafo (in Inform, dopo un "^" un `newline`).[10] Inoltre non necessariamente sono visualizzate correttamente più di una immagine per ogni margine (Figura 14.10, Figura 14.11).

Il quarto parametro della funzione `glk_image_draw` in questo caso è a zero, quindi di può omettere. In questo modo, ricapitolando, per mostrare un'immagine in un buffer di testo è sufficiente una chiamata del tipo

```
glk_image_draw(gg_mainwin, IMG_SCIMMIA,
```

[9] L'allineamento verso il basso nella versione distribuita di garglk-mod è difettoso! È già stato corretto ma per una sola linea di codice non vale la pena fare una release, considerando che *nessuno* ha mai usato questo allineamento.

[10] garglk-mod le accetta anche a metà paragrafo ma in compenso è limitato ad *una* immagine per margine alla volta.

Figura 14.7. Allineamento imagealign_InlineUp

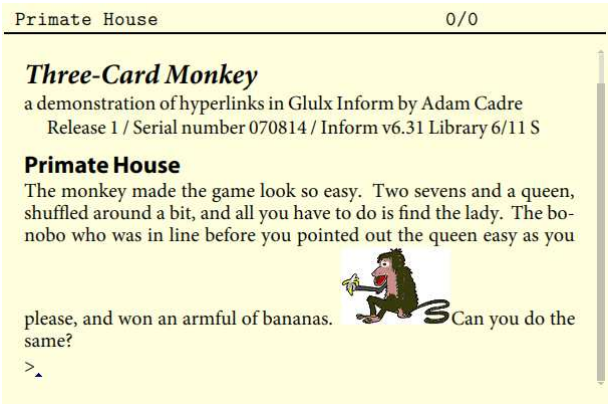


Figura 14.8. Allineamento imagealign_InlineCenter

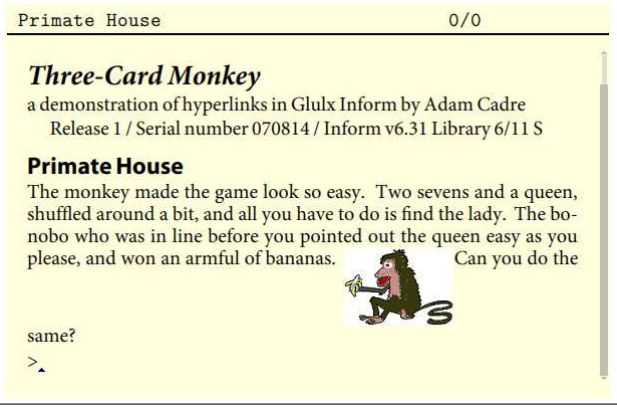


Figura 14.9. Allineamento imagealign_InlineDown

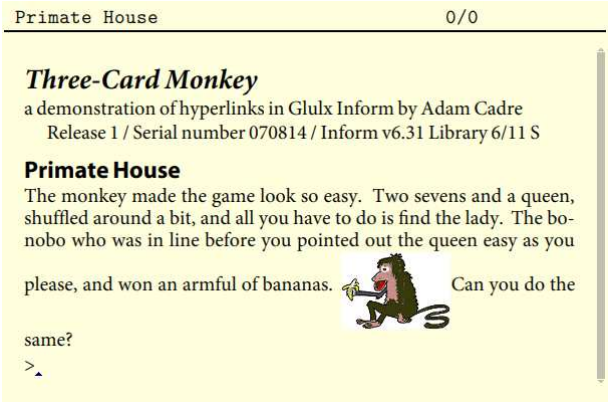


Figura 14.10. Allineamento imagealign_MarginLeft

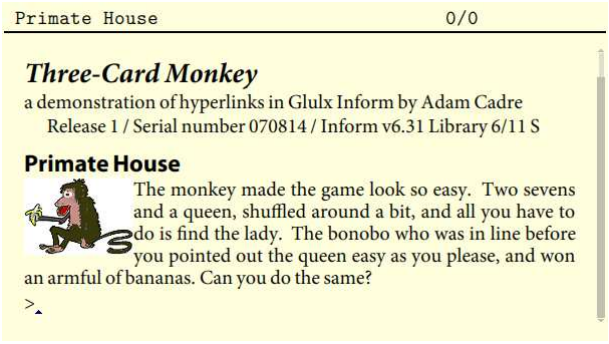
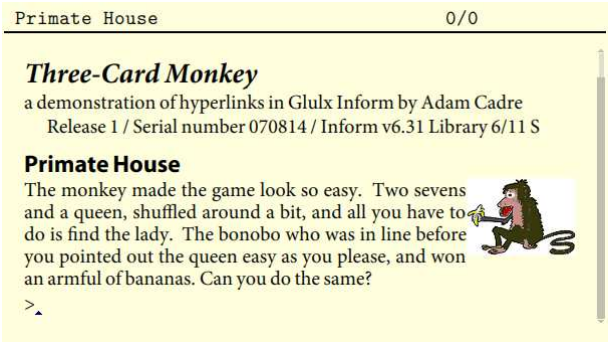


Figura 14.11. Allineamento imagealign_MarginRight




```
    imagealign_MarginLeft);
```

per ottenere un risultato simile a quello di Figura 14.10.

Naturalmente IMG_SCIMMIA deve essere una costante contenente il numero di risorsa e la risorsa stessa deve essere accessibile durante l'esecuzione (in caso contrario, semplicemente, niente immagine!).

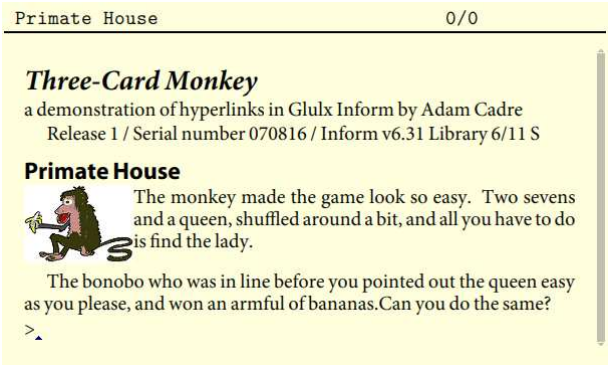
Sempre nel caso di immagini a margine, è anche possibile forzare dello spazio in modo da superare l'immagine (e quindi tornare a margini pieni). La chiamata per questo effetto è `glk_window_flow_break` (Figura 14.12)[11]

14.13.2. Grafica in una finestra grafica

Per visualizzare un'immagine in una finestra grafica il procedimento è del tutto simile. In questo caso il primo parametro di `glk_image_draw` sarà il riferimento ad una finestra grafica e secondo il numero della risorsa, esattamente allo stesso modo.

[11] L'implementazione di garglk-mod attualmente impone il break a inizio paragrafo. Le specifiche ufficiali non sono chiare al riguardo.

Figura 14.12. Effetto della funzione `glk_window_flow_break`



Il terzo e il quarto parametro questa volta indicano le coordinate (in pixel) alle quali disegnare l'immagine nella finestra grafica. Per la precisione l'angolo in alto a sinistra dell'immagine verrà posto nella posizione specificata.

Tutto molto semplice. Ma ci sono delle complicazioni:

- Ovviamente bisogna avere una finestra grafica in cui disegnare;

- Le finestre grafiche *non* vengono ripulite automaticamente. Per sostituire l'immagine visualizzata è possibile ripulire la finestra o disegnare un'altra immagine sopra alla precedente. In questo caso dovrà essere almeno della stessa dimensione, altrimenti sotto rimangono pezzi della vecchia!

  Questo comportamento è però utile per comporre dinamicamente immagini complesse, specialmente se la libreria in uso supporta la trasparenza.

  > Per esempio, ammettiamo di voler mostrare una *mappa del tesoro* con la "X" nel punto dove scavare. Ma magari il punto da scavare *non è sempre lo stesso*. Oppure il giocatore lo scopre solo in seguito.

  In questo caso basta avere una immagine della mappa da usare come sfondo e una immagine, piccola e trasparente, da visualizzare eventualmente *sopra* la prima, nella posizione voluta.

  L'alternativa sarebbero tante immagini diverse, per ogni posizione della X accettabile. Un'altra possibilità sarebbe di comporre la mappa a pezzi in base alle locazioni conosciute dal giocatore (con la proprietà `visited`).

- Le finestre grafiche possono venir cancellate senza preavviso. La libreria in tal caso manda un evento di ridisegno, e questo deve essere gestito;
- Ultimo ma non meno importante: non c'è alcun collegamento fra la gestione delle locazioni della libreria e le immagini. Tanto peggio se nel frattempo viene ripristinato o riavviato il gioco.

Per chiarire meglio questo ultimo punto (che tornerà a tormentarci a riguardo dei canali audio). Una implementazione *ingenua* per mostrare la tipica immagine rappresentante la locazione potrebbe essere composta da una chiamata a `glk_draw_image` nella proprietà `description` di ogni locazione (o qualcosa del genere).

Apparentemente tutto va per il verso giusto (eventualmente con delle routine appoggio, una proprietà per contenere la risorsa immagine, e altri accorgimenti per semplificarci la vita).

Ma mettiamo il caso che il giocatore *ripristini* una posizione precedente, o faccia *undo* dopo un movimento. Lo stato interno del gioco è stato ripristinato. *La grafica visualizzata non è stata toccata*. È lo stesso fenomeno che richiede la *riassociazione* delle finestre e `IdentifyGlkObject`.

E la soluzione infatti si basa su un meccanismo molto simile. Per la cronaca, vi siete ricordati di riassociare correttamente la finestra grafica? No vero?

14.14. Finestre grafiche aggiuntive

Dove con *aggiuntiva* si intende: dalla prima in poi, visto che il layout di default non contempla finestre grafiche.

Ecco quel che bisogna fare per gestire una finestra grafica (e, con alcune differenze, una finestra aggiuntiva di qualunque tipo):

- Una variabile globale per il riferimento alla finestra:

  ```
  Global gg_picwin;
  ```

- Una costante da mettere sotto il sasso per identificare la finestra in seguito:

  ```
  Constant GG_PICWIN_ROCK 210;
  ```

- Un mezzo per ridisegnare il contenuto della finestra grafica (Figura 14.13); `current_pic` è nell'esempio una variabile globale che contiene l'immagine da mostrare (il suo numero di risorsa);

  L'idea è che ogniqualvolta si voglia cambiare l'immagine si aggiorna questa variabile e poi si chiama `RidisegnaGrafica`.

Figura 14.13. Ridisegno di una finestra grafica

```
[ RidisegnaGrafica;
    if (gg_picwin ~= 0) {
        glk_window_clear(gg_picwin);
        glk_image_draw(gg_picwin, current_pic, 0, 0);
    }
];
```

- Una funzione IdentifyGlkObject per tenere aggiornato il valore della variabile gg_picwin (Figura 14.14).

  Il meccanismo di funzionamento di questa funzione è spiegato in dettaglio nella sezione 14.6. Quando viene, per esempio, ripristinato lo stato di gioco, i riferimenti ad oggetti Glk non sono più validi. Grazie alla costante GG_PICWIN_ROCK la routine è in grado di ripristinare il valore corretto.

  Se la finestra non esiste più (o non esiste ancora) la funzione di ridisegno gestisce il caso (in fase 0 il riferimento viene azzerato).

- Una funzione HandleGlkEvent per gestire gli eventi di ridisegno e ridimensionamento (Figura 14.15).

  Questa funzione gestisce i due casi in cui una finestra grafica deve essere ridisegnata per motivi indipendenti dal programma:

  - La finestra è stata ridimensionata (evtype_Arrange, sezione 4.5).
  - La finestra deve essere ridisegnata (evtype_Redraw, sezione 4.6).

- Per finire, la prima cosa da fare: creare la finestra grafica, per esempio in InitGlkWindow (nella fase 0 o 1) o comunque nel momento in cui è necessaria (Figura 14.16). Il controllo sul valore di gg_picwin serve per il caso in cui il programma venga riavviato con la finestra ancora aperta. Potrebbe essere necessaria qualche gestione particolare se il formato della finestra non fosse costante durante l'esecuzione (nell'esempio viene utilizzata una finestra con un altezza fissa di 200 pixel).

Queste funzioni lavorano assieme per far funzionare la finestra. Non è complesso estenderle per gestire più di una finestra. Nel caso in cui il layout non fosse costante la situazione è decisamente più complessa.

Fortunatamente, è possibile prevedere (fino ad un certo punto) quando sarà necessaria una riassociazione (la parte più complessa della gestione). Per la precisione *non* sarà mai necessario gestirla se al giocatore non è concesso

Figura 14.14. Riassociazione di una finestra aggiuntiva

```
[ IdentifyGlkObject phase type ref rock;
    ! In fase zero si azzerano i riferimenti, che
    ! potrebbero essere non piu' validi
    if (phase == 0) {
        gg_picwin = 0;
        return;
    }

    ! In fase uno vengono passati tutti gli oggetti
    ! non riconosciuti dalla libreria, in modo da
    ! riassociarli al loro riferimento
    if (phase == 1) {
        switch (type) {
        0: ! Bisogna identificare una finestra
        switch (rock) {
        ! Se e' la nostra finestra la salviamo
        GG_PICWIN_ROCK: gg_picwin = ref;
        }
        1: ! Bisogna identificare uno stream
            ! Nel caso ne avessimo creati
        2: ! Bisogna identificare una fileref
            ! Nel caso ne avessimo creati
        ! 3: Ma non vedremo mai canali audio, qui!
        }
        return;
    }

    ! In fase 2 l'enumerazione e' stata completata
    if (phase == 2) {
        ! Per esser sicuri ridisegnamo la finestra
        RidisegnaGrafica();
    }
];
```

Figura 14.15. Gestione degli eventi di ridisegno

```
[ HandleGlkEvent ev context;
    switch (ev-->0) {
    evtype_Redraw, evtype_Arrange:
        RidisegnaGrafica();
    }
];
```

Figura 14.16. Creazione di una finestra grafica

```
if (gg_picwin == 0) {
    gg_picwin = glk_window_open(gg_mainwin,
        (winmethod_Above+winmethod_Fixed), 200,
        wintype_Graphics, GG_PICWIN);
}
```

ripristinare, riavviare o annullare un'operazione (tipico è il caso dei menù). Fino a quando non esiste la possibilità di una di queste operazioni non si rischia una riassociazione (in sostanza: fino alla fine del turno).

14.15. Suoni nei canali audio

E dove altrimenti?

Scherzi a parte, la gestione dei canali audio non si discosta di molto dalla gestione di una finestra aggiuntiva (sezione 14.14). Anche in questo caso bisogna creare un oggetto Glk, farci qualcosa quando si vuole (riprodurre suoni, in questo caso) e gestire in modo appropriato le chiamate di riassociazione della libreria (pensare a quel che accade ripristinando la posizione con un canale audio aperto e in esecuzione... il gioco riparte, il sonoro prosegue).

L'unica leggera differenza è data dal mancato supporto della libreria per l'enumerazione in automatico dei canali audio. Ma, sempre dentro alla funzione IdentifyGlkObject, la soluzione è comunque molto semplice.

Come per le risorse grafiche, Glk utilizza anche risorse sonore. Come queste ultime gli identificativi sono interi a piacere (non solo, può esistere la risorsa grafica numero 1 *e simultaneamente* la risorsa audio numero 1, senza alcun conflitto). Anche in questo caso presumibilmente lo strumento che crea il file Blorb genererà un file con la definizione delle costanti corrispondenti alle risorse disponibili.[12]

[12] Se non viene utilizzata un file di risorse Blorb, garglk-mod è comunque in grado di caricare

Sono disponibili due tipi di risorsa audio:

- Suoni campionati (lo standard specifica lo standard AIFF, parente dal lato Macintosh del ben più noto WAV), non compressi.
- Musica di tipo MOD (un formato vecchiotto nato su Amiga).
- Suoni campionati in formato compresso (lo standard impone il formato OGG/Vorbis).

da file con nomi SND1, SND2, eccetera.

Inoltre, come formati *non standard*, garglk-mod è in grado di riconoscere e riprodurre:

- File WAV non compressi.
- File compressi in formato MP3.

Le specifiche fanno una distinzioni fra *suoni* e *musiche*: il motivo è che alcune librerie possono avere limitazioni diverse al riguardo. Si intende con *musica* un file di tipo MOD; tutti gli altri formati (compresi OGG e MP3) vengono considerati come *suoni*.

Se una libreria supporta l'audio in generale metterà a disposizione almeno un canale per riprodurre suoni (con questo si intende: al massimo una risorsa di tipo sonoro in riproduzione simultaneamente). Librerie più evolute potranno avere limitazioni in altri sensi.[13]

Le risorse audio vengono riprodotte in canali audio, proprio come le risorse grafiche vengono visualizzate nelle finestre grafiche. Ignorando le chiamate preliminari per stabilire se l'audio è disponibile (le solite chiamate gestalt), ecco ripetuto l'esempio riportato in *Gull* (la fonte di molti altri esempi).

> Quello che si vuole ottenere è che all'interno di una locazione *Ascensore* venga riprodotta della musica quando il giocatore è all'interno.

Il procedimento è identico sia per la musica che per suoni di altro genere. La distinzione viene fatta solo per descrivere le limitazioni di una particolare implementazione.

Quello che è necessario per un lavoro ben fatto è:

- Una variabile globale per il riferimento al canale audio:

  ```
  Global gg_musicchan;
  ```

- Una costante da mettere sotto il sasso per identificare il canale audio:

  ```
  Constant GG_MUSICCHAN_ROCK 410;
  ```

  Perché 410? Anche se nessuno ci obbliga, nella progressione logica i canali audio andrebbero identificati dal 410 in poi, credo (Tabella 14.2). Alla fine basta che non sia uguale ad altri sassi in uso...

[13] garglk-mod ha, indicativamente, supporto per un canale musicale e 8 canali per risorse di tipo suono. Ma dipende dalla configurazione in compilazione.

Figura 14.17. Creazione di un canale audio

```
if (gg_musicchan == 0) {
    gg_musicchan=glk_create_schannel(
        GG_MUSICCHAN_ROCK);
}
```

Figura 14.18. Riproduzione di musica di sottofondo

```
[ RiavviaMusica;
    if (gg_musicchan) {
        if (current_music == 0)
            glk_schannel_stop(gg_musicchan);
        else
            glk_schannel_play_ext(gg_musicchan,
                current_music, -1, 0);
    }
];
```

- Per semplificarci la vita, una variabile per contenere la musica di sottofondo da tenere in esecuzione (proprio come current_pic nell'esempio grafico; anche in questo caso si può automatizzare il processo di cambio musica quanto si vuole):

  ```
  Global current_music;
  ```

- Un canale audio in cui riprodurre la risorsa (ovviamente, deve esserci *qualcosa* da riprodurre e *qualcosa* in cui riprodurla).

  Creare un canale audio è molto più semplice che creare una finestra grafica: non ci sono divisioni, parametri o dimensionamenti. Ogni canale esiste indipendentemente dagli altri. Se qualcosa va male, tanto per cambiare si ottiene uno zero (GLK_NULL).

  Dato che i canali audio anche se non si usano comunque non si vedono (a differenza delle finestre grafiche), una buona strategia potrebbe essere quella di crearli nella funzione Initialise (Figura 14.17). Come promesso, nulla di complicato (il confronto con zero copre il caso del restart, al solito).

- Una funzione per avviare l'audio. Parallelamente a RidisegnaGrafica, ecco la funzione RiavviaMusica (Figura 14.18).

Nonostante la sua estrema semplicità questa routine contiene un paio di finezze: per cominciare se *non* c'è musica da suonare arresta la riproduzione; inoltre se la risorsa da riprodurre è la stessa già in esecuzione la libreria Glk *dovrebbe* essere sufficientemente intelligente da *non riavviarla*. In questo modo la funzione `RiavviaMusica` può essere chiamata ripetutamente senza disturbare la musica già in esecuzione (se non è cambiata, ovviamente).

Cosa dire dei parametri di `glk_schannel_play_ext`? I primi due sono più che ovvi (il canale e la risorsa da riprodurre).

Il terzo parametro è il *numero di volte* per cui la risorsa deve essere riprodotta. Passare -1 in questo caso significa *per sempre* (la musica degli ascensori non finisce mai!).

Per finire il quarto parametro è normalmente zero, a meno che non si voglia ricevere un evento quando la riproduzione è completata (*se* la libreria supporta questo genere di notifiche). Ovviamente, con un numero di ripetizioni infinito la riproduzione non terminerà *mai*, quindi in questo caso non sarebbe comunque utile. Se viene specificato un intero questo verrà passato all'interno di un evento `SoundNotify` alla fine della riproduzione della risorsa (vedere anche la sezione 8.3).

Esiste anche la funzione `glk_schannel_play` che semplicemente considera gli ultimi due argomenti a zero.

- Per finire, un segmento in `IdentifyGlkObject` per gestire la riassociazione del canale audio. In questo caso il codice è più complesso perché la libreria standard non enumera automaticamente i canali, per cui viene svolto il lavoro nella fase 2 (Figura 14.19).

  La funzione in figura illustra *solo* la gestione di un canale audio. Deve essere ovviamente completata se si usano altri oggetti Glk non-standard.

Con questo codice all'interno del programma, dovrebbe essere possibile cambiare musica di sottofondo semplicemente settando `current_music` e chiamando `RiavviaMusica()`.

Tutto questo lavoro è necessario *solo* se si hanno delle parti sonore di *lunga durata* che girano in sottofondo. Se sono sufficienti brevi effetti sonori (nell'ordine di un paio di secondi, e comunque *non* in ripetizione infinita!) il sistema di riassociazione *non* è necessario (appena terminata la risorsa il canale si libera automaticamente, non c'è bisogno della `RiavviaMusica` in questo caso).

Figura 14.19. Riassociazione di un canale audio

```
[ IdentifyGlkObject phase type ref rock res id;
    ! Come sempre in fase 0 si fa pulizia
    if (phase == 0) {
        gg_musicchan = 0;
        return;
    }

    ! La fase 1 per i canali audio non è utilizzata
    ! Ovviamente nulla impedisce di gestire altri
    ! oggetti Glk qui (come una finestra)
    ! if (phase == 1) {
    !     return;
    ! }

    ! La fase 2 itera sui canali audio presenti e cerca
    ! quello che e' stato creato. Poi lo reinizializza
    ! con la musica corrente
    if (phase == 2) {
        id = glk_schannel_iterate(0, gg_arguments);
        while (id) {
            switch (gg_arguments-->0) {
            GG_MUSICCHAN_ROCK:
                gg_musicchan = id;
            }
            id = glk_schannel_iterate(id,
                gg_arguments);
        }
        RiavviaMusica();
    }
];
```

14.16. Gli eventi del mouse

Per una volta non c'è bisogno di *creare* nulla. Se la libreria Glk in uso è in grado di ricevere i click del mouse, basta richiederli sulle finestre desiderate (ma *solo* finestre grafiche e griglie di testo; per i buffer di testo si usano i collegamenti, che sono animali completamente diversi).

Gli eventi del mouse sono in grado di indicare due cose, a seconda della finestra dalla quale provengono:

- Un evento mouse in una finestra grafica indica le coordinate *del pixel* che è stato selezionato con il mouse;
- Un evento mouse da una griglia di testo indica le coordinate *del carattere* che è stato selezionato;[14]
- Altri tipi di finestra semplicemente *non generano* eventi del mouse.

Molto semplicemente per richiedere un evento (banalmente, dalla finestra di stato per non complicarci la vita):

```
glk_request_mouse_event(gg_statuswin);
```

Dopo questa chiamata Glk ritornerà *uno ed un solo* evento di click del mouse nella finestra specificata.[15] Ricevuto (e gestito) il primo evento bisogna ripetere la chiamata per ottenerne altri in seguito.

Gli eventi del mouse (come la stragrande maggioranza degli eventi) vengono gestiti solo in due occasioni: quando il programma si aspetta la pressione di un tasto o quando si aspetta una linea intera di testo dal giocatore.

In entrambi i casi la libreria chiama la funzione `HandleGlkEvent` (se definita) per ogni evento che non gestisce direttamente. In questo caso per gestire gli eventi del mouse si può definire la funzione come in Figura 14.20. Nell'esempio è mostrato solo il codice relativo agli eventi del mouse; nel caso si dovrà estendere la struttura switch in modo appropriato.

Anche se ricevere gli eventi del mouse è relativamente semplice, implementare il codice per un determinato effetto non lo è necessariamente.

Tutti gli eventi non-standard vengono eseguiti *al di fuori* del normale turno di gioco. Inoltre tutti i tipi di eventi di tutte le finestre (eventi che poi avverranno in qualunque locazione del gioco!) devono necessariamente passare per `HandleGlkEvent`. Bisogna fare attenzione allo *smistamento* di questi eventi per non complicarsi troppo la vita (in pratica bisogna fare manualmente lo smistamento che il parser fa con i comandi tastiera; a seconda dell'effetto desiderato la questione può essere parecchio complessa.

[14] Alcune versioni di garglk sono difettose in questo senso; ritornano la posizione del pixel anche per le griglie di testo. garglk-mod dovrebbe funzionare correttamente

[15] *Solo* click: il semplice spostamento del mouse non genera eventi.

Figura 14.20. Ricezione degli eventi mouse

```
[ HandleGlkEvent ev context abortres;
    switch (ev-->0) {
    evtype_MouseInput:
        ! Chiediamo altri eventi mouse
        glk_request_mouse_event(gg_statuswin);
        ! Qui si fanno considerazioni sulla
        ! posizione che e' stata cliccata
    }
];
```

I dati per effettuare lo smistamento sono tutti disponibili nell'array `ev` che, per gli eventi del mouse, ha il seguente contenuto:

- In `ev-->0` la costante `evtype_MouseInput`. In questo modo sappiamo che si tratta di un evento del mouse e non altro (la switch di livello superiore).
- in `ev-->1` il riferimento alla finestra che ha generato l'evento (ovvero la finestra su cui ha cliccato il giocatore). Se si richiedono eventi da più di una finestra occorrerà uno smistamento aggiuntivo in base a questo campo.
- in `ev-->2` e `ev-->3` rispettivamente la colonna e la riga del carattere o pixel selezionato (a seconda del tipo di finestra).

Specialmente quando si utilizzano finestre grafiche è conveniente confrontare le coordinate (gli elementi 2 e 3dell'array `ev`) con un intervallo di valori (per esempio, `ev-->2` compreso fra 100 e 150; in caso contrario il giocatore dovrebbe cliccare *esattamente* su un pixel!).

Per far eseguire un comando dal parser come se fosse stato digitato, si può utilizzare `abortres` e il valore di ritorno nel modo già visto in Figura 14.6.

14.17. Collegamenti (hyperlink)

In maniera simile agli eventi del mouse è anche possibile gestire dei *collegamenti ipertestuali* all'interno di buffer e griglie di testo. Ovviamente se disponibili nell'implementazione Glk in uso.

Mentre gli eventi del mouse sono attivi su *tutta* la finestra, i collegamenti sono associati a particolari sezioni di testo, marcate durante l'inserimento da chiamate a `glk_set_hyperlink`.

Il valore passato alla funzione `glk_set_hyperlink` viene associato al testo visualizzato in seguito (che, tipicamente, sarà evidenziato in modo appropriato: sottolineato, per esempio) fino a quando il valore viene riportato a zero (o comunque cambiato, cambiando la *destinazione* del collegamento).

Un esempio banale per chiarire:

```
print "Puoi␣andare␣a␣";
glk_set_hyperlink(42);
print "nord";
glk_set_hyperlink(0);
print "␣oppure␣a␣";
glk_set_hyperlink(43);
print "sud";
glk_set_hyperlink(0);
print ".^";
```

In una situazione di questo genere, prevedibilmente, verrebbe generato l'evento relativo al collegamento 42 con la selezione di "*nord*" o un evento con valore di collegamento 43 con una selezione di "*sud*".

Il valore di collegamento può essere pensato come un *attributo* assegnato ad una sezione di testo (proprio come lo stile). Nulla vieta di riaprire un valore di collegamento dopo averlo chiuso: in sostanza si possono avere più collegamenti con la stessa destinazione. Inoltre è possibile cambiarlo senza necessariamente averlo reimpostato a zero (in questo caso si creano due collegamenti *adiacenti*).

In seguito, quando il giocatore seleziona un collegamento, viene generato un evento di tipo `evtype_Hyperlink` (se richiesto, ovviamente) con il valore associato nell'array `ev`. Starà poi alla routine `HandleGlkEvent` decidere come agire di conseguenza.

I valori associati ai collegamenti possono essere interi arbitrari, ma diversi da zero (il valore zero *chiude* il collegamento).

Una volta visualizzato del testo con dei collegamenti associati, per ricevere eventi è sufficiente chiamare la funzione `glk_request_hyperlink_event`, in modo del tutto analogo a quanto già visto a riguardo del mouse.

```
glk_request_hyperlink_event(gg_mainwin);
```

Anche in questo caso verrà ritornato *solo un evento*. Per cui sarà probabilmente necessario richiamare la funzione come nell'esempio di Figura 14.21.

Nel caso di un evento di questo tipo i campi di `ev` assumono il seguente significato:

- In `ev-->0` la costante `evtype_Hyperlink`. Per distinguere il tipo evento.

Figura 14.21. Ricezione degli eventi di collegamento

```
[ HandleGlkEvent ev context abortres;
    switch (ev-->0) {
    evtype_Hyperlink:
        ! Chiediamo altri eventi
        glk_request_hyperlink_event(gg_mainwin);
        ! Qui si fanno considerazioni sul
        ! collegamento che e' stato selezionato
    }
];
```

- in `ev-->1` il riferimento alla finestra che ha generato l'evento. Se sono presenti collegamenti su più finestre potrebbe essere necessario considerare questo valore, a seconda di come vengono utilizzati i valori di collegamento.
- in `ev-->2` la *destinazione* del collegamento, ovvero il valore specificato a `glk_set_hyperlink` per il testo selezionato.

Come per gli eventi del mouse, anche la gestione dei collegamenti (lo *smistamento*) può essere complessa, a seconda di cosa si vuole ottenere. Una gestione intelligente dei valori di collegamento può semplificare la vita in certi casi (il valore di collegamento può essere qualunque intero a 32 bit diverso da zero).

14.18. Aspettare e ritardare

Nelle precedenti versioni di Inform, una routine di attesa (utilizzando gli eventi del timer di Glk) era una questione abbastanza complessa. Infatti, oltre a richiedere gli eventi con `glk_request_timer_events` (cosa facile di per se) era necessario implementare una mini-pompa degli eventi con `glk_select` per attendere l'evento di *tempo scaduto*.

D'altra parte però, gli eventi Glk possono arrivare in qualunque momento quindi, per esempio, ci si poteva ritrovare a dover gestire *correttamente* eventi di ridisposizione o di ridisegno, nonché a dover chiamare *sempre correttamente* la funzione `HandleGlkEvent` per tutti gli altri eventi non-standard.

Fortunatamente, la libreria standard 6.11 fornisce la chiamata `KeyDelay`, di funzionamento analogo a `KeyCharPrimitive`: aspetta la pressione di un tasto, con un tempo limite (specificato in decimi di secondo). Se viene premuto

Figura 14.22. Attesa fissa di un periodo di tempo

```
[ NoKeyDelay tenths  done ix;
    glk($00D6, tenths*100); ! request_timer_events
    while (~~done) {
        glk($00C0, gg_event); ! select
        ix = HandleGlkEvent(gg_event, 1, gg_arguments);
        if (ix == 2) {
            done = true;
        }
        else if (ix >= 0 && gg_event-->0 == 1) {
            done = true;
        }
    }
    glk($00D6, 0); ! request_timer_events
];
```

un tasto, il tempo viene accorciato (in altre parole, la funzione ritorna in anticipo).

Il valore ritornato è il tasto premuto (oppure zero se il tempo è scaduto).

Incidentalmente, KeyDelay *non* gestisce nessun evento standard, ma almeno integra correttamente la gestione di HandleGlkEvent (compresa la gestione di abortres).

Nulla vieta di richiedere in proprio gli eventi del timer per poi gestirli in altro modo (in HandleGlkEvent o in una pompa degli eventi personalizzata). Ma attenzione: KeyDelay resetta il timer di Glk! (fortunatamente *nessuna* funzione di libreria usa KeyDelay).

Se invece si vuole attendere *forzatamente* un periodo di tempo (ma state in guardia dai giocatori frettolosi e iracondi!) non c'è altro modo che creare una funzione ad-hoc. La stessa KeyDelay può essere facilmente adattata allo scopo (ed è anche un buono scheletro per simili cicli di gestione eventi) come mostrato in Figura 14.22.

14.19. I/O su file

In una normale Avventura Testuale l'autore generalmente non si preoccupa di file esterni all'avventura stessa. Al più avrà a che fare con un file di risorse (gestito comunque in modo automatico dall'interprete) e con i file contenenti la partita salvata (gestiti dalla libreria standard di Inform, in verblibm.h).

Tuttavia, per gestirli, Inform usa comunque le routine di I/O di Glk. Inoltre, in una qualche avventura d'avanguardia, l'autore potrebbe decidere di tenere dei dati in qualche file esterno; prima dell'avvento di glulx una simile tecnica poteva essere utile per avventure *veramente* grandi, tali da riempire persino un file z8 (e, a dire il vero, *Curses* quasi ci riesce visto che le note sono state riportate in un *separato* file z8 per motivi di spazio!). In seguito poi glulx ha praticamente rimosso i precedenti limiti di spazio al riguardo: è stata progettata appositamente allo scopo; l'utilizzo di file esterni viene quindi relegato a informazioni con un ciclo di vita *indipendente* da quello della normale partita.

❖ ❖ ❖

All'autore *veramente* d'avanguardia, potrebbe interessare qualche variazione sul comportamento in salvataggio e ripristino della partita: operazione comunque da effettuare con estrema cautela, per pietà nei rispetti del giocatore.

La macchina virtuale glulx è in grado di salvare/ripristinare lo stato del gioco in maniera semiautomatica. Ha tuttavia bisogno di uno stream *già aperto* sul quale lavorare (i turpi dettagli sono nelle specifiche glulx).

Tutto il codice relativo può essere trovato in `verblibm.h` (con un piccolo aiuto per riassociare lo stream di salvataggio). Fortunatamente si tratta di una ventina di linee di codice in tutto (ben commentate, fra l'altro). Il sadico autore interessato potrà quindi metter mano alle due funzioni `SaveSub` e `RestoreSub`.

◆ ◆ ◆

Creare (e utilizzare) un file con Inform/glulx non è davvero una faccenda tanto diversa rispetto ad altri linguaggi di programmazione. Esistono però alcune particolarità: sfortunatamente il concetto di *nome del file* è qualcosa che varia ampiamente da un sistema operativo all'altro: le specifiche arrivano a consigliare, come minimo comune denominatore, otto caratteri alfanumerici in maiuscolo.

Inoltre è stato definito il concetto di *utilizzo dichiarato* del file; questo per permettere (sui sistemi in grado di gestire, per esempio, le estensioni al nome) di proporre al giocatore una lista di file adatti durante la selezione. Per la precisione, Glk indica quattro possibili utilizzi:

SavedGame per contenere i dati di salvataggio della partita (utilizzata dai verbi *save* e *restore*);

Transcript per un file su cui inviare la trascrizione dell'avventura (verbo *script*);

InputRecord per un elenco di comandi dati dal giocatore da rieseguire (verbi *commands* e *replay*);

Data per file di *altro tipo*. La libreria standard non ne usa.

L'utilizzo dichiarato del file, in ogni caso, influisce solo sul *nome* del file, quello che in Glk viene identificato da una fileref.

Oltre all'utilizzo dichiarato, è necessario anche specificare se il file contiene (o conterrà) solo testo o dati binari. Questo al fine di creare un file eventualmente utilizzabile con gli strumenti di manipolazione testi della piattaforma in uso. Inutile dire che le conversioni applicate ad un file di testo *potrebbero* alterarne il contenuto: non necessariamente si rileggerà *esattamente* quel che vi è stato scritto originariamente.

Le linee guida delle specifiche consigliano la modalità testo solo per i file di trascrizione; le altre tipologie di file sono infatti intese per essere rilette in seguito dall'avventura stessa. Tuttavia un file contenente, per esempio, delle statistiche conclusive, dovrebbe essere scritto in modalità testo, se lo scopo è quello di creare un file leggibile dal giocatore.

Per tutti questi motivi, all'apertura di un file non ne viene specificato il *nome*. Al contrario, bisogna prima creare una fileref (uno degli oggetti opachi di Glk) che lo rappresenti; in ogni caso non è una questione complicata; ci sono solo quattro modi in cui è possibile creare una fileref:

- Richiedendo un file temporaneo (`glk_fileref_create_temp`);
- Chiedendo all'utente di scegliere (`glk_fileref_create_by_prompt`);
- Specificando esplicitamente un nome (`glk_fileref_create_by_name`), ma attenzione alle limitazioni della piattaforma in uso;
- Da una fileref esistente, conservando il nome ma con un utilizzo diverso (`glk_fileref_create_from_fileref`).

Il caso più comune sarà quello di richiedere all'utente il nome da utilizzare: in base all'utilizzo dichiarato (e la modalità che si prevede di utilizzare) il sistema molto probabilmente aprirà una dialog box filtrata in maniera opportuna; è necessario specificare anche la modalità (se in sola lettura o in lettura/scrittura) per permettere, per esempio, la digitazione di un nome *nuovo* nel caso il file debba essere scritto.

Un file temporaneo non ha un nome né una posizione ben definita; finché se ne possiede la fileref è accessibile, dopo la sua sorte è segnata (non necessariamente verrà rimosso dal disco ma comunque non sarà più accessibile agevolmente).

Specificare il nome da dare al file potrebbe sembrare una buona idea, inizialmente. Dopo aver considerato però le varie limitazioni presentate dalle varie piattaforme, forse converrà chiedere *comunque* il nome al giocatore. In ogni caso la funzione è presente, se qualcuno la volesse utilizzare. Attenzione al passaggio del nome in infglk (vedere la sezione 13.3 per il metodo corretto di specificare una stringa a Glk).

Per finire è anche possibile *duplicare* una fileref, per creare un file con un nome simile ma contenuti diversi; anche in questo caso ci sono dei particolari di cui tenere conto; le specifiche della sezione 6.1 raccontano tutta la storia.

Una volta in possesso del *nome* del file, è possibile aprirlo e lavorarci sopra. Le primitive a disposizione sono quelle classiche, utilizzate anche per l'output su finestra; l'input è simile (anche se non basato su eventi) ma chiunque fosse interessato deve solo riferirsi alle sezioni del Capitolo 5.

Una cosa importante: come tutti gli oggetti Glk anche le fileref devono essere distrutte, quando si è finito di utilizzarle. Dal punto di vista del file, la fileref è necessaria *solo* durante l'apertura.

Può essere in qualche caso utile tenere attiva la fileref per un tempo più lungo, per evitare di chiedere un'altra volta il nome all'utente, per esempio. In questo caso sarà necessario implementare `IdentifyGlkObject` opportunamente. La tecnica è la stessa vista per canali audio e finestre; in ogni caso la stessa libreria standard la utilizza per il file di trascrizione, se servisse un esempio.

❖ ❖ ❖

Anche i file aperti (gli stream per la precisione) devono essere identificati e riassociati; ma solo nel caso in cui rimangano aperti durante un *save/restore* o *undo*.

Tipicamente le operazioni su un file sono relativamente brevi e quindi ristrette all'interno di un turno; in questo caso non c'è da preoccuparsi, è impossibile che si presenti un ripristino in queste circostanze.

La libreria tuttavia è costretta a riassociare lo stream di salvataggio, durante il salvataggio stesso; la necessità deriva dall'interruzione di flusso creata dalla stessa istruzione `@save`: un salvataggio termina nel momento in cui viene ripristinato (è paradossale, ma è così anche nella Z-machine) e lo stream è *ancora* aperto quando il gioco riprende; ovviamente il riferimento non è più valido e deve essere riassociato.

◆ ◆ ◆

15

Gli esempi di Gull

Questo capitolo tratta in rassegna gli esempi presentati con il tutorial *Gull*; sfortunatamente, anche se menzionati, non sono spiegati nel tutorial stesso (e nonostante tutto utilizzano anche tecniche diverse).

Attenzione, non sono giochi nel senso tradizionale della parola! Sono principalmente esempi (pure abbastanza ridicoli, per certi versi) concepiti per mostrare l'utilizzo di funzionalità specifiche di Inform/glulx.

15.1. Download!

L'idea di questo esempio è di mostrare come utilizzare gli stili di testo e una barra di stato personalizzata.

Si tratta di una... ehm... simulazione di una tipica situazione dell'era pre-internet, quando i modem viaggiavano alla mostruosa velocità di 300 baud[1] e gli schermi erano in verde e nero (Figura 15.1).

```
Constant Story "Download!";
Constant Headline "^a␣demonstration␣of␣customized␣text␣in␣Glulx␣Inform
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣by␣Adam␣Cadre^";

Global pct = 94;
Replace DrawStatusLine;
Include "Parser";

Object LibraryMessages
   with
      before [;
         Score: rtrue;
      ];

Include "VerbLib";
Include "Infglk";
```

[1] Per chi ha visto almeno un terminale VideoTel, beh, quello è *molto* più veloce! Il VideoTel viaggiava all'incredibile velocità di 1200 baud...

Figura 15.1. Download!

Le solite dichiarazioni di inizio gioco: nome della storia, banner, variabili globali, messaggi della libreria e inclusioni. Il comando *score* viene disattivato in un modo alquanto poco elegante però...

Viene dichiarata una variabile `pct` per tener conto dello *stato di gioco* e dichiarato che si sostituirà la routine di libreria `DrawStatusLine` (in quanto personalizzata).

```
Object Den "Den"
   with description
      "Having␣~logged␣on~␣to␣a␣local␣~BBS~␣with␣your␣new␣300␣baud
␣␣␣␣␣␣~modem~␣--␣wow,␣all␣this␣futuristic␣terminology␣to␣keep␣track
␣␣␣␣␣␣of!␣--␣you␣are␣now␣busily␣~downloading~␣a␣computer␣game␣called
␣␣␣␣␣␣~Lunar␣Lander~.␣It's␣been␣over␣an␣hour,␣but␣it's␣almost␣done!",
   has light;
```

Il teatro della tragedia che sta per compiersi: la descrizione della stanza (la *tana*) serve anche come testo introduttivo in questo caso (da un esempio non si può chiedere molto di più!).

```
Object -> computer "computer"
   with article "your",
      name 'computer' 'screen' 'friendly' 'green' 'glow',
      describe "^Your␣computer␣screen␣emits␣a␣friendly␣green␣glow.",
```

L'unico oggetto con cui si può interagire: una delle due cose interessanti di questo listato. Nella proprietà `description` infatti...

```
      description [;
         print (g) "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣^";
         print (g) "␣␣␣DOWNLOADING:␣LUNRLNDR.EXE␣␣␣^";
         if (pct < 98) {
            print (g) "␣␣␣";
            glk_set_style(style_User1);
            print pct;
            glk_set_style(style_Normal);
            print (g) "%␣done...␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣^";
            print (g) "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣^";
```

```
      } else {
         print (g) "␣␣␣98%␣done...$H*#Q&P%%%␣␣␣␣␣␣␣^";
         print (g) "␣␣␣%␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣^";
         print (g) "␣␣␣%%%␣NO␣CARRIER␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣^";
         print (g) "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣^";
      }
```

... viene mostrato come effettuare un cambio di stile da manuale. Lo stile User1 verrà inizializzato per simulare uno schermo di terminale (verde su nero a spaziatura fissa). La funzione di comodo g è definita sotto come scorciatoia per cambiare stile e migliorare la leggibilità. In questo caso è importante mettere gli spazi in coda al testo in quando lo sfondo nero simula lo schermo del computer.[2]

```
      if (pct == 98) {
         deadflag = 3;
         "^Hmm.␣You␣seem␣to␣have␣thrown␣the␣~monitor~␣out␣the␣window.";
      }
      rtrue;
   ],
   each_turn [;
      if (pct < 98) pct++;
      if (pct == 98) "^The␣computer␣beeps.";
      rtrue;
   ],
has static;
```

La definizione dell'oggetto è completa. Fra l'altro è anche chiaro come finirà la storia no? Nulla di eccezionale da segnalare.

```
[ InitGlkWindow winrock;
   switch (winrock) {
      GG_MAINWIN_ROCK:
         glk_stylehint_set(wintype_TextBuffer, style_User1,
            stylehint_TextColor, $00FF00);
         glk_stylehint_set(wintype_TextBuffer, style_User1,
            stylehint_BackColor, $000000);
         glk_stylehint_set(wintype_TextBuffer, style_User1,
            stylehint_Proportional, 0);
   }
   rfalse;
];
```

Ecco una delle funzioni care a Inform/glulx: InitGlkWindow. Gli stili da applicare ad una finestra di testo devono essere configurati *prima* di creare la finestra stessa. La fase intermedia che si sfrutta permette di fare proprio questo: lo stile User1 viene settato nell'ordine:

- Con un colore del testo verde;
- un colore di sfondo nero;
- e per finire con un font a spaziatura fissa.

[2] Alcune versioni di garglk-mod possono sbagliare l'allineamento nella riga con ... se il font non è *ufficialmente* a spaziatura fissa.

Ovviamente l'ordine di definizione non è importante, ma deve essere tutto completato prima di ritornare dalla funzione, che termina con `rfalse` per continuare con la gestione di default (sarebbe stato utile ritornare `rtrue` se la creazione della finestra principale fosse stata non-standard).

Come si può notare nell'immagine garglk-mod mostra la prima riga (vuota) di altezza diversa. Il problema potrebbe essere corretto specificando il leading dello stile nel file di configurazione; sfortunatamente gli hint sono piuttosto limitativi.

```
[ Initialise;
   location = Den;
   rtrue;
];

[ DeathMessage;
   switch (deadflag) {
      3: print "You␣have␣failed␣to␣download␣Lunar␣Lander";
   }
];
```

Nulla di eccezionale da vedere... l'inizio e la fine, come da consuetudine.

```
[ DrawStatusLine;
   ! If we have no status window, we must not try to redraw it.
   if (gg_statuswin == 0)
      return;

   ! If there is no player location, we shouldn't try either.
   if (location == nothing || parent(player) == nothing)
      return;

   glk_set_window(gg_statuswin);
   glk_window_clear(gg_statuswin);
   glk_window_move_cursor(gg_statuswin, 5, 0);
   print "Download:␣", pct-1, "%␣done";
   glk_set_window(gg_mainwin);
];
```

Ecco la gestione di una linea di stato personalizzata (essendo di una sola linea non si è dovuto far altro in `InitGlkWindow`). Dopo i controlli scaramantici del caso, viene mostrato il testo alternativo (la forma `pct-1` è per compensare la differenza di tempismo fra la `each_turn` dell'oggetto e l'aggiornamento della barra di stato (una finezza).

Non viene applicato alcuno stile alla finestra di stato per cui viene utilizzato lo stile *Normal*; ma attenzione: lo stile corrente è specifico per ogni finestra! Anche variando lo stile corrente della finestra di stato non si va ad alterare quello in uso nella finestra principale; è comunque bene cercare di ripristinare lo stile dopo un cambio, se possibile.

```
[ g text;
   glk_set_style(style_User1);
   print (string) text;
   glk_set_style(style_Normal);
];
```

La funzione di comodo di cui si parlava sopra: mostra il testo passato nello stile *User1*, permettendo quindi la sintassi `print (g)"Testo";`.

Figura 15.2. Bipolar Bear


```
[ PrintRank; rtrue; ]; ! to suppress spurious period when game ends

Include "Grammar";
```

Altri dettagli per omettere il punteggio e la grammatica per concludere.

15.2. *Bipolar Bear*

Come è noto(?!) gli orsi polari sono migratori... l'eroe della nostra avventura però prende la migrazione un po' troppo sul serio!

In ogni caso è la situazione perfetta per mostrare come usare le risorse grafiche è mostrare delle immagini in una finestra dedicata. Il risultato è visibile in Figura 15.2

```
Constant Story "Bipolar␣Bear";
Constant Headline "^a␣demonstration␣of␣graphics␣window␣handling␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Glulx␣Inform␣by␣Adam␣Cadre^";

Constant GG_BIGWIN_ROCK 210;
Constant GG_SMALLWIN_ROCK 211;

Replace DrawStatusLine;

Global gg_bigwin = 0;
Global gg_smallwin = 0;
Global curr_pic;

Include "Parser";
Include "VerbLib";
Include "Infglk";
Include "bipolar.bli";
```

Le solite definizioni a inizio file: di particolare importanza le due costanti per identificare le *due* finestre grafiche che verranno create, i loro riferimenti e curr_pic per contenere la risorsa immagine da mostrare.

❖ ❖ ❖

Vengono create *due* finestre grafiche, anche se non è evidente; *tutta* l'area a disposizione di Glk deve far parte di una finestra, quindi anche l'area *libera* visibile sulla destra è una finestra. In questa particolare applicazione poteva anche essere utilizzata una finestra vuota; Adam Cadre ha invece preferito sfruttare lo spazio per un'altra finestra grafica.

◆ ◆ ◆

Il file bipolar.bli viene generato dal blorbificatore e contiene le costanti identificative delle risorse.

```
Object Arctic "The␣Arctic"
   with description
      "It's␣pretty␣darn␣cold␣here.␣Warmer␣climes␣lie␣to␣the␣south.",
      s_to [;
         curr_pic = Antarctic_pic;
         MyRedrawGraphicsWindows();
         PlayerTo(Antarctic);
         rtrue;
      ],
   has light;

Object Antarctic "The␣Antarctic"
   with description
      "Whoops!␣You␣walked␣all␣the␣way␣to␣Antarctica.␣Guess␣you␣just
␣␣␣␣␣␣don't␣know␣when␣to␣quit.␣Most␣of␣the␣rest␣of␣the␣world␣lies␣to
␣␣␣␣␣␣the␣north.",
      n_to [;
         curr_pic = Arctic_pic;
         MyRedrawGraphicsWindows();
         PlayerTo(Arctic);
         rtrue;
      ],
   has light;
```

Le due locazioni a disposizione del giocatore; il cambio immagine viene gestito durante lo spostamento; la scelta è discutibile in quanto, in un'avventura di proporzioni tipiche, ogni spostamento dovrebbe contenere un cambio immagine alla *destinazione*. Non molto pratico, secondo me. Una migliore soluzione potrebbe essere una proprietà della stanza e gestire, ad ogni turno, un eventuale cambio di posizione.

È interessante notare anche che gli orsi polari sono anche *ottimi nuotatori*!

```
[ Initialise;
   location = Arctic;

   if (~~glk_gestalt(gestalt_Graphics, 0)) {
      print "^^^This␣interpreter␣doesn't␣support␣graphics!␣To␣play␣this
␣␣␣␣␣␣␣␣␣demo,␣you␣need␣one␣that␣does.^";
   }

   if (gg_bigwin == 0) {
      gg_bigwin = glk_window_open(gg_mainwin,
         (winmethod_Above+winmethod_Fixed), 279, wintype_Graphics,
```

```
        GG_BIGWIN_ROCK);
    }
    if (gg_smallwin == 0 && gg_bigwin) {
        gg_smallwin = glk_window_open(gg_bigwin,
        (winmethod_Left+winmethod_Fixed), 425, wintype_Graphics,
        GG_SMALLWIN_ROCK);
    }
    if (gg_statuswin) {
        glk_window_close(gg_statuswin, 0);
    }

    curr_pic = Arctic_pic;
    MyRedrawGraphicsWindows();

];
```

Nella Initialise viene effettuato il completamento del layout dello schermo (dopo le due finestre iniziali vengono aperte le due finestre grafiche) e la locazione iniziale (con immagine relativa). Notare anche l'utilizzo del selettore gestalt_Graphics per rendere *obbligatorio* il supporto grafico; tuttavia è clemente, si tratta solo di un avviso al giocatore (i prossimi esempi utilizzeranno metodi più persuasivi!).

La parte centrale costituisce il succo dell'esempio: la finestra principale viene prima divisa orizzontalmente per ottenere una *fetta* alta 279 pixel. Questa viene poi divisa verticalmente per ottenere una finestra grande esattamente 425×279 pixel. Lo spazio residuo rimane nella finestra piccola: cioè quel che rimane di gg_bigwin; la scelta dei nomi non è stata particolarmente felice.

Per finire viene chiusa la finestra di stato, se esistente (sarebbe anche buona idea forzare gg_statuswin=0 dopo averla chiusa, per evitare di utilizzarla inavvertitamente. Ma in questo caso DrawStatusLine viene *vuotata* quindi non si corrono rischi.

Un'alternativa più elegante potrebbe essere stata la creazione dell'intero layout in InitGlkWindow, in fase 0, evitando in primo luogo la creazione della finestra di stato. Ma il risultato netto sarebbe lo stesso.

```
[ IdentifyGlkObject phase type ref rock;
    if (phase == 0) {
        ! Zero out references to our objects.
        gg_bigwin = 0;
        gg_smallwin = 0;
        return;
    }

    if (phase == 1) {
        switch (type) {
            0: ! it's a window
                switch (rock) {
                    GG_BIGWIN_ROCK: gg_bigwin = ref;
                    GG_SMALLWIN_ROCK: gg_smallwin = ref;
                }
            1: ! it's a stream
                ! But we don't create any.
            2: ! it's a fileref
                ! But we don't create any.
        }
        return;
    }
```

```
    if (phase == 2) {
        ! We just changed game state, and the contents of the graphics
        ! windows are probably wrong. Insert code here to redraw them.
        MyRedrawGraphicsWindows();
    }
];
```

Il codice classico in IdentifyGlkObject per riassociare le finestre, semplicemente esteso per *due* finestre. I dettagli sono nella sezione 14.14.

```
[ HandleGlkEvent ev context;
    context = 0; ! suppress ignored warning
    switch (ev-->0) {
      evtype_Redraw, evtype_Arrange:
          MyRedrawGraphicsWindows();
    }
];
```

Bisogna gestire gli eventi per ridisegnare nelle finestre grafiche. Questo è il modo standard per farlo; non ci interessano altri eventi.

```
[ MyRedrawGraphicsWindows;
   if (gg_smallwin && glk_gestalt(gestalt_Graphics, 0)) {
      glk_image_draw(gg_smallwin, curr_pic, 0, 0);
   }
];

[ DrawStatusLine; ];

Include "Grammar";
```

Per finire la funzione che disegna l'immagine corrente e la forzatura di DrawStatusLine per evitare problemi (oltre all'immancabile grammatica).

Dato che la dimensione della finestra è costante (per come è stata creata!) e le immagini sono tutte di questa dimensione (e non hanno parti trasparenti) non è necessario pulire la finestra a priori.

15.3. Sieppo's Diner

Per questo esempio nessuna immagine: sfortunatamente non è possibile allegare un campione musicale, dato che è tutto incentrato sull'utilizzo delle risorse audio...

```
Constant Story "Seppo's␣Diner";
Constant Headline "^a␣demonstration␣of␣sound␣in␣Glulx␣Inform
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣by␣Adam␣Cadre^";

Constant GG_MUSICCHAN_ROCK 410;

Global gg_musicchan = 0;
Global current_music = 0;

Include "Parser";
Include "VerbLib";
Include "Infglk";
Include "jukebox.bli";
```

L'inizio è praticamente identico a quello di *Bipolar Bear*. La sola differenza è che invece delle costanti e variabili necessarie per tener conto di una finestra grafica, qui si dichiarano quelle per un canale audio. Anche in questo

caso jukebox.bli contiene le costanti per riferirsi alle risorse ed è generato dal blorbificatore.

```
Object Diner "Seppo's Diner"
   with description "It's been years since Seppo's Diner closed its
       doors. It seems like only yesterday that old Seppo was working
       the grill -- every day you'd come in and he'd have burned yet
       another finger, and the two of you would laugh and laugh... but
       then came the lawnmower accident... poor, poor Seppo. Now the
       grill and the ovens are gone, as is all the seating... all that
       remains is a jukebox full of Finnish techno, and even that is
       on its last legs.",
   has light;

Object -> jukebox "jukebox"
   with name 'jukebox',
      number 0,
      describe [;
         new_line;
         switch (self.number) {
            0: "The jukebox is silent.";
            1: "The jukebox plays the ominous ~Tune 1~ by Riku
                Nuottajarvi.";
            2: "The jukebox plays the strange ~Tune 2~ by Riku
                Nuottajarvi.";
            3: "The jukebox plays the peppy ~Tune 4~ by Riku
                Nuottajarvi.";
            4: "The jukebox plays the stirring ~Tune 6~ by Riku
                Nuottajarvi.";
         }
      ],
      description "The jukebox is on its last legs. It doesn't accept
          coins anymore; the only way to get it to play anything is to
          hit it, a la the Fonz. And you never know what you'll get --
          though chances are good it'll be something by Riku Nuottajarvi,
          the only musician Seppo ever liked.",
```

L'ambientazione e l'oggetto con cui interagire. La proprietà number in un certo senso *segue* la variabile globale current_music, per adattare la descrizione del jukebox.

```
      before [x;
         Attack:
            x = random(5) - 1;
            while (x == self.number) x = random(5) - 1;
            self.number = x;
            switch (self.number) {
               0: current_music = 0;
                  MyRestartMusicChannel();
                  "The jukebox falls silent. Let's hope you didn't break
                      it permanently!";
               1: current_music = Tune1;
                  MyRestartMusicChannel();
                  "The jukebox begins playing the ominous ~Tune 1~ by
                      Riku Nuottajarvi.";
               2: current_music = Tune2;
                  MyRestartMusicChannel();
                  "The jukebox begins playing the strange ~Tune 2~ by
                      Riku Nuottajarvi.";
               3: current_music = Tune4;
                  MyRestartMusicChannel();
                  "The jukebox begins playing the peppy ~Tune 4~ by Riku
                      Nuottajarvi.";
               4: current_music = Tune6;
                  MyRestartMusicChannel();
                  "The jukebox begins playing the stirring ~Tune 6~ by
                      Riku Nuottajarvi.";
            }
      ],
   has static;
```

Come descritto nell'introduzione, il jukebox opera per mezzo di colpo fonzarelliano... viene scelta a caso una traccia *diversa* da quella corrente (il ciclo while serve a questo) e poi viene riavviato il canale audio (anche in questo caso il codice è assolutamente paragonabile a quello relativo ad una finestra grafica).

```
[ Initialise;
   if (~~gg_musicchan)
      gg_musicchan = glk_schannel_create(GG_MUSICCHAN_ROCK);
   location = Diner;
];

[ MyRestartMusicChannel;
   if (gg_musicchan) {
      if (current_music == 0) glk_schannel_stop(gg_musicchan);
      else glk_schannel_play_ext(gg_musicchan, current_music, -1, 0);
   }
];
```

Initialise a parte (dove viene creato il canale audio se non è già presente a causa di un restart), la funzione MyRestartMusicChannel aggiorna lo stato del canale audio. Essendo musica di sottofondo è stato specificato -1 come numero di ripetizioni.

Ma attenzione: cosa succede se l'interprete corrente non supporta la riproduzione audio? Molto semplicemente glk_schannel_create ritornerà 0. E, per l'appunto, gg_musicchan viene sempre confrontato con 0 prima di essere utilizzato (o anche con GLK_NULL, che è la stessa cosa).

Questo perché la musica di sottofondo è una parte *facoltativa*, in questo caso, e anche se non ci fosse non succederebbe nulla di grave. Per contro in *Bipolar Bear* il supporto grafico era richiesto come obbligatorio (anche se solo come avvertimento). La stessa tecnica potrebbe essere applicata in questa funzione.[3]

❖ ❖ ❖

A dire il vero ci si aspetterebbe che il jukebox suoni solo una volta la canzone... ma in questo caso sarebbe necessario gestire l'evento di notifica audio (il quarto parametro) e, probabilmente, interrompere l'input dell'utente per segnalare la situazione. Inoltre si inserirebbe una situazione in *tempo reale* che non necessariamente viene ben vista in una avventura testuale.

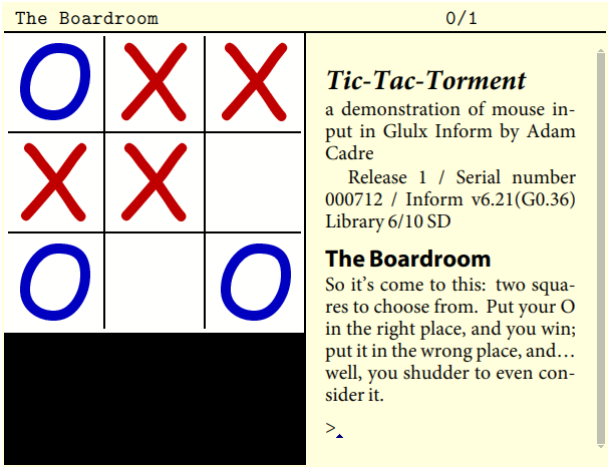
Presumiamo, quindi, che un jukebox che funziona a sberle ripeta in continuazione lo stesso disco... ragionevole, no?

◆ ◆ ◆

```
[ IdentifyGlkObject phase type ref rock res id;
    if (phase == 0) {
        ! Zero out references to our objects.
        gg_musicchan = 0;
        return;
    }
```

[3] In questo caso è anche necessario verificare che siano supportati i MOD!

Figura 15.3. Tic-Tac-Torment




```
    if (phase == 1) {
        ! Nothing to do for sound in this phase.
        return;
    }

    if (phase == 2) {
        ! Iterate through all the existing channels -- there should
        ! be either none or one -- and identify ours.
        id = glk_schannel_iterate(0, gg_arguments);
        while (id) {
            switch (gg_arguments-->0) {
                GG_MUSICCHAN_ROCK: gg_musicchan = id;
            }
            id = glk_schannel_iterate(id, gg_arguments);
        }

        ! Now, we just changed game state, so it might be necessary
        ! to set a different piece playing on the jukebox.
        MyRestartMusicChannel();
    }
];

Include "Grammar";
```

Questo è il codice di riassociazione necessario per un canale audio. Durante la fase 2 di IdentifyGlkObject si itera sui canali in quanto la libreria standard non lo fa in fase 1.

154. Tic-Tac-Torment

Ecco come *perdere* al gioco. In maniera sanguinosa, oltretutto... Per consolazione viene mostrato l'uso degli eventi del mouse (oltre ad un ripasso sulle finestre grafiche, Figura 15.3).

L'orribile blocco nero è *voluto*. Forse per mostrare come cambiare il colore di sfondo, o forse per ricordare la triste fine che capita ai giocatori perdenti...

```
Constant Story "Tic-Tac-Torment";
Constant Headline "^a␣demonstration␣of␣mouse␣input␣in␣Glulx␣Inform
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣by␣Adam␣Cadre^";

Constant GG_TTTWIN_ROCK 210;

Global gg_tttwin = 0;

Include "Parser";

Object LibraryMessages
   with
      before [;
         Score: if (deadflag == 0) "Your␣side␣is␣O,␣your␣score␣is␣0.";
                else rtrue;
      ];

Include "VerbLib";
Include "Infglk";
Include "ttt.bli";

Object Boardroom "The␣Boardroom"
   with description
      "So␣it's␣come␣to␣this:␣two␣squares␣to␣choose␣from.␣Put␣your␣O
␣␣␣␣␣␣in␣the␣right␣place,␣and␣you␣win;␣put␣it␣in␣the␣wrong␣place,
␣␣␣␣␣␣and...␣well,␣you␣shudder␣to␣even␣consider␣it.",
   has light;
```

Ormai nulla che sia degno di commenti; si capisce però già che verrà utilizzata una finestra aggiuntiva. E delle risorse (il file `ttt.bli`).

```
Object -> block
   with name 'block',
      before [;
         Try: glk_image_draw(gg_tttwin, TTO, 207, 105);
              glk_image_draw(gg_tttwin, TTX, 107, 205);
              deadflag = 1;
              "^You␣go␣for␣the␣block␣in␣the␣center␣right␣square␣--
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣and␣the␣X-Ecutioner␣gleefully␣plays␣at␣bottom␣center,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣securing␣the␣win.␣Having␣lost,␣your␣life␣is␣forfeit.
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣The␣X-Ecutioner␣produces␣an␣axe;␣it␣is␣the␣last␣thing
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣you␣see.";
      ],
   has scenery;

Object -> win
   with name 'win',
      before [;
         Try: glk_image_draw(gg_tttwin, TTO, 107, 205);
              deadflag = 2;
              "^You␣gleefully␣play␣at␣bottom␣center,␣securing␣the␣win.
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣A␣fanfare␣of␣trumpets␣announces␣your␣triumph!␣Huzzah!";
      ],
   has scenery;

Object -> invalid
   with name 'invalid',
      before [;
         Try: "^That's␣not␣an␣empty␣square!";
      ],
   has scenery;
```

Tre oggetti di comodo, che rappresentano le scelte possibili (un po' come viene fatto con i menù). Si possono vedere gli aggiornamenti allo schema grafico (le `glk_image_draw`) a seconda della mossa che verrà scelta. La selezione del giocatore verrà mappata sul verbo *Try*.

```
[ Initialise;
```

```
    if (~~glk_gestalt(gestalt_Graphics, 0)) {
        print "^^^This␣interpreter␣doesn't␣support␣graphics!␣To␣play␣this
␣␣␣␣␣␣␣␣␣demo,␣you␣need␣one␣that␣does.^";
        KeyCharPrimitive();
        quit;
    }
    if (~~glk_gestalt(gestalt_MouseInput, wintype_Graphics)) {
        print "^This␣interpreter␣doesn't␣support␣mouse␣input!␣To␣play
␣␣␣␣␣␣␣␣␣this␣demo,␣you␣need␣one␣that␣does.^";
        KeyCharPrimitive();
        quit;
    }

    if (~~gg_tttwin) {
        gg_tttwin = glk_window_open(gg_mainwin,
            (winmethod_Left+winmethod_Fixed), 300,
            wintype_Graphics, GG_TTTWIN_ROCK);
    }

    glk_request_mouse_event(gg_tttwin);

    MyRedrawGraphicsWindows();

    location = Boardroom;
];
```

La solita Initialise dove è obbligatorio avere sia il supporto per la grafica che il mouse. Inoltre viene aperta una finestra grafica (larga 300 pixel) sulla sinistra e su questa viene richiesto un evento del mouse.

❖ ❖ ❖

Nella gestione dei gestalt falliti c'è un particolare anomalo: prima di eseguire quit per uscire, viene chiamata KeyCharPrimitive per richiedere la pressione di un tasto.

Questo, da specifiche, non è necessario. Anzi. La maggior parte delle implementazioni Glk richiede di premere un tasto all'uscita: le specifiche *impongono* che l'ultimo testo mostrato sia visibile all'utente!

Per questo motivo, su molte implementazioni, in caso di errore sarà necessario premere *due* tasti per uscire. Non so se sia una svista o se effettivamente qualche interprete abbia dei bug in questo senso...

◆ ◆ ◆

```
[ IdentifyGlkObject phase type ref rock;
    if (phase == 0) { ! Zero out references to our objects.
        gg_tttwin = 0;
        return;
    }

    if (phase == 1) { ! Reset our windows, streams and filerefs.
        switch (type) {
            0: ! it's a window
                switch (rock) {
                    GG_TTTWIN_ROCK: gg_tttwin = ref;
                }
            1: ! it's a stream
                    ! But we don't create any.
            2: ! it's a fileref
                    ! But we don't create any.
        }
        return;
    }
```

```
    if (phase == 2) { ! Update our objects.
        MyRedrawGraphicsWindows();
    }
];
```

La riassegnazione della finestra grafica. Copia & incolla sono utili in questo caso...

```
[ HandleGlkEvent ev context abortres newcmd cmdlen;
    switch (ev-->0) {
        evtype_Redraw, evtype_Arrange:
            MyRedrawGraphicsWindows();
\end{intermission}
La |HandleGlkEvent| comincia nel modo tipico per ridisegnare le finestre
grafiche\dots
\index{glk\_cancel\_line\_event!esempio Inform}
\index{PrintAnyToArray!esempio Inform}
\index{glk\_set\_style!esempio Inform}
\index{input a linea!simulazione}
\begin{intermission}
        evtype_MouseInput:
            glk_request_mouse_event(gg_tttwin);
            if (ev-->2 >= 200 && ev-->2 < 300
               && ev-->3 >= 100 && ev-->3 < 200) {
                glk_cancel_line_event(gg_mainwin, 0);
                newcmd = "try␣block";
                cmdlen = PrintAnyToArray(abortres+WORDSIZE,
                    INPUT_BUFFER_LEN-WORDSIZE, newcmd);
                abortres-->0 = cmdlen;
                glk_set_style(style_Input);
                print "(mouse␣click)";
                glk_set_style(style_Normal);
                new_line;
                return 2;
            }
            if (ev-->2 >= 100 && ev-->2 < 200
               && ev-->3 >= 200 && ev-->3 < 300) {
                glk_cancel_line_event(gg_mainwin, 0);
                newcmd = "try␣win";
                cmdlen = PrintAnyToArray(abortres+WORDSIZE,
                    INPUT_BUFFER_LEN-WORDSIZE, newcmd);
                abortres-->0 = cmdlen;
                glk_set_style(style_Input);
                print "(mouse␣click)";
                glk_set_style(style_Normal);
                new_line;
                return 2;
            }
            else {
                glk_cancel_line_event(gg_mainwin, 0);
                newcmd = "try␣invalid";
                cmdlen = PrintAnyToArray(abortres+WORDSIZE,
                    INPUT_BUFFER_LEN-WORDSIZE, newcmd);
                abortres-->0 = cmdlen;
                glk_set_style(style_Input);
                print "(mouse␣click)";
                glk_set_style(style_Normal);
                new_line;
                return 2;
            }
    }
];
```

...ecco finalmente qualcosa di nuovo! Questi tre blocchi (sostanzialmente identici) convertono i click del mouse in verbi *Try* da passare al parser.

Le parti interessanti sono, ovviamente:

- Il riarmo degli eventi del mouse;
- il confronto delle coordinate del click (`ev-->2` e `ev-->3`) per delimitare delle aree rettangolari;
- l'utilizzo di `glk_cancel_line_event` e `PrintAnyToArray` per forzare il comando in esecuzione al parser, ritornando 2.

❖ ❖ ❖

Tecnicamente c'è un bug in questa implementazione: dato che viene simulato l'input da tastiera, è sufficiente (per esempio) digitare al prompt *try win* per vincere, senza nemmeno sapere dove cliccare!

Certo, chi si sognerebbe di scrivere qualcosa del genere, però?

Ma non è l'unico bug in questo esempio...

◆ ◆ ◆

```
[ MyRedrawGraphicsWindows;
    if (gg_tttwin==0) rtrue;
    glk_window_set_background_color(gg_tttwin, $000000);
    glk_window_clear(gg_tttwin);
    glk_window_fill_rect(gg_tttwin, $FFFFFF, 0, 0, 300, 300);
    glk_window_fill_rect(gg_tttwin, $000000, 99, 4, 2, 292);
    glk_window_fill_rect(gg_tttwin, $000000, 199, 4, 2, 292);
    glk_window_fill_rect(gg_tttwin, $000000, 4, 99, 292, 2);
    glk_window_fill_rect(gg_tttwin, $000000, 4, 199, 292, 2);

    glk_image_draw(gg_tttwin, TTX, 107, 5);
    glk_image_draw(gg_tttwin, TTX, 207, 5);
    glk_image_draw(gg_tttwin, TTX, 7, 105);
    glk_image_draw(gg_tttwin, TTX, 107, 105);

    glk_image_draw(gg_tttwin, TTO, 7, 5);
    glk_image_draw(gg_tttwin, TTO, 7, 205);
    glk_image_draw(gg_tttwin, TTO, 207, 205);
];
```

Il ridisegno della finestra: viene utilizzato uno sfondo nero (per qualche strano motivo...), l'area della finestra viene cancellata (in modo che sia *tutto nero* e per finire viene disegnata l'area di gioco.

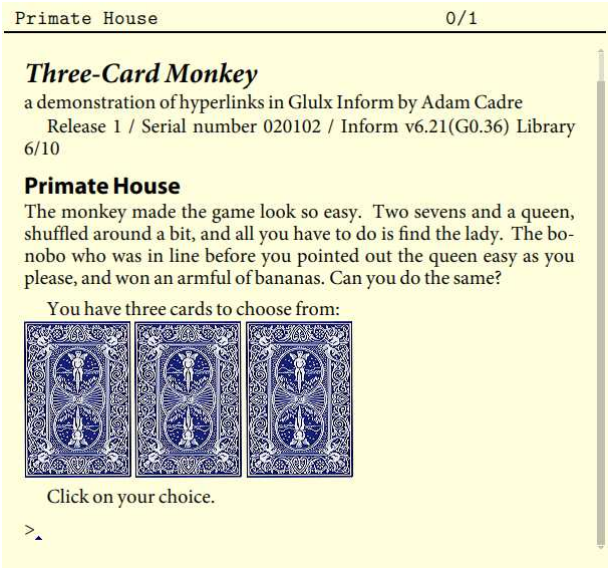
Lo sfondo del *campo di gioco* viene riempito in bianco, mentre le righe dello schema sono realizzate con riquadri neri (sfortunatamente non è possibile disegnare linee diagonali, a meno di non farlo pixel per pixel con un ciclo).

Per finire vengono mostrati tutti i simboli della situazione.

❖ ❖ ❖

È evidente ora il problema? Vengono sempre mostrati i simboli della situazione *iniziale*. Anche in un demo banale come questo, dopo avere effettuato la mossa (che si sia vinto o perso non importa) si arriva al prompt ricarica/ricomincia/annulla. E *in quel momento* il giocatore può ridimensionare la finestra... facendo scomparire i simboli aggiunti!

Figura 15.4. Three-Card Monkey



Certo, in un demo di questa portata non è importante, ma attenzione a questi casi limite.

◆ ◆ ◆

```
[ PrintRank; rtrue; ]; ! to suppress spurious period at end of game

Include "Grammar";

[ TrySub; "You␣can't␣try␣that."; ];

Verb "try" * noun -> Try;
```

Un verbo e poco più.

15.5. *Three-Card Monkey*

Ancora sul gioco d'azzardo... chi non ha mai *perso* al gioco delle tre carte? Anche contro una scimmia? Vedere la Figura 15.4.

Questo esempio su garglk non funziona del tutto correttamente, in quanto vengono utilizzati i collegamenti (hyperlink). È però anche utile per vedere come inserire immagini in un buffer di testo (la finestra principale, in questo caso); si può comunque digitare *try left*, *try center* o *try right* per proseguire con la selezione (il motivo è la tecnica impiegata, come in *Tic-Tac-Torment*).

```
Constant Story "Three-Card␣Monkey";
Constant Headline "^a␣demonstration␣of␣hyperlinks␣in␣Glulx␣Inform
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣by␣Adam␣Cadre^";

Include "Parser";

Object LibraryMessages
    with
```

```
      before [;
         Score: rtrue;
      ];

Include "VerbLib";
Include "Infglk";
Include "monkey.bli";

Object House "Primate␣House"
   with description
      "The␣monkey␣made␣the␣game␣look␣so␣easy.␣Two␣sevens␣and␣a
␣␣␣␣␣␣queen,␣shuffled␣around␣a␣bit,␣and␣all␣you␣have␣to␣do␣is
␣␣␣␣␣␣find␣the␣lady.␣The␣bonobo␣who␣was␣in␣line␣before␣you␣pointed
␣␣␣␣␣␣out␣the␣queen␣easy␣as␣you␣please,␣and␣won␣an␣armful␣of
␣␣␣␣␣␣bananas.␣Can␣you␣do␣the␣same?",
   has light;
```

Nulla degno di nota. Questa volta non ci sono variabili aggiuntive in quanto tutto svolge all'interno della finestra principale. Tuttavia monkey.bli contiene i numeri delle risorse grafiche utilizzate.

```
Object -> cards
   with describe [;
      print "^You␣have␣three␣cards␣to␣choose␣from:";
      if (glk_gestalt(gestalt_Graphics, 0)) {
         new_line;
         glk_set_hyperlink(1);
         glk_image_draw(gg_mainwin, Back, imagealign_InlineUp, 0);
         glk_set_hyperlink(0);
         print "␣";
         glk_set_hyperlink(2);
         glk_image_draw(gg_mainwin, Back, imagealign_InlineUp, 0);
         glk_set_hyperlink(0);
         print "␣";
         glk_set_hyperlink(3);
         glk_image_draw(gg_mainwin, Back, imagealign_InlineUp, 0);
         glk_set_hyperlink(0);
      }
      else {
         print "␣the␣";
         glk_set_hyperlink(1);
         print "left";
         glk_set_hyperlink(0);
         print "␣card,␣the␣";
         glk_set_hyperlink(2);
         print "center";
         glk_set_hyperlink(0);
         print "␣card,␣and␣the␣";
         glk_set_hyperlink(3);
         print "right";
         glk_set_hyperlink(0);
         "␣card.";
      }
      "^^Click␣on␣your␣choice.";
   ];
```

Oltre a mostrare come definire i collegamenti, questa funzione gestisce in maniera alternativa la mancanza di funzionalità grafiche, visualizzando del testo sostitutivo (viene utilizzata la proprietà describe per mostrare le scelte fin da subito, nella descrizione della locazione).

Se la grafica è disponibile vengono mostrati tre dorsi in linea, separati da spazi; la libreria garglk originale non supportava correttamente le immagini

in linea quindi le mostra *una sotto l'altra*. Come da specifiche, il link è assegnato anche alle immagini.

Sfortunatamente il test gestalt *non è corretto*. Quello che interessa, più che sapere se la libreria supporta la grafica, è se è in grado di visualizzare immagini in una finestra di testo! Il test corretto per questa circostanza è quindi `glk_gestalt(gestalt_DrawImage, wintype_TextBuffer)`. Anche se qui viene eseguito ogni volta tipicamente si utilizzano delle variabili globali per sapere quel che è disponibile.

❖ ❖ ❖

Ad essere pignoli, lo stesso si poteva dire anche per *Tic-Tac-Torment*. Ma se si riesce ad aprire una finestra grafica, a dire il vero, *necessariamente* sarà possibile usarla per mostrare delle immagini, no?

Teoricamente nessuno ce lo garantisce, ma sarebbe poco sensata una finestra in grado di mostrare *solo rettangoli colorati*...

◆ ◆ ◆

In questo caso l'allineamento prescelto per le immagini non è significativo, visto che non c'è testo sulla stessa linea.

```
Object -> left "left␣card"
   with name 'left' 'card',
      before [;
         Try:
         if (glk_gestalt(gestalt_Graphics, 0)) {
             glk_image_draw(gg_mainwin, Club, imagealign_InlineUp, 0);
             print "␣";
             glk_image_draw(gg_mainwin, Back, imagealign_InlineUp, 0);
             print "␣";
             glk_image_draw(gg_mainwin, Back, imagealign_InlineUp, 0);
             new_line;
         }
         deadflag = 3;
         "^^The␣seven␣of␣clubs!␣That␣monkey␣has␣outsmarted␣you␣again!";
      ],
   has scenery;

Object -> center "center␣card"
   with name 'center' 'card',
      before [;
         Try:
         if (glk_gestalt(gestalt_Graphics, 0)) {
             glk_image_draw(gg_mainwin, Back, imagealign_InlineUp, 0);
             print "␣";
             glk_image_draw(gg_mainwin, Heart, imagealign_InlineUp, 0);
             print "␣";
             glk_image_draw(gg_mainwin, Back, imagealign_InlineUp, 0);
             new_line;
         }
         deadflag = 3;
         "^^The␣seven␣of␣hearts!␣That␣monkey␣has␣outsmarted␣you␣again!";
      ],
   has scenery;

Object -> right "right␣card"
   with name 'right' 'card',
      before [;
         Try:
         if (glk_gestalt(gestalt_Graphics, 0)) {
```

```
            glk_image_draw(gg_mainwin, Back, imagealign_InlineUp, 0);
            print "␣";
            glk_image_draw(gg_mainwin, Back, imagealign_InlineUp, 0);
            print "␣";
            glk_image_draw(gg_mainwin, Spade, imagealign_InlineUp, 0);
            new_line;
        }
        deadflag = 3;
        "^^The␣seven␣of␣spades!␣That␣monkey␣has␣outsmarted␣you␣again!";
    ],
  has scenery;
```

Le tre carte... ovviamente *non c'è traccia della donna*. Tutti sanno che è in mano alla scimmia no?

Il verbo *Try* come la funzione describe gestisce *quasi* correttamente l'assenza di supporto grafico.

```
[ Initialise;
   if (~~glk_gestalt(gestalt_Hyperlinks, 0)) {
      print "^^^This␣interpreter␣doesn't␣support␣hyperlinks!␣To␣play␣this
␣␣␣␣␣␣␣␣demo,␣you␣need␣one␣that␣does.^";
      KeyCharPrimitive();
      quit;
   }

   glk_request_hyperlink_event(gg_mainwin);

   location = House;
];
```

La Initialise questa volta si assicura che sia presente il supporto per i collegamenti e ne richiede un evento per la finestra principale.

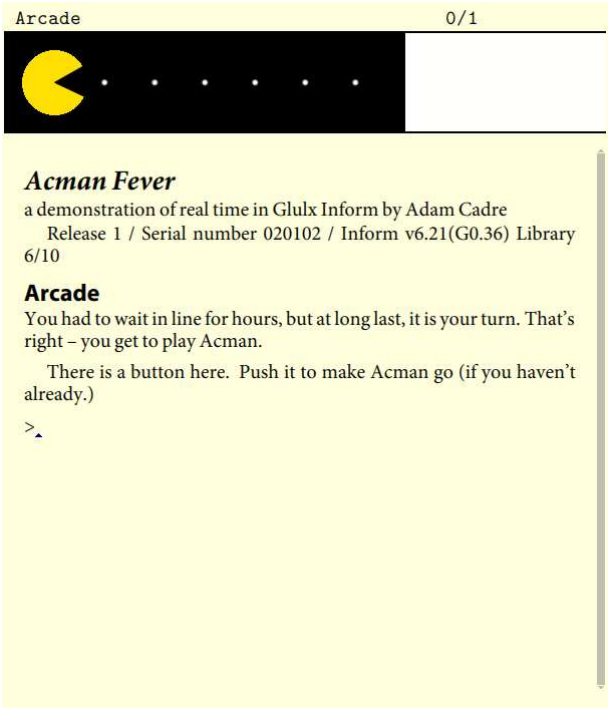
```
[ HandleGlkEvent ev context abortres newcmd cmdlen;
   switch (ev-->0) {
      evtype_Hyperlink:
         glk_cancel_line_event(gg_mainwin, 0);
         switch (ev-->2) {
            1: newcmd = "try␣left";
            2: newcmd = "try␣center";
            3: newcmd = "try␣right";
         }
         cmdlen = PrintAnyToArray(abortres+WORDSIZE,
            INPUT_BUFFER_LEN-WORDSIZE, newcmd);
         abortres-->0 = cmdlen;
         new_line;
         return 2;
   }
];
```

La gestione dei collegamenti è un po' più semplice di quella del mouse. Almeno non bisogna calcolare tutte quelle coordinate; il valore del link è stato specificato in fase di visualizzazione.

La tecnica di sostituzione della linea di comando, come per *Tic-Tac-Torment* permette di digitare il comando *senza* utilizzare il collegamento. In questo caso (nonostante sia comunque un bug) può essere un esempio di come gestire la *mancanza* degli hyperlink.

```
[ DeathMessage;
   switch (deadflag) {
      3: print "You␣have␣lost";
   }
```

Figura 15.5. Acman Fever




```
];

[ PrintRank; rtrue; ]; ! to suppress spurious period at end of game

Include "Grammar";

[ TrySub; "You␣can't␣try␣that."; ];

Verb "try" * noun -> Try;
```

Il triste destino di chi sfida una scimmia alle tre carte e il verbo *Try* utilizzato per le carte.

15.6. Acman Fever

Un'accurata diversione per evitare copyright e marchi registrati (anche Ackman è già stato preso... è un personaggio di Akira Toriyama).

In ogni caso, è la volta di mostrare il funzionamento del timer. In congiunzione con una animazione. Circa. In Figura 15.5

```
Constant Story "Acman␣Fever";
Constant Headline "^a␣demonstration␣of␣real␣time␣in␣Glulx␣Inform
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣by␣Adam␣Cadre^";

Constant GG_ACMANWIN_ROCK 210;

Global gg_acmanwin = 0;
Global acman_count = 0;
```

```
Include "Parser";
Include "VerbLib";
Include "Infglk";
Include "acman.bli";

Object Arcade "Arcade"
   with description
      "You␣had␣to␣wait␣in␣line␣for␣hours,␣but␣at␣long␣last,␣it␣is
␣␣␣␣␣␣your␣turn.␣That's␣right␣--␣you␣get␣to␣play␣Acman.",
   has light;

Object -> button "button"
   with name 'button',
      describe "^There␣is␣a␣button␣here.␣Push␣it␣to␣make␣Acman␣go
␣␣␣␣␣␣␣␣␣(if␣you␣haven't␣already.)",
      description "There's␣really␣nothing␣more␣to␣it␣than␣was␣already
␣␣␣␣␣␣␣␣␣stated.",
```

Una finestra grafica, delle risorse, una locazione e un pulsante...

```
      before [;
         Push: if (self has general) "Acman␣has␣left␣the␣building.";
            give self general;
            glk_request_timer_events(200);
            "Go,␣Acman,␣go!";
      ],
   has static;
```

...da premere! Anche se solo per una volta (come imposto dal controllo su general). La cosa interessante è che viene avviato il timer di Glk, con un periodo di 200 millisecondi.

```
[ Initialise;
   if (~~glk_gestalt(gestalt_Graphics, 0)) {
      print "^^^This␣interpreter␣doesn't␣support␣graphics!␣To␣play␣this
␣␣␣␣␣␣␣␣␣demo,␣you␣need␣one␣that␣does.^";
      KeyCharPrimitive();
      quit;
   }
   if (~~glk_gestalt(gestalt_Timer, 0)) {
      print "^This␣interpreter␣doesn't␣support␣real␣time!␣To␣play
␣␣␣␣␣␣␣␣␣this␣demo,␣you␣need␣one␣that␣does.^";
      KeyCharPrimitive();
      quit;
   }

   if (~~gg_acmanwin) {
      gg_acmanwin = glk_window_open(gg_mainwin,
         (winmethod_Above+winmethod_Fixed), 100,
         wintype_Graphics, GG_ACMANWIN_ROCK);
   }

   MyRedrawGraphicsWindows();

   location = Arcade;
   acman_count = First_Frame;
];
```

La solita inizializzazione... ci vuole la grafica, il timer ed una finestra alta 100 pixel. Inoltre viene inizializzata la variabile che tiene conto dello stato dell'animazione.

```
[ IdentifyGlkObject phase type ref rock;
   if (phase == 0) { ! Zero out references to our objects.
      gg_acmanwin = 0;
      return;
   }
```

```
   if (phase == 1) { ! Reset our windows, streams and filerefs.
      switch (type) {
         0: ! it's a window
            switch (rock) {
               GG_ACMANWIN_ROCK: gg_acmanwin = ref;
            }
         1: ! it's a stream
               ! But we don't create any.
         2: ! it's a fileref
               ! But we don't create any.
      }
      return;
   }

   if (phase == 2) { ! Update our objects.
      MyRedrawGraphicsWindows();
   }
];
```

Il solito codice di riassociazione di una finestra grafica.

```
[ HandleGlkEvent ev context;
   switch (ev-->0) {
      evtype_Redraw, evtype_Arrange:
         MyRedrawGraphicsWindows();
```

La solita gestione del ridisegno in caso di necessità...

```
      evtype_Timer:
         if (acman_count > Last_Frame) glk_request_timer_events(0);
         else {
            glk_image_draw(gg_acmanwin, acman_count, 0, 0);
            acman_count++;
         }
   }
];
```

E finalmente la parte interessante di questo esempio, la gestione degli eventi del timer. A differenza di tutti gli eventi di input, il timer continua a funzionare, fino a che non viene fermato esplicitamente.

Se non è terminato si disegna un nuovo frame e si procede con l'animazione.

```
[ MyRedrawGraphicsWindows;
   if (gg_acmanwin==0) return;
   glk_window_set_background_color(gg_acmanwin, $FFFFFF);
   glk_window_clear(gg_acmanwin);

   if (acman_count == 0)
      glk_image_draw(gg_acmanwin, First_Frame, 0, 0);
   else
      glk_image_draw(gg_acmanwin, Last_Frame, 0, 0);
];

Include "Grammar";
```

Questa è la funzione di ridisegno della finestra. È molto semplice e pulisce lo sfondo (in bianco) prima di mostrare il frame successivo.

❖ ❖ ❖

Nonostante la sua semplicità, questa funzione non è corretta! `acman_count` in realtà non vale *mai* zero, dato che viene inizializzato a `First_Frame`.

Di conseguenza, se si ridimensiona la finestra *prima* di premere il pulsante viene disegnato *l'ultimo* frame, non il primo, come dovrebbe.

E questo è il motivo per cui lo screenshot è così alto...

◆ ◆ ◆

15.7. Ork 1 e Ork 2

Questa volta ci sono *due* esempi, che mostrano come (in linea di principio) si possono passare informazioni da un'avventura ad un'altra. In sostanza, l'uso degli stream su disco.

15.7.1. Ork 1

La prima "avventura" crea un file su disco contenente il bottino, guadagnato presumibilmente durante l'avventura stessa.

```
Constant Story "Ork␣1";
Constant Headline "^a␣demonstration␣of␣file␣I/O␣in␣Glulx␣Inform
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣by␣Adam␣Cadre^";

Global fref;
Global str;

Include "Parser";

Object LibraryMessages
   with
      before [;
         Score: rtrue;
      ];

Include "VerbLib";
Include "Infglk";

Object Credits "Ork␣1"
   with description
      "a␣demonstration␣of␣file␣I/O␣in␣Glulx␣Inform␣by␣Adam␣Cadre",
   has light;
```

Questo programma è *estremamente* breve. Possono essere degne di nota le due variabili per fileref e stream. Anche se in realtà potevano essere utilizzate tranquillamente due variabili locali nella funzione successiva.

```
[ Initialise;
   location = Credits;
   print "^^^You␣slay␣the␣dragon␣with␣a␣swing␣of␣your␣mighty
␣␣␣␣␣␣battle-axe!␣The␣dragon's␣gold␣is␣yours!";
   KeyCharPrimitive();
   print "^^Select␣a␣filename␣to␣use␣to␣save␣your␣character.";
   fref = glk_fileref_create_by_prompt(fileusage_Data+fileusage_BinaryMode,
            filemode_Write, 0);
   str = glk_stream_open_file(fref, filemode_Write, 0);
   glk_fileref_destroy(fref);
   glk_stream_set_current(str);
   print "GOLD:␣5^";
   glk_set_window(gg_mainwin);
   glk_stream_close(str, 0);

   print "^^This␣has␣been...";
   deadflag = 2;
   return 2;
```

```
];

Include "Grammar";
```

Ecco. Tutto qui. L'avventura è *così* breve che il giocatore non ha nemmeno visto la descrizione della locazione, per cui l'oggetto Credits è sostanzialmente inutile. O meglio, quasi visto che la libreria standard riesce a mostrare la descrizione *subito prima* di gestire deadflag. Un comportamento al limite.

La parte interessante è ovviamente la scrittura del file. Viene chiesto il nome al giocatore (per un file dati, binario, in scrittura). La fileref viene distrutta appena il file è stato aperto e, dopo una ridirezione dello stream corrente, vengono messi dei *dati* sullo stream.

Dopodiché, chiuso lo stream, e fine del gioco.

15.7.2. Ork 2

Il sequel di *Ork 1* permette di sfruttare il bottino accumulato per divertimento e profitto.

E per cominciare, ogni avventuriero necessita di equipaggiamento...

```
Constant Story "Ork␣2";
Constant Headline "^a␣demonstration␣of␣file␣I/O␣in␣Glulx␣Inform
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣by␣Adam␣Cadre^";

Global fref;
Global str;
Global gold;
Global ch;

Include "Parser";

Object LibraryMessages
   with
      before [;
         Score: rtrue;
      ];

Include "VerbLib";
Include "Infglk";

Object Showroom "Showroom"
   with description
      "Grignr's␣Used␣Battle-Axes␣is␣where␣you␣buy␣all␣your␣gear.
␣␣␣␣␣␣You␣don't␣see␣a␣salesman,␣though.␣Maybe␣it's␣self-serve.",
   has light;
```

Un cumulo di variabili globali! Anche in questo caso la maggior parte potrebbero essere tranquillamente locali; ma tanto questo è un esempio. gold in particolare dovrebbe essere l'unica dichiarata come tale.

```
Object -> goodaxe "Dragon-Slayer␣8000"
   with name 'battle-axe' 'battle' 'axe' 'dragon-slayer' 'dragon'
             'slayer' '8000',
      price 8,
      describe "^The␣Dragon-Slayer␣8000␣catches␣your␣eye␣--␣but␣it
␣␣␣␣␣␣␣␣␣␣might␣be␣out␣of␣your␣price␣range.",
      description "This␣looks␣almost␣brand-new.␣Maybe␣it␣was␣owned␣by
␣␣␣␣␣␣␣␣␣␣a␣little␣old␣lady␣who␣only␣used␣it␣on␣Sundays␣or␣something.",
      before [;
         Take, Buy:
            if (gold < self.price) "You␣can't␣afford␣it.";
```

```
            else {
               gold = gold - self.price;
               deadflag = 3;
               "The␣salesman␣emerges␣as␣you␣take␣the␣axe␣and␣you␣hand
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣him␣your␣gold␣pieces.␣~Been␣hacking␣the␣data␣file,␣eh?~
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣he␣says.␣~Well,␣I␣won't␣tell␣if␣you␣won't.␣Enjoy␣your
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣axe!~";
            }
      ];
```

Comincia la mercanzia, il top della gamma. Ovviamente anche con il bottino di cinque monete d'oro, il nostro eroe non sarebbe mai in grado di permettersela. Ma d'altra parte Grignr non si preoccupa della provenienza dell'oro con cui viene pagata la merce.

```
Object -> okayaxe "Elf-Chopper␣5000"
   with name 'battle-axe' 'battle' 'axe' 'elf-chopper' 'elf'
             'chopper' '5000',
      price 4,
      describe "^The␣Elf-Chopper␣5000␣is␣another␣option.",
      description "This␣looks␣fairly␣well-used.␣No␣wonder␣the␣elf
␣␣␣␣␣␣␣␣␣population␣has␣been␣dropping␣around␣here.",
      before [;
         Take, Buy:
            if (gold < self.price) "You␣can't␣afford␣it.";
            else {
               gold = gold - self.price;
               deadflag = 3;
               "The␣salesman␣emerges␣as␣you␣take␣the␣axe␣and␣you␣hand
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣him␣your␣gold␣pieces.␣~A␣fine␣choice,~␣he␣says.␣~Enjoy
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣your␣axe!~";
            }
      ];
```

Un altro oggetto in vendita, questa volta più a buon mercato. Non per niente è un'ascia usata! Per potercela permettere, come si vedrà, sarà necessario utilizzare il bottino lasciato da *Ork 1*, visto che altrimenti il budget è di due misere monete d'oro.

```
Object -> stick "stick"
   with name 'stick',
      description "It's␣just␣a␣stick.",
      before [;
         Take, Buy:
            deadflag = 3;
            "Ah,␣the␣economical␣option.␣Well,␣it'll␣do.";
      ];
```

E per finire, il *modello base*. È gratis, di cosa ci si può lamentare?

```
[ Initialise;
   location = Showroom;
   print "^^^Would␣you␣like␣to␣load␣a␣character?␣>";
   if (YesOrNo()) {
      print "^^Select␣the␣file␣to␣load.";
      fref =
         glk_fileref_create_by_prompt(fileusage_Data+fileusage_BinaryMode,
            filemode_Read, 0);
      str = glk_stream_open_file(fref, filemode_Read, 0);
      glk_fileref_destroy(fref);

      ch = glk_get_char_stream(str);
      while (ch < 48 || ch > 57)
         ch = glk_get_char_stream(str); ! get to the digit

      gold = ch - 48; ! '0' is char code 48, '1' is char code 49, etc.
```

```
      glk_stream_close(str, 0);
   }
   else gold = 2;

   "^Time␣for␣your␣next␣great␣adventure!␣But␣first,␣time␣to␣stock␣up
␣␣␣on␣supplies.^";
];

[ DeathMessage;
   switch (deadflag) {
      3: print "You␣have␣acquired␣your␣gear";
   }
];

Include "Grammar";
```

Ecco la parte interessante (almeno la prima parte). Se non si ha un file dati da caricare il budget a disposizione è di due monete d'oro.

In caso contrario, si chiede il nome del file (questa volta per *leggerlo* però!) lo si apre e lo si legge.

Il parser del file dati è... peculiare; Semplicemente legge byte dallo stream (è aperto in modalità binaria) fino a che non trova un cifra. E quella rappresenta il bottino recuperato (con una sola cifra quindi non si potrà mai partire con più di nove monete); ovviamente questo *non è* un esempio di come interpretare il contenuto di file dati.

16
Blorbificare

> "**blorbificare** *s. m.* l'atto di prendere un pugno di file di risorse e con essi creare un file Blorb utilizzabile da Glk."

Per chi ancora non lo sapesse Blorb è il formato standard (ovvero, l'unico standard al riguardo) per distribuire risorse ed eseguibile in un solo file (il file Blorb, per l'appunto).

Inventato da (*tanto per cambiare*) Andrew Plotkin è, in fin dei conti, un semplice archivio indicizzato (simile allo ZIP, ma non compresso). Sfortunatamente, pur essendo standard (ma non *così* standard), gli strumenti per manipolarlo sono abbastanza scarsi.

❖ ❖ ❖

Perché Plotkin abbia inventato *ancora un altro* formato di archiviazione non ci è dato di sapere. Si poteva, per esempio, utilizzare un *banalissimo* file ZIP senza compressione (ovvero in sola modalità *stored*); oppure un archivio *jar*, che è sostanzialmente la stessa cosa... per questi formati *già esistono* decine di strumenti e di librerie già pronte per ogni piattaforma immaginabile, e sono oltremodo ben documentati.

Questa volta però almeno si conosce l'origine del nome:

> "This format is named 'Blorb' because it wraps your possessions up in a box, and because the common save file format was at one point named 'Gnusto'. That has been changed to 'Quetzal', but I'm not going to let that stop me." – Andrew Plotkin

◆ ◆ ◆

16.1. Contenuto di un file Blorb

Come sia fatto e come si legga un file Blorb esula dalle conoscenze necessarie per il suo utilizzo in una Avventura Testuale (scritta in Inform, o in qualun-

que altro linguaggio supporti Blorb); quello che è importante sapere è *cosa* può entrare in un archivio in questo formato. Ogni archivio è una collezione di *risorse* (chiamate, nell'ambito di Blorb, *chunk*, essendo un formato basato su AIFF); una risorsa può appartenere ad una delle seguenti tre categorie:

Risorse grafiche; il formato permette di memorizzare immagini raster (bitmap) da visualizzare a schermo. Sono stati scelti due formati largamente diffusi:

- JPEG per immagini a tono continuo (fotografie) con compressione *lossy*;
- PNG per immagini al tratto, con compressione reversibile; inoltre le immagini PNG sono in grado di rappresentare aree trasparenti, anche in modo parziale (canale alpha).

Il formato prevede anche un *segnaposto rettangolare* ma solo ai fini della conversione delle Avventure Grafiche Infocom.

Risorse sonore; i formati possibili per l'audio sono relativamente più rari, ma sono comunque reperibili gli strumenti in grado di gestirli. Si possono usare risorse audio di tre tipi:

- AIFF, audio digitale non compresso (approssimativamente paragonabile al più comune WAV).
- OGG, per la precisione Ogg/Vorbis; è una alternativa al diffusissimo MP3, ma senza vincoli brevettuali. Il codec e gli strumenti sono disponibili su http://www.vorbis.com.
- MOD, un formato ibrido campionato basato su pattern. Diffuso su Amiga ma ora relativamente raro; *molto approssimativamente* equivalente al formato *midi*. Per la precisione il formato supportato è il ProTracker 2.0 (alcuni dettagli sono presenti in http://www.eblong.com/zarf/blorb/mod-spec.txt).

Il formato *song* è stato deprecato in Blorb 2.0, dato che *nessuno* lo ha mai utilizzato.

Risorse di codice; all'interno del file Blorb può essere presente *solo uno* di questi chunk, rappresentante l'eseguibile del programma. Fra tutti i formati previsti, può contenere un eseguibile Glulx ma non è una risorsa direttamente accessibile da Glk (se non per il fatto che l'interprete è in grado di estrarla da un file Blorb).

Il file Blorb può anche contenere altri chunk (non considerati come risorse), come, per esempio, informazioni sul gioco, un'immagine da utilizzare

come frontespizio ed altri dettagli (principalmente per compatibilità con la Z-machine).

Per tutti i dettagli al riguardo è consigliabile leggere le specifiche Blorb (http://www.eblong.com/zarf/blorb/blorb.html e il *Treaty of Babel* (http://babel.ifarchive.org).

16.2. Blorb e Inform

Tutte le risorse contenute in file Blorb vengono identificate con un numero intero; i numeri devono essere univoci *per tipo di risorsa*: anche se possono coesistere un'immagine e un suono con identificativo 3, non è possibile, per esempio avere un *png* e un *jpeg* con lo stesso numero di risorsa.

La risorsa codice ha sempre identificativo 0 (anche se per l'autore non è importante). Per convenzione non si crea una risorsa grafica o audio identificata da 0: in questo modo è possibile definire la risorsa 0 come *nessuna risorsa* (per esempio, nessuna musica di sottofondo).

Per comodità, programmando in Inform, si utilizza generalmente un file di inclusione per dare un nome a tutte queste risorse; la tradizione creata da iblorb detta che l'estensione di questo file sia `.bli`; per il compilatore Inform è comunque un file di inclusione come tutti gli altri (`.h`).

Il file `.bli` è molto semplicemente una lunga definizione di costanti: una costante per ogni risorsa, per la precisione; iblorb genera automaticamente questo file nella prima fase di compilazione; gli utenti di altri blorbificatori (come blorbtar) devono crearlo manualmente (non che sia uno sforzo micidiale, in ogni caso).

Lo strumento originale per creare file Blorb si chiama *blurb*; blurb è anche il nome del formato del suo file di configurazione. Attualmente non è più in uso in quanto non aggiornato all'ultima versione delle specifiche.

Attualmente, per la creazione di file Blorb sono disponibili due programmi distinti: il primo, blorbtar è estremamente semplice. Più sofisticato è invece iblorb, che, fra l'altro, genera automaticamente il file di definizione `.bli`.

16.3. blorbtar

blorbtar è il modo più semplice per creare un file Blorb. È uno script perl, quindi sconsigliato agli utenti Windows (che generalmente non hanno a disposizione questo linguaggio). L'ultima versione è la 0.1, con una patch per gestire correttamente i file Glulx (per Inform è quindi *vitale*.

Figura 16.1. Correzione per il supporto audio in blorbtar 0.1

```
sub SndType
{
    my $magic;
    seek CHUNK, 8, 0;
    read CHUNK, $magic, 4;
    if($magic eq "AIFF") {
        return "AIFF";
    }
    seek CHUNK, 1080, 0;
    read CHUNK, $magic, 4;
    if($magic eq "M.K." || $magic eq "M!K!") {
        return "MOD ";
    }
    seek CHUNK, 0, 0;
    read CHUNK, $magic, 4;
    if($magic eq "OggS") {
        return "OGGV";
    }

    die "Unknown sound type!\n";
}
```

❖ ❖ ❖

Al momento, inoltre, non sono supportati i file OGG, quindi probabilmente blorbtar non è ancora aggiornato alla versione 2 dello standard Blorb.

Inoltre è presente un bug per cui *raramente* è possibile che un file AIFF non venga riconosciuto correttamente (tecnicamente: se la lunghezza del chunk FORM contiene un byte 0A).

Entrambi i problemi sono banalmente risolvibili: basta sostituire la routine SndType (definita circa a linea 110 di blorbtar) e sostituirla con quella riportata in Figura 16.1 (in sostanza viene corretto il controllo per i file *aiff* e aggiunto il rilevamento degli *ogg*). I più pignoli potrebbero anche aggiungere alla definizione di %extlist a inizio file l'associazione "OGGV" => "ogg"...

◆ ◆ ◆

L'interfaccia del programma è minimale, e si rifà al tradizionale programma tar disponibile sotto Unix; blorbtar può funzionare in tre modalità distinte, selezionate con il primo argomento:

`c` crea un nuovo file Blorb con le risorse specificate;

`x` estrae il contenuto del file Blorb specificato nella directory attuale;

Tabella 16.1. Nomi dei file per la creazione con blorbtar

Nome del file	Chunk creato
PIC*nnn*	Risorsa grafica (PNG o JPEG)
SND*nnn*	Risorsa sonora (AIFF, MOD o OGG)
STORY0	Eseguibile (il file .ulx creato da Inform)
IDENT	Identificativo della storia (se usato con Glulx *deve* contenere i primi 128 byte del file .ulx)
PALETTE	Palette dei colori utilizzati (generalmente ignorata)
RELEASE	Significativo solo per la Z-machine
RESOL	Significativo solo per la Z-machine
LOOPING	Significativo solo per la Z-machine
AUTH	File contenente il nome dell'autore
ANNO	File contenente annotazioni varie
COPY	File contenente la clausola di copyright

`t` elenca le risorse presenti all'interno del file Blorb specificato.

16.3.1. Creazione di un file Blorb

La modalità principale di blorbtar viene invocata con la seguente sintassi (al prompt della shell o dell'interprete dei comandi):

```
blorbtar.pl c fileblorb.blb risorse
```

fileblorb.blb è il file da creare, che verrà comunque sovrascritto. risorse invece è la lista dei file che verranno inseriti nell'archivio. Il tipo dei file (e dei chunk creati) viene determinato *automaticamente* sia dal loro nome che dal loro contenuto, come indicato in Tabella 16.1.

Se un file specificato non ha un nome fra quelli indicati o il suo contenuto non è riconoscibile (per quanto riguarda i file grafici e sonori) la creazione dell'archivio fallisce.

È importante, in particolar modo, che il file contenente il codice Glulx venga reso disponibile (copiato, rinominato o collegato) sotto il nome di STORY0.

❖ ❖ ❖

I nuovi chunk specificati nella release 2 delle specifiche Blorb non sono supportati: il frontespizio (Fspc) e i metadati (IFmd), entrambi definiti nel *Treaty of Babel*.

Essendo però due semplici chunk di testo è possibile definirli in blorbtar aggiungendo le due righe seguenti a %namelist (a inizio file).

```
FRONTIS  => [ 0, "Fspc" ],
METADATA => [ 0, "IFmd" ],
```

◆ ◆ ◆

16.3.2. Elencare il contenuto di un file Blorb

Si può ottenere una lista dei chunk presenti all'interno del file Blorb con il seguente comando:

```
blorbtar.pl t fileblorb.blb
```

Se si volesse avere una lista dettagliata (comprendente i tipi dei chunk e la loro dimensione) si può usare l'opzione "v":

```
blorbtar.pl tv fileblorb.blb
```

16.3.3. Estrarre il contenuto di un file Blorb

Per estrarre tutte le risorse presenti in un file Blorb (un file per ogni singolo chunk) bisogna usare blorbtar con il comando "x":

```
blorbtar.pl x fileblorb.blb
```

In questo modo viene estratto ogni chunk in un file distinto, con un nome adatto per poter ricomporre in seguito l'archivio. È anche possibile far aggiungere automaticamente l'estensione corretta ai file creati (ma attenzione, in questo modo non potranno essere utilizzati da blorbtar in seguito senza essere rinominati!):

```
blorbtar.pl xe fileblorb.blb
```

In questo modo verranno creati file .jpg per le risorse grafiche JPEG, file .png per immagini di tipo PNG e così via.

164. iblorb

Il blorbificatore iblorb è molto più avanzato, anche se necessita di più passaggi e di un file di definizione delle risorse da utilizzare (blorbtar funziona invece completamente da riga di comando).

È disponibile già compilato per DOS e Windows; si possono anche scaricare i sorgenti per la compilazione su altre piattaforme. La versione 0.5b supporta ufficialmente le specifiche Blorb 2.0.

❖ ❖ ❖

Sfortunatamente, almeno su Linux, la compilazione non è così immediata... sono presenti dei riferimenti all'header `dos.h` e alla struct `date` (sono usati solo per generare un commento, quindi si possono rimuovere con estremo pregiudizio); inoltre, per qualche strano motivo, le versioni troppo recenti di flex non compilano correttamente (sono riuscito senza problemi con flex 2.5.4).

In caso di problemi si può sempre tentare la fortuna con l'eseguibile non testato sul sito di iblorb (http://justice.loyola.edu/~lraszews/if/iblorb.html) oppure semplicemente usare blorbtar.

Io non sono riuscito a farlo funzionare in nessuno dei due modi...

◆ ◆ ◆

iblorb non è un solo programma: è una collezione di programmi utili per creare un file Blorb. Per la precisione, sono presenti sei eseguibili:

blc il generatore di file Blorb propriamente detto;

bres il compilatore di risorse; genera il file `.bli` per Inform e il file `.blc` per blc;

front un programma di utilità per automatizzare la sequenza bres, Inform, blc;

bpal un editor di palette Blorb;

bmerge permette di combinare più file Blorb in uno;

bdiff estrae le risorse differenti fra due file Blorb.

I comandi bpal, bmerge e bdiff sono utilizzati solo in occasioni particolari e non verranno discussi; il cuore del pacchetto iblorb è composto da bres e blc (con front che, anche se non indispensabile, è una bella comodità).

164.1. Invocazione di iblorb

La compilazione utilizzando iblorb avviene in tre fasi:

- Per prima cosa bres legge il file di definizione delle risorse `.res` (che deve essere preparato dall'autore).

  In base al contenuto del file bres genera un file `.bli` (con lo stesso nome di base del file `.res`) e un file `.blc` (utilizzato in seguito da blc, nella terza fase);

- Si compila normalmente con Inform; in questa fase vengono utilizzate le costanti simboliche definite nel file `.bli` generato in precedenza per identificare le risorse da utilizzare. Inform genera quindi un file `.ulx`;

- Per finire, blc utilizza il file `.blc` generato nella prima fase, il file `.ulx` creato da Inform nella seconda e tutte le risorse definite nel file `.res` per creare l'archivio Blorb di destinazione.

I comandi bres e blc si utilizzano semplicemente invocandoli con i nomi dei file da trattare (bres in particolare vuole il nome senza estensione).

Il modo più pratico per utilizzare iblorb è comunque utilizzando il programma front in dotazione. Questo programma chiama *automaticamente* i tre programmi nell'ordine corretto (bres, Inform, blc) per creare un file Blorb:

```
front risorse gioco.inf opzioni
```

Nella chiamata, `risorse` è il nome senza estensione del file `.res` da elaborare; tutto il resto della linea di comando viene passato ad Inform (quindi `gioco.inf` e `opzioni` sono in realtà destinate ad Inform). front a questo punto richiama in sequenza i tre comandi e (assumendo che non ci siano errori) il risultato è un file chiamato `risorse.blb`.

Per funzionare correttamente front necessita degli eseguibili bres, blc ed inform disponibili nel path di esecuzione. È anche possibile creare nella directory corrente un file `infb.rc` per definire la posizione del compilatore Inform le opzioni di default; vedere il file di documentazione `front.txt` fornito con iblorb per un esempio.

164.2. Il file di definizione delle risorse

Per poter funzionare correttamente, bres e blc necessitano di un file (scritto dall'autore) con l'elenco di tutte le risorse utilizzate e altre informazioni che andranno a comporre il file Blorb.

Questo file, un comunissimo file di testo, deve *necessariamente* avere estensione `.res`.

Ogni riga del file specifica una risorsa o un chunk del file di destinazione. È possibile commentare il file utilizzando righe che cominciano con "!".

❖ ❖ ❖

Nel file di documentazione `front.txt` presente nella distribuzione iblorb è presente un file `.res` di esempio; anche se relativo ad un gioco per Z-machine, è perfettamente applicabile per programmi Inform/glulx (tranne che per la direttiva RELEASE che non è significativa, in questo caso).

Una piccola pignoleria: la dichiarazione COPYRIGHT non dovrebbe contenere i caratteri `(c)` (sono aggiunti esternamente da un eventuale visualizzatore, secondo le specifiche Blorb).

◆ ◆ ◆

Definizione delle risorse

Fra tutte le direttive, l'unica *essenziale* è la CODE, per specificare il file eseguibile all'interno dell'archivio (a meno che non si voglia creare un file Blorb di sole risorse). SOUND e PICTURE sono ovviamente necessarie per definire le risorse grafiche e sonore utilizzate.

Tutte le altre dichiarazioni sono facoltative e influiscono in misura minima sull'esecuzione (possono fornire però dati aggiuntivi utili per *altri* programmi).

`CODE file`

Dichiara `file` come risorsa eseguibile del file Blorb (l'equivalente del file STORY0 per blorbtar). In questo caso si specifica il file `.ulx` generato da Inform.

`FRONTISPIECE file`

Definisce una risorsa immagine e la dichiara come frontespizio, secondo il *Treaty of Babel*. L'immagine è disponibile da Inform attraverso la costante FRONTISPIECE_ib. Il formato dell'immagine è determinato automaticamente (PNG o JPEG), ma per un'immagine di frontespizio sono presenti raccomandazioni extra, specie a riguardo delle dimensioni consigliate.

`SOUND nome file`

Definisce una risorsa audio. Nel file `.bli` verrà aggiunta una dichiarazione Constant per l'identificatore `nome`. Il formato della risorsa è determinato automaticamente.

```
PICTURE nome file
```

Definisce una risorsa grafica. Nel file `.bli` verrà aggiunta una dichiarazione `Constant` per l'identificatore `nome`. Il formato dell'immagine è determinato automaticamente.

Definizione dei chunk informativi

Tutti i chunk informativi sono facoltativi e intesi principalmente come commento interno al file Blorb.

```
AUTHOR testo
```

Specifica l'autore del file Blorb.

```
COPYRIGHT testo
```

Specifica i termini di copyright (senza "`(c)`").

```
NOTE testo
```

Questa definizione può essere ripetuta più di una volta; inserisce il testo specificato come commento all'interno del file Blorb.

Definizione dei file ausiliari

Tutti questi chunk sono facoltativi; ne esistono anche altri che però sono significativi solo per la Z-machine. Consultare la documentazione di iblorb per l'elenco completo.

```
PALETTE file
HIGHCOLOR
TRUECOLOR
```

Queste tre direttive sono mutuamente esclusive e permettono di definire la palette utilizzata per le risorse grafiche del file Blorb; se viene utilizzata la direttiva `PALETTE` allora `file` deve essere un file palette nel formato richiesto da Blorb. `HIGHCOLOR` e `TRUECOLOR` specificano l'assenza di palette e una profondità colore richiesta di, rispettivamente, 16 e 32 bit.

Se qualcuno ai giorni nostri dovesse *ancora* aver bisogno di specificare una palette, il programma bpal (facente parte di iblorb) permette di editarla in modo pseudo-interattivo (la documentazione è nel file `front.txt` della distribuzione iblorb).

HEADER `file`

Inserisce il file specificato come header (chunk `IFhd`). Le specifiche Glulx lo definiscono come i primi 128 byte del file `.ulx`.

METADATA `file`

Inserisce il file specificato come metadati (chunk `IFmd`). Il formato dei metadati è un file XML, contenente le informazioni specificate nel *Treaty of Babel*.

17

sgw - a Simple Glulx Wrapper

È giunto il momento di considerare alcune soluzioni *pronte all'uso* per utilizzare grafica e sonoro all'interno delle proprie avventure. Fino a questo punto, tutto quello che si aveva a disposizione erano le nude e crude API Glk e una manciata di funzioni di supporto fornite dalla libreria di Inform.

Sebbene siano i componenti indispensabili per questo genere di operazione, nulla vieta di utilizzare librerie più evolute, con vari gradi di funzionalità e limitazioni, ma che consentono di iniziare a sviluppare più rapidamente Avventure Testuali Non Solo Testuali.

La prima di queste libreria è *sgw - a Simple Glulx Wrapper*; nonostante il nome inglese, si tratta di un prodotto italianissimo, sviluppato da Alessandro Schillaci, Vincenzo Scarpa e Paolo Maroncelli. Io stesso, nel corso della trattazione, proporrò alcune migliorie e correzioni alla stessa.

Derivata estraendo il sottosistema Glk da *Little Falls* è ora una libreria indipendente a tutti gli effetti: può essere usata così com'è per un discreto numero di funzionalità; a causa però della necessaria presenza degli entry point utilizzati dalla libreria (`HandleGlkEvent` e `IdentifyGlkObject`, in particolar modo) necessiterà di variazioni nel caso in cui si vogliano utilizzare altre funzionalità di Glk (il mouse, per esempio). È comunque un ottimo punto di partenza, abbastanza completo per gli utilizzi normali.

L'ultima versione della libreria (attualmente la 1.6.1) è liberamente scaricabile dal sito http://slade.altervista.org.

17.1. Le funzionalità proposte

Il risultato a cui mira la libreria *sgw* è una classica Avventura Testuale con finestra grafica, ed eventualmente il supporto audio (è anche pos-

sibile utilizzarla con le sole routine audio, definendo un simbolo prima dell'inclusione).

Per la precisione, l'autore che la utilizza ha a disposizione:

- Un layout di schermo esteso composto da, oltre alle finestre standard principale e di stato, una finestra grafica di altezza prefissata;
- la possibilità di cambiare i colori utilizzati per il testo;
- tre canali audio: uno per la musica e due per gli effetti sonori, dati le limitazioni delle implementazioni Glk attuali; ogni canale ha il proprio volume indipendente (se supportato dall'implementazione Glk in uso);
- la possibilità di mostrare nella finestra grafica un'immagine (presa da una risorsa, ovviamente), centrata o allineata su uno dei lati;
- una serie completa di routine di comodo per poter visualizzare il testo con uno stile a scelta;
- tutte le routine standard di gestione degli eventi e riassociazione definite, senza bisogno di lavoro extra.

Essendo una libreria orientata ai principianti non promette cose strabilianti (anzi, è essa stessa *leggermente* bacata... per esempio qualche volta utilizza dei riferimenti a oggetti nulli, ma fortunatamente tutte le librerie in uso sono clementi al riguardo); tuttavia è una base su cui eventualmente sviluppare funzionalità aggiuntive specifiche per il gioco.

17.2. Come si usa

Come descritto nella documentazione presente all'interno del file `sgw.h`, la prima cosa da fare è includere il file stesso assieme a `infglk.h` (su cui si basa), ma *prima* del parser:

```
Include "infglk";
Include "sgw";
Include "Parser";
```

Prima dell'inclusione, è possibile definire alcune costanti per meglio configurarne il comportamento (Tabella 17.1).

L'assegnazione dei colori agli stili è peculiare: i colori assegnati ad ogni stile sono indicati in tabella. Oltre a questo, lo stile *Alert* è in negativo mentre *Normal* è completamente giustificato.

Per cambiare queste assegnazioni (ed eventualmente assegnare altri hint e/o configurare lo stile *User2*) è necessario modificare il blocco che compone

Tabella 17.1. Costanti di configurazione per *sgw*

Costante	Significato
NOGRAPHICS	Disattiva la gestione della finestra grafica
SCBACK	Colore di sfondo
SCTEXT	Colore normale del testo
SCSOFT	Colore per lo stile *User1*
SCEMPH	Colore per gli stili *Emphasized* e *Header*
SCHEAD	Colore per lo stile *SubHeader*
SCINPU	Colore per gli stili *Blockquote*, *Input*, *Preformatted* e *Note*

il grosso della funzione InitGlkWindow; ovviamente se l'assegnazione dei colori agli stili è soddisfacente basta definire le costanti relative *prima* di includere la libreria; alcuni valori indicativi di colori da provare sono riportati nell'Appendice A.

L'altra operazione indispensabile è chiamare la funzione initializeSGW all'inizio della propria Initialise , passando l'altezza in pixel della finestra grafica (se compilata con NOGRAPHICS ovviamente quest'ultima non sarà significativa).

```
initializeSGW(h); ! h = altezza grafica in pixel
```

❖ ❖ ❖

Attenzione però: anche se, a prima vista, si potrebbe passare come h un valore costante (visto che *si conoscono a priori* le dimensioni delle immagini che saranno utilizzate), questo approccio non è del tutto corretto; in linea di principio la libreria è libera di ridimensionare le immagini (per compensare prefenze dell'utente o altro).

Il modo *completamente corretto* per determinare h è chiamare l'apposita funzione glk_image_get_info (sezione 7.1) applicata su una immagine campione; *sgw* presume che le immagini siano tutte della stessa altezza: in caso contrario basta utilizzare la più alta.

◆ ◆ ◆

A questo punto il grosso del lavoro è fatto, le finestre sono aperte, i canali audio pure e tutto quel che rimane da fare è richiedere i singoli servizi della libreria.

❖ ❖ ❖

Durante l'inizializzazione la libreria segnala nella finestra principale le funzionalità gestalt che trova *non supportate*.

Esiste anche un punto che può essere fuorviante per chi volesse modificare il codice: nella funzione inizializeGlulx (chiamata da initializeSGW) viene *apparente-*

mente creata anche la finestra di stato, alta due righe, subito prima della finestra grafica.

Sfortunatamente, in questo punto dell'esecuzione, la finestra di stato *è già presente* in quanto non sono state date indicazioni al riguardo in InitGlkWindow (per dettagli su *come* dare questo genere di indicazioni, vedere la sezione 14.9).

Oltre a questo, ci sono altri problemi minori:

glk_window_set_background_color viene purtroppo utilizzata sulla finestra principale (una pratica teoricamente illecita ma necessaria su alcune implementazioni Glk non del tutto conformi) e, leggermente più grave, non è gestito in alcun modo il fallimento nella creazione della finestra grafica.

Inoltre alla prima esecuzione i tre canali audio vengono silenziati *prima* di venir creati (durante la fase 2 di IdentifyGlkObject). Per correggere questa banale svista è sufficiente sostituire la chiamata a ResetMusicChannel con una a silenceAll (la funzione ResetMusicChannel può poi essere rimossa visto che rimane inutilizzata dopo questa modifica).

◆ ◆ ◆

17.3. Utilizzo degli stili

Grazie alla contribuzione di Vincenzo Scarpa, la libreria *sgw* contiene un set completo di funzioni per mostrare testo con uno stile a piacere; sono tutte funzioni del tipo illustrato nella sezione 14.10, Figura 14.2. La lista di queste funzioni (e dello stile che producono) è in Tabella 17.2. I nomi non sono *esattamente* riferiti allo stile, ma alla formattazione *tipica* che assume questo stile (per la precisione, gli stili utilizzati tradizionalmente nella Z-machine); per cui non necessariamente hanno un nome accurato. Non sono nemmeno presenti *tutti* gli stili, ma completare la serie è banale (copia & incolla possono aiutare in questo frangente).

Per esempio, le seguenti chiamate sono del tutto equivalenti:

```
print "E␣adesso␣", (s_emph) "crepa!", "^";
print "E␣adesso␣", (s_bold) "crepa!", "^";
```

17.4. Mostrare un'immagine nella finestra superiore

Ora che la finestra grafica è stata aperta (ed è dell'altezza desiderata) è possibile richiedere alla libreria di mostrarvi un'immagine presa dalle risorse a disposizione.

A seconda dell'allineamento desiderato (a sinistra, al centro, oppure a destra) basta chiamare una delle seguenti funzioni:

```
viewImageLeft(image);
viewImageCenter(image, image_width);
viewImageRight(image, image_width);
```

Tabella 17.2. Funzioni per gli stili di testo in *sgw*

Funzione	**Stile risultante**
s_emph	Emphasized
s_bold	Emphasized
s_pref	Preformatted
s_fixed	Preformatted
s_head	Header
s_subhead	Subheader
s_alert	Alert
s_reverse	Alert
s_note	Note
s_underline	Note
s_block	BlockQuote
s_input	Input

Ovviamente image è il numero della risorsa da caricare, mentre il parametro image_width (necessaria solo se l'immagine deve essere centrata nella finesta oppure allineata al margine destro) è la larghezza dell'immagine stessa. Se *non* si vuole più mostrare un'immagine, si passa 0 come numero di risorsa

Il ridisegno in caso di ripristino, ridimensionamento o evento apposito è gestito in modo automatico dalla libreria.

❖ ❖ ❖

La domanda a questo punto sorge spontanea: perché dobbiamo passare alle routine di visualizzazione la larghezza dell'immagine? Ovviamente affinché la libreria sia in grado di calcolare la posizione allineata.

Come si potrebbe determinare il valore da passare in questo parametro? Semplicemente con glk_image_get_info (sezione 7.1). Perché allora non farla calcolare alla libreria la larghezza?

In effetti la soluzione è abbastanza semplice. Basta modificare la funzione viewImageSGW e ripulire le altre funzioni. La variabile globale curr_pic_width diventa così inutile, come mostrato in Figura 17.1.

Se necessario, è anche molto semplice allineare verticalmente l'immagine (i dati sono già tutti disponibili, basta utilizzarli).

◆ ◆ ◆

Figura 17.1. Modifiche proposte per semplificare la visualizzazione delle immagini (libreria *sgq*).

```
!Global curr_pic_width; -- Non serve più!

! Centralized visualization routine - added by Paolo Maroncelli
[ viewImageSGW finestra_larghezza img_larghezza x_pos;
    ! Abbiamo aggiunto una variabile per la larghezza dell'immagine
    #Ifdef TARGET_GLULX;
    #Ifndef NOGRAPHICS;
    ! Anticipiamo il controllo poiché glk_image_get_info fa parte del
    ! pacchetto grafico
    if (~~glk_gestalt(gestalt_Graphics,0)) {
        return;
    }

    glk_window_clear(gg_bigwin);
    ! glk_image_get_info, oltre a dare la dimensione effettiva
    ! dell'immagine consente di sapere anche se l'immagine esiste
    if (glk_image_get_info(curr_pic,
            gg_arguments, gg_arguments+WORDSIZE)) {
        ! Ecco la larghezza. Se servisse per un allineamento
        ! verticale, l'altezza dell'immagine è in gg_arguments-->1
        img_larghezza = gg_arguments-->0;
        glk_window_get_size(gg_bigwin,
            gg_arguments, gg_arguments+WORDSIZE);
        finestra_larghezza = gg_arguments-->0;
        switch (curr_pic_pos) {
            POS_GG_LEFT:   x_pos = 0;
            POS_GG_CENTER: x_pos = (finestra_larghezza/2)
                -(img_larghezza/2);
            POS_GG_RIGHT:  x_pos = (finestra_larghezza)
                -(img_larghezza);
        }
        if (gg_bigwin) {
            glk_image_draw(gg_bigwin, curr_pic, x_pos, 0);
        }
    }
    #endif;
    #endif;
];
    ! Versioni semplificate delle routine di set, non hanno più bisogno
    ! di considerazioni sulla larghezza dell'immagine
! View an image on the left of the main graphic window
[ viewImageLeft image;
    #Ifdef TARGET_GLULX;
        curr_pic = image;
        curr_pic_pos = POS_GG_LEFT;
        viewImageSGW();
    #endif;
];
! View an image on the center of the main graphic window
[ viewImageCenter image;
    #Ifdef TARGET_GLULX;
        curr_pic = image;
        curr_pic_pos = POS_GG_CENTER;
        viewImageSGW();
    #endif;
];
! View an image on the right of the main graphic window
[ viewImageRight image;
    #Ifdef TARGET_GLULX;
        curr_pic = image;
        curr_pic_pos = POS_GG_RIGHT;
        viewImageSGW();
    #endif;
];
```

17.5. Pulire le finestre

Quando si vuole fare *tabula rasa*, questa è la funzione da chiamare per ottenere uno schermo lindo, pulito e vuoto (tranne per la barra di stato):

```
clearMainWindow();
```

... pulisce sia la finestra principale che la finestra grafica.

❖ ❖ ❖

Ma attenzione! Anche se *apparentemente* la finestra grafica è stata cancellata, la libreria rimetterà l'ultima immagine richiesta alla prima opportunità di disegno!

Per evitare questo inconveniente (sempre che si voglia *veramente* rimanere con la finestra grafica *completamente vuota*) basta richiedere l'immagine numero 0. Per farlo in automatico è sufficiente modificare l'istruzione `glk__window__clear(gg__bigwin);` della funzione `clearMainWindow` con, per esempio, `viewImageLeft(0);`.

◆ ◆ ◆

17.6. Musica ed effetti sonori

La libreria *sgw* mette a disposizione tre canali audio. L'idea è di sfruttarne uno per una eventuale musica di sottofondo, mentre gli altri due permettono riprodurre simultaneamente due effetti sonori.

Nulla vieta però di non utilizzare la musica di sottofondo e, al suo posto, ottenere un terzo canale per audio campionato.

La funzione principale per attivare l'audio è `playSound`:

```
playSound(channel, sound, length, volume);
```

`channel` è uno dei tre canali audio a disposizione:

- `music` per la musica;
- `chan1` per gli effetti sonori (primo canale);
- `chan2` per gli effetti sonori (secondo canale).

`sound` è il numero della risorsa audio da riprodurre; `length` è il numero di volte che la risorsa deve essere ripetuta: nel caso tipico della musica di sottofondo infinita si può passare -1; con zero la risorsa non viene riprodotta nemmeno una volta (ovviamente).

volume è il volume con cui riprodurre il canale audio; non tutte le librerie Glk lo supportano, ma in caso contrario è possibile utilizzare una delle 3 costanti definiti nella libreria:

- VOLUME_HIGH, volume alto (5);
- VOLUME_NORMAL, volume alto (3);
- VOLUME_LOW, volume alto (2).

È possibile utilizzare anche valori intermedi o fuori da questo range: se si vuole sentire qualcosa è meglio che il volume sia maggiore di 0, valori maggiori di 5 potrebbero portare ad un suono distorto per la troppa amplificazione; il risultato dipende molto dalla libreria Glk in uso, però.

In generale *sgw* non permette di sapere se la riproduzione di un effetto sonoro è terminata; questa possibilità è comunque facoltativa per Glk e necessita la gestione di un evento (e quindi modifiche alla routine HandleGlkEvent o un ciclo di poll scritto ad hoc.

Il problema non si dovrebbe presentare finché gli effetti sonori sono relativamente brevi o la musica è in ripetizione infinita; in caso contrario bisogna tenere conto che la nuova risorsa *interrompe bruscamente* quella attualmente attiva sul canale. Risolvere il problema con un effetto di fading è possibile, ma non banale (richiede il supporto di Glk sia per il controllo del volume che degli eventi del timer), in ogni caso ad un livello ben più avanzato di quello a cui si rivolge *sgw*.

Quando si vuole arrestare la riproduzione di un canale (specialmente nel caso di un canale in ripetizione infinita!) si può usare silenceChannel:

```
silenceChannel(channel);
```

... dove channel è uno dei tre canali definiti; se, infine, quel che necessita è il vero silenzio di tomba (quindi azzittire tutti e tre i canali) la funzione silenceAll è quello che serve:

```
silenceAll();
```

Per finire, è possibile variare il volume di un canale già in riproduzione:

```
setVolume(volume, channel);
```

I valori del volume sono gli stessi visti in precedenza (ma attenzione, *prima* il volume, poi il canale; l'ordine dei parametri in questa funzione è un po' anomalo).

Tabella 17.3. Entry point della libreria standard già implementati da *sgw*

`IdentifyGlkObject`
`InitGlkWindow`
`HandleGlkEvent`

17.7. Integrazione

La libreria *sgw*, anche se orientata ai programmatori principianti, fornisce un discreto numero di funzionalità a disposizione. Inoltre è anche piuttosto breve, e adeguatamente commentata.

Nel caso si volessero utilizzare funzionalità di Glk non previste (per esempio una ulteriore finestra, o gli eventi del mouse) bisogna tenere presente che tutte le funzioni necessarie per l'integrazione con la libreria standard *sono già definite* all'interno della libreria.

Per questo motivo, qualunque estensione necessiti di codice in una delle funzioni in Tabella 17.3 deve essere implementata modificando la libreria stessa.

Sono inoltre utilizzati i valori 210, 410, 411 e 412 per gli oggetti Glk creati (sotto i sassi della finestra grafica e dei tre canali audio).

18

GWindows

A differenza di sgw (la cui "s" sta per *simple*), la libreria GWindows (sviluppata da L. Ross Raszewski) è probabilmente la più completa (nonché complessa) a disposizione per la gestione del layout delle finestre Glk sotto Inform.

❖ ❖ ❖

La libreria GWindows è alquanto complessa, e il codice non è nemmeno un capolavoro di leggibilità, a mio parere. Sfortunatamente la documentazione è vaga, in alcuni punti, particolarmente al riguardo dei widget definiti. L'intera libreria GREX poi non è documentata (e a parte quello, pare sia addirittura molto lenta!).

Per questo motivo la discussione non è così approfondita come per la libreria sgw; è consigliabile tenere come riferimento gli esempi forniti con la distribuzione nel caso si volesse utilizzare in produzione questa libreria.

◆ ◆ ◆

La copertura delle funzionalità di GWindows è però ridotta alla *sola gestione del layout*; altre sezioni di Glk non sono gestite, a parte una libreria ausiliaria per la gestione dell'audio (il GWindows Sound System). Rimangono quindi sostanzialmente scoperte la gestione degli stream e delle fileref, che d'altra parte sono di uso limitato (proprio come in sgw).

❖ ❖ ❖

Anche il numero di versione è una questione complicata per questa libreria: ogni singolo componente ha il suo numero di versione; per questo motivo la distribuzione in esame contiene:

- GWindows 0.9b
- GREX 0.1b
- GSound 0.2b

- GWindows Core 1.0

GWindows è l'implementazione delle finestre, GREX la libreria grafica ausiliaria, GSound il sound system (prevedibilmente) e per finire, GWindows Core è la parte che fa stare il tutto assieme (facile no?).

Inoltre sono necessarie le librerie infglk e le utility di Ross Raszewski (che sono comunque incluse nel pacchetto).

◆ ◆ ◆

La differenza di approccio è radicale: mentre con la comune programmazione (e anche utilizzando sgw) si implementano mano a mano le parti richieste (compresi gli entry point della libreria standard), GWindows copre fin dall'inizio *tutti* i punti possibili (dalla funzione `DrawStatusLine` alla `HandleGlkEvent` e li rende disponibili con una interfaccia orientata agli oggetti.

Per la precisione con GWindows è necessario dichiarare un oggetto per *ogni finestra* che si intende utilizzare (comprese le finestre paio!). Il vantaggio è che permette di variare automaticamente il layout e di creare situazioni dinamiche (come una finestra *a tendina* che appare solo in determinate circostanze).

Allo stesso modo, il sistema di organizzazione ad oggetti permette di separare la gestione degli eventi, e persino di creare delle finestre-tipo dal comportamento standardizzato (note come *widget*, in GWindows). Sfortunatamente, a causa della complessità della struttura delle finestre, il cambio di layout non è indolore: tutte le finestre vengono richiuse e riaperte, in questo caso (tranne nelle circostanze più semplici); la conseguenza è la perdita del contenuto delle finestre e dell'eventuale scrollback memorizzato.

18.1. Come utilizzarla

Data la sua complessità e la possibilità di modularizzare il codice, la libreria GWindows è divisa in una miriade di file di inclusione; il minimo essenziale per farla funzionare è composto da:

- `gwincls.h` da includere *prima* del parser;
- eventuali costanti di configurazione; esistono quattro costanti che possono essere definite per alterare il funzionamento della libreria:

  `GW_INPUT_ECHO` (default 1) Se diversa da zero, l'input del giocatore viene riportato nella finestra principale anche se la finestra di input non è quella principale;

  `GW_ECHO_OVERRIDE` Se definita, mostra nella finestra principale *anche* i comandi che sono stati forzati durante la gestione di un evento (tipicamente a seguito di azioni del mouse o simili);

GW_Echo_Prompt (default ">") il prompt che viene riportato nella finestra principale quando viene utilizzata la variabile Set_Prompt (vedere la sezione 18.4);

GW_BUFFER_SIZE (default 128) è la dimensione di un buffer interno a GWindows; principalmente limita la lunghezza delle righe di menù.

- gwindefs.h da includere *dopo* il parser;
- gwindows.h da includere *prima* della grammatica.

Ogni singola funzionalità aggiuntiva (widget) ha poi il proprio file di include (per esempio gpopmem.h per le finestre a pop-up).

❖ ❖ ❖

La libreria GWindows, per quanto indubbiamente potente, ha il difetto di essere abbastanza pesante. È molto facile, in fase di compilazione, incappare in uno dei limiti di preallocazione di Inform (per esempio MAX_PROP_TABLE_SIZE); sarà quindi necessario sperimentare con le opzioni di compilazione!

Inoltre, per la maggior parte delle operazioni, a mio parere è decisamente troppo intricata... se quindi non si ha bisogno di un layout dinamico o di una interfaccia particolarmente complessa, potrebbe convenire per semplicità l'utilizzo delle normali chiamate infglk.

◆ ◆ ◆

18.2. La gestione degli eventi

GWindows prende il controllo di *tutti* gli entry point forniti dalla libreria standard per personalizzare il comportamento sotto Glk (vedere anche la sezione 18.5); ovviamente fornisce i mezzi per gestire indipendentemente gli eventi all'interno delle varie finestre. Altri eventi (come per esempio quelli di ridisposizione e di ridisegno) sono gestiti in maniera automatica, così come il meccanismo di riassociazione degli oggetti Glk in caso di *restart* o *restore*.

18.3. Finestre e widget

Come già accennato, il layout delle finestre su schermo viene definito da una gerarchia di oggetti Inform sfruttando lo stesso concetto di *contenimento* definito dal modello del mondo di Inform stesso.

L'interfaccia utenti è quindi definita con una gerarchia di oggetti (rappresentanti finestre) appartenenti a classi definite da GWindows.

Un oggetto della classe WindowPair rappresenta una finestra paio, e quindi un punto di divisione dell'area dello schermo. Ogni oggetto di questa classe conterrà *due* oggetti-finestra (altre finestre paio o finestre foglia).

Le finestre foglia sono invece rappresentate dalla classe GWindow, e determinano (oltre al tipo di finestra da creare) la modalità di divisione della finestra paio a cui appartengono (attenzione: in realtà la modalità di divisione è definita all'interno della finestra paio, ma questa libreria delega il dimensionamento alle finestre sottostanti; è facile far confusione, ma il risultato finale è lo stesso).

Le tre sottoclassi di GWindow sono (ovviamente) TextBuffer, TextGrid e GraphWin; la finestra vuota non è considerata, in quanto completamente inutile.

Per finire, esistono delle sottoclassi specializzate di GWindow, già programmate per i casi più comuni. Queste sottoclassi vengono chiamate *widget*; a seconda che la finestra risultante sia direttamente utilizzabile oppure no, si parla di *widget concreti* o *widget astratti*, rispettivamente.

Ogni widget è definito in un proprio file di inclusione.

Tutti gli oggetti discendenti dalla classe GWindow possiedono la proprietà winid, contenente il riferimento all'oggetto Glk corrispondente. È importante ricordarsi che l'oggetto Inform *non è* una finestra Glk; piuttosto ne contiene una (accessibile con winid).

184. Variabili globali

Anche GWindows usa alcune variabili globali per tener traccia dello stato del layout; la maggior parte di queste devono essere considerate *di sola lettura*, tranne dove diversamente specificato.

Active_UI indica l'oggetto Inform radice del layout attualmente attivo, tipicamente una WindowPair. Questa variabile è utile per stabilire quale interfaccia è visibile al momento (se sono definiti più layout).

Inform usa la variabile gg_mainwin quando deve operare sulla finestra della storia (la finestra principale). La variabile Main_GWindow indica sempre il corrispondente oggetto GWindows (una TextBuffer, normalmente); per questo motivo vale sempre la relazione Main_GWindow.winid=gg_mainwin.

Inoltre, se si volesse ricevere l'input dei comandi da una finestra diversa da quella principale (una finestra separata), si può settare la variabile Input_GWindow all'oggetto relativo. Similmente, nella stessa circostanza, è possibile settare la variabile set_prompt ad un valore diverso da zero per far apparire il prompt *anche nella finestra principale* quando il giocatore conferma l'inserimento.

La variable GW_Abilities è un riepilogo dei descrittori gestalt più importanti; si possono verificare i singoli flag contenuti in questa variabile con una operazione di *and logico*: GW_Abilities & GWIN_xxxx. In questo modo si può adattare l'interfaccia presentata alle possibilità dell'interprete in uso.

Le costanti relative sono:

`GWIN_GWOK` Segnala che le finestre grafiche sono disponibili;

`GWIN_DROK` Se è possibile disegnare immagini nelle finestre grafiche;

`GWIN_MTOK` Se il mouse viene gestito nelle griglie di testo;

`GWIN_MGOK` Se il mouse viene gestito nelle finestre grafiche.

❖ ❖ ❖

Stranamente, è possibile sapere se è possibile disegnare immagini in una finestra grafica (il che è *praticamente garantito*, se ci sono finestre grafiche, ma non se anche i buffer di testo possono contenere immagini.

Nulla di grave, solo una chiamata gestalt in più da fare, nel caso.

◆ ◆ ◆

La variabile `cmd_override` ha una funzione del tutto analoga al parametro `abortres` di `HandleGlkEvent`: durante la gestione di un evento si può mettere in questa variabile *qualunque cosa possa essere passata alla funzione* `PrintAnything`; la stringa risultante verrà passata al parser come comando da eseguire.

La libreria GWindows utilizza spesso la funzione `PrintAnything`: un parametro valido per tale funzione è chiamato *valore stampabile* ("*printable*" nella documentazione originale).

Per finire, l'ultima variabile globale non è *esattamente* una variabile nel senso comune del termine. `GW_Massive_Error` è un catch point[1] gestito da GWindows. Basti sapere che per segnalare un errore fatale, anche durante la gestione di un evento, si può utilizzare lo statement:

```
@throw GW_Massive_Error msg;
```

In questo modo viene invocata la routine di gestione degli errori con la stringa diagnostica `msg`. Ovviamente è una funzionalità intesa principalmente durante lo sviluppo o per errori eccezionali.

18.5. Funzioni utente chiamate da GWindows

Anche se gli entry point standard della libreria di Inform sono già stati intercettati da GWindows, è comunque possibile (anzi, in almeno un caso necessario) scrivere funzioni che vengono richiamate *all'incirca nelle stesse circostanze*; in pratica GWindows rimane in mezzo e, nel caso, effettua la chiamata in modo simile a quella fatta dalla libreria standard.

[1] I catch point di glulx sono simili alle eccezioni di altri linguaggi di programmazione: una volta segnalati il programma ritorna indietro attraverso le chiamate fino ad arrivare al punto di catch.

Per la precisione, la chiamata di HandleGlkEvent è sostituita da una chiamata a GWindowsGlkEvent e ogni chiamata a IdentifyGlkObject è sostituita da una a GWindowsGlkIdentify; per quanto riguarda InitGlkWindow la situazione è più complessa.

GWindows chiama la funzione InitGWindows quando deve creare il layout iniziale (quindi in un momento paragonabile alla fase 0 di InitGlkWindow);

L'autore *deve* definire questa funzione, ma non solo, deve anche preparare il layout iniziale dello schermo. In particolare, deve assicurarsi che le variabili Active_UI, Main_GWindow ed eventualmente Input_GWindow (vedere la sezione 18.4 siano settate ai valori corretti, se ancora a zero. In caso contrario si tratta di un restart e quindi il meccanismo di riassociazione automatica le ha già valorizzate correttamente.

Per finire, l'ultimo stub, GWindowsHandleError viene chiamato in circostanze fatali (ovvero quando la libreria non è in grado di continuare in alcun modo); viene passato un parametro che indica l'errore che si è generato: la libreria ne definisce tre, al momento:

GW_ERR_ABS_GRAPH tipicamente indica che la grafica non è disponibile e si è richiesta una finestra grafica di dimensione prefissata; dato che le finestre grafiche, se non disponibili, vengono trasformate in finestre vuote, non è possibile effettuare il layout;

GW_ERR_NO_OPEN indica che la creazione di una finestra è fallita;

GW_ERR_RED_NO_WIN viene segnalato quando viene richiesto il ridisegno di una finestra non aperta, in genere quando tale richiesta è stata fatta manualmente.

È possibile ritornare dalla funzione GWindowsHandleError con il valore 2 per riprendere l'esecuzione dal punto dell'errore; generalmente questo non è sensato, comunque.

18.6. Definire il layout delle finestre

Come anticipato, per definire il layout delle finestre si definiscono degli oggetti Inform appartenenti alla classe WindowPair o ad una classe derivata da GWindow; nei casi più semplici si utilizzeranno le TextBuffer, TextGrid e GraphWin.

Come dettagliato nel Capitolo 3 ogni finestra che non è una *foglia* nell'albero deve essere una finestra paio; ed ogni finestra paio ha sempre *esattamente* due finestre figlie.

I parametri di divisione (la direzione di divisione, se fissa o in proporzione e la dimensione richiesta) vengono definite nelle foglie (esattamente come

Figura 18.1. Il layout di default dichiarato in GWindows

```
WindowPair root;
    TextBuffer -> mainwin;
    TextGrid -> statuswin
        with split 1,
            split_dir winmethod_Above
        has abssplit;
```

Figura 18.2. Il layout di sgw dichiarato in GWindows

```
WindowPair root;
    WindowPair -> mainarea;
        TextBuffer -> mainwin;
        GraphWin -> graphwin
            with split 240,
                split_dir winmethod_Above
            has abssplit;
    TextGrid -> statuswin
        with split 1,
            split_dir winmethod_Above
        has abssplit;
```

se si stesse chiamando glk_window_open); per la precisione, nella proprietà split_dir si specifica la posizione della nuova finestra (winmethod_Left, winmethod_Above, eccetera); nella proprietà split invece si specifica la dimensione (in percentuale o in unità). Se la divisione deve essere di tipo fisso, si assegna l'attributo abssplit. Il significato è identico a quello dettagliato nella sezione 3.2.

Per fare un esempio banale (un esempio *molto più* complesso si può trovare nella documentazione di GWindows), il layout di default può essere dichiarato come in Figura 18.1.

In questo caso è stato utilizzato l'operatore -> per comodità (in quanto la relazione di contenimento è immediata). La finestra di stato viene aperta come al solito di una riga al di sopra della finestra principale.

Allo stesso modo il layout, leggermente più complesso di un gioco sgw è dichiarato in modo simile a quello in Figura 18.2.

Ora che il layout è stato definito, bisogna attivarlo; se si utilizza un solo layout, basta farlo all'inizio in InitGWindows (che è una funzione da definire obbligatoriamente); il minimo indispensabile è visibile in Figura 18.3 (anche se le due variabili dovrebbero essere considerate di sola lettura, questa è

Figura 18.3. Una funzione InitGWindows minimale

```
InitGWindows [;
    Active_UI = root;
    Main_GWindow = mainwin;
];
```

l'inizializzazione e fa eccezione!).

A questo punto, *automaticamente*, il layout definito verrà gestito e visualizzato (con la gestione di default).

❖ ❖ ❖

È possibile definire come metodo (proprietà) di ogni finestra una funzione init. Queste funzioni vengono chiamate quando le finestre sono state create, ma prima del primo ridisegno.

◆ ◆ ◆

Attenzione, a questo punto la libreria standard di Inform conosce la finestra principale (è stata configurata con Main_GWindow) ma non sa nulla della finestra di stato! D'altra parte GWindows intercetta *anche* DrawStatusLine...

18.7. Disegnare nelle finestre

Come è noto, il contenuto delle finestre grafiche deve essere gestito in modo particolare; in particolare Glk non è tenuta a ricordarsi del contenuto di una finestra grafica e *in qualunque momento* può chiedere di ridisegnarne il contenuto.

Questo viene fatto tradizionalmente con gli eventi di ridisegno o di ridisposizione (anche se il giocatore ridisegna le finestre si pone il problema); il meccanismo fornito da GWindows è simile: viene chiamato il metodo redraw in una grafica nelle seguenti condizioni:

- È stato ricevuto un evento di ridisegno;
- è stato ricevuto un evento di ridimensionamento;
- la finestra è stata appena creata (ovviamente, per il contenuto iniziale);
- alla fine del turno, solo se la finestra ha l'attributo general (utile per forzare un ridisegno, per esempio durante un cambio di locazione);
- immediatamente, se viene chiamata la funzione GW_ForceRedraw.

Figura 18.4. Aggiornamento della finestra di stato in GWindows

```
! ... nel contesto della gerarchia
TextGrid -> status
    with split 1,
    split_dir winmethod_Above,
    update [;
        ! Il minimo sindacale della barra di stato
        glk_window_clear(self.winid);
        glk_window_move_cursor(0,0);
        print (name) location;
        glk_window_move_cursor(60,0);
        print score;
    ],
    has abssplit;
```

```
GW_ForceRedraw(finestra);
```

La funzione GW_ForceRedraw permette di forzare il ridisegno di una finestra, generalmente *prima della fine del turno*. Il sistema di ridisegno automatico (basato sull'attributo general) viene eseguito solo a fine turno; per questo motivo per aggiornare una finestra in anticipo (per esempio, un'azione di scorrimento all'interno della finestra, che non provoca azioni del parser) è necessario chiamare questa funzione.

18.8. Aggiornare le finestre

In aggiunta al metodo di redraw, GWindows chiama su *tutte* le finestre il metodo update alla fine del turno. In questo modo si può effettuare la *normale manutenzione* del contenuto delle finestre (durante il ridisegno bisogna ripulire la finestra, durante l'aggiornamento, non necessariamente; il contenuto precedente è intatto).

Un esempio tipico è la gestione della linea di stato ad ogni turno (Figura 18.4).

❖ ❖ ❖

Per la precisione, i metodi update delle finestre vengono fatti girare ad ogni turno quando la libreria standard utilizza DrawStatusLine; ed anche chiamate esterne a DrawStatusLine provocano l'esecuzione delle routine di update!

◆ ◆ ◆

18.9. Eventi del mouse

Per utilizzare il mouse (se disponibile) con GWindows, non è necessario richiedere manualmente l'evento alla finestra: in generale GWindows tiene attivi tutti gli eventi che possono esserlo simultaneamente (l'eccezione è l'input a linea, che avviene per default nella sola finestra di input designata, in quanto incompatibile con l'input a carattere).

Per questo motivo, la gestione del mouse si riduce a definire un metodo nella finestra in cui si vuole ricevere il click:

```
click_event [x, y;
! ...
]
```

GWindows chiama questo metodo con le coordinate ricevute da Glk: coordinate di cella nel caso di griglie di testo o coordinate di pixel per le finestre grafiche.

18.10. Eventi di input a carattere

Allo stesso modo del mouse, GWindows è sempre pronta a ricevere eventi a carattere (tranne che nella finestra di input designata); in questo caso viene chiamato il metodo `char_event`.

```
char_event [ch;
! ...
]
```

Prevedibilmente, in `ch` viene passato il codice del tasto premuto (sezione 2.4).

18.11. Forzare la linea di comando

In risposta ad alcuni eventi, tipicamente quelli del mouse, si può voler *convertire* l'azione del giocatore in un comando da eseguire (e quindi in una stringa per il parser).

GWindows offre due modalità per fare ciò: i due metodi sono completamente indipendenti e agiscono in modo distinto sulla linea di input.

18.11.1. Generare un comando sintetico

Il modo più semplice, gestibile senza grandi difficoltà anche con la sola libreria Inform, permette di generare completamente il comando da passare al parser e mandarlo immediatamente in esecuzione.

Quello che è stato eventualmente digitato nella linea di input è *completamente ignorato* e il parser viene attivato con il comando sostitutivo (l'evento termina quindi l'input di linea e passa il turno).

Per farlo basta semplicemente settare la variabile `cmd_override` ad un valore stampabile (qualunque cosa possa essere gestita da `PrintAnything`); di conseguenza, un banale esempio quale

```
cmd_override = "INVENTARIO";
```

... posto dentro alla gestione di un evento (per esempio, il click in un'area grafica) provoca l'esecuzione del comando "*inventario*" come se fosse stato digitato dal giocatore.

In base alle costanti di configurazione attive (sezione 18.1) il comando può anche essere riportato nella finestra della storia.

18.11.2. Modificare la riga di comando digitata

Un'altra possibilità, più utile per comandi parziali, è quella di *aggiungere del testo* in coda alla linea già digitata dal giocatore. La linea così modificata viene poi riproposta sulla linea di input per essere completata o corretta.

La funzione da usare in questo caso è `StreamWord`:

```
StreamWord("PRENDI␣");
```

La stringa passata viene accodata semplicemente a quanto è già stato digitato; lo spazio in coda permette al giocatore di inserire direttamente il nome dell'oggetto in questo caso.

❖ ❖ ❖

Ma attenzione: quello che è stato digitato precedentemente *non* viene in alcun modo alterato; per modifiche più radicali è necessario lavorare sulla variabile globale `buffer`, non documentata. Ovviamente il sorgente di `StreamWord` in `gwincls.h` è un buon punto di partenza per implementare questo genere di operazioni.

◆ ◆ ◆

18.12. Cambiare layout

Uno dei maggiori vantaggi con GWindows è la possibilità si sostituire, praticamente in qualsiasi momento, il layout di schermo con un altro precedentemente dichiarato; può essere utile, per esempio, per mostrare interfacce alternative o *modi* di gioco distinti, come un pannello di conversazione.

Per cambiare layout bisogna cambiare il riferimento nella variabile globale `Active_UI` (e le collegate `Main_GWindow` e `Input_GWindow`); il fatto che fossero in sola lettura era in realtà una vile menzogna; tuttavia, dopo averle

Figura 18.5. Una funzione InitGWindows in grado di gestire layout multipli

```
InitGWindows [;
    if (Active_UI == 0) {
        Active_UI = root;
        Main_GWindow = mainwin;
    }
];
```

riassegnate, è necessario resettare tutta l'interfaccia chiamando la funzione RestartGWindows().

Attenzione inoltre: la funzione InitGWindows (definita dall'autore) viene richiamata in ogni caso; è necessario quindi proteggere l'assegnazione dell'interfaccia di default (altrimenti rimarrebbe attiva *sempre quella*!). L'esempio di Figura 18.3 viene quindi modificato come in Figura 18.5.

18.13. Layout dinamici

Cambiare layout con la funzione RestartGWindows è efficace, ma generalmente provoca la chiusura e la riapertura di tutte le finestre, perdendone quindi il contenuto (e il buffer di scrollback); spesso questo comportamento non è accettabile: si pensi ad esempio ad un menù popup per le azioni disponibili; resettare l'interfaccia ogni volta è chiaramente improponibile.

La soluzione di GWindows al problema consiste nel *ridimensionare* le finestre; si crea un layout comprendente *tutte le finestre possibili* con un valore di split a zero per quelle inizialmente non visibili; in caso di necessità poi vengono *espanse* per essere rese visibili e poi *ridotte* quando devono essere nascoste.

Il meccanismo è ovviamente incentrato su glk_window_set_arrangement ed è spiegato in dettaglio nella documentazione di GWindows. Fortunatamente, è un'operazione così comune che esiste un widget pronto all'uso: GPopupWin (definito in gpopup.h). Un utilizzo tipico (preso direttamente dalla documentazione) è mostrato in Figura 18.6.

Le parti importanti sono, ovviamente la dichiarazione class GPopupWin che inserisce il codice del widget nell'oggetto, e la proprietà asplit che indica il valore di divisione *quando la finestra è aperta* e sostituisce normalmente la split.

Per implementare manualmente questo meccanismo (se proprio necessario) è necessario aggiungere alla finestra il metodo validate; questa funzione viene richiamata dopo la creazione e dopo la riassociazione delle finestre

Figura 18.6. Come gestire una finestra in popup

```
#include "gpopup.h"
! ... sempre nella gerarchia
TextGrid -> popupmenu
    class GPopupWin,
    with asplit 27,
        split_dir winmethod_Right;

[ ShowPopup;
  popupmenu.activate();
];

[ HidePopup;
  popupmenu.deactivate();
];
```

e il suo compito è di rendere la finestra visibile conforme alle proprietà dell'oggetto; basti pensare ad un restart con la finestra aperta: la finestra viene riassociata correttamente con l'oggetto corrispondente, ma è ancora aperta (cioè della dimensione indicata da asplit). Compito della validate, in questo caso, è di ridimensionare la finestra a dimensione 0 (cioè *chiuderla*, dal punto di vista del giocatore). Lo stesso discorso vale se si ricarica una posizione con una finestra aperta: in questo caso validate serve per *aprirla* alla dimensione specificata da asplit. Il codice in gpopup.h può servire da guida per l'implementazione di comportamenti simili.

18.14. Stili del testo

Il cambio dello stile del testo visualizzato all'interno di una finestra viene fatto con delle semplici chiamate a glk_style_set; la configurazione dei suggerimenti deve però essere fatta *prima* che la finestra venga creata, al momento opportuno. Fortunatamente GWindows fornisce un modo molto semplice per l'impostazione degli stili.

È sufficiente definire, nell'oggetto finestra desiderato, la proprietà stylehints ad un array di valori; per la precisione, tale proprietà deve essere composta da triplette stile hint valore; copiando come sempre un esempio dalla documentazione per chiarire:

```
! ... definizione della finestra
stylehints
```

```
style_Normal stylehint_Oblique 1
style_User1 stylehint_ReverseColor 1;
```

18.15. I widget in dotazione

❖ ❖ ❖

La documentazione in questo punto scade notevolmente in qualità... purtroppo al momento non ho il tempo e la voglia per provare tutti questi moduli; sono indicati solo quelli *probabilmente* più utili.

◆ ◆ ◆

Il sistema di modularizzazione di GWindows consente di definire delle classi di finestre con un comportamento in parte già definito. Si è già visto, per esempio, il widget GPopupWin (Figura 18.6).

Alcuni di queste (come lo stesso GPopupWin) sono *widget astratti*: forniscono qualche funzionalità aggiuntiva, ma la finestra deve essere comunque completata dall'autore.

Altre classi invece sono pronte all'uso: è sufficiente inserirle nella gerarchia delle finestre; gli oggetti di questo tipo vengono chiamati anche *widget concreti*.

18.15.1. La barra di stato

Il widget concreto GStatusWin fornisce una finestra di stato standard, di una linea, simile a quella della Z-machine. È sufficiente inserirla come foglia nel layout.

18.15.2. La finestra per citazioni

In sostituzione alla tradizionale finestra per citazioni (creata da Inform con lo statement box), il widget concreto GQuoteWin è meglio integrato in GWindows.

Per funzionare correttamente deve essere inserito come foglia nel layout, al di sotto di una divisione orizzontale.

Anche se è possibile gestire manualmente questa finestra (con il metodo quote, a cui deve essere passata una tabella di stringhe), il modo più semplice per utilizzarla è fare in modo che GWindows prenda il controllo *anche* della routine di visualizzazione utilizzata dallo statement box.

Per fare questo è necessario dichiarare Replace Box___Routine; prima di includere gquote.h (e comunque prima del parser); poi, durante l'inizializzazione del layout (tipicamente in InitGWindows) bisogna settare la variabile globale Quote_GWindow all'istanza di questa classe da controllare.

Fatto questo, il normale statement box utilizzerà la finestra specificata.

18.15.3. Finestre per immagini

Nel caso in cui si voglia una semplice finestra contenente un'immagine grafica, e nulla più, è disponibile la classe GImageWin.

Dopo averla inserita nella posizione desiderata della gerarchia, è possibile utilizzare il metodo setImage(img) per visualizzare la risorsa grafica img al suo interno (passando zero non viene mostrata nessuna immagine, ma il solo colore di sfondo); l'immagine viene automaticamente ridimensionata alla dimensione effettiva della finestra: per evitare deformazioni è necessario settare l'attributo aspected.

Per finire, il colore di sfondo della finestra è controllato dalla proprietà col (il colore codificato nel modo usuale).

18.154. Altri widget

Sono presenti altri widget (alcuni peraltro molto interessanti) nella distribuzione di GWindows; sfortunatamente la documentazione al riguardo è lacunosa e rimanda in genere ai singoli file di include.

È possibile che questa sezione venga espansa in successive edizioni di questa guida; per il momento verranno solo menzionati i widget presenti.

Per la gestione dei menù, sono presenti differenti classi: il menù standard (GMenu), a popup (GPopupMenu), su più colonne a larghezza fissa (GColumnMenu), su più colonne a larghezza automatica (GAutoMenu), con a capo automatico (GWrapMenu);

i menù grafici vengono gestiti dai widget GImageMap, GDrawImageMap, GImageGrid e GDrawImageGrid.

Versioni alternative a GImageWin che ripetono l'immagine invece di ridimensionarla sono GTileWin, GVTileWin e GHTileWin.

18.16. Trace e debug di GWindows

Includendo il file gconsole.h fra gwindefs.h e gwindows.h viene creata una finestra aggiuntiva sul lato inferiore della finestra, destinata a contenere i messaggi di debug generati dalla libreria (e dal gioco, volendo).

Per redirigere l'output sulla console è possibile chiamare GConsole.penon(); l'operazione viene terminata da una corrispondente GConsole.penoff(). Per semplici messaggi è utile anche la chiamata GConsole.write(stampabile).

18.17. Il GWindows Sound System

Il modulo audio `gsound.h` è stato progettato per l'uso in congiunzione con GWindows; deve essere incluso prima del parser; è anche utilizzabile *senza* GWindows, includendo in precedenza `gcdefs.h`.

La gestione dei canali audio Glk viene fatta in parallelo ad oggetti GWindows rappresentanti *canali virtuali*; in sostanza ogni canale virtuale occupa un canale Glk solo mentre è in esecuzione. Attenzione però: molte delle funzionalità di gsound si basano sulla possibilità di ricevere eventi di notifica audio; su alcune implementazioni Glk alcune cose potrebbero comportarsi quindi in modo strano.

Durante la riassociazione (quindi su *restart*, *restore* e *undo*) i canali Glk vengono resettati ed eventualmente riavviati in maniera automatica: sfortunatamente non è possibile riavviarli *dal punto in cui erano rimasti* quindi la sincronizzazione audio non è garantita (come nel resto di Glk, del resto).

18.17.1. Il multicanale

Per i casi più semplici (del tipo: "riproduci questo suono e facciamola finita"), gsound fornisce un oggetto di comodo, `GSMultiChannel`.

All'interno del multicanale le risorse audio sono sostanzialmente indipendenti fra di loro; vengono riprodotte per il numero di volte specificato (un numero finito, non può essere usato per la musica di sottofondo) e poi vengono sostanzialmente dimenticate. Inoltre non è disponibile la notifica al programma di fine riproduzione.

In linea di principio, il multicanale può far coesistere in riproduzione fino a 64 risorse audio, ma difficilmente una implementazione Glk sarà in grado di gestirle simultaneamente.

```
GSMultiChannel.play(risorsa, volume, ripet);
```

Questa chiamata avvia la riproduzione di una risorsa audio (specificata da `risorsa`) sul multicanale. Il `volume` indica il volume desiderato (se la libreria Glk in uso consente la regolazione del volume). I valori validi sono compresi fra `0` e `65536` (`65536` è il volume massimo, mentre `0` ovviamente è completamente muto). L'ultimo parametro `ripet` è il numero di volte per cui si vuole che venga ripetuta la risorsa audio; ovviamente deve essere almeno `1`.

18.17.2. Canali audio indipendenti

Nel caso in cui si voglia utilizzare della musica di sottofondo in formato MOD o ripetere all'infinito una risorsa (tipico sempre della musica di sottofondo) è necessario dichiarare un canale audio.

Un canale audio è un oggetto di tipo GSChannel o GSMusicChannel; entrambe le classi (derivate da GSvChannel) hanno le stesse funzionalità e la stessa interfaccia: l'unica differenza è che GSMusicChannel è in grado di evitare errori a runtime nel caso l'interprete non supporti risorse di tipo MOD (ed è pertanto destinata a riprodurre solo questo tipo di file audio).

Indicativamente, ad ogni canale creato per gsound corrisponderà un canale Glk, ma non necessariamente: esiste infatti una logica di virtualizzazione che tenta di rendere disponibili più canali audio di quelli effettivamente gestibili dalla libreria. Ovviamente, in questo caso, alcune delle risorse in esecuzione verranno interrotte per far posto alle nuove.

Il numero di ripetizioni della risorsa sul canale *non è* specificato durante la chiamata a play (che rimane identica, escluso questo argomento): viene, piuttosto, configurato nella proprietà number del canale. La costante GS_LOOP_FOREVER indica una riproduzione continua, senza termine.

È anche possibile sapere se il canale è attivo (attributo on), se associato ad un canale Glk (attributo open) e quale risorsa è attualmente attiva su di esso (proprietà playing).

Sono disponibili anche altre funzionalità avanzate (di scarsa utilità); la maggior parte di esse richiedono il supporto della notifica di eventi audio, comunque.

❖ ❖ ❖

gsound ha il concetto di *effetto audio*. In pratica sono due metodi del canale che vengono chiamati prima dell'inizio e dopo la fine della risorsa.

A causa del diverso grado di bufferizzazione delle varie librerie Glk, della frequenza della lettura degli eventi e di chissà quali fattori, la segnalazione di *fine riproduzione* può in effetti avvenire *leggermente prima o dopo* il termine della risorsa!

Bisogna quindi trattare con estrema cautela le (eventuali) notifiche di fine riproduzione.

◆ ◆ ◆

Per interrompere la risorsa in riproduzione su un canale si può utilizzare il metodo cancel().

Parte III.

Appendici

A

I Colori standard

Inutile dirlo, dal bianco-su-blu delle prime avventure testuali (anzi, no, dal verde-su-nero dei primi terminali a caratteri[1]) la resa *cromatica* delle avventure testuali ha fatto passi da gigante.

E, in effetti, la pluri-premiata *Photopia* giocata in monocromatico perde molto del suo feeling (anzi, probabilmente non sarebbe così osannata se non fosse per l'uso innovativo dei colori).

Certo, ognuno è libero di scegliere i colori che preferisce per la sua avventura. Ma volendo mantenere, come dire, lo stile *vintage* esistono in effetti una serie di colori standardizzati.

La lista dei colori (con i corrispondenti valori esadecimali da utilizzare per configurare gli stili è in Tabella A.1.

❖ ❖ ❖

Nota importante: le seguenti tabelle in glorioso TEXnicolor sono rappresentative dei colori *solo* quando osservate su schermo, per esempio in un documento PDF. La resa stampata potrebbe non essere rappresentativa.

◆ ◆ ◆

Lo standard proviene dalle specifiche per la Z-machine 1.1. Sono i colori utilizzati dall'interprete Infocom Amiga, non vi obbliga nessuno ad utilizzarli, ma *a me* piacciono, e sono quelli che Frotz richiede a garglk-mod.

A dire il vero con Glk non sarebbe possibile implementare il modello di colori della Z-machine. La Z-machine richiede, per ogni singolo carattere

[1] Posso testimoniare personalmente del corretto funzionamento di Frotz sulla console seriale di un RS/6000 a 120 MHz! (su AIX 3...). Se non fosse per il fatto che è ormai stata rottamata scommetto che anche glulxe/curses funzionerebbe egregiamente.

visualizzato, di poter impostare indipendentemente il colore del testo e dello sfondo.

Ma esistono, nelle profondità di garglk-mod, delle estensioni ad-hoc che permettono a Frotz di cambiare il colore carattere per carattere. E, prima che venga chiesto, *no*, non sono state rese disponibili esternamente in modo da essere utilizzate direttamente da glulx.

Forse con *adeguata* persuasione potrei pensarci...

Un altro insieme di colori utili è fornito dalla libreria *sgw*, descritta nel Capitolo 17. Per completezza, sono riportati in Tabella A.2.

Tabella A.1. I colori standard della Z-machine (e perché non usarli?)

Colore	Valori RGB	Glk (esadecimale)
Bianco	255, 255, 255	00FFFFFF
Grigio chiaro	176, 176, 176	00B0B0B0
Grigio medio	136, 136, 136	00888888
Grigio scuro	88, 88, 88	00585858
Nero	0, 0, 0	00000000
Rosso	232, 0, 0	00E80000
Verde	0, 208, 0	0000D000
Giallo	232, 232, 0	00E8E800
Blu	0, 104, 176	000068B0
Magenta	248, 0, 248	00F800F8
Ciano	0, 232, 232	0000E8E8

Tabella A.2. I colori definiti dalla libreria *sgw*

Colore	Costante	Glk (esadecimale)
Bianco	CLR_GG_WHITE	00FFFFFF
Nero	CLR_GG_BLACK	00000000
Grigio	CLR_GG_GREY	00BFBFBF
Rosso	CLR_GG_RED	00FF3030
Verde	CLR_GG_GREEN	0030FF30
Blu	CLR_GG_BLUE	000000A0
Ciano	CLR_GG_CYAN	0030FFFF
Magenta	CLR_GG_MAGENTA	00FF30FF
Giallo	CLR_GG_YELLOW	00FFFF30
Arancio	CLR_GG_ORANGE	00FF7F00
Marrone	CLR_GG_BROWN	007F3F00
Rosa	CLR_GG_PINK	00FF7FFF

B

I Caratteri Latin-1

La libreria Glk, fino alla versione 0.6.1 e nelle funzione di I/O di base per quanto riguarda la versione 0.7, utilizza caratteri con codifica Latin-1. Sono riportate per comodità le stesse due tavole preparate da Plotkin. In circolazione ovviamente ce ne sono di meglio e anche di peggio. I caratteri a disposizione tanto sono sempre quelli!

La pagina bassa della codifica (Figura B.1) corrisponde al set di caratteri ASCII e sicuramente funzionerà su qualunque implementazione Glk. La parte alta (Figura B.2) è invece specifica a Latin-1: in caso di dubbio sulla effettiva disponibilità di un carattere sarebbe bene utilizzare la rispettiva funzione gestalt.

Ovviamente, per chi utilizzasse le funzioni Unicode, queste tabelle sono solo parte della storia (i primi 256 code point di Unicode corrispondono infatti a Latin-1); la codifica Unicode *è un intero libro a se stante!*

Figura B.1. La codifica Latin-1: parte comune

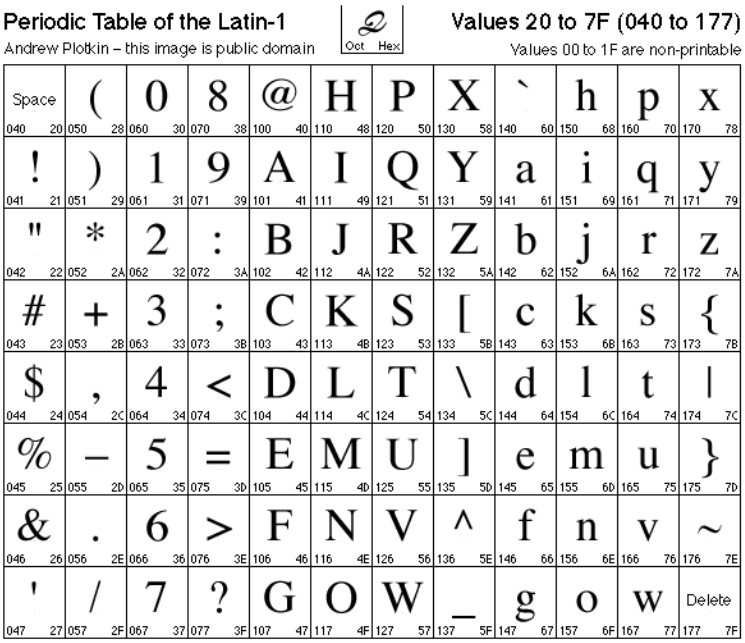
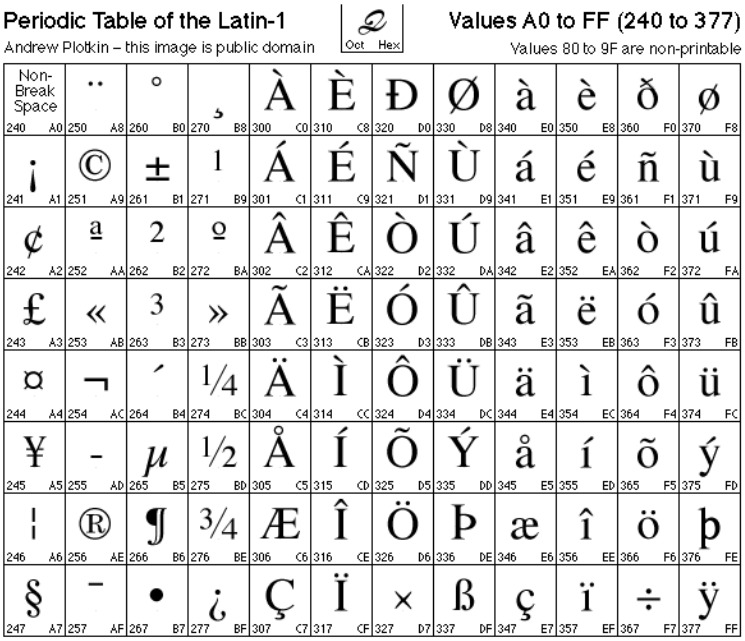


Figura B.2. La codifica Latin-1: parte estesa



Indice analitico