# MMIFIF

(**M**aniac **M**ansion **I**nterface **F**or **I**nteractive **F**iction)

Manuale d’uso

**V 0.8**

# SOMMARIO:

# Introduzione

MMIFIF è un template per Twine 2 che comprende un subset di macro e funzioni appositamente create per facilitare lo sviluppo di giochi del tipo “avventura testuale”. Si avvale del formato Sugarcube (v > 2.32) ed è scritto senza l’ausilio di codice esterno a Twine/Sugarcube. Alcune funzioni Javascript sono utilizzate per controllare elementi dinamici, come l’effetto di type-text e il “roteare” del loader nella schermata iniziale, ma si tratta sempre di implementazioni di trigger già presenti e documentati sulla reference ufficiale di Sugarcube, per cui il template è Twine “puro”, senza magheggi, accrocchi e tapulli di terze parti.

Il tipo di gioco che è possibile realizzare con MMIFIF è un ibrido tra l’impostazione di design delle “vecchie” avventure testuali con parser (es. “Avventura nel castello” di Enrico Colombini) e il sistema “visuale” utilizzato nel 1987 dall’adventure “Maniac Mansion”, e poi divenuto quasi uno standard “de facto” per molti titoli successivi, ovvero l’interfaccia “coi verbi sotto” alla scena in cui avviene l’azione, tramite i quali il giocatore costruisce i comandi da impartire al proprio avatar combinando oggetti e azioni in frasi di senso compiuto.

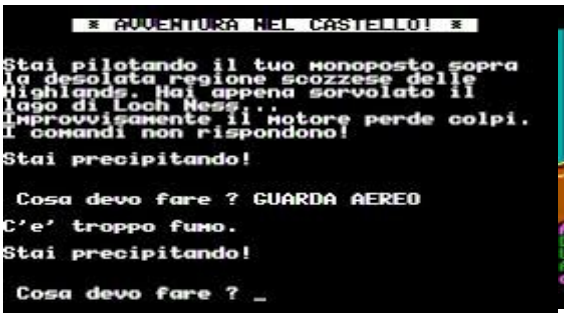



*Figura 1: Avventura nel castello (1982)*


*Figura 2: Maniac Mansion (1987)*

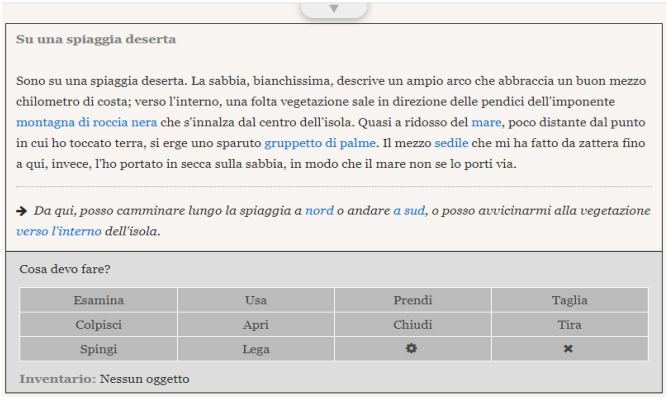



*Figura 3: MMIFIF*

Il concetto alla base del template è quello di rendere l'esperienza di una avventura testuale "vecchio stile" più adatta ai supporti moderni, eliminando la necessità di digitare direttamente verbi e comandi tramite l'adozione di una impostazione prettamente visuale, mediante la quale è possibile sfruttare adeguatamente i sistemi di input touchscreen di cellulari e tablet.

Una lista di verbi è posta nella parte sottostante della schermata, mentre nella parte superiore abbiamo la rappresentazione della scena; all'interno di questa, le parti di testo interattive rappresentano gli "hotspot" della scena stessa, e possono essere elementi dello scenario, altri personaggi o oggetti che il giocatore può raccogliere o con i quali può genericamente interagire. Si costruisce il comando da eseguire cliccando sui verbi e sugli oggetti, in modo da formare frasi di senso compiuto che il parser poi interpreterà per fornire un risultato: tutto sarà molto più chiaro e comprensibile provando il mini-gioco allegato a questo manuale.

I giochi realizzati con MMIFIF sono giochi Twine al 100%, quindi non necessitano di altro software per essere sviluppati. L'output è semplice HTML + Javascript, per cui i titoli realizzati con questo sistema possono funzionare perfettamente su qualsiasi dispositivo in grado di lanciare un browser internet moderno, o possono essere impacchettati tramite svariati software per produrre app per cellulari Android/iOS, o applicazioni Windows, Mac, Linux, etc…

Sebbene realizzare un qualche tipo di programma funzionante non sia mai troppo complesso in questo genere di scenari, la predisposizione di un ambiente che renda semplice la scrittura di AT complesse anche a chi non è avvezzo o non si è mai occupato di programmazione è un altro paio di maniche: ho cercato di racchiudere tutte le funzioni più complesse in widget che non sia mai necessario editare direttamente, tentando di fornire all'autore tutta una serie di strumenti per interagire con essi tramite comandi il più possibile semplici ed intuitivi.

Di seguito, sono presentate le funzionalità nello specifico, tramite un tutorial che guiderà l'autore nella creazione di un gioco completo partendo da zero.

Buona lettura!

# 1 - Twine: nozioni di base

Dalla homepage di Twine, www.twinery.org: "Twine è un tool open-source per raccontare storie interattive, non lineari". Nasce dall'esigenza del suo autore, Chris Klimas, di avere un tool in grado di organizzare e semplificare la stesura di quello che in origine era un semplice progetto di libro-game, ovvero quel genere di giochi in cui la narrazione procede a bivi e si dirama in base alle scelte del giocatore. Trovando il semplice hyperlink tra pagine html non proprio comodo in termini di progettazione (troppi piccoli file generati senza una organizzazione visuale dei collegamenti tra loro, oppure un solo grande file anch'esso senza una vera e propria struttura intellegibile, a meno di non usare tabelle e riferimenti), Chris iniziò a lavorare ad una specie di sotto-linguaggio che rendesse semplici le più comuni operazioni di scripting, e contemporaneamente sviluppò un'interfaccia utente chiara, che rendesse possibile leggere a colpo d'occhio la struttura di un intero progetto. Nel corso degli anni, Twine si è evoluto in un tool semplice ma molto efficiente, e l'adozione degli "Story Formats" rende possibile scegliere il tipo di linguaggio di scripting che più si adatta alle nostre esigenze; sia il nostro intento quello di scrivere una semplice storia a scelte multiple o quello di realizzare un simulatore di guerre navali, potremo "tarare" la complessità del nostro lavoro scegliendo di scrivere in un linguaggio semplice e chiaro, anche se magari limitato in alcuni aspetti, o in uno che rasenta la programmazione Javascript, più potente ma meno immediato per chi non mastica pane e funzioni void.

Di base, è possibile realizzare semplici giochi fondati sul paradigma del collegamento ipertestuale (link) senza neppure dover scomodare uno Story Format: Twine si presenta con alcune funzioni di markup che consentono immediatamente di creare "passaggi" (ovvero le singole parti della nostra storia, le locations della nostra mappa, o i paragrafi del nostro librogame, etc...) e collegarli tra loro. Vediamo come.

## 1.a - Scaricare e installare Twine 2 (o usarlo direttamente dal web)

L'indirizzo è:

http://twinery.org/

E' piuttosto spartano (come piace ai duri e puri dell'open-source). A destra ci sono i link per scaricare Twine sottoforma di applicazione. L'installazione è semplicissima, se comparisse un qualche messaggio paranoico di Windows che bolla l'applicazione come non sicura, è per via dei certificati. Al momento, Twine dovrebbe avere ottenuto la certificazione tramite la IF Foundation, ma nel caso il vostro sistema operativo vi mettesse in guardia, scavalcate tranquillamente gli avvisi e installate il programma. In alternativa, potete usarlo direttamente online cliccando sul link che dice "Use it online" (ma guarda un po').

## 1.b - Storie e salvataggi

Twine salva i vostri lavori nella cache, mentre lavorate, per cui non esiste una vera e propria opzione di salvataggio. In tanti anni di utilizzo non ho mai subito perdite di dati o avuto nessun grosso problema da questo punto di vista, ma per andare sul sicuro è sempre meglio fare una copia di backup del proprio lavoro ogni tanto; per fare questa semplice operazione basta cliccare su "Pubblica come file", nel menu opzioni di ogni storia, nella homepage del programma, o aprendo una storia tra quelle presenti in memoria e selezionando la stessa opzione dal menu basso a sinistra. In questo modo, creeremo un file html che conterrà il nostro progetto: per "lanciarlo" all'infuori di Twine, sarà sufficiente aprirlo con un qualsiasi browser, mentre per caricarlo nuovamente in Twine dovremo cliccare su "Importa da file", nella home. E' anche possibile creare un archivio di backup di TUTTE le storie che abbiamo attualmente in home.

## 1.c - Passaggi e collegamenti

Proviamo a cliccare sul bottone “+ Story” presente nella barra a destra, in home. Scegliamo un titolo per il nostro nuovo racconto interattivo e saremo subito di fronte alla nostra storia, vuota, con solo il primo passaggio impostato e pronto per essere editato. Facciamo doppio click su questo; come prima cosa cambiamo il suo titolo, mettendo qualcosa come “Prima stanza”. Il titolo sarà l’elemento utilizzato da Twine per identificare un determinato passaggio, per cui è meglio utilizzare dei rimandi chiari; i passaggi sono delle unità logiche di dati, possono rappresentare i paragrafi di un libro, le location di una mappa, le stanze di una casa, le parti di un dialogo etc…; in MMIFIF utilizzeremo i passaggi come scene: ogni passaggio rappresenterà una “stanza” o un generico ambiente, ed il giocatore potrà spostarsi tra una location e l’altra utilizzando dei semplici link.

Proviamo a scrivere un po’ di testo nel corpo del passaggio, a mo’ di descrizione:

```
“Sono in una piccola stanza fredda e buia.”
```

Sotto, aggiungiamo un link per uscire da questo tugurio: il sistema di markup di Twine risponde a quello che scriviamo tra una doppia parentesi quadra; proviamo ad aggiungere:

```
[[Esci dalla stanza fredda e buia]]
```

Ecco il testo completo da copiare e incollare nel passaggio (lo so che nessuno si mette davvero a scrivere le cose che legge su un manuale, anche se sarebbe meglio...):

```
Sono in una piccola stanza fredda e buia.

[[Esci dalla stanza fredda e buia]]
```

Istantaneamente, Twine creerà un secondo passaggio, nominandolo automaticamente "Esci dalla stanza fredda e buia". Avviando la storia (cliccando sul tasto play in basso a sinistra) ci ritroveremo nella nostra piccola stanza fredda e buia con un link sottostante che ci consentirà di spostarci immediatamente al secondo passaggio. E' già qualcosa, tuttavia il titolo del

secondo passaggio non è granché; sarebbe meglio qualcosa come "Seconda stanza", per cui sostituiamo il contenuto del link tra le due parentesi quadre con qualcosa tipo:

```
[[Esci dalla stanza fredda e buia|Seconda stanza]]
```

Twine creerà un altro passaggio, nominandolo "Seconda stanza" e linkandolo automaticamente al primo. Il funzionamento è semplicissimo: si scrive il testo che desideriamo visualizzare a schermo e lo si separa con un pipe (|) dal titolo del passaggio di destinazione. Cancelliamo pure il passaggio "Esci dalla stanza fredda e buia" che oramai non ha più referenti ed è quindi un passaggio "orfano".

Il sistema di link tra i passaggi è grossomodo tutto qui;  è possibile includere ulteriori comandi all'interno di un link Twine, come per esempio l'assegnazione di variabili, ma per addentrarci in questo aspetto del sistema serve introdurre almeno il concetto di Story Format. (Lasciatemi sfogare un attimo con una nota a margine: il 90% dei giochi prodotti con Twine si limita ad utilizzare il sistema di link tra passaggi e basta, ragion per cui molto spesso si confonde la piattaforma con il tipo di giochi realizzati con essa - "Ho appena giocato ad un Twine, fa schifo, è sempre la solita solfa!" - perchè naturalmente è molto semplice per chiunque scrivere delle storie a bivi utilizzando il sistema di markup, e quindi esistono moltissimi giochi realizzati seguendo unicamente questo semplice paradigma. In realtà, questa storia dei link è solo la punta dell'iceberg che Twine rappresenta, ed è possibile produrre pressoché qualsiasi tipo di gioco statico in 2D utilizzando Twine e i suoi sistemi di scripting; e se non bastasse, integrarlo per esempio  in Unity o Godot, per utilizzarlo come sistema di coding per, che ne so, un gioco tipo "Walking Dead" con grafica 3D, in cui tutto il sistema di decisioni è in realtà un "gioco" scritto in Twine, o fare mille altre cose: non credete a chi dice che Twine non serve a niente perchè è limitato, il più delle volte queste persone non hanno perso nemmeno tempo a provare effettivamente il sistema, o non sanno di cosa parlano).

## 1.d - Gli Story Formats

E se, oltre a muoverci tra un passaggio e l'altro, volessimo introdurre cose come un sistema di punti, un inventario, o un set di caratteristiche del giocatore (come Forza, Velocità, Destrezza etc...), o volessimo in qualche modo controllare se un determinato oggetto è posseduto o una certa azione è stata compiuta dal giocatore in precedenza?

Per questo genere di cose, che implicano la gestione di variabili, l'uso di costrutti etc..., è necessario utilizzare un sistema di scripting (= un linguaggio di programmazione) che ci consenta di accedere e manipolare questo tipo di informazioni. Come già detto, Twine non ha un unico "linguaggio": oltre al suo sistema di markup, che è comune a tutti i progetti, è possibile scegliere uno tra i tanti Story Formats a disposizione, o crearne uno da zero. Uno Story Format è un set di istruzioni, preconfezionate in una sorta di "dialetto" di Javascript, il cui scopo è quello di rendere quanto più semplici e chiare - all'interno di Twine - tutte quelle funzionalità che potrebbero servirci in corso d'opera, senza dover "sporcare" i nostri passaggi con orribili porzioni di codice Javascript. In questo modo, all'utente non viene più richiesto di occuparsi di problemi da programmatore, tipo lo "scope" di una funzione o la gestione della cache, ma gli viene fornito un linguaggio molto semplificato che ben si integra nel testo dei passaggi Twine. E' molto più difficile spiegarlo a parole che vederlo in pratica, ma prima di fare un esempio concreto dobbiamo scegliere che Story Format utilizzare: Twine propone di default "Harlowe", che è un formato molto adatto ai neofiti e pensato per rendere quanto più

possibile leggibile il "codice" prodotto, dato che questo dovrà essere integrato nelle pagine testuali del nostro racconto/avventura. Ho usato Harlowe per molti giochi, ma nel caso di MMIFIF ho dovuto muovermi verso un formato che si avvicinasse un po' di più al concetto di "linguaggio" vero e proprio: Sugarcube è uno Story Format storico di Twine, introdotto fin dalle prime versioni, e, oltre ad offrire praticamente qualsiasi funzionalità di base di un normale linguaggio di scripting, si presenta in una forma molto "classica" fatta di istruzioni inserite in tag. Prima di passare alla pratica, dobbiamo dire a Twine che la nostra storia utilizzerà Sugarcube come formato: nella home, clicchiamo sul link "Formati" nella barra a destra. Nel pop-up che si apre, scegliamo il tab "Add a New Format" (quello più a destra) ed inseriamo nel campo il link:

https://cdn.jsdelivr.net/gh/tmedwards/sugarcube-2/dist/format.js

Confermando, Twine andrà a scaricare e installare l'ultima versione di Sugarcube. NOTA: questo va fatto una volta sola, dopodiché Sugarcube sarà aggiornato e lo troverete nella lista dei formati utilizzabili. E' necessario (per lo meno fino alla nuova release di Twine) perchè la versione di Sugarcube fornita di default con Twine 2.3.9 è precedente a quella attuale, necessaria per far funzionare correttamente MMIFIF.

Apriamo ora il nostro progetto, clicchiamo sul menu in basso a sinistra e scegliamo "Cambia il Formato Racconto". Sezioniamo dal menu "Sugarcube 2.xx.x" e saremo pronti per iniziare.

## 1.e - Sugarcube e comandi base

Un'ultima infarinatura di concetti generali prima di entrare nel vivo del tutorial MMIFIF è necessaria per comprendere appieno la logica che sta dietro il template ed utilizzarlo senza timori di sorta, magari anche in modi "creativi" che non sono stati neppure previsti dal suo autore ;)

Mettiamo il caso che, tornando alla nostra piccola stanza fredda e buia, volessimo quantomeno rendere meno angusto l'ambiente accendendo una lampadina. Potremmo - è vero - mettere un bel link [[Accendi la lampadina|Prima stanza con luce accesa]] e creare l'illusione dell'accensione della lampadina "teletrasportando" il giocatore in un nuovo passaggio che è la riproposizione di quello precedente MA con la lampadina accesa, ma in questo modo avremmo due passaggi a rappresentare due stati diversi di uno solo, e alla lunga, con tutte le opzioni e le possibilità che di volta in volta vorremo rendere disponibili al giocatore, il nostro progetto diventerebbe un mostro ingestibile con migliaia di passaggi-doppione quasi identici tra loro. Meglio, in questo caso, utilizzare un po' di programmazione e mantenere una logica sensata in quel che facciamo. L'ambiente è uno - la piccola stanza buia - e la lampadina è al suo interno. Quindi, vogliamo che il testo all'interno della descrizione di questo singolo passaggio (la stanza) cambi in funzione del fatto che la lampadina sia accesa o spenta, SENZA dover creare un nuovo passaggio per ogni possibilità. Ci serve innanzitutto una cosa chiamata "variabile", una sorta di "post-it" su cui segnare se la lampadina è accesa o meno. In Sugarcube, una variabile si setta in questo modo:

```
<<set $variabile to "qualcosa">>
```

Utilizziamo nomi e descrizioni che abbiano senso per noi:

```
<<set $lampadinaAccesa to false>>
```

e mettiamo questa dichiarazione in un nuovo passaggio (clicchiamo su +Passaggio in basso a destra) che chiameremo StoryInit. Attenzione: questo particolare titolo fa parte di una serie di "titoli di passaggio riservati" con cui Sugarcube identifica particolari passaggi ben definiti, che contengono informazioni specifiche per la nostra storia. In particolare, "StoryInit" contiene tutte quelle operazioni che devono essere svolte una volta sola al lancio della storia. Ribadiamo: tutto quello che scriveremo in un passaggio intitolato StoryInit verrà processato dal sistema una volta sola all'inizio della storia. E' dunque molto comodo per compiere tutta quella serie di "preparativi" che in genere vengono approntati all'inizio dell'esecuzione di un programma; tra questi, naturalmente, c'è la dichiarazione di variabili. Non è assolutamente necessario dichiarare ogni variabile in StoryInit; se ad un certo punto della nostra storia settiamo una nuova variabile in un qualunque passaggio, Twine lo accetterà di buon grado, ma bisognerà tenere sempre presente che - fino a quel punto - avremo in giro una variabile "non definita" (che non vuol dire "vuota", attenzione), cosa che potrebbe causare qualche problema. Può sembrare un dettaglio di poco conto, ma già nel nostro caso, se non dichiarassimo la variabile $lampadinaAccesa come "falsa" all'inizio della storia, dovremmo pensare a qualche magheggio per non fare pasticci dichiarandola nello stesso passaggio in cui magari andremo anche a cambiare il suo stato; ma evitiamo divagazioni e fidiamoci dell'autore di questo manuale: scriviamo in StoryInit la nostra dichiarazione, dicendo a Twine che all'inizio abbiamo questa benedetta lampadina spenta, e torniamo alla nostra stanza buia:

```
Sono in una piccola stanza fredda e buia.

[[Esci dalla stanza fredda e buia]]
```

Modifichiamo un po' il testo, inserendo un paio di nuovi tag Sugarcube: <<if>> ed <<else>>. "If" ("se") controlla se una condizione è vera, e se lo è esegue quello che c'è all'interno del tag fino alla sua chiusura (ovvero <</if>>; ogni tag Sugarcube si chiude con un backslash di fronte al tag utilizzato in apertura, ad esempio: <<link>>fai qualcosa<</link>>, <<button>>fai qualcosa<</button>>, etc..., in modo similare al normale HTML), altrimenti esegue quello che c'è sotto "else" ("altrimenti", per l'appunto) :

`<<if $lampadinaAccesa is false>>` ⬅ SE la lampadina non è accesa...

`Sono in una piccola stanza fredda e buia.` ⬅ ALLORA fai questo.

`<<else>>` ⬅ ALTRIMENTI...

`Sono in una piccola stanza fredda illuminata da una lampadina accesa.` ⬅ fai quest'ALTRO.

`<</if>>` ⬅ fine.

E inseriamo la possibilità di accendere la lampadina sottoforma di link:

```
[[Accendi la luce|Prima stanza][$lampadinaAccesa to true]]
```

Analizziamo questo nuovo link: "Accendi la luce" è il testo che apparirà a schermo per il giocatore, mentre il "luogo" dove dovremo ritrovarci sarà sempre lo stesso, ergo "ricaricheremo" la stessa location "Prima stanza". Invece che chiudere qui il link, gli facciamo prima fare una cosa chiamata "setting": la parte "$lampadinaAccesa to true" dice a Twine di settare la variabile $lampadinaAccesa a "vera" PRIMA di mandare il giocatore al passaggio indicato precedentemente (che nel nostro caso è sempre la stessa stanza). In sostanza, la forma generica per un link con setter annesso è dunque:

```
[[Testo da visualizzare|titolo del passaggio][$variabile1 to qualcosa,
$variabile2 to qualcosa, etc...]]
```

Combiniamo queste nuove cose tra loro in modo da cambiare anche l'opzione per il giocatore, di modo che se la lampadina è accesa avremo la possibilità di spegnerla di nuovo:

```
<<if $lampadinaAccesa is false>>

[[Accendi la luce|Prima stanza][$lampadinaAccesa to true]]

<<else>>

[[Spegni la luce|Prima stanza][$lampadinaAccesa to false]]

<</if>>
```

In questo modo, anche il link relativo all'interazione con la lampadina cambierà in funzione del fatto che questa sia accesa o spenta, coerentemente con la descrizione della stanza. Ecco il testo completo da mettere in "Prima stanza":

```
<<if $lampadinaAccesa is false>>

Sono in una piccola stanza fredda e buia.

<<else>>

Sono in una piccola stanza fredda illuminata da una lampadina accesa.

<</if>>

<<if $lampadinaAccesa is false>>

[[Accendi la luce|Prima stanza][$lampadinaAccesa to true]]

<<else>>

[[Spegni la luce|Prima stanza][$lampadinaAccesa to false]]

<</if>>

[[Esci dalla stanza fredda e buia|Seconda stanza]]
```

E la dichiarazione da scrivere in StoryInit:

```
<<set $lampadinaAccesa to false>>
```

Già che ci siamo, scriviamo qualcosa anche in "Seconda stanza" e forniamo un link per tornare indietro, collegando correttamente le due locations:

```
Sono nella seconda stanza, qui è tutto molto più confortevole.

[[Torna alla stanza fredda|Prima stanza]]
```

Dovremmo ora avere un totale di tre passaggi nel nostro progetto: StoryInit, Prima stanza e Seconda stanza. Clicchiamo play e valutiamo il risultato. Ora possiamo divertirci ad accendere e spegnere l'interruttore, fantastico!

Esistono molti altri comandi/tag (Sugarcube li chiama "macro") oltre a <<if>>, <<else>> e <<set>>, ma il funzionamento di base è sempre lo stesso, e li vedremo via via che ci addentreremo nella costruzione di un gioco con MMIFIF.

## 2 - Tutorial: "Rapina a casa Berluti"

Entriamo nel vivo proponendoci di realizzare un brevissimo gioco di avventura intitolato "Rapina a casa Berluti", in cui impersoniamo un ladruncolo che si deve intrufolare in una ricca magione per rubare un prezioso cimelio: non sarà niente di che, ma cercheremo di metterci dentro un po' tutto quello che si potrebbe dover affrontare nella stesura di una avventura più grande ed elaborata, in modo da fare chiarezza su tutti i vari aspetti del template.

Ora, quando si progetta un gioco di avventura, la prima cosa da fare, che si stia lavorando in Twine o in qualsiasi altro sistema di sviluppo, è chiudere tutto quello che non sia un editor di testo - o ancor meglio spegnere il pc e prendere un bloc notes e una penna - e mettersi a scrivere. L'implementazione di ambienti, personaggi, dialoghi, azioni, oggetti, etc... deve necessariamente avvenire DOPO aver steso un quadro completo e chiaro del gioco. C'è chi (come il sottoscritto) ama fare il contrario e mettersi a buttar giù idee e pezzi di storie MENTRE sviluppa l'engine per tenerle assieme, ma vi posso assicurare che è un modo SBAGLIATO di fare le cose... nessuno vi punirà o vi farà la ramanzina se decidete di fare di testa vostra, ma dover buttare via sei mesi di lavoro perchè non avevate previsto quella singola possibilità che si è poi rivelata assolutamente necessaria alla coerenza narrativa della storia, beh, NON è una bella sensazione. E se ci passate una volta, a partire da quella successiva avrete tutto l'interesse a fare le cose a modo: preparate PRIMA un documento di progetto (si chiama proprio così), controllatelo e pulitelo più che potete, e POI passate ad implementarlo su computer. Che un gioco sia piccolo, medio, grande o enorme come le odierne produzioni tripla A, non si scappa: anche alla Ubisoft, prima di muovere un dito, vogliono avere un pacco così di documenti, storie, dialoghi, immagini, concept-art, e Dio sa cos'altro; per cui, armiamoci di umiltà e facciamolo anche noi. Non ci vorrà molto.

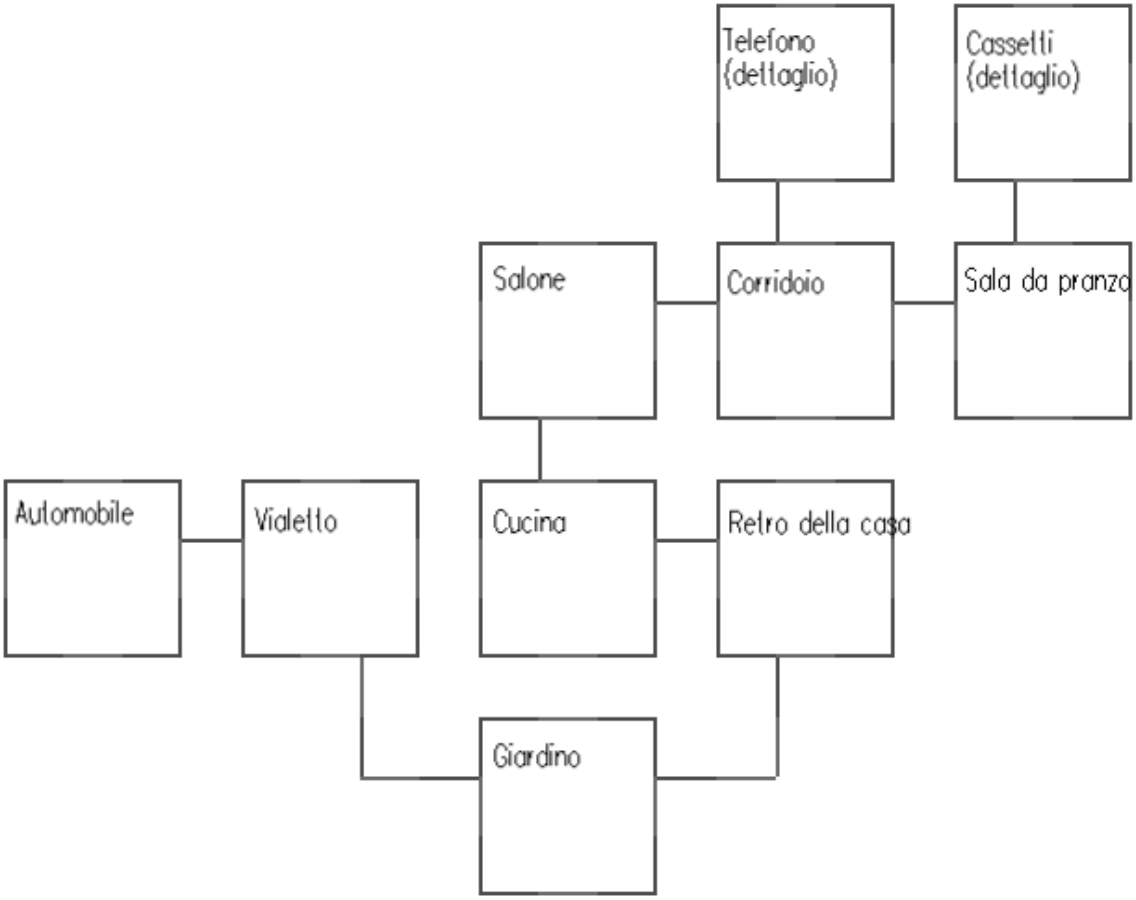
### 2.a - Il documento di progetto

Il soggetto è semplice e l'abbiamo già esplicitato: siamo nei panni di un ladro che si deve intrufolare a casa Berluti per rubare un cimelio. Il cimelio in oggetto è una preziosa bottiglia di vino custodita in una cassaforte. Inizieremo fuori dalla casa, seduti in auto con il nostro compare Bosko, parcheggiati di fronte all'ingresso della villetta. Bosko ci sarà utile per due motivi: ci fornirà qualche dettaglio utile, e fungerà da "guardiano del cancello"; se gli daremo la preziosa bottiglia, la missione sarà stata un successo e partiremo sgommando verso la prosperità, ma fino a quel momento il nostro compare si rifiuterà di riaccendere il motore dell'auto e portarci via.

Definita la situazione iniziale e la condizione di vittoria finale, dobbiamo occuparci di quello che c'è in mezzo: per appropriarci della bottiglia, dovremo logicamente entrare in casa Berluti. Approfitteremo del fatto che la famiglia è in vacanza all'estero, e dell'ora tarda, per scassinare la serratura della porta secondaria con un arnese da scasso che ci faremo consegnare dal nostro complice, ed entreremo così dal retro, ritrovandoci in cucina. Da qui ci sposteremo verso il salone d'ingresso, dove troveremo un cappotto appeso all'appendiabiti. Nella tasca del cappotto, troveremo una piccola chiave. Ci sposteremo lungo il corridoio, dove troveremo uno

strano telefono vecchio stile posto su un mobiletto sotto uno specchio. Oltre, entreremo in sala da pranzo, dove troveremo una credenza con dei grandi cassetti. Utilizzando la chiave, apriremo uno dei cassetti, ma sarà vuoto. Tuttavia, potremo tirarlo a noi per sfilarlo dal mobile, e attaccata dietro di esso con del nastro adesivo, troveremo una busta contenente una nota. La nota è un codice di 8 cifre, che comporremo con il telefono in corridoio causando l'apertura della cassaforte segreta dietro lo specchio. Qui troveremo la bottiglia: la prenderemo, torneremo in auto e la consegneremo a Bosko concludendo l'avventura.

Estrapoliamo da questa descrizione sommaria i dati che ci servono; prima di tutto, stendiamo una mappa delle ambientazioni necessarie:




Per quanto riguarda la geografia del nostro gioco, possiamo scegliere il paradigma che più ci viene comodo: si possono organizzare le locations in una "griglia" di ambienti collegati tra loro secondo la logica dei punti cardinali, con le varie direzioni (nord, sud, est, sudest, etc...) come opzioni di movimento, in modo simile alle avventure testuali vecchia-scuola; o possiamo scegliere un sistema più astratto, in cui cui i collegamenti tra scene sono meno rigorosi e più in armonia con l'ambiente, nello stile delle avventure grafiche... o si può fare un misto: quel che importa, è avere chiaro lo schema dell'intero ambiente di gioco e sintetizzarlo in "locations", ambienti, stanze, quello che sia, purché segua una sua logica.

Definiti i luoghi, passiamo a stilare una lista di azioni che saranno necessarie per portare a termine l'avventura. Tralasciando i movimenti tra le scene, concentriamoci sulle combinazioni di verbi e oggetti che ci porteranno alla risoluzione dell'enigma:

1. - (in automobile) parla con Bosko: all'inizio del gioco saremo seduti in auto, in compagnia di Bosko. Vogliamo fare in modo che il giocatore scambi almeno un paio di battute con questo personaggio, per lo meno per farsi consegnare l'arnese da scasso standard in dotazione ad ogni aspirante rapinatore.
2. - (sul retro) apri la porta con l'arnese da scasso: apre la porta sul retro.
3. - (nel salone) esamina il cappotto: fa cadere la piccola chiave dalla tasca dell'indumento appeso all'appendiabiti.
4. - (nel salone) prendi la piccola chiave: prende la chiave.
5. - (in sala da pranzo) esamina la credenza: effettua uno "zoom" sulla credenza, dando accesso al tag "cassetto".
6. - (guardando la credenza) apri il cassetto con la piccola chiave: apre il cassettone.
7. - (guardando la credenza) tira il cassetto: estrae il cassettone dalla sede nel mobile e rileva la presenza di una busta appiccicata sul retro dello stesso, prendendola automaticamente.
8. - (ovunque) apri la busta: apre la busta ed estrae un foglietto con una nota scritta a penna.
9. - (ovunque) esamina la nota: legge il foglietto, è un codice di 8 cifre.
10. - (in corridoio) esamina il telefono: effettua uno "zoom" sull'oggetto, dando la possibilità di comporre un numero.
11. - (guardando il telefono) componi sulla tastiera del telefono il codice contenuto nella nota: causa l'apertura della cassaforte segreta, che si rivela trovarsi dietro lo specchio a scomparsa, e dentro la quale troveremo e prenderemo la bottiglia.
12. - (in automobile) dai la bottiglia a Bosko: fine del gioco.

Da questa lista di azioni, estrapoliamo gli oggetti e i personaggi strettamente necessari a portare a termine l'avventura:

1. - (in automobile) Bosko, il nostro compare.
2. - (da farci consegnare previo dialogo con Bosko) un arnese da scasso.
3. - (sul retro) la porta sul retro della casa.
4. - (nel salone) il cappotto appeso all'appendiabiti.
5. - (da prendere) piccola chiave nella tasca del cappotto.
6. - (in sala da pranzo) la credenza.
7. - (esaminando la credenza) il cassettone.
8. - (da prendere/leggere) la nota dentro la busta.
9. - (in corridoio) il telefono.
10. - (esaminando il telefono) i tasti numerati.
11. - (da prendere) la bottiglia.

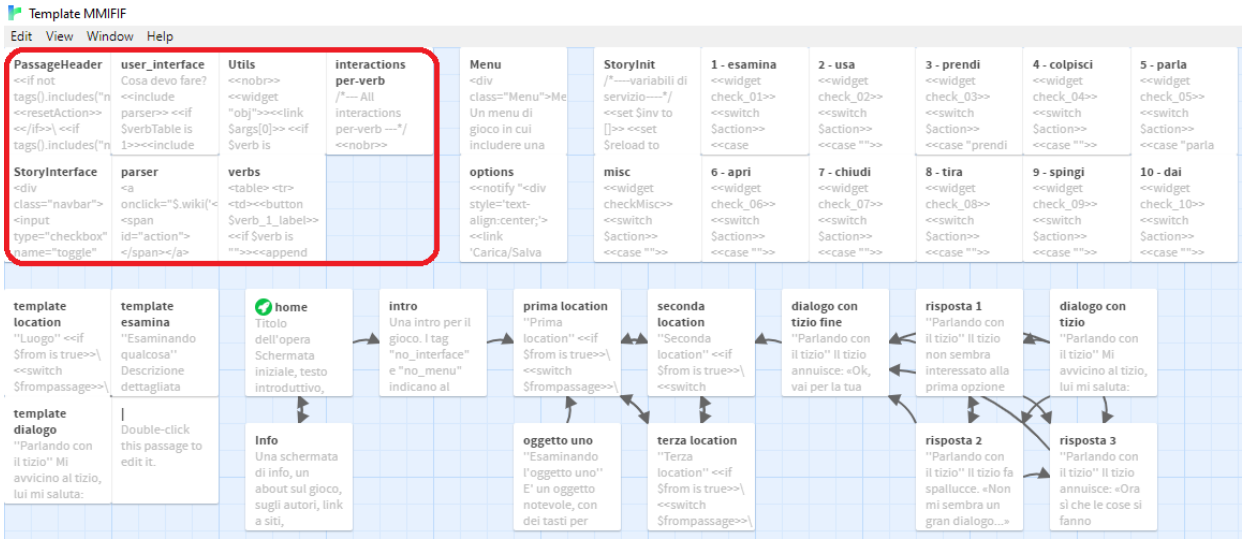
Questi sono gli 11 oggetti e le 12 azioni strettamente necessarie a portare a termine il gioco. Naturalmente potremo includere molti altri dettagli, per esempio fornendo delle battute non generiche in risposta a tentativi infruttuosi ma non illogici (per esempio, tentare di aprire la porta sul retro senza utilizzare l'arnese adatto). Potremo inserire una quantità di oggetti secondari, magari di puro arredo, e la possibilità di esaminarli o manipolarli, e quant'altro ci detta la fantasia, ma di base dobbiamo definire per lo meno quello che sarà strettamente necessario a dare un capo e una coda al nostro progetto. Il resto può venire dopo, una volta che la meccanica di base sarà implementata senza errori e ci saranno un inizio ed una serie di azioni che portano ad una logica conclusione. Avremo tempo, in successive revisioni, di inserire una miriade di oggetti inutili con descrizioni dettagliate di qualsiasi cosa, risposte divertenti ai tentativi più assurdi, opzioni di dialogo inedite, e magari la possibilità di ordinare una pizza per Bosko chiamando col telefono un numero particolare, ma per ora concentriamoci sull'essenziale e completiamo il nostro documento di progetto con il minimo indispensabile: abbiamo una storia, una mappa, una lista di oggetti e una di azioni da implementare. Non serve altro. Cominciamo.

## 2.b - Configurazione di MMIFIF

Apriamo Twine e clicchiamo “Importa da file” nel menu a sinistra. Importiamo il file “Template_MMIFIF.html”: Twine creerà una nuova storia intitolata “Template MMIFIF” e la aggiungerà a quelle presenti in home page. Clicchiamo sull’icona a forma di ingranaggio a destra del titolo, e selezioniamo “Duplica storia”, scegliamo un titolo (per esempio “Rapina a casa Berluti”) e ci ritroveremo con una copia del nostro modello pronta per essere editata. Questo passaggio non è strettamente necessario (potremmo semplicemente lavorare su “Template MMIFIF” rinominando la storia) ma è consigliato per due ragioni: la prima è che in questo modo avremo sempre da parte una versione “pulita” del template iniziale, utile come riferimento se dovesse capitare qualche pasticcio, e l’altra – più importante – è che Twine assegna automaticamente un “IFID” ad ogni storia creata, e se lavorassimo direttamente su “Template MMIFIF” senza prima crearne un “clone”, ci ritroveremmo ad avere lo stesso “IFID” del template e – potenzialmente – di ogni altra storia creata direttamente dal template stesso. Cos’è un “IFID”? E’ un codice seriale che identifica un’opera di interactive fiction a livello globale, simile al codice “ISBN” per i libri. Esistono software che permettono di generare un codice IFID autonomamente, ma Twine integra già la funzione di default. E’ possibile vedere l’IFID associato alla nostra storia nelle statistiche della storia stessa, cliccando sul menu in basso a sinistra e selezionando “Statistiche storia”: oltre al codice IFID, qui potremo leggere alcuni dati riassuntivi sul nostro progetto, come per esempio la quantità di parole e di passaggi utilizzati, e l’eventuale presenza di “link rotti” (in genere, si tratta di errori da correggere causati da link che non portano da nessuna parte).

Diamo ora un’occhiata alla nostra storia, che è ancora una semplice copia del template di base. Nella parte superiore sono presenti tutti i passaggi che servono per far funzionare la “logica” del nostro gioco, mentre sotto c’è la mappa, gli oggetti che generano una risposta di tipo “zoom” o “primo piano” (e che hanno quindi un loro passaggio dedicato) e gli eventuali dialoghi con altri personaggi. Sono anche già predisposte una schermata “Home” per il nostro gioco, un passaggio di “info/about” e i tre template di base per la creazione veloce di luoghi, oggetti “zoom” e dialoghi. Analizziamo nello specifico quello che vediamo sullo schermo, per

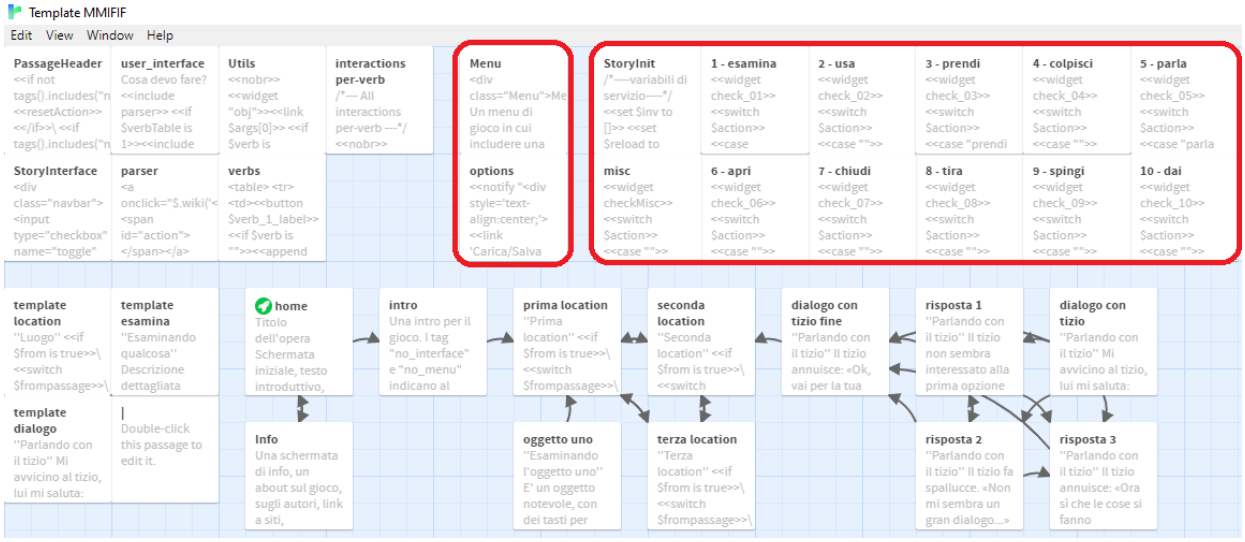
fugare ogni possibile dubbio su “cosa serve a cosa” (i nomi e la disposizione di alcuni passaggi potrebbero differire LEGGERMENTE in base alla versione del template):




I sette passaggi in alto a sinistra contengono tutta una serie di script e funzioni che non vanno modificati (a meno di non sapere cosa si sta facendo). Nello specifico, solo per chiarezza:

- StoryInterface contiene la struttura dell’interfaccia-utente. E’ un nome di passaggio riservato, come StoryInit, che abbiamo visto in precedenza, e serve per definire lo “scheletro” html per le pagine dell’opera;
- PassageHeader è un altro passaggio riservato: ciò che contiene viene eseguito all’inizio di ogni altro passaggio;
- user_interface contiene l’interfaccia utente vera e propria, ovvero i verbi, il parser e l’inventario;
- parser è il pezzettino di testo che va scrivendosi via via che il giocatore clicca su verbi e oggetti;
- verbs è la tabella dei verbi;
- Utils contiene tutti gli script di servizio, gran parte dei quali verranno usati in questo tutorial. Si tratta di un insieme di funzioni pensate per facilitare la vita all’autore, che racchiudono in singoli “widget” delle porzioni di codice che si occupano di vari aspetti del gioco, come per esempio la gestione dell’inventario, il sistema di notifiche pop-up, il controllare se un determinato oggetto è posseduto o meno etc…
- interactions per-verb è una sorta di “smistatore”: identifica quale verbo regge la frase impostata dal giocatore, e spedisce l’azione al passaggio corrispondente.

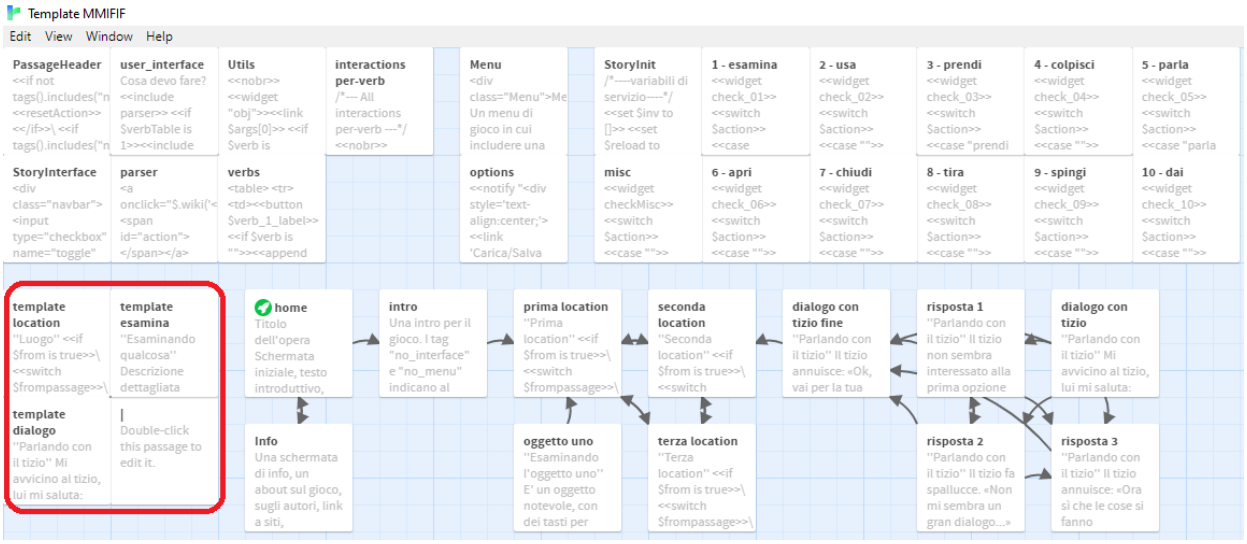
A destra di questi passaggi “da non toccare”, troviamo una serie di passaggi che andranno in qualche modo editati dall’autore:




Vediamo brevemente questi passaggi:

- StoryInit lo conosciamo già;
- options contiene le opzioni che compaiono al giocatore quando clicca sull’icona a forma di ingranaggio. Andrà editato solo se si desidera modificare in qualche modo il menu opzioni.
- In Menu c’è il contenuto che vorremo visualizzare nella “tendina” che scende dall’alto cliccando sul piccolo triangolino rivolto verso il basso.
- A esamina, usa, apri, chiudi, tira, spingi, dai, prendi, parla e colpisci vengono “passate” le corrispettive azioni dal passaggio “interactions per-verb”. Tutte le azioni che rispondono al verbo “esamina” saranno dunque contenute nel passaggio “esamina”, tutte quelle che rispondono al verbo “usa” saranno contenute nel passaggio “usa” e così via.
- misc contiene tutte quelle azioni che non sono previste nei casi precedenti, come per esempio la composizione di un numero di telefono o la combinazione di una cassaforte.

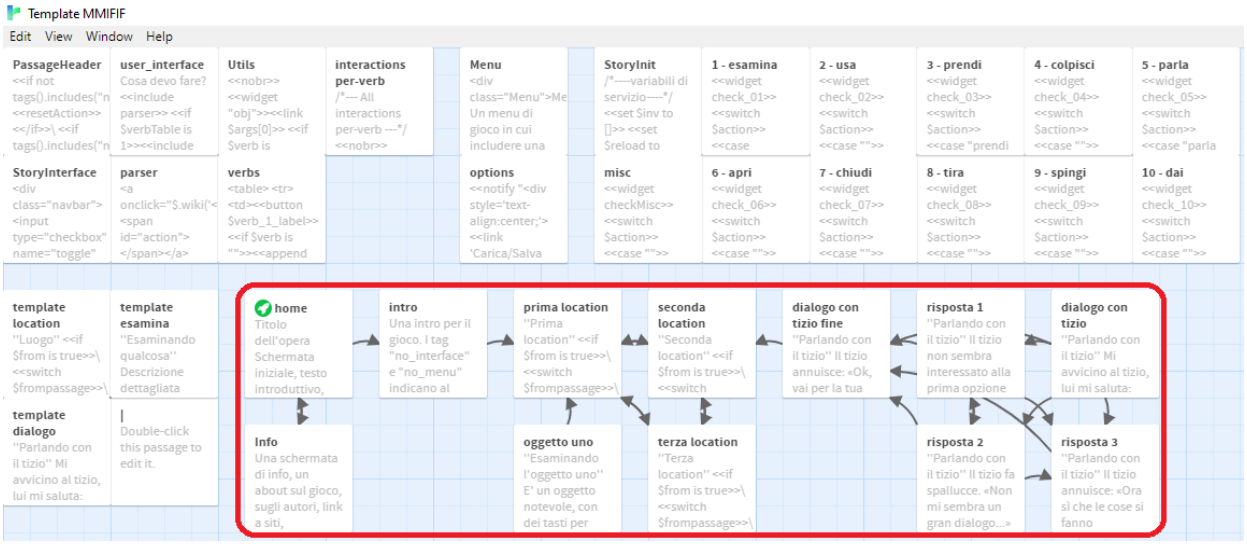
Nella parte sottostante, troviamo altri quattro passaggi che sostanzialmente servono solo da riferimento:




In dettaglio:

- template location, template esamina e template dialogo contengono i modelli di riferimento per i luoghi, le "zoomate" sugli oggetti e i dialoghi: sono semplicemente uno scheletro vuoto da copiare e incollare, per rendere più veloce la creazione di mappe, dettagli di oggetti (vedremo poi a cosa ci riferiamo con questa storia dello "zoom"), e dialoghi;
- il passaggio “|” è creato automaticamente da Twine per via del fatto che nei template precedenti i link sono “vuoti”, ragion per cui il sistema legge l’unico carattere “|” come titolo del passaggio di destinazione: non preoccupiamoci, cambierà a automaticamente quando popoleremo i nostri passaggi con i giusti link.

A destra dei template, c’è il nostro gioco vero e proprio:




Come già detto, ho provveduto a creare una struttura di base che comprende una pagina Home, una di informazioni, una di introduzione e tre locations (o stanze) correttamente linkate tra loro. Sono anche presenti un paio di oggetti di scena e un personaggio con cui dialogare, in

modo da avere la gran parte delle funzionalità di MMIFIF già implementate “sul campo” e pronte per essere modificate a piacimento. Caricato il template e capito (più o meno) com'è strutturato, possiamo iniziare a creare qualcosa. Inizieremo dalla mappa.

## 2.c - Mappa e luoghi: come si costruiscono gli ambienti

Nelle tre stanze di esempio già presenti nel template sono comprese gran parte delle funzionalità che reggono la costruzione degli ambienti di gioco: il sistema riconosce se si sta provenendo da un’altra location e da dove, e se siamo già stati nella location stessa, e predispone il testo di conseguenza, proponendo pezzetti di descrizione che variano in funzione della “strada” che abbiamo dovuto affrontare per arrivare lì dove ci troviamo, e decidendo se mostrarci una descrizione estesa o meno dell’ambiente a seconda che sia o meno la prima volta che visitiamo un determinato luogo. Altresì, è possibile costruire i link per le altre locations in modo dinamico, proponendo di “tornare” al luogo in cui ci trovavamo in precedenza oltre che di dirigersi verso nuove strade. Questo serve soprattutto a rendere più “vivo” il testo, che cambierà a seconda di tutti questi parametri in modo coerente con la mappa di gioco: il sistema “vecchio stile” delle avventure testuali già proponeva - ad esempio - lo switch intelligente tra “descrizione lunga” e “descrizione breve”, per evitare di riproporre al giocatore una quantità di testo (un “muro”, come si dice in gergo) ogni volta che questi rientrava in un ambiente già conosciuto. Ho cercato di estendere il concetto, includendo come già detto la possibilità di manipolare il testo di apertura di ogni passaggio in funzione del luogo da cui si proviene: se per esempio ci troviamo alle pendici di un’altura e la mappa ci consente di arrampicarci su per le rocce, una volta arrivati in vetta il testo dovrebbe aprirsi con qualcosa tipo: “Dopo una faticosa scalata, sono finalmente arrivato in cima alla montagna”, piuttosto che “teletrasportare” semplicemente il giocatore nella location di destinazione. Con la stessa logica, se ci spostiamo dalla strada entrando in un centro commerciale, una volta dentro, tra i link per muoverci in giro, troveremo “Torna all’esterno (sottinteso: “da cui provengo”)”. Questo serve più che altro a rendere meno difficoltoso per il giocatore l’orientarsi tra gli ambienti di gioco: durante le prime fasi di esplorazione di un’isola deserta o di un bosco maledetto, per esempio, per il giocatore potrebbe risultare stancante tenere una sorta di “mappa mentale” dei luoghi; per questa ragione, gli evitiamo per lo meno l’imbarazzo di tornare per sbaglio nello stesso luogo da cui proveniva, e mettiamo un punto certo sul fatto che ha percorso PROPRIO quella strada per arrivare dov’è. So che sembra contorto e inutile, ma una volta messo in pratica, questo sistema permette di rendere le descrizioni e le “transizioni” tra luoghi molto più vive e più “fluide”, consentendo una narrativa meno “robotica” e fredda.

Ad ogni modo, uno degli aspetti migliori del template è che tutto quanto detto sopra può essere bellamente ignorato, o possiamo decidere di utilizzare solo alcune delle funzionalità proposte, senza che il sistema ne risenta. Di base, possiamo scrivere solo una descrizione e fornire sempre gli stessi link per le uscite, senza badare a luogo di provenienza, link di “ritorno” etc…: il template funzionerà ugualmente.

Vediamo ora nel dettaglio la struttura di un’ambientazione MMIFIF; apriamo il passaggio "template location" e diamo un'occhiata a cosa contiene (ho numerato le righe per comodità):

1. `''Luogo''`

```
2. <<if $from is true>>\
3. <<switch $frompassage>>\
4. <<case "">>\
5. \
6. <<case "">>\
7. \
8. <<case "">>\
9. \
10. <</switch>>\
11. <<set $from to false>>\
12. <<else>>\
13. \
14. <</if>>\
15. <<if $longDesc is true>>\
16. <<set $longDesc to false>><<else>>\
17. <</if>>\
18. ----
19. <<iconforward>> //Da qui, posso <<if $frompassage is ""
    >>tornare<<else>>dirigermi<</if>> [[|][$from to true, $frompassage to passage()]]//
```

La riga 1 contiene sostanzialmente il nome della location tra due apici. Gli apici fanno parte del sistema di markup di Twine, e significano "scrivi in grassetto quello che sta tra noi": qualsiasi testo scriviamo tra una coppia di apici ('') e un'altra, Twine lo interpreterà come carattere "bold". Dalla riga 2 alla riga 14 c'è il sistema per controllare da dove viene il giocatore e le frasi da proporre caso per caso. Nel dettaglio, alla riga 2 il costrutto <<if>>, che abbiamo già visto, verifica se la variabile $from è vera o falsa. Questa variabile fa parte di una serie di variabili di servizio che vengono utilizzate per controllare i comportamenti di base del sistema, impareremo a conoscerle via via che si renderà necessario, ma una lista completa di tutte le variabili e di tutte le funzioni utilizzate è presente in appendice a questo manuale. Tornando alla nostra variabile, $from, molto semplicemente si occupa di dire al sistema se è il caso o no di visualizzare la parte di testo che descrive la "transizione" da un ambiente all'altro. Se $from è vera, il testo viene visualizzato, altrimenti no. Serve per fare in modo che se dovessimo per caso ricaricare il passaggio in cui ci troviamo a seguito - che ne so - di un'azione, o dopo un dialogo, la parte in cui si dice "Provengo dalla tal location" NON venga più visualizzata: è passato del tempo, magari abbiamo smanettato con qualche oggetto, parlato con qualcuno, non stiamo più "arrivando", eravamo già qui. Senza contare, ovviamente, tutti quei casi in cui non c'è spostamento, per esempio nel luogo in cui iniziamo l'avventura, o dopo che siamo stati svegliati nel cuore della notte da una telefonata. Non tenere conto di questo tipo di dettagli

renderebbe la scrittura molto più meccanica e fredda, se non ridondante. In ogni caso, è bene ribadire che possiamo tranquillamente non utilizzare questa funzionalità.

Torniamo a noi; la riga 2 dice sostanzialmente: "SE provengo direttamente da un'altra location, allora fai quello che segue". La riga 3 è un nuovo costrutto: <<switch>>. E' un altro classico di ogni linguaggio di programmazione: verifica il contenuto di una variabile (nel nostro caso $frompassage) e lo confronta con una serie di casi. SE uno di questi casi si verifica, esegue il codice relativo, altrimenti esegue il codice di default. La variabile $frompassage è un'altra variabile di servizio: essa contiene sempre il passaggio in cui ci trovavamo prima di quello in cui ci troviamo attualmente. Se da "passaggio uno" mi sposto in "passaggio due", qui $frompassage avrà come valore "passaggio uno". Se poi mi sposto da "passaggio due" a "passaggio tre", giunto qui la variabile $frompassage avrà come valore "passaggio due". E' ora molto semplice capire il resto: alla riga 4 si confronta la variabile $frompassage con qualcosa, e nel caso il raffronto dia esito positivo si esegue il codice sottostante, altrimenti si passa alla riga 6, dove un secondo "case" controlla una seconda possibilità, e così via. Le possibilità da controllare saranno naturalmente i passaggi di provenienza: prendiamo il passaggio "prima location" presente nel nostro template:

```
1. ''Prima location''


2. <<if $from is true>>\
3. <<switch $frompassage>>\
4. <<case "seconda location">>\
5. Provengo dalla seconda location. \
6. <<case "terza location">>\
7. Provengo dalla terza location. \
8. <</switch>>\
9. <<set $from to false>>\
10. <<else>>\
11. Non provengo da nessuna parte, ero già qui in precedenza. \
12. <</if>>\
```

Questa prima stanza è collegata ad altre due, per cui entrando nella stanza 1 i casi sono due: o veniamo dalla stanza 2, o veniamo dalla stanza 3. Alla riga 4 controlliamo la prima eventualità: scriviamo tra le virgolette ("") il titolo del passaggio di provenienza, esattamente così com'è (occhio alla sintassi e occhio a questo tipo di dettagli se in corso d'opera vi viene in mente di cambiare i titoli dei passaggi, tenete sempre a mente che sono un dato "sensibile" nella logica del gioco). Nella riga successiva, la 5, c'è il "codice" da eseguire: in questo caso è semplicemente il testo che vogliamo visualizzare in apertura di passaggio SE il giocatore proviene dalla seconda location. Stesso identico comportamento se il giocatore proviene dalla terza location: alla riga 6 scriviamo il successivo passaggio di provenienza, e in quella successiva il testo da visualizzare in questo caso. Possiamo aggiungere tutti i casi che ci pare, o averne anche uno solo (o nessuno), basta continuare ad inserire:

```
<<case "passaggio di provenienza">>
```

e il testo che vogliamo far apparire. La riga 8 chiude il costrutto <<switch>>; non abbiamo inserito l'opzione <<default>> perchè non capiterà mai che uno dei casi precedenti non si verifichi, in quanto la variabile $frompassage sarà sempre valorizzata. La riga 9 cambia lo stato della variabile $from, "spegnendo" il sistema di visualizzazione per le iterazioni successive: in pratica fa in modo che questo famoso messaggio di transizione venga visualizzato solo una volta all'ingresso del giocatore nella stanza. E se invece il giocatore era già presente nella stanza, e ha passato del tempo magari dialogando con un tizio? E' quello che succede alla riga 11: in sostanza, SE non provengo da alcun passaggio (ovvero mi trovavo già qui), allora fai quello che c'è a riga 11. La riga 12 chiude l'<<if>> aperto all'inizio. Ah, il carattere "\", presente alla fine di ogni riga, dice a Twine di non considerare l'interruzione di linea. Serve a fare in modo che tutta questa pappardella sia in verità "concentrata" in una sola riga in fase di interpretazione, e va inserito alla fine di ogni riga perchè altrimenti avremo un "a capo" alla fine del messaggio visualizzato a schermo.

Tutto chiaro? Ovviamente no. Ma per tirarvi su di morale, lasciatemi dire che tutta questa spiegazione era solo per chiarezza; nella pratica, basterà editare SOLO le zone evidenziate in uno scintillante GIALLO CANARINO; non c'è mai bisogno di toccare il resto:

```
''Prima location''


<<if $from is true>>\
<<switch $frompassage>>\
<<case "seconda location">>\
Provengo dalla seconda location. \
<<case "terza location">>\
Provengo dalla terza location. \
<</switch>>\
<<set $from to false>>\
<<else>>\
Non provengo da nessuna parte, ero già qui in precedenza. \
<</if>>\
```

Passiamo ora al corpo vero e proprio del passaggio, ovvero le descrizioni lunga e corta dell'ambiente (ho nuovamente numerato le righe per comodità):

1. `<<if $longDesc is true>>\`
2. `Questa è la descrizione lunga della prima location, che descrive l'ambiente in modo prolisso evidenziando tutti gli oggetti di scena, come <<obj "l'oggetto uno" "oggetto uno" "l'" "all'">> (che sembra molto interessante) e un po' più in là <<obj "l'oggetto due" "oggetto due" "l'" "all'">>, e magari le possibili uscite, come quelle che vanno in direzione della seconda e della terza location. <<invContiene "oggetto">><<if $invCheck is false>>Anche gli oggetti raccoglibili sono presenti nella descrizione lunga, come per esempio l'<<obj "oggetto raccoglibile" "oggetto raccoglibile" "l'" "all'">> qui presente.<</if>>`
3. `<<set $longDesc to false>><<else>>\`

4. `Questa è la descrizione breve della prima location. Qui vedo <<obj "l'oggetto uno" "oggetto uno" "l'" "all'">> e <<obj "l'oggetto due" "oggetto due" "l'" "all'">>.<<invContiene "oggetto">><<if $invCheck is false>> Per terra, vedo un <<obj "oggetto raccoglibile" "oggetto raccoglibile" "l'" "all'">>.<</if>>`
5. `<</if>>\`
6. `----`

La riga 1 è un costrutto <<if>> che controlla se bisogna visualizzare la descrizione lunga o corta in funzione di $longDesc. Come avrete già intuito, $longDesc è una nuova variabile di servizio: se è vera, il sistema restituirà la descrizione "lunga" dell'ambiente, altrimenti verrà visualizzata quella corta. Alla "riga" numero 2 abbiamo il contenuto della descrizione "lunga": per ora non badiamo al fatto che al suo interno ci sono nuove funzioni ("widgets") e costrutti - servono per generare gli oggetti contenuti nella stanza - e concentriamoci sul fatto che questa sezione deve contenere una descrizione dettagliata dell'ambiente, che naturalmente vorremo leggere per lo meno la prima volta che entriamo in un nuovo luogo, e che potremo richiamare successivamente "guardandoci attorno" tramite l'esecuzione del solo comando "Esamina" (in modo simile a quanto accadeva nelle avventure vecchio-stile, in cui si poteva rileggere la "descrizione lunga" di un ambiente digitando il comando "look" o simili).

La riga 3 "spegne" la visualizzazione della descrizione lunga per le successive iterazioni: rientrando nella stessa location, $longDesc sarà "false" e quindi la descrizione visualizzata sarà quella più stringata ed essenziale, contenuta nella sezione a riga 4.

La riga 5 chiude l'<<if>> di apertura, mentre la 6 è un nuovo markup di Twine: quattro trattini su una riga vuota vogliono dire "disegna una riga" (il tag <hr> in html). Molto semplice.

Anche qui, ecco la versione con evidenziata la parte che andrà editata:

```
<<if $longDesc is true>>\

Questa è la descrizione lunga della prima location, che descrive l'ambiente in modo prolisso evidenziando tutti gli oggetti di scena, come <<obj "l'oggetto uno" "oggetto uno" "l'" "all'">> (che sembra molto interessante) e un po' più in là <<obj "l'oggetto due" "oggetto due" "l'" "all'">>, e magari le possibili uscite, come quelle che vanno in direzione della seconda e della terza location. <<invContiene "oggetto">><<if $invCheck is false>>Anche gli oggetti raccoglibili sono presenti nella descrizione lunga, come per esempio l'<<obj "oggetto raccoglibile" "oggetto raccoglibile" "l'" "all'">> qui presente.<</if>>

<<set $longDesc to false>><<else>>\

Questa è la descrizione breve della prima location. Qui vedo <<obj "l'oggetto uno" "oggetto uno" "l'" "all'">> e <<obj "l'oggetto due" "oggetto due" "l'" "all'">>.<<invContiene "oggetto">><<if $invCheck is false>> Per terra, vedo un <<obj "oggetto raccoglibile" "oggetto raccoglibile" "l'" "all'">>.<</if>>

<</if>>\

----
```

Per concludere, non ci resta che occuparci dei link che ci permettono di muoverci tra le varie stanze:

```
<<iconforward>> //Da qui, posso <<if $frompassage is "seconda location">>tornare<<else>>dirigermi<</if>> verso la [[seconda location|seconda location][$from to true, $frompassage to passage()]], oppure <<if $frompassage is "terza location">>tornare<<else>>dirigermi<</if>> verso la
```

```
[[terza location|terza location][$from to true, $frompassage to passage()]].//
```

Il widget <<iconforward>> contiene semplicemente l'immaginetta della freccia che identifica il "muoversi" tra una stanza e l'altra. Tutto quello che segue è in carattere "corsivo", ragion per cui chiudiamo tutta la sezione dei link tra una doppia coppia di slash (//) e un'altra: anche questo fa parte del sistema di markup di Twine, in pratica qualsiasi cosa scriviamo tra una doppia coppia di barre (//..../) verrà interpretata dal sistema come "testo in corsivo". Appena dopo il "Da qui posso..." ecco un costrutto <<if>> che controlla da dove stiamo arrivando e compone il testo di conseguenza: nel caso in cui stiamo componendo il collegamento per la seconda stanza, SE provengo dalla location "seconda stanza", allora scrivi: "tornare", altrimenti scrivi: "andare" (o simili). Il link vero e proprio è la parte contenuta tra le parentesi quadre: abbiamo già visto come funziona nelle nozioni base, e qui non si fa niente di diverso; dopo aver detto al sistema cosa scrivere a video e il nome del passaggio verso cui saremo destinati, separati da un "|", seguono un paio di assegnazioni di variabili, nello specifico viene resettata la variabile $from di modo che nel prossimo passaggio venga visualizzata la parte in cui si dice "Provengo da...", e viene correttamente valorizzata la variabile $frompassage con il titolo del passaggio attuale. Ad ogni modo, anche qui, nella pratica è necessario editare SOLO il testo evidenziato:

```
<<iconforward>> //Da qui, posso <<if $frompassage is "seconda location">>tornare<<else>>dirigermi<</if>> verso la [[seconda location|seconda location][$from to true, $frompassage to passage()]], oppure <<if $frompassage is "terza location">>tornare<<else>>dirigermi<</if>> verso la [[terza location|terza location][$from to true, $frompassage to passage()]].//
```

## 2.d - Implementiamo la nostra mappa

Ora che abbiamo capito come si costruiscono gli ambienti, abbiamo tutto quel che serve a costruire il nostro piccolo mondo di gioco. Visivamente, Twine si presta particolarmente bene a costruire mappe fatte di ambienti collegati tra loro; basterà replicare esattamente la mappa che avevamo ideato durante la stesura del documento di progetto. Selezioniamo tutti i passaggi che costituiscono la mappa del template (le tre stanze con il tizio) e mettiamoli via, spostandoli da qualche parte dove non diano fastidio (il desktop di lavoro di Twine è infinito, vi basterà metterle in alto a destra, dopo i verbi). Lasciamo invece al loro posto i passaggi "home", "intro" e "info", perchè comunque ci serviranno. Non preoccupiamoci della lunga freccia che collega "intro" con "prima location", Twine la cambierà non appena modificheremo il passaggio "intro" con il link verso la prima effettiva location della nostra storia: "in automobile", che andremo subito a costruire. Clicchiamo su "+Passaggio": un nuovo passaggio sarà comparso al centro dello schermo. Facciamoci doppio click sopra e modifichiamo subito il titolo, scrivendo "in automobile". Chiudiamo il passaggio, apriamo il passaggio "intro" e modifichiamo il link in modo che punti a "in automobile":

```
[[Inizia il gioco!|in automobile]]
```

Ora apriamo il passaggio "template location" e copiamo tutto il suo contenuto nel passaggio "in automobile". Per ora ci interessa creare le ambientazioni e collegarle in modo corretto tra loro, per cui tralasciamo il contenuto dei passaggi e le descrizioni e lavoriamo solo sulla sezione link. Il link di base del template si presenta così:

```
<<iconforward>> //Da qui, posso <<if $frompassage is """>>tornare<<else>>dirigermi<</if>> [[|][$from to true, $frompassage to passage()]]//
```

E' troppa roba se consideriamo il fatto che siamo seduti su un'auto al posto del passeggero; in una situazione del genere, una singola opzione per aprire la portiera e scendere è più che sufficiente, per cui togliamo tranquillamente il costrutto <<if>> che controlla se andiamo o torniamo e scriviamo qualcosa di più adatto:

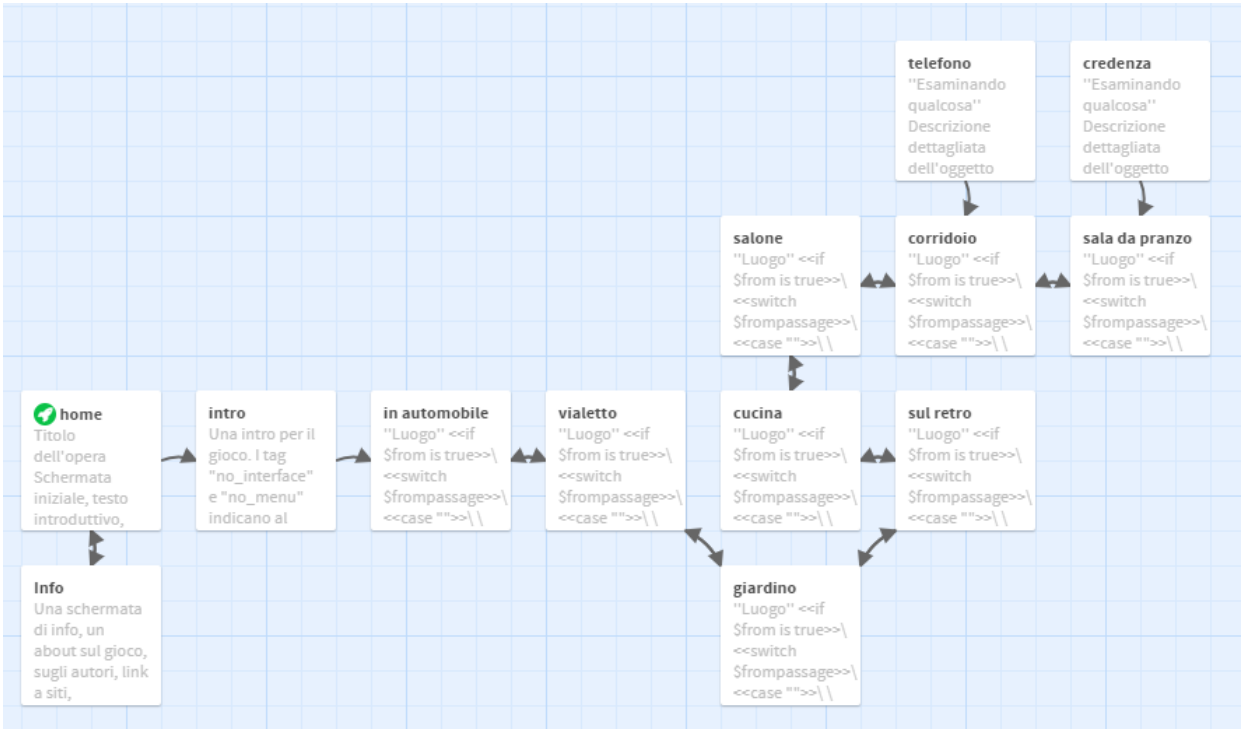
```
<<iconforward>> //[[Apri la portiera|vialetto][$from to true, $frompassage to passage()]] e scendi dall'auto.//
```

Twine creerà istantaneamente un nuovo passaggio, "vialetto", perchè lo abbiamo indicato come passaggio di destinazione e questo non era ancora presente nel gioco, risparmiandoci l'agonia di un +Passaggio per ogni nuovo ambiente :)

Andiamo subito ad editare il nuovo passaggio, incollandoci dentro il contenuto di "template location" e andando immediatamente alla sezione link. Questa volta, il sistema di andare/tornare può starci, per cui lo useremo, e forniremo un secondo link per raggiungere il giardino:

```
<<iconforward>> //Da qui, posso <<if $frompassage is "in automobile">>tornare <<else>>salire <</if>> [[in auto|in automobile][$from to true, $frompassage to passage()]], oppure posso farmi prossimo alla villa, <<if $frompassage is "giardino">>tornando<<else>>dirigendomi<</if>> [[verso il giardino|giardino][$from to true, $frompassage to passage()]].//
```

Procediamo allo stesso modo costruendo tutti i passaggi, prendendo a modello questa sezione di link per costruire tutte le altre: basterà copiarla in calce ad ogni passaggio e modificare di volta in volta le sezioni evidenziate per avere in breve tutti gli ambienti che ci servono:




La location "sala da pranzo" (come la prima, "in automobile") non ha altre uscite che quella da cui si proviene, per cui anche qui possiamo fare a meno del sistema "andare/tornare" e usare solo un link di ritorno. I passaggi "telefono" e "credenza" sono "zoom" sui relativi oggetti. Il funzionamento è simile a quello delle classiche avventure grafiche, in cui se si osserva un particolare oggetto, invece di avere in risposta un messaggio di testo si va a "guardare da

vicino" l'oggetto in questione, che può avere una sua schermata dedicata o apparire magari in un pop-up, ma che sempre si propone al giocatore come successivo "ambiente interattivo": la tastiera di un telefono, un meccanismo, etc..., quando c'è bisogno di interagire con singole parti di un oggetto particolare, lo inseriremo in un passaggio apposta, prendendo a modello "template esamina". Questo template è in realtà una versione molto semplificata del normale template per le locations: non c'è il sistema "andare/tornare" nè descrizioni lunghe o brevi (se stiamo specificatamente esaminando un oggetto, va da se che vogliamo la descrizione dettagliata), e fornisce sempre e solo un unico link di ritorno per - appunto - ritornare all'ambiente in cui ci trovavamo. Unica differenza, <<iconback>> al posto di <<iconforward>>: è la freccetta "a rientro", più adatta al concetto di "torna indietro".

Ora che abbiamo completato la struttura della nostra mappa, possiamo dedicarci a popolare le ambientazioni (per ora desolatamente vuote) con oggetti e personaggi.

## 2.e - Gli oggetti

Partiamo dal primo ambiente in cui ci troveremo una volta passato il passaggio dedicato all'introduzione: siamo in automobile con il nostro compare Bosko, al quale dovremo parlare per farci consegnare l'arnese da scasso (dotazione standard) senza il quale non ci sarà modo di intrufolarci in casa Berluti. Iniziamo a editare il passaggio "in automobile", che ora dovrebbe presentarsi così:

```
''Luogo''


<<if $from is true>>\
<<switch $frompassage>>\
<<case "">>\
\
<<case "">>\
\
<<case "">>\
\
<</switch>>\
<<set $from to false>>\
<<else>>\
\
<</if>>\
<<if $longDesc is true>>\


<<set $longDesc to false>><<else>>\
```

```
<</if>>\

----

<<iconforward>> //[[Apri la portiera|vialetto][$from to true, $frompassage to passage()]] e scendi dall'auto.//
```

Inseriamo il titolo del passaggio, le transizioni (quello da visualizzare se il giocatore è appena salito in auto) e le descrizioni, lunga e corta:

```
''In automobile''


<<if $from is true>>\

<<switch $frompassage>>\

<<case "vialetto">>\

Salgo in auto e richiudo la portiera. \

<</switch>>\

<<set $from to false>>\

<<else>>\

\

<</if>>\

<<if $longDesc is true>>\

Sono seduto al posto del passeggero, a bordo della vecchia Volkswagen di Bosko. Lui è al volante, come sempre, con la sua canonica sigaretta spenta che gli pende dal labbro. L'abitacolo è sporco, le cicche riempono il posacenere a cassetto così tanto da rendere impossibile richiuderlo, i tappetini sono logori e il cambio è privo di pomello: questa macchina all'apparenza fa schifo, ma sia io che Bosko sappiamo che è quello che c'è sotto il cofano e dietro al volante, a fare la differenza in caso di necessità.

<<set $longDesc to false>><<else>>\

Sono seduto al posto del passeggero, a bordo della Volkswagen di Bosko.

<</if>>\

----

<<iconforward>> //[[Apri la portiera|vialetto][$from to true, $frompassage to passage()]] e scendi dall'auto.//
```

Provando ad avviare il gioco, possiamo verificare che il funzionamento di questa prima location è quello che ci aspettiamo: uscendo dall'auto e rientrando viene visualizzato correttamente il messaggio "di transizione", e la descrizione proposta è quella breve. Se volessimo rileggere la descrizione lunga, sarà sufficiente il comando "Esamina".

Fin qui tutto ok, ma abbiamo detto che per ottenere il necessario arnese da scasso dobbiamo parlare con Bosko: per far questo, dobbiamo creare un "oggetto", ovvero un'entità che il sistema sia in grado di interpretare correttamente. In pratica, si tratta di un link ipertestuale che ci consentirà, unitamente ai verbi nella parte sottostante dell'interfaccia, di interagire con l'ambiente di gioco.

Nello specifico, vogliamo poter cliccare su “Bosko”, in modo da poterci parlare. In MMIFIF ogni oggetto combinabile con i verbi sottostanti viene definito tramite la funzione <<obj>>, che costruisce il link utilizzando quattro parametri:

```
<<obj “testo del link” “oggetto” “articolo” “preposizione”>>
```

- Il primo parametro è il testo che vogliamo far comparire a video: può essere o meno diverso dal testo che costituirà l’oggetto in se.
- Il secondo parametro è il testo che comparirà nell’interfaccia, subito dopo il “Cosa devo fare?”.
- Il terzo parametro è l’articolo da anteporre all’oggetto.
- Il quarto ed ultimo parametro è la particella da anteporre all’oggetto in frasi rette da verbi che vogliono la preposizione “a”, e verrà utilizzato con tutti i verbi che lo prevedono, per esempio “Lega” (‘lega la corda ALL’albero), “Dai” (‘dai l’oggetto A Tizio’), etc…

Per chiarire, facciamo un esempio. Vogliamo implementare nel nostro gioco un bel sasso, uno di quei sassi notevoli che si trovano solo una volta ogni tanto. Nella descrizione dell'ambiente, dovrà apparire come "un sasso notevole", in modo da inserirlo in una cosa tipo:

```
Sul sentiero, posso vedere - tra gli altri - un sasso notevole.
```

E vogliamo che nell'interfaccia utente venga scritto solo "il sasso", senza troppi fronzoli. L'oggetto da creare è:

```
<<obj "un sasso notevole" "sasso" "il " "al ">>
```

Con tutte queste informazioni, possiamo ora costruire, coi verbi e con questo sasso, frasi coerenti. Mettiamo di avere con noi un martello e di voler provare a scalfire il nostro sasso: cliccando su "colpisci" impostiamo il verbo; al secondo click, sul sasso, la frase verrà generata automaticamente e sarà: "colpisci il sasso". Se poi clicchiamo sul martello nel nostro inventario (che avremo definito nello stesso identico modo, ma vedremo in seguito come aggiungere e togliere oggetti dalle tasche del nostro protagonista), il sistema genererà correttamente "colpisci il sasso con il martello". Se invece avessimo con noi una corda e trovassimo un albero sulla nostra via, una volta definiti correttamente gli oggetti:

```
<<obj "un albero piegato dal vento" "albero" "l'" "all'">>
```

nella descrizione dell'ambiente, e:

```
<<obj "una robusta corda" "corda" "la " "alla ">>
```

come oggetto dell'inventario, potremmo legare la corda all'albero cliccando sul verbo "lega", poi sulla corda, e infine sull'albero, generando con tre click la frase "lega la corda all'albero". Vedremo successivamente come configurare i verbi, ma per ora basti sapere che sono di due categorie: quelli che vogliono la preposizione "con" e quelli che vogliono la preposizione "a". I verbi di default sono ovviamente già predisposti correttamente: quando si crea un nuovo oggetto basterà impostare come terzo parametro l'articolo  corretto per l'oggetto presente nel secondo parametro, e nel quarto la preposizione che risponde alla domanda: "a cosa?", sempre riferita al secondo parametro. Un'ultima cosa: occhio agli spazi. Devono essere inseriti (se ci vanno) nei parametri 3 e 4: se l'articolo è tipo "il" o "la", andrà dichiarato come:

```
"il "
```

o

```
"la "
```

con lo spazio prima delle virgolette di chiusura, mentre se c'è un apostrofo di mezzo, possiamo togliere lo spazio per ottenere una tipografia corretta.

Ora mettiamo in pratica quanto detto costruendo l'oggetto "Bosko", il nostro compare: trattandosi di una persona, l'articolo lo ometteremo (per evitare di trovarci di fronte a frasi tipo "Parla con IL Bosko"), dichiarandolo vuoto, e il testo a video sarà lo stesso di quello generato dal link:

```
<<obj "Bosko" "Bosko" "" "a ">>
```

E sostituiamo questo widget al testo originale, sia nella descrizione lunga che in quella breve:

```
''In automobile''


<<if $from is true>>\
<<switch $frompassage>>\
<<case "vialetto">>\
Salgo in auto e richiudo la portiera. \
<</switch>>\
<<set $from to false>>\
<<else>>\
\
<</if>>\
<<if $longDesc is true>>\
Sono seduto al posto del passeggero, a bordo della vecchia Volkswagen di <<obj "Bosko" "Bosko" "" "a ">>. Lui è al volante, come sempre, con la sua canonica sigaretta spenta che gli pende dal labbro. L'abitacolo è sporco, le cicche riempono il posacenere a cassetto così tanto da rendere impossibile richiuderlo, i tappetini sono logori e il cambio è privo di pomello: questa macchina all'apparenza fa schifo, ma sia io che il mio compare sappiamo che è quello che c'è sotto il cofano e dietro al volante, a fare la differenza in caso di necessità.
<<set $longDesc to false>><<else>>\
Sono seduto al posto del passeggero, a bordo della Volkswagen di <<obj "Bosko" "Bosko" "" "a ">>.
<</if>>\
----
<<iconforward>> //[[Apri la portiera|vialetto][$from to true, $frompassage to passage()]] e scendi dall'auto.//
```

Notiamo come cliccando direttamente su "Bosko", il testo generato sia "esamina Bosko": il verbo "Esamina" è automatico se clicchiamo su un oggetto qualunque nella descrizione dell'ambiente.

Procediamo in modo orizzontale, ovvero occupiamoci di portare a questo livello anche tutte le altre ambientazioni prima di passare ad implementare la logica e le azioni per ogni singolo oggetto: scriviamo tutte le descrizioni per ogni ambientazione della nostra mappa di gioco, con le relative transizioni, e gli oggetti necessari a completare l'avventura, così come li abbiamo elencati nel nostro prezioso documento di progetto.

L'automobile l'abbiamo già vista, passiamo al vialetto:

```
''Sul vialetto''


<<if $from is true>>\
<<switch $frompassage>>\
<<case "in automobile">>\
Sono sceso dall'auto e ho camminato in direzione della villa. \
<<case "giardino">>\
Ho attraversato il giardino, ritrovandomi di fronte alla villa. \
<<case "">>\
\
<</switch>>\
<<set $from to false>>\
<<else>>\
\
<</if>>\
<<if $longDesc is true>>\
Sono in mezzo al grazioso vialetto che conduce all'ingresso di casa Berluti.
<<set $longDesc to false>><<else>>\
Sono sul vialetto di ingresso.
<</if>>\
----
<<iconforward>> //Da qui, posso <<if $frompassage is "in automobile">>tornare
<<else>>salire <</if>> [[in auto|in automobile][$from to true, $frompassage
to passage()]], oppure posso farmi prossimo alla villa, <<if $frompassage is
"giardino">>tornando<<else>>dirigendomi<</if>> [[verso il giardino|giardino]
[$from to true, $frompassage to passage()]].//
```

Qui non ci sono oggetti strettamente necessari al compimento del gioco, in pratica si tratta di una "stanza di passaggio" in cui sostanzialmente non si può interagire con niente, ma sicuramente nelle successive revisioni vorremo inserire degli elementi di contorno, per esempio per approfondire la descrizione della villa, o ribadire che non è possibile scassinare il portone principale con il nostro modesto arnese da scasso. Per il momento, però, va bene così: ricordiamo che la prima stesura di un gioco (o di una sua parte) consiste nel dargli un capo e una coda. Tutto quello che non è assolutamente necessario a completare il percorso si può aggiungere in un secondo momento. Passiamo quindi al giardino:

```
''In giardino''
```

```
<<if $from is true>>\
<<switch $frompassage>>\
<<case "vialetto">>\
Ho attraversato il vialetto e mi sono spostato verso il prato. \
<<case "sul retro">>\
Dal retro, mi sono spostato verso il prato. \
<<case "">>\
\
<</switch>>\
<<set $from to false>>\
<<else>>\
\
<</if>>\
<<if $longDesc is true>>\
Mi trovo nel giardino della casa, sull'erba che cresce tra aiuole fiorite e ben curate.
<<set $longDesc to false>><<else>>\
Sono nel giardino della casa.
<</if>>\
----
<<iconforward>> //Da qui, posso <<if $frompassage is "sul retro">>tornare<<else>>dirigermi<</if>> [[verso il retro|sul retro][$from to true, $frompassage to passage()]] o <<if $frompassage is "vialetto">>tornare<<else>>dirigermi<</if>> [[verso il vialetto|vialetto][$from to true, $frompassage to passage()]].//
```

Anche qui, per ora, nessun oggetto. Ci basta che il giardino sia calpestabile e i messaggi di transizione e nei link siano disposti correttamente. Ci spostiamo sul retro:

```
''Sul retro''


<<if $from is true>>\
<<switch $frompassage>>\
<<case "giardino">>\
Mi sposto con circospezione oltrepassando il giardino. \
<<case "cucina">>\
Sono uscito dalla porta sul retro. \
<<case "">>\
\
<</switch>>\
```

```
<<set $from to false>>\

<<else>>\

\

<</if>>\

<<if $longDesc is true>>\

Sono sul retro dell'abitazione: una palizzata separa la proprietà da quella della villa vicina, mentre un breve porticato di legno precede <<obj "l'ingresso posteriore" "ingresso posteriore" "l'" "all'">>.

<<set $longDesc to false>><<else>>\

Sono sul retro dell'abitazione. Qui vedo la <<obj "porta d'ingresso posteriore" "ingresso posteriore" "l'" "all'">>.

<</if>>\

----

<<iconforward>> //Da qui, posso <<if $frompassage is "cucina">>tornare<<else>>dirigermi<</if>> [[verso l'ingresso posteriore|cucina][$from to true, $frompassage to passage()]] o <<if $frompassage is "giardino">>tornare<<else>>dirigermi<</if>> [[verso il giardino|giardino][$from to true, $frompassage to passage()]].//
```

Qui abbiamo inserito l'oggetto "porta", con cui dovremo interagire con l'arnese da scasso al fine di guadagnare l'accesso a casa Berluti. Notiamo subito che c'è qualcosa che non torna: per come stanno le cose, adesso è possibile entrare in casa senza problemi. Dovremo modificare in qualche modo la sezione dei link per fare in modo che se non abbiamo ancora forzato la serratura, il link per entrare NON sia disponibile: lo vedremo tra poco, per ora lavoriamo sempre in orizzontale e continuiamo con l'allestimento delle stanze. Passiamo alla cucina:

```
''In cucina''


<<if $from is true>>\

<<switch $frompassage>>\

<<case "sul retro">>\

Apro la porta e mi introduco in casa. \

<<case "salone">>\

Mi sposto dal salone in pochi passi. \

<<case "">>\

\

<</switch>>\

<<set $from to false>>\

<<else>>\

\

<</if>>\

<<if $longDesc is true>>\
```

```
Mi trovo nella cucina. Attorno a me ci sono una quantità di elettrodomestici, un ampio piano-cottura con sei fuochi e un forno gigantesco.

<<set $longDesc to false>><<else>>\

Sono in cucina.

<</if>>\

----

<<iconforward>> //Da qui, posso <<if $frompassage is "salone">>tornare<<else>>dirigermi<</if>> [[verso il salone d'ingresso|salone][$from to true, $frompassage to passage()]] o <<if $frompassage is "sul retro">>tornare<<else>>dirigermi<</if>> [[all'esterno|sul retro][$from to true, $frompassage to passage()]] della casa.//
```

Un'altra stanza di giunzione, senza oggetti rilevanti. Il salone:

```
''Nel salone d'ingresso''


<<if $from is true>>\

<<switch $frompassage>>\

<<case "cucina">>\

Esco dalla cucina e mi sposto verso l'atrio. \

<<case "corridoio">>\

Attraverso il corridoio spostandomi verso l'atrio. \

<<case "">>\

\

<</switch>>\

<<set $from to false>>\

<<else>>\

\

<</if>>\

<<if $longDesc is true>>\

Sono nel salone. Un ampio tappeto è steso sul pavimento, mentre un appendiabiti, da cui pende un solitario <<obj "cappotto" "cappotto" "il " "al ">>, è posizionato a lato della pesante porta d'ingresso.

<<set $longDesc to false>><<else>>\

Sono nel salone d'ingresso. Qui vedo un <<obj "cappotto" "cappotto" "il " "al ">> pendere dall'appendiabiti.

<</if>>\

----

<<iconforward>> //Da qui, posso <<if $frompassage is "corridoio">>tornare<<else>>dirigermi<</if>> [[verso il corridoio|corridoio][$from to true, $frompassage to passage()]], o <<if $frompassage is "cucina">>tornare<<else>>dirigermi<</if>> [[verso la cucina|cucina][$from to true, $frompassage to passage()]].//
```

Qui abbiamo inserito il cappotto appeso all'appendiabiti: frugando nella tasca, faremo cadere a terra una chiave, ma per ora non serve altro. Passiamo oltre.

```
''In corridoio''


<<if $from is true>>\
<<switch $frompassage>>\
<<case "salone">>\
Mi lascio il salone alle spalle. \
<<case "sala da pranzo">>\
Esco dalla sala da pranzo. \
<<case "">>\
\
<</switch>>\
<<set $from to false>>\
<<else>>\
\
<</if>>\
<<if $longDesc is true>>\
Sono in mezzo ad un lungo corridoio, verso il centro del quale, a ridosso della parete, è posizionato un mobiletto che regge quello che sembra un <<obj "vecchio telefono" "telefono" "il " "al ">>, di quelli con la rotella per selezionare i numeri.
<<set $longDesc to false>><<else>>\
Sono in mezzo ad un lungo corridoio. Su un mobiletto è posizionato un <<obj "vecchio telefono" "telefono" "il " "al ">>.
<</if>>\
----
<<iconforward>> //Da qui, posso <<if $frompassage is "sala da pranzo">>tornare<<else>>dirigermi<</if>> [[verso la sala da pranzo|sala da pranzo][$from to true, $frompassage to passage()]], o <<if $frompassage is "salone">>tornare<<else>>dirigermi<</if>> [[verso l'ingresso|salone][$from to true, $frompassage to passage()]].//
```

In corridoio troviamo il telefono: non c'è un link diretto per passare ad esaminarlo, dovremo farlo rispondendo al comando del giocatore: "esamina il telefono". Anche questo lo vedremo tra poco, andiamo avanti:

```
''In sala da pranzo''


<<if $from is true>>\
<<switch $frompassage>>\
<<case "corridoio">>\
Attraverso il corridoio in pochi passi ben distesi. \
```

```
<<case "">>\

\

<<case "">>\

\

<</switch>>\

<<set $from to false>>\

<<else>>\

\

<</if>>\

<<if $longDesc is true>>\

Sono in sala da pranzo: al centro della stanza è posto un ampio tavolo circolare, circondato da bellissime sedie imbottite e ornate di merletti. A ridosso della parete in fondo, è posta una <<obj "grande credenza" "credenza" "la " "alla ">> di legno massello.

<<set $longDesc to false>><<else>>\

Sono in sala da pranzo. Qui posso vedere una <<obj "grande credenza" "credenza" "la " "alla ">> di legno massello.

<</if>>\

----

<<iconforward>> //Da qui, posso solo tornare [[verso il corridoio|corridoio][$from to true, $frompassage to passage()]].//
```

Anche qui, passeremo allo "zoom" sulla credenza in risposta a "esamina la credenza". Concludiamo prima la stesura della nostra mappa proprio con questi ultimi due passaggi, gli oggetti "visti da vicino" con cui sarà possibile interagire. Il primo è proprio la credenza nella sala da pranzo: aggiungiamo il passaggio, copiamoci dentro il contenuto di "template esamina", e modifichiamo il testo aggiungendo quello che serve:

```
''Esaminando la credenza''


E' un mobile antico, con due ampi sportelli per i piatti e un <<obj "grosso cassettone" "cassetto" "il " "al ">> per le posate. Il legno, nonostante l'età, sembra ben mantenuto e molto resistente.

<<set $longDesc to false>>\

----

<<iconback>> //[[Torna alla sala|sala da pranzo]].//
```

Qui abbiamo l'oggetto "cassetto", che dovremo aprire. L'ultimo passaggio da allestire è il vecchio telefono:

```
''Esaminando il vecchio telefono''


E' un apparecchio vecchio stile, di colore nero, di quelli con il selettore a rotella da girare col dito. Sembra più un oggetto d'arredo che altro, posizionato com'è sotto l'elegante specchio posto alla parete. Posso comunque provare a comporre un numero di telefono:
```

```
/ <<cmb "1">> / <<cmb "2">> / <<cmb "3">> / <<cmb "4">> / <<cmb "5">> / <<cmb
"6">> / <<cmb "7">> / <<cmb "8">> / <<cmb "9">> / <<cmb "0">> /

<<set $longDesc to false>>\

----

<<iconback>> //[[Torna al corridoio|corridoio]].//
```

Qui troviamo un nuovo "widget", <<cmb>>. Serve per definire oggetti "liberi", senza cioè tutta la parte di preposizioni e verbi automatizzati che servono a costruire una frase di senso compiuto: <<cmb>> "sputa fuori" il testo che gli passiamo come parametro esattamente così com'è. Serve sostanzialmente per tutte quelle situazioni in cui, per esempio, si deve poter comporre un codice cliccando su dei numeri, o per giochi di parole, password, o un sistema di parola d'ordine tipo quello di Monkey Island 2. La sintassi, per chiarezza, è:

```
<<cmb "testo">>
```

Abbiamo finito di implementare la mappa e tutti i passaggi che ci servono sono collegati tra loro, popolati con le corrette descrizioni e pieni di oggetti pronti all'uso: iniziamo ad occuparci delle azioni e dei verbi.

## 2.f - I verbi

Torniamo alla nostra automobile: abbiamo già allestito la scena con tutto quello che serve, infatti cliccando su Bosko e sui vari verbi possiamo già comporre una moltitudine di frasi. Genericamente il sistema compone automaticamente le frasi da eseguire senza richiedere all'utente di inserire nient'altro che un verbo e un oggetto: "prendi la mela", "colpisci l'albero" etc…, mentre è sempre possibile (a discrezione del giocatore) aggiungere un ulteriore complemento cliccando su un secondo oggetto: "apri la porta con la chiave", "lega la briglia al cavallino", "dai il portafoglio a Ciro", "usa la benzina con il fuoco". Come abbiamo visto, il verbo "Esamina" si comporta in modo semi-automatico: se clicchiamo su un oggetto all'inizio di una nuova frase, esso comparirà di default a completare il comando, mentre se lo utilizziamo da solo otteniamo la descrizione lunga dell'ambiente in cui ci troviamo. Un altro verbo che fa una piccola eccezione rispetto al comportamento di tutti gli altri, è "Parla", che richiede una costruzione leggermente diversa: impostando la frase su usto verbo, l'oggetto seguente verrà collegato a "parla" direttamente tramite il "con", e non sarà più possibile impostare un secondo complemento (del resto, "parla con Bosko" è già sufficiente, non servirebbe in nessun caso avere un secondo oggetto nella frase, e infatti il sistema non permette neppure di inserirlo).

"Esamina" e "Parla" sono gli unici due verbi che è meglio non modificare (proprio perché si dovrebbe intervenire su questi automatismi, che sono "hard-coded" nei widget di sistema). Potrebbe darsi il caso in cui stiamo scrivendo un'avventura che si svolge completamente in solitaria, o dove comunque non incontreremo nessuno con cui dialogare (un'isola deserta, una escape-room, etc…): in una situazione del genere, possiamo tranquillamente rimuovere il verbo "parla" e sostituirlo con qualcos'altro di più utile al nostro gioco. Per quanto riguarda "esamina", non mi viene in mente nessun caso in cui potrebbe essere utile eliminarlo.

Tutti gli altri verbi sono tranquillamente modificabili, sia nel testo che nel “tipo” di verbo (se vuole la preposizione “con” o quella “a”); sarà sufficiente aprire il passaggio StoryInit e cercare la sezione apposita:

```
/*----verbi----*/


<<set $verb_1_label to "Esamina">>
<<set $verb_1_cont to "esamina">>
<<set $verb_1_part to "con ">>

<<set $verb_2_label to "Usa">>
<<set $verb_2_cont to "usa">>
<<set $verb_2_part to "con ">>

<<set $verb_3_label to "Prendi">>
<<set $verb_3_cont to "prendi">>
<<set $verb_3_part to "con ">>

<<set $verb_4_label to "Colpisci">>
<<set $verb_4_cont to "colpisci">>
<<set $verb_4_part to "con ">>

<<set $verb_5_label to "Parla">>
<<set $verb_5_cont to "parla">>
<<set $verb_5_part to "con ">>

<<set $verb_6_label to "Apri">>
<<set $verb_6_cont to "apri">>
<<set $verb_6_part to "con ">>

<<set $verb_7_label to "Chiudi">>
<<set $verb_7_cont to "chiudi">>
<<set $verb_7_part to "con ">>

<<set $verb_8_label to "Tira">>
<<set $verb_8_cont to "tira">>
<<set $verb_8_part to "con ">>
```

```
<<set $verb_9_label to "Spingi">>
<<set $verb_9_cont to "spingi">>
<<set $verb_9_part to "con ">>

<<set $verb_10_label to "Dai">>
<<set $verb_10_cont to "dai">>
<<set $verb_10_part to "a ">>
```

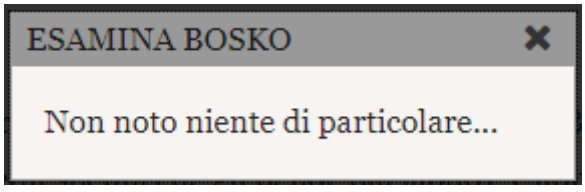
E’ piuttosto semplice: i verbi sono numerati da 1 a 10 e ogni verbo è definito da tre variabili, la variabile $verb_nn_label, la variabile $verb_nn_cont, e la variabile $verb_nn_part (dove “nn” è il numero del verbo). La prima, $verb_nn_label, contiene il testo da scrivere nella tabella dei verbi; la seconda, $verb_nn_cont, è quello che il sistema scriverà nella linea di comando, mentre la terza variabile, $verb_nn_part, è la preposizione voluta dal verbo. In sostanza, se volessimo per esempio sostituire il verbo “Chiudi” con il verbo “Lega”, sarebbe sufficiente editare il settimo verbo della lista come segue:

```
<<set $verb_7_label to "Lega">>
<<set $verb_7_cont to "lega">>
<<set $verb_7_part to "a ">>
```

Facciamo sempre attenzione allo spazio prima delle virgolette di chiusura nella sezione “part”. E’ possibile cambiare a piacere anche l’ordine dei verbi; ricordiamoci sempre, in caso di customizzazione dei verbi, di cambiare anche il nome del passaggio che contiene le interazioni specifiche per verbo (il sistema funzionerebbe correttamente anche se non lo facessimo, ma ovviamente la leggibilità del progetto ne risentirebbe). E' inoltre possibile cambiare ogni verbo durante lo svolgersi del gioco, semplicemente ridichiarando la tripletta di variabili quando necessario (in questo caso, meglio scrivere come titolo del passaggio che contiene le azioni relative, qualcosa tipo "chiudi/lega", ovvero i due verbi che occupano a turno lo "slot" numero 7 nella nostra lista dei verbi) .

## 2.g - Le azioni

Compreso il funzionamento di oggetti e verbi, possiamo finalmente iniziare a costruire le azioni, che costituiranno l’effettiva interazione del giocatore con l’ambiente di gioco. Per cominciare, proviamo ad esaminare Bosko, tanto per dargli un’occhiata: il sistema accetta il comando “esamina Bosko” e fa saltar fuori un pop-up con la risposta:




Come possiamo notare, l’azione appena eseguita viene scritta in maiuscolo come titolo del pop-up, mentre il corpo del messaggio contiene l’effettiva risposta del sistema. La risposta, “Non noto niente di particolare…”, è la risposta di default quando si imposta una frase con il verbo “esamina” e non viene trovata un’azione corrispondente. Potrebbe anche andar bene

(forse non c'è davvero molto di particolare in Bosko), ma vogliamo quantomeno fornire qualche informazione in più. Apriamo il passaggio "esamina":

```
<<widget check_01>>
<<switch $action>>

<<case "esamina ">>
<<set $longDesc to true>>
<<refresh>>

<<case "esamina l'oggetto uno ">>
<<goto "oggetto uno">>

<<default>>

<<notify "Non noto niente di particolare...">>

<</switch>>
<</widget>>
```

Il costrutto utilizzato per valutare un comando inserito dal giocatore è <<switch>>, che abbiamo già visto, e che qui valuta il contenuto della variabile $action. Questa è una variabile di servizio; essa contiene sempre il comando composto dal giocatore tramite l'interfaccia. I "casi" sono dunque rappresentati dalle frasi che verranno composte, e qui possiamo vederne già due, il primo: <<case "esamina ">> è quello che fa in modo che ci venga riproposta la descrizione lunga di una location cliccando sul solo verbo "Esamina". Nello specifico, <<set $longDesc to true>> imposta la variabile di servizio $longDesc a "vera" e successivamente <<refresh>> ricarica la pagina: trovando $longDesc impostata, il sistema di passaggi (come abbiamo già visto) ripropone la descrizione lunga.

Il secondo "case" che incontriamo è un "residuo" del template iniziale che abbiamo spostato all'inizio della lavorazione. Possiamo cancellarlo - anzi, possiamo ormai cancellare anche i vecchi passaggi preimpostati del template, "prima location", "seconda location" etc..., che ormai non servono più.

Inseriamo ora un nuovo "case", che risponderà al comando "esamina Bosko":

```
<<case "esamina Bosko ">>
```

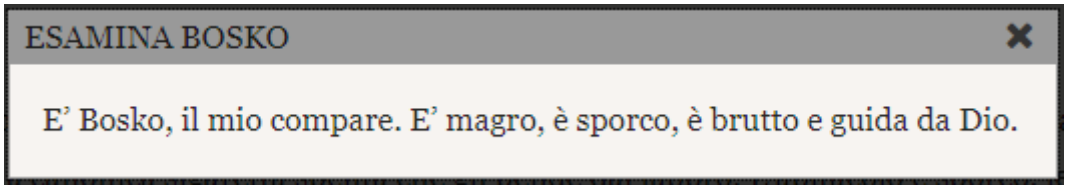
Facciamo attenzione allo spazio finale prima delle virgolette di chiusura. Ora, tutto quello che seguirà fino al prossimo "case", verrà eseguito solo se il giocatore scriverà "esamina Bosko" nell'interfaccia. Per impostare una risposta, utilizzeremo il widget <<notify>>, che fa comparire un pop-up. La sintassi è semplice:

```
<<notify "messaggio">>
```

L'unico parametro da impostare è il corpo del messaggio, pertanto l'azione completa "esamina Bosko" diventa:

```
<<case "esamina Bosko ">>
<<notify "E' Bosko, il mio compare. E' magro, è sporco, è brutto e guida da Dio.">>
```

Non serve altro. Proviamo ora a chiudere il passaggio "esamina" e a lanciare il gioco. Una volta in auto, esaminiamo Bosko:




Tutto secondo i piani. Passiamo ora a qualcosa di più complicato: vogliamo parlare con il nostro compare, quindi non basterà una semplice notifica pop-up. Useremo la macro <<goto>>, già predisposta in SugarCube, che ha la funzione di "teletrasportare" il giocatore verso un determinato passaggio, che viene indicato come parametro. La sintassi è:

```
<<goto "passaggio di destinazione">>
```

Prima di far questo, però, abbiamo bisogno di creare per lo meno il primo passaggio di quelli dedicati al dialogo con Bosko. Creiamo un nuovo passaggio con +Passaggio, lo intitoliamo "dialogo con Bosko" e copiamo al suo interno il contenuto del passaggio "Template dialogo":

```
''Parlando con il tizio''

Mi avvicino al tizio, lui mi saluta:
«Saluti, straniero! Qui c'è un bel dialogo di mezzo! Che cosa mi dici?»
<<set $longDesc to false>>\
----
//Mi faccio coraggio e gli rispondo:
[[||]]
[[||]]
[[||]]//
```

E' molto semplice, metteremo il corpo del testo nella parte superiore, e in fondo, dei semplici link (ci sono già tre "slot" vuoti preimpostati), manderanno ai passaggi relativi alle risposte, in una sorta di diagramma ad albero tipico del sistema di dialoghi "a scelta multipla" delle classiche avventure grafiche in stile Lucas. Unica accortezza che dobbiamo avere: durante un dialogo, proprio come accadeva in quei vecchi giochi, l'interfaccia utente coi verbi non sarà visibile. Per far questo, è sufficiente utilizzare un tag. Un "tag" è – per l'appunto – un'etichetta che si può mettere ad un particolare passaggio: nel nostro caso, aggiungendo il tag "no_interface" al dialogo con Bosko, il sistema saprà che qui non andrà visualizzata l'interfaccia. A passaggio aperto, clicchiamo quindi su +Etichetta e scriviamo "no_interface" nel campo di testo. Confermiamo e vedremo apparire il tag in alto, appena sotto il titolo del passaggio. In modo simile, il tag "no_menu" farà in modo che il sistema non visualizzi il menù a

tendina che si può richiamare premendo il piccolo triangolo in alto. Modifichiamo ora il passaggio secondo le nostre esigenze (tutte le modifiche sono evidenziate in giallo):

```
''Parlando con Bosko''


Bosko si volta ad osservarmi, guardandomi di sbieco, senza staccare le mani dal volante.

«Ehi amico, che c'è?»

<<set $longDesc to false>>\

----

//[[«Bosko, mi piacerebbe sapere cosa ne pensi di questo colpo…»|cosa ne pensi]]

[[«Hai qualche consiglio, prima di cominciare?…»|qualche consiglio]]

[[«Bosko, mi serve l'arnese da scasso per entrare in casa.»|dammi l'arnese]]//
```

Come al solito, Twine creerà automaticamente tutti i passaggi dichiarati nei link. Scriviamo qualcosa per ogni opzione, e copiamo&incolliamo la sezione dei link modificandoli di volta in volta all'occorrenza; ecco il passaggio "cosa ne pensi":

```
''Parlando con Bosko''


Bosko fa spallucce:

«E' un buon colpo, lo sai. Devi solo trovare la bottiglia. Ho già il compratore che ci aspetta coi soldi in mano.»

<<set $longDesc to false>>\

----

//[[«Hai qualche consiglio, prima di cominciare?…»|qualche consiglio]]

[[«Bosko, mi serve l'arnese da scasso per entrare in casa.»|dammi l'arnese]]

[[«Beh, ora credo sia meglio mettersi all'opera. Ti saluto.»|fine dialogo con Bosko]]//
```

Ed ecco "qualche consiglio":

```
''Parlando con Bosko''


I suoi occhi si fanno fessure, mentre ripassa mentalmente i punti salienti del colpo:

«La porta di ingresso principale è allarmata, oltre ad essere quasi inespugnabile. Dovrai trovare un'altra via di accesso. E una volta dentro, cerca un qualche tipo di dispositivo, una cassaforte segreta, qualcosa del genere... il padrone di casa è un fanatico di queste cose.»

<<set $longDesc to false>>\

----

//[[«Bosko, mi piacerebbe sapere cosa ne pensi di questo colpo…»|cosa ne pensi]]
```

```
[[«Bosko, mi serve l’arnese da scasso per entrare in casa.»|dammi l’arnese]]

[[«Beh, ora credo sia meglio mettersi all'opera. Ti saluto.»|fine dialogo con Bosko]]//
```

Il passaggio “fine dialogo con Bosko” fa da “transizione” tra il dialogo e l’ambiente in cui ci trovavamo quando abbiamo iniziato a parlare con Bosko:

```
''Parlando con Bosko''


Bosko annuisce:

«Ok, adesso muoviti e cerca di non dare troppo nell'occhio.»

<<set $longDesc to false>>\

----

<<iconback>> //[[Torna all'automobile|in automobile]].//
```

Da ultimo, il passaggio in cui chiediamo al nostro compare l’arnese da scasso:

```
''Parlando con Bosko''


Bosko annuisce, come se si fosse ricordato solo ora:

«Ah, giusto. Ecco qui, il miglior arnese da scasso che abbiamo trovato. A questo prezzo, s'intende.»

Si fruga in tasca per un secondo e poi mi porge il pezzo; l'arnese da scasso passa di mano. <<prendi "l’ arnese da scasso" "arnese" "l'" "all'">>

<<set $longDesc to false>>\

----

//[[«Bosko, mi piacerebbe sapere cosa ne pensi di questo colpo…»|cosa ne pensi]]

[[«Hai qualche consiglio, prima di cominciare?…»|qualche consiglio]]

[[«Beh, ora credo sia meglio mettersi all'opera. Ti saluto.»|fine dialogo con Bosko]]//
```

Oltre al normale testo, evidenziato, abbiamo un nuovo widget: <<prendi>>. La sintassi è identica a quella di <<obj>>, ma, invece di inserire l’oggetto nella descrizione, <<prendi>> lo inserisce nell’inventario del giocatore, senza scrivere niente a video (avremmo potuto posizionarlo ovunque nella descrizione senza alterare il risultato finale). Se proviamo ad avviare il gioco e a parlare con Bosko, dopo averci fatto quattro chiacchiere potremo chiedergli l’arnese, e a dialogo terminato troveremo l’arnese da scasso nel nostro inventario. Tutto ok? Non proprio: se parliamo di nuovo con Bosko (o se non lasciamo il dialogo subito dopo esserci fatti consegnare l’arnese), noteremo che l’opzione di dialogo che porta a “dammi l’arnese” è sempre valida, il che non è proprio logico (l’arnese ce l’abbiamo già) e se la selezioniamo di nuovo il sistema si rifiuterà (correttamente) di inserire un “doppione” dell’oggetto in inventario, ma il passaggio sarà lo stesso di prima e la logica della narrazione andrà a farsi benedire. Occorre cambiare qualcosa, in modo che, nel caso abbiamo già preso l'arnese, l’opzione per chiederlo a Bosko non sia più disponibile.

Per ottenere questo risultato, useremo un nuovo widget di MMIFIF: <<invContiene>>. Questa funzione va sempre utilizzata in coppia con un costrutto <<if>> e funziona così: si controlla la

presenza nell'inventario di un determinato oggetto, che passeremo come parametro al widget, e nel caso esso sia presente, la variabile di servizio $invCheck verrà settata a "vera", mentre nel caso contrario a "falsa". L'<<if>> successivo, come avremo già intuito, si occuperà di verificare $invCheck e agirà di conseguenza. La sintassi è semplice:

```
<<invContiene "oggetto">><<if $invCheck>> Cosa fare se l'oggetto è presente
<<else>> Cosa fare se l'oggetto non è presente <</if>>
```

Utilizziamolo per modificare il link che porta il giocatore a chiedere l'arnese a Bosko nei passaggi in cui compare ("cosa ne pensi", "qualche consiglio" e "dialogo con Bosko"):

```
<<invContiene "l'arnese da scasso">><<if not $invCheck>>[[«Bosko, mi serve
l'arnese da scasso per entrare in casa.»|dammi l'arnese]]

<</if>>\
```

In pratica, abbiamo "rinchiuso" il link precedente in un costrutto <<if>> che viene eseguito SOLO se $invCheck è falsa. Una piccola nota: l'<</if>> che chiude il costrutto su una riga a parte, seguito dal terminatore "\", fa in modo che se il link non è visibile, NON compaia una riga vuota tra le altre due risposte. La sintassi utilizzata nel costrutto (if not $invCheck) è un modo più sintetico di dire "if $invCheck is false", in pratica:

```
<<if $invCheck>>
```

è uguale a:

```
<<if $invCheck is true>>
```

mentre:

```
<<if not $invCheck>>
```

è lo stesso che scrivere:

```
<<if $invCheck is false>>
```

A questo punto, sostituiti correttamente i link in tutti e tre i passaggi, il dialogo e la sua logica saranno corretti: sarà possibile chiedere a Bosko l'arnese da scasso SOLO se non lo avremo già in nostro possesso. Non abbiamo utilizzato <<else>>, in questo caso un'opzione "alternativa" non serve. Possiamo continuare: preso l'arnese e sbrigato il dialogo con Bosko, attraversiamo il vialetto, passiamo per il giardino e ci troviamo nel retro della villa. La porta, al momento, non è un grande ostacolo: possiamo tranquillamente entrare in casa senza dover fare alcunché. Vogliamo invece che il passaggio non sia consentito al giocatore fintanto che questi non scassinerà la porta con il suo arnese.

Apriamo il passaggio "sul retro" e diamo un'occhiata alla sezione dei link, là dove dovremo agire per impedire il facile ingresso:

```
<<iconforward>> //Da qui, posso <<if $frompassage is
"cucina">>tornare<<else>>dirigermi<</if>> [[verso l'ingresso posteriore|
cucina][$from to true, $frompassage to passage()]] o <<if $frompassage is
"giardino">>tornare<<else>>dirigermi<</if>> [[verso il giardino|giardino]
[$from to true, $frompassage to passage()]].//
```

Ho evidenziato la parte che dovremo "rinchiudere" nel costrutto <<if>>, ma quale sarà la condizione da verificare per vedere la porta è già stata forzata o no? Questa volta, andare a vedere se un oggetto è o meno nelle tasche del giocatore non serve a niente, e dovremo utilizzare quella che chiameremo una "variabile di controllo". Le variabili di controllo rappresentano una sorta di "memorandum" che ci permetterà di tenere traccia di tutti i

progressi fatti dal giocatore, degli stati degli oggetti, e di quant'altro dovesse definire la logica della nostra storia.

Possiamo avere quante variabili di controllo vogliamo, basta definirle all'inizio del gioco in StoryInit nella sezione apposita, e possono essere del tipo che più ci serve (booleane, numeriche, stringhe di testo etc...). Io di norma utilizzo una serie di variabili boolenae (quelle che possono avere solo due stati: vero o falso) con dei nomi adatti, in modo da usarle come "interruttori" on/off per le varie sezioni logiche. Attenzione: nulla ci vieta di usare variabili di controllo ovunque, ma la pratica migliore è quella di limitare la loro presenza allo stretto necessario. Per questo prima abbiamo utilizzato <<invContiene>> per decidere se visualizzare o meno il link a "dammi l'arnese", invece di instanziare una variabile tipo $arnesePosseduto come falsa e metterà a "true" una volta ottenuto l'oggetto: avendo la possibilità di verificare direttamente se abbiamo o meno l'oggetto, sarebbe stato inutile avere in ballo una nuova variabile. Come regola generale, è sempre preferibile dichiarare meno variabili di controllo possibile (nei limiti della decenza), per questioni di chiarezza, leggibilità ed editabilità del codice.

Detto questo, apriamo StoryInit e aggiungiamo una variabile che ci permetterà di tenere traccia del fatto che la porta sia stata forzata o meno:

```
<<set $portaForzata to false>>
```

La settiamo a "false" perchè, naturalmente, all'inizio della nostra storia la porta è ben serrata.

Adesso abbiamo tutti gli elementi che ci servono per modificare correttamente il link nel passaggio "sul retro":

```
<<iconforward>> //Da qui, posso <<if $portaForzata>><<if $frompassage is
"cucina">>tornare<<else>>dirigermi<</if>> [[verso l'ingresso posteriore|
cucina][$from to true, $frompassage to passage()]] o<</if>> <<if $frompassage
is "giardino">>tornare<<else>>dirigermi<</if>> [[verso il giardino|giardino]
[$from to true, $frompassage to passage()]].//
```

Tutta la parte compresa tra <<if $portaForzata>> e <</if>> comparirà SOLO se la variabile di controllo $portaForzata sarà VERA; fino a quel momento, non verrà visualizzato niente e il link che ne risulterà sarà solo:

*Da qui, posso tornare verso il giardino.*

Ora bisogna vedere COME cambiare lo stato di questa variabile, in modo da poter poi varcare la soglia. Dovremo rispondere all'azione "apri l'ingresso posteriore con l'arnese", per cui apriamo il passaggio relativo alle azioni del verbo "apri" e inseriamo un nuovo "case":

```
<<case "apri l'ingresso posteriore con l'arnese ">>
```

e di seguito le istruzioni relative:

```
<<notify "Con un po' di pazienza, riesco nell'impresa di forzare la porta con
l'arnese da scasso, che alla fine dell'operazione risulta ormai
inutilizzabile.">>

<<set $portaForzata to true>>

<<lascia "l'arnese da scasso">>

<<refresh>>
```

Il <<notify>> fa comparire un messaggio in cui informiamo il giocatore dell'esito dell'azione. Immediatamente dopo, settiamo a "vera" la variabile $portaForzata. Alla riga successiva, un nuovo widget: <<lascia>>. Serve per cancellare un oggetto dall'inventario, la sintassi è semplicissima:

```
<<lascia "oggetto">>
```

L'oggetto da cancellare è definito nel parametro, e corrisponde a quello che leggiamo nell'inventario (che sarebbe il primo parametro passato a <<prendi>> nella creazione dell'oggetto).

L'ultima riga, <<refresh>>, serve a "ricaricare" la pagina. Ricordiamoci sempre che stiamo lavorando in ambito statico: tutti i cambiamenti e le impostazioni di variabili etc... hanno effettivamente luogo al caricamento del passaggio successivo, per cui per visualizzare correttamente la nuova sezione di link che sarà affetta da <<set $portaForzata to true>> e la "sparizione" dell'oggetto "l'arnese da scasso" dall'inventario, è necessario "ricaricare" il passaggio in cui ci troviamo: <<refresh>> fa appunto questo e non necessita di alcun parametro; andrà usato ogni qualvolta una nostra azione comporta un qualche cambiamento in descrizioni o inventario.

Da ultimo, modifichiamo leggermente i vari "dirigermi"/"tornare" presenti nei link del passaggio "sul retro", in modo che rispondano in maniera più coerente agli accadimenti:

```
<<iconforward>> //Da qui, posso <<if $portaForzata>><<if $frompassage is
"cucina">>tornare dentro casa attraversando<<else>>introdurmi in casa
passando per<</if>> [[l'ingresso posteriore|cucina][$from to true,
$frompassage to passage()]] o<</if>> <<if $frompassage is
"giardino">>tornare<<else>>dirigermi<</if>> [[verso il giardino|giardino]
[$from to true, $frompassage to passage()]].//
```

L'azione successiva, secondo il nostro documento di progetto, sarà esaminare il cappotto appeso nel salone, dalla tasca del quale cadrà una piccola chiave che dovremo raccogliere.

Aggiungiamo l'azione "esamina il cappotto" nel passaggio relativo al verbo "esamina":

```
<<case "esamina il cappotto ">>
```

e riflettiamo un attimo su cosa dovrà succedere. Vogliamo che:

1. frugando nel cappotto causiamo la caduta a terra di una piccola chiave;
2. una volta successo questo, non sia più possibile che quanto sopra accada nuovamente ripetendo l'azione;
3. la scena alla fine dell'azione sia cambiata, mostrando la piccola chiave a terra nella descrizione dell'ambiente.

Per fare questo, prima di tutto bisognerà verificare che sia o meno la prima volta che il giocatore fruga le tasche del cappotto: ci servirà una variabile di controllo che chiameremo $cappottoFrugato e dichiareremo come falsa in StoryInit; inoltre, ci servirà una variabile anche per controllare se la chiave è caduta (potremmo usarne una sola, ma per chiarezza è meglio averne due, in modo da contravvenire subito alle regole esposte poco fa). Il messaggio da mostrare al giocatore dovrà cambiare in funzione del fatto che abbia o meno, in precedenza, già frugato le tasche del cappotto:

1. `<<if $cappottoFrugato is false>>`

2. <<notify "Frugo alla svelta le tasche del vecchio cappotto: quando mi rendo conto di star tastando uno schifoso Kleenex usato, ritraggo la mano e qualcosa cade sul tappeto: è una piccola chiave dorata.">>
3. <<set $chiaveCaduta to true>>
4. <<set $cappottoFrugato to true>>
5. <<refresh>>
6. <<else>>
7. <<notify "A parte fazzoletti usati, non c'è nient'altro nelle tasche di quel cappotto.">>
8. <</if>>

Alla riga 1 viene controllato se il giocatore ha o meno già frugato il cappotto. Se NON lo ha fatto, alla riga 2 viene visualizzato il messaggio; alla riga 3 viene cambiato in "vero" lo stato della variabile di controllo $chiaveCaduta (che all'inizio era falso, essendo stato dichiarato tale in StoryInit), stessa cosa alla riga 4 (abbiamo frugato il cappotto proprio ora); il <<refresh>> serve perchè sarà comparsa nell'ambiente di gioco la piccola chiave (la inseriremo tra un attimo). Se, invece, avevamo già frugato il cappotto, il sistema risponderà semplicemente con quanto a riga 7. E' molto semplice e lineare.

Vediamo adesso come modificare l'ambiente e far comparire la piccola chiave: come già si sarà intuito, useremo un semplice costrutto <<if>> nelle descrizioni lunga e breve della stanza:

```
''Nel salone d'ingresso''


<<if $from is true>>\
<<switch $frompassage>>\
<<case "cucina">>\
Esco dalla cucina e mi sposto verso l'atrio. \
<<case "corridoio">>\
Attraverso il corridoio spostandomi verso l'atrio. \
<<case "">>\
\
<</switch>>\
<<set $from to false>>\
<<else>>\
\
<</if>>\
<<if $longDesc is true>>\
Sono nel salone. Un ampio tappeto è steso sul pavimento, mentre un appendiabiti, da cui pende un solitario <<obj "cappotto" "cappotto" "il " "al ">>, è posizionato a lato della pesante porta d'ingresso.<<invContiene "una
```

```
piccola chiave">><<if $invCheck is false and $chiaveCaduta is true>> Sul tappeto, vedo la piccola chiave caduta dal cappotto.<</if>>

<<set $longDesc to false>><<else>>\

Sono nel salone d'ingresso. Qui vedo un <<obj "cappotto" "cappotto" "il " "al ">> pendere dall'appendiabiti.<<invContiene "una piccola chiave">><<if $invCheck is false and $chiaveCaduta is true>> Sul tappeto, vedo la piccola chiave caduta dal cappotto.<</if>>

<</if>>\

----
```

Soffermiamoci un attimo sulla parte evidenziata (è uguale in entrambe le descrizioni). Viene chiamato <<invContiene>> a controllare se la chiave è già presente nell'inventario, e nel costrutto troviamo una doppia condizione. Si possono legare tra loro due condizioni con vari operatori, i più comuni dei quali sono "or" e "and". In questo caso, com'è intuibile, stiamo dicendo: SE l'oggetto "piccola chiave" NON è presente nell'inventario (ovvero se non l'abbiamo già preso) E SE la piccola chiave è già caduta dal cappotto, ALLORA scrivi quanto segue. Questo serve ovviamente a fare in modo che quando successivamente prenderemo la chiave, la descrizione scompaia di nuovo: a quel punto una delle due condizioni necessarie non sarà più vera ($invCheck sarà true) e il contenuto del costrutto non verrà eseguito.

Per prendere la chiave, prima di tutto definiamo l'oggetto che ci servirà, sostituendo al testo il nostro widget <<obj>> che definirà la piccola chiave nelle descrizioni lunga e corta:

```
''Nel salone d'ingresso''


<<if $from is true>>\

<<switch $frompassage>>\

<<case "cucina">>\

Esco dalla cucina e mi sposto verso l'atrio. \

<<case "corridoio">>\

Attraverso il corridoio spostandomi verso l'atrio. \

<<case "">>\

\

<</switch>>\

<<set $from to false>>\

<<else>>\

\

<</if>>\

<<if $longDesc is true>>\

Sono nel salone. Un ampio tappeto è steso sul pavimento, mentre un appendiabiti, da cui pende un solitario <<obj "cappotto" "cappotto" "il " "al ">>, è posizionato a lato della pesante porta d'ingresso.<<invContiene "una
```

```
piccola chiave">><<if $invCheck is false and $chiaveCaduta is true>> Sul
tappeto, vedo la <<obj "piccola chiave" "chiave" "la " "alla ">> caduta dal
cappotto.<</if>>

<<set $longDesc to false>><<else>>\

Sono nel salone d'ingresso. Qui vedo un <<obj "cappotto" "cappotto" "il " "al
">> pendere dall'appendiabiti.<<invContiene "una piccola chiave">><<if
$invCheck is false and $chiaveCaduta is true>> Sul tappeto, vedo la <<obj
"piccola chiave" "chiave" "la " "alla ">> caduta dal cappotto.<</if>>

<</if>>\

----
```

Ora scriviamo l'azione "prendi la chiave" nel passaggio relativo al verbo "prendi":

```
<<case "prendi la chiave ">>

<<notify "Ho raccolto la piccola chiave e l'ho messa in tasca.">>

<<prendi "una piccola chiave" "piccola chiave" "la " "alla ">>

<<refresh>>
```

Testiamo il gioco: esaminando il cappotto la chiave cade a terra. Riprovando ad esaminarlo otteniamo un messaggio di diniego. Prendendo la chiave questa viene correttamente rimossa dalla descrizione. Tutto perfetto; stiamo andando alla grande. Passiamo ora a collegare tramite l'azione "esamina" i due oggetti "zoom", il telefono in corridoio e la credenza in sala da pranzo:

```
<<case "esamina il telefono ">>

<<goto "telefono">>


<<case "esamina la credenza ">>

<<goto "credenza">>
```

Ora che abbiamo accesso alla credenza, possiamo implementare la prossima azione, ovvero aprire il cassettone con la piccola chiave in nostro possesso:

```
<<case "apri il cassetto con la piccola chiave ">>

<<if $cassettoEstratto is false and $cassettoAperto is false>>

<<notify "Giro la chiavetta nella toppa del cassettone: si apre senza
problemi, ma purtroppo è vuoto!">>

<<set $cassettoAperto to true>>

<<else>>

<<notify "Il cassetto è già aperto.">>

<</if>>
```

Solo una piccola nota sul costrutto evidenziato: come da documento di progetto, non basterà aprire il cassetto con la chiave, ma estrarlo dalla sede, per trovare poi la busta. Dobbiamo quindi controllare che questo non sia già accaduto, usando un paio di variabili di controllo: nello specifico, $cassettoEstratto verifica che il cassetto non sia ancora stato estratto dalla credenza (lo useremo anche per modificare la descrizione), mentre $cassettoAperto verifica che abbiamo già aperto la serratura del cassettone. SOLO SE queste due variabili sono

ENTRAMBE false, l'azione avverrà. In qualsiasi altro caso, vorrà dire che il cassetto è già stato aperto.

Scriviamo anche una risposta alla logica "Apri il cassetto", senza chiave: anche qui, oltre a negare l'azione se tentiamo semplicemente di aprire il cassetto senza chiave, prevediamo che abbiamo già aperto la serratura e rispondiamo di conseguenza (questo è solo un abbellimento, non è strettamente necessario alla logica del gioco, ma rispondere "non si può aprire" una volta che abbiamo aperto effettivamente il cassetto sarebbe proprio brutto; diciamo che è un abbellimento "quasi" necessario):

```
<<case "apri il cassetto ">>

<<if $cassettoAperto is true>>

<<notify "Il cassetto è vuoto. VUOTO.">>

<<else>>

<<notify "E' chiuso a chiave.">>

<</if>>
```

Scriviamo subito anche l'azione che ci consentirà di estrarre il cassetto dalla credenza (sempre andando ad editare il passaggio relativo al verbo, in questo caso "tira"):

```
<<case "tira il cassetto ">>

<<if $cassettoAperto is true and $cassettoEstratto is false>>

<<notify "Estraggo il cassetto dalla sede tirandolo completamente fuori dalla credenza; nel poggiarlo a terra, mi rendo conto che dietro l'asse posteriore è appiccicato qualcosa: due pezzi di scotch tengono incollata al legno quella che sembra una piccola busta. La prendo e la apro immediatamente: dentro c'è solo un foglietto spiegazzato con scritto qualcosa.">>

<<prendi "un foglietto spiegazzato" "foglietto" "il " "al ">>

<<set $cassettoEstratto to true>>

<<refresh>>

<<elseif $cassettoAperto is false>>

<<notify "Non posso, è chiuso a chiave.">>

<<elseif $cassettoEstratto is true>>

<<notify "Ho già smontato il cassetto.">>

<</if>>
```

Diamo uno sguardo alle sezioni evidenziate: la prima è il costrutto che verifica che abbiamo aperto il cassetto con la chiave ma che questo sia ancora al suo posto nella credenza. La seconda e la terza parte in giallo sono una nuova "opzione" del costrutto <<if>>; come <<else>> si occupa di eseguire una parte di codice se la condizione specificata in <<if>> non si verifica, <<elseif>> ci permette di verificare una seconda condizione, e mettendone più di uno in serie si possono verificare varie possibilità in un unico costrutto <<if>>, in modo simile a quello che fa <<switch>>. Nello specifico, la prima condizione serve per impostare la risposta da dare al giocatore nel caso in cui il cassetto sia ancora da aprire con la chiave, mentre la seconda per la risposta da dare nel caso in cui il cassetto sia già stato sia aperto che estratto dalla credenza.

Ci resta solo da modificare la descrizione della credenza in funziona del fatto che il cassetto sia ancora al suo posto o meno:

```
''Esaminando la credenza''


E' un mobile antico, con due ampi sportelli per i piatti e un <<obj "grosso cassettone" "cassetto" "il " "al ">> per le posate <<if $cassettoEstratto is true>> che giace ora posato di traverso sul pavimento<</if>>. Il legno, nonostante l'età, sembra ben mantenuto e molto resistente.

<<set $longDesc to false>>\

----
```

Ora le descrizioni e azioni agiscono coerentemente; non ci resta che leggere la nota appena trovata, per cui aggiungiamo l'azione corrispondente nel verbo "esamina":

```
<<case "esamina il foglietto ">>

<<notify "E' una nota scritta a penna su un foglio di bloc notes; contiene solo una serie di numeri: 1 2 3 4">>
```

Manca veramente poco: la penultima azione che implementiamo è di tipo leggermente diverso dalle altre. Non si tratta di una frase di senso compiuto con un verbo e uno o più oggetti, ma di una azione di tipo "combinazione", in cui possiamo comporre dei numeri con la rotella del nostro telefono vecchio stile presente in corridoio. Questo tipo di azioni vanno definite nel passaggio "misc", che agisce in modo simile a tutti gli altri passaggi-verbo, con l'unica differenza di processare tutto quello che NON contiene un verbo. Basterà inserire l'azione in questo passaggio:

```
<<case "1 2 3 4 ">>

<<invContiene "la bottiglia di Stravecchio">><<if not $invCheck>>

<<notify "Non appena la rotella torna alla sua posizione originale, ruotando in senso antiorario con il classico ticchettio, sento una specie di //clock// provenire da dietro lo specchio a parete: subito dopo, vedo lo specchio ritrarsi di qualche centimetro nel muro e scomparire scivolando verso il basso: una cassaforte segreta. E al suo interno, posso finalmente vedere la bottiglia di Stravecchio per cui io e Bosko siamo qui. La prendo immediatamente.">>

<<refresh>>

<<prendi "la bottiglia di Stravecchio" "bottiglia" "la " "alla ">>

<<else>>

<<notify "La cassaforte è già aperta, non succede niente.">>

<</if>>
```

Come possiamo vedere, non abbiamo utilizzato nessuna variabile di controllo: per verificare che la cassaforte sia già stata aperta o meno, andremo a vedere se abbiamo o meno già preso la bottiglia. L'azione di prendere la bottiglia viene eseguita contestualmente all'apertura della cassaforte, per cui se è accaduta una cosa è accaduta necessariamente pure l'altra. Useremo lo stesso sistema per modificare la descrizione "zoom" del telefono, in modo che lo specchio non ci sia più e al suo posto sia correttamente posizionata la cassaforte appena svaligiata:

```
''Esaminando il vecchio telefono''
```

```
E' un apparecchio vecchio stile, di colore nero, di quelli con il selettore a rotella da girare col dito.<<invContiene "la bottiglia di Stravecchio">><<if not $invCheck>> Sembra più un oggetto d'arredo che altro, posizionato com'è sotto l'elegante specchio posto alla parete.<<else>> Sopra il telefono, nella parete, la cassaforte segreta nascosta dietro lo specchio è ora vuota.<</if>> Posso comunque provare a comporre un numero di telefono:

/ <<cmb "1">> / <<cmb "2">> / <<cmb "3">> / <<cmb "4">> / <<cmb "5">> / <<cmb "6">> / <<cmb "7">> / <<cmb "8">> / <<cmb "9">> / <<cmb "0">> /

<<set $longDesc to false>>\

----
```

Non resta che definire l'ultima azione, dare la bottiglia al nostro complice, nel passaggio relativo al verbo "dai":

```
<<case "dai la bottiglia a Bosko ">>

<<goto "finale">>
```

Creiamo un nuovo passaggio, lo intitoliamo "finale", aggiungiamo i tag "no_interface" e "no_menu" (non vogliamo più nessuna interfaccia, è il gran finale!) e scriviamo qualcosa di consono per porre fine alla storia.

La prima stesura della nostra avventura in MMIFIF è terminata.

## 2.h - Possibili miglioramenti

Naturalmente, possiamo e dobbiamo continuare a lavorare alla nostra avventura, se non vogliamo che resti proprio ad un livello basico di funzionamento. Possiamo incrementare le possibilità di dialogo con Bosko, per esempio, facendo in modo che nel finale, se abbiamo la bottiglia con noi, compaia un'opzione di dialogo apposita (in alternativa al più diretto "dai la bottiglia a Bosko"); ci sono poi da aggiungere oggetti di contorno e relative descrizioni, nonché tutta una possibile serie di risposte non standard ai tentativi fallimentari del giocatore: lascio ai lettori e alla loro creatività questo compito, proponendo una versione "basica" dell'avventura "Rapina a casa Berluti" così come l'abbiamo sviluppata assieme finora.

Una cosa che non abbiamo usato, è il menu a tendina che si può richiamare cliccando sul piccolo triangolo in alto: come abbiamo visto all'inizio, il contenuto di questo menu è definibile nel passaggio "Menu". In un gioco piccolo e breve come "Rapina a casa Berluti", effettivamente, non serve a molto, ma potrebbe essere utilizzato per:

- Una mappa dinamica che si costruisce in base alle locations già esplorate o meno dal giocatore;
- Una "lista di cose da fare" che serva al giocatore per tenere traccia degli obiettivi già raggiunti e delle prossime sfide, nello stile del "bloc-notes" di tante avventure grafiche;
- Un registro di indizi, prove e sospetti in un gioco di tipo investigativo;
- Una scheda personaggio, nel caso vogliamo prevedere meccaniche di tipo RPG, con tutte le varie caratteristiche tipo "Forza", "Destrezza" etc...

- Qualsiasi altra cosa, è un passaggio come gli altri...

In caso il menu a tendina non sia utile, è possibile escluderlo da ogni passaggio della nostra storia tramite il tag "no_menu".

Un'altra cosa che potremmo voler inserire, sono le immagini: non c'è nessuna limitazione da questo punto di vista, si può utilizzare il sistema di markup di Twine linkando delle immagini esterne o "impacchettarle" direttamente nei passaggi tramite il sistema "base64"; tutta la documentazione occorrente è presente nella documentazione ufficiale di Sugarcube:

http://www.motoslave.net/sugarcube/2/docs/#markup-image

Lo stesso nel caso in cui vogliamo inserire suoni e musiche, basterà applicare il codice di esempio che si trova nella documentazione:

http://www.motoslave.net/sugarcube/2/docs/#simpleaudio-api

E' possibile cambiare cose come il formato del testo, il font e i colori di ogni cosa; lo "stile" è editabile direttamente da Twine, aprendo il menu in basso a sinistra e cliccando su "Modifica i Fogli di Stile del Racconto": si tratta di semplice CSS.

# 3 - Debugging e appendici di riferimento

## 3.a - Il sistema di debug di SugarCube

SugarCube implementa un sistema di debug che si può utilizzare cliccando sul bottone "Testa": questo lancerà il gioco in "modalità debug", riempiendo lo schermo con tutti i comandi e le istruzioni normalmente "nascoste" (e generando un bel caos). La cosa potrebbe essere utile o meno, ad ogni modo possiamo tornare ad una visualizzazione "umana" cliccando sul bacherozzo in basso a destra: si aprirà una barra di debug, spegniamo la visualizzazione "alla matrix" cliccando sul bottone "Views".

La barra di debug di SugarCube ci consente di fare varie cose molto utili quando si scrive un gioco complesso: possiamo avere a video una lista di tutte o alcune variabili, vedendo "in diretta" che valore hanno e come cambiano nel corso del gioco; possiamo avere una lista storica dei passaggi visitati, nonché saltare "avanti e indietro" tra le azioni che abbiamo svolto e i passaggi in cui le abbiamo svolte.

Il sistema di debug è molto utile quando vogliamo partire da un passaggio che non è quello iniziale, per esempio per testare una stanza che magari è raggiungibile solo verso la fine del gioco e non vogliamo - ovviamente - dover rifare tutto quello che serve per arrivarci TUTTE le volte che vogliamo verificare una qualche modifica.

Per partire da un passaggio qualsiasi, basta cliccare sul tasto "Play" che compare passando il mouse sopra un passaggio: il sistema avvierà il gioco a partire da quel passaggio, in modalità debug (ricordiamo di spegnere la vista-matrix cliccando su "Views" nella barra di debug.)

## 3.b - Variabili di servizio

Queste sono le variabili utilizzate internamente. Di norma, NON vanno toccate, ma potrebbe essere utile poter cambiare lo stato di alcune di esse in passaggi complessi:

`$inv` = è un array che contiene l'inventario del giocatore.

`$reload` = è una variabile booleana che accende o spegne il refresh di un passaggio.

`$action` = è una stringa che contiene l'azione impostata dal giocatore.

`$obj1` = è il primo oggetto utilizzato nell'azione.

`$obj2` = è il secondo oggetto utilizzato nell'azione.

`$from` = è il passaggio di provenienza del giocatore.

`$invCheck` = è una variabile booleana che è vera se l'oggetto passato a <<invContiene>> è presente nell'inventario.

`$conCheck` = è una variabile interna al sistema di costruzione delle frasi.

`$verb` to = contiene il verbo selezionato dal giocatore.

`$verb_part` = è una variabile interna al sistema di costruzione delle frasi.

## 3.c - Funzioni del template

### <<obj>>

Serve per inserire un oggetto nell'ambiente di gioco. Sintassi:

```
<<obj “testo del link” “oggetto” “articolo” “preposizione”>>
```

Esempio:

```
<<obj “un bellissimo sasso” “sasso” “il ” “al ”>>
```

### <<cmb>>

Serve per inserire un oggetto di tipo "combinazione" nell'ambiente di gioco. Sintassi:

```
<<cmb "oggetto">>
```

Esempio:

```
<<cmb "1">>
```

### <<refresh>>

Serve per ricaricare una pagina. Sintassi:

```
<<refresh>>
```

### <<notify>>

Serve per far comparire un pop-up di notifica. Sintassi:

```
<<notify "messaggio">>
```

Esempio:

```
<<notify "Hello, world!">>
```

### <<prendi>>

Serve per inserire un oggetto nell'inventario. Sintassi:

```
<<prendi “testo nell'inventario” “oggetto” “articolo” “preposizione”>>
```

Esempio:

```
<<prendi “un grosso sasso” “sasso” “il ” “al ”>>
```

### <<lascia>>

Serve per cancellare un oggetto dall'inventario. Sintassi:

```
<<lascia “testo nell'inventario”>>
```

Esempio:

```
<<lascia “un grosso sasso”>>
```

### <<invContiene>>

Serve per verificare se un oggetto è presente nell'inventario, nel qual caso valorizza la variabile di servizio $invCheck come "vera", altrimenti "falsa". Sintassi:

```
<<invContiene "oggetto">>
```

Esempio:

```
<<invContiene "un sasso">>
```

---

Per ulteriore documentazione riguardo ai costrutti base di SugarCube (come <<if>>, <<switch>> e <<set>>), se qualcosa fosse ancora poco chiaro o se servissero ulteriori informazioni, la documentazione ufficiale è raggiungibile a questo indirizzo:

http://www.motoslave.net/sugarcube/2/docs/

---

rev 0.8