Lorenzo De Simone
N86/1008

aMAZEing

Progetto finale per l'esame di Laboratorio
di Algoritmi e Strutture Dati 2013/2014

Sommario

Introduzione

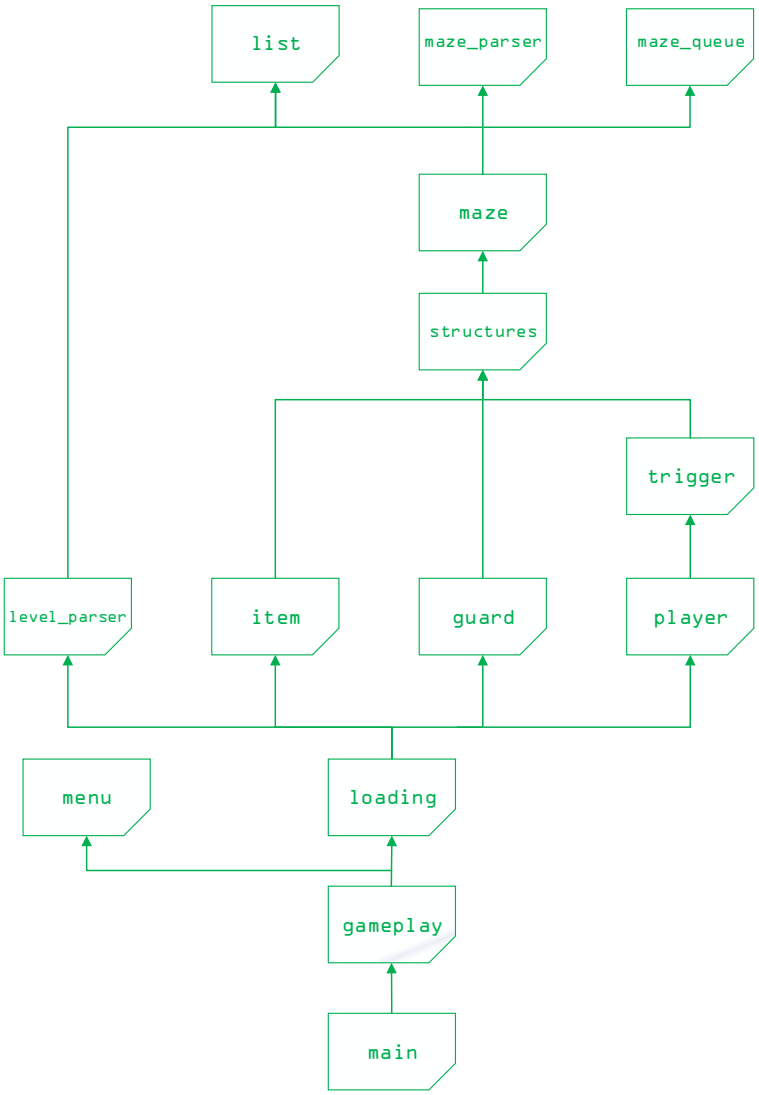
Il progetto consiste in un gioco interattivo sviluppato in linguaggio C nel quale il giocatore guida un personaggio all'interno di labirinti alla ricerca di un certo numero di chiavi che, una volta raccolte, permettono al personaggio di aprire la porta per il livello successivo.
Ogni guardia pattuglia la chiave di cui è responsabile fino a quando il personaggio non entra nel loro riquadro di guardia: quando ciò avviene, esse lasciano la chiave incustodita per inseguire il personaggio. Tutte le chiavi vanno recuperate entro un tempo limite dipendente dal livello. Quando il tempo a disposizione scade, se il personaggio ha altre vite, ne viene decrementato il numero e vengono dati altri 30 secondi di tempo. Quando una guardia intercetta un personaggio, quest'ultimo perde una vita, viene teletrasportato nella locazione di partenza e diviene invincibile per 3 secondi.
Se le vite rimanenti dovessero arrivare a 0, si farà Game Over.
Questa semplice dinamica di gioco viene arricchita dalla presenza di oggetti supplementari collezionabili dal giocatore, da bonus/malus che influenzano sia guardie che giocatori e da un set di abilità in possesso del giocatore, tra le quali, per esempio, la possibilità di rallentare le guardie in un range e di scavare muri. Queste ultime avranno un costo in punti e permetteranno al giocatore di interagire in maniera creativa con il labirinto e le guardie.

Impostazione generale del progetto

Per massimizzare la riutulizzabilità di codice, si è prestata particolare attenzione alla disposizione delle librerie e allo sfruttamento delle potenzialità della riflessione. Si cercherà di descrivere la bontà della strutturazione proposta e si motiveranno le scelte implementative effettuate. Si è scelta la libreria grafica ncurses poiché compatibile con Linux e Windows: su alcuni terminali Linux, dei caratteri potrebbero non essere visualizzati correttamente. La grandezza del terminale va impostata ad almeno 100x35. L'applicazione è stata compilata ed eseguita con successo su Windows utilizzando MinGW con le opzioni

`-lpdcurses -static`(per rendere l'exe eseguibile ovunque).

Struttura delle librerie e dipendenze



list

In questa libreria viene proposta una struttura dati molto semplice ma allo stesso tempo estremamente versatile: una lista doppiamente linkata. Questa particolare scelta, ispirata alla Java Collection Framework, permette di utilizzare la stessa struttura in modalità differenti: può essere usata come stack, coda o lista linkata scorrendola con i metodi di avanzamento di `has_next()` e `next()`, il cui uso ricorda vagamente gli Iteratori Java. I campi info sono ovviamente void* ed è una struttura dati d'appoggio: la rimozione da essa non comporta una deallocazione dei dati contenuti. Si è scelto di non deallocare, non solo per snellire il codice e non richiedere passaggi di funzioni callback apposite, ma per permettere di usare le liste anche qualora i dati vadano solo gestiti e non cancellati. In caso si debbano deallocare, vanno deallocati dopo l'estrazione dall'utilizzatore della libreria.

Non supporta alcun tipo di ordinamento ed è la scelta ideale per effettuare inserimenti ed estrazioni dalle estremità a tempo costante o per routine che necessitano di fare qualcosa su tutti gli elementi della lista a tempo lineare.

level_parser

Questa libreria fornisce funzioni per effettuare il parsing delle informazioni con le quali riempire i labirinti letti dal parser più generico dei labirinti. `level_parser` ritorna una struttura `parsed_level` apposita, che viene letta dal chiamante per costruire livelli e successivamente deallocata. Per ulteriori dettagli sul caricamento di livelli, si veda la sezione apposita.

maze_parser

Questa libreria fornisce funzioni per effettuare il parsing dei labirinti generici. `maze_parser` ritorna una struttura `parsed_maze` apposita, che viene letta dal chiamante per costruire labirinti generici e successivamente deallocata. Si è scelto di scindere le due letture in file diversi per permettere il riutilizzo del parser di labirinti per altri eventuali giochi futuri, coerentemente alla generalità offerta nella libreria `maze`. In questo modo ogni futuro gioco dovrà unicamente fornire il proprio parser nel quale si codificano le informazioni per riempire, con i propri dati specifici, un labirinto generico letto con la libreria `maze_parser`. Per ulteriori dettagli sul caricamento di livelli, si veda la sezione apposita.

maze_queue

In questa libreria viene proposta una struttura dati piuttosto specifica, ovvero una coda a priorità il cui funzionamento però è limitato a strutture di tipo `pos`: essa è una piccola struct contenente due interi che rappresentano le coordinate (y,x) della casella di un labirinto. Si è scelto di perdere di generalità per motivi di efficienza. Poiché in algoritmi di ricerca quali dijkstra e A* viene fatto largo uso di funzioni di increase key e decrease key, si è preferito ridurre la generalità ed aumentare la complessità di spazio per ottimizzare la complessità di tempo.

In pratica oltre al consueto heap tree, viene creata anche una matrice di puntatori a nodi heap. In questo modo, ad ogni chiamata di increase key o decrease key, il nodo corrispondente nello heap viene trovato a tempo costante. Per esempio, increase key sulla posizione (y,x) controlla la posizione (y,x) della sopracitata matrice: se questa locazione è diversa da NULL, implica che il nodo è nello heap ed è all'indirizzo specificato. Individuato il nodo con questo procedimento, si procede con il normale algoritmo di increase key di una coda a priorità su uno heap.

maze

Questa libreria offre una struttura dati che vuole modellare un labirinto. Quest'ultimo è simile ad un grafo, ma le uniche adiacenze possibili per ogni singolo nodo sono le quattro in direzione dei punti cardinali.

La struttura `maze` contiene due campi interi indicanti altezza e larghezza e un puntatore ad una matrice di puntatori a caselle.

La presenza di un arco è codificata in maniera implicita, infatti se due caselle sono presenti e sono posizionate in maniera adiacente esiste fra loro un arco: ciò implica che il grafo rappresentato è non orientato.

Perdendo potenzialità espressive dal punto di vista del numero di adiacenze rispetto ai grafi, si ottiene una struttura dati molto più compatta ed utilizzabile facilmente per molti scopi, quali giochi 2D.

Ogni nodo è una struct `square`, avente quattro float ed un void* che punta ad un dato generico, rendendo utilizzabile la libreria per un'infinità di scopi futuri. La scelta di utilizzare quattro float è stata effettuata per preservare potenzialità espressiva; ognuno di questi quattro valori è legato al peso degli archi

uscenti nelle quattro direzioni e permette a chi usa la libreria di modellare in giochi futuri cose come salite, discese e simili.

Se la casella è NULL, si ha un muro: questa codifica permette di risparmiare spazio e di inizializzare facilmente un labirinto pieno di muri. La coerenza del void* contenuto in ogni casella è lasciata in gestione all'utilizzatore della libreria: in questo progetto, verrà riempita con una struttura di tipo `level_square`.

Oltre alla definizione delle strutture dati, ci si vuole soffermare sulla scelta dei prototipi per tutte le funzioni di algoritmi di ricerca.

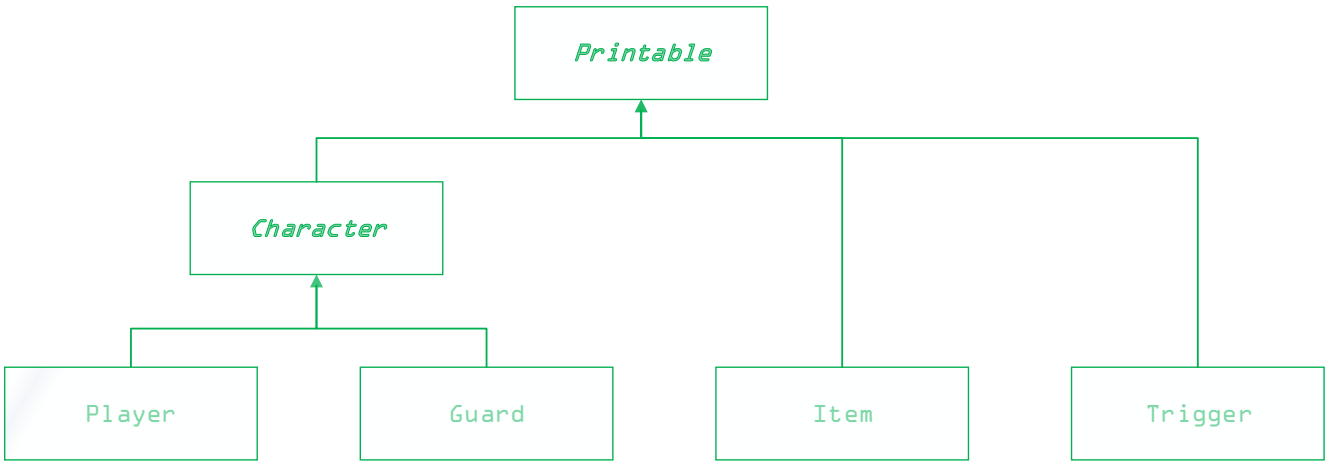
Gli algoritmi funzionano sempre prendendo in input coordinate di partenza ed arrivo, slegandosi così totalmente dal tipo di dato memorizzato. È inoltre possibile richiamare versioni di questi algoritmi con funzioni di adiacenza parametriche, per realizzare effetti particolari: a tal proposito, si veda la strategia della guardia di tipo B.

Infine, vengono fornite funzioni per la generazione di labirinti random: `random_maze()` garantisce la creazione di labirinti che sono componenti connesse. Per verificare se un labirinto in input è connesso o meno, è possibile utilizzare la funzione `is_connected()`. Si è data una certa importanza alla questione delle componenti connesse perchè permettono di garantire la raggiungibilità di ogni elemento presente in un labirinto.

structures

Questa libreria contiene le principali strutture dati del gioco `aMAZEing` e le funzioni per gestirli correttamente: da questo punto in poi si parlerà di strutture dati relative unicamente a questo gioco. Pur con le grosse limitazioni di un linguaggio non Object Oriented come il C, si è cercato di modellare una sorta di sistema di classi, con costruttori, distruttori e metodi di get() e set(). Per realizzare ciò, le sottoclassi sono struct con vari attributi e puntatori alle loro superclassi chiamati super*.

Di seguito si riporta la gerarchia di classi implementate nel gioco:



Le classi printable e character sono astratte e quindi, in teoria, non istanziabili. In structures sono presenti tutti i metodi ad esse relativi. I metodi e le istanze realizzate (effetti specifici di tutti gli oggetti, tutti bonus ecc.) delle classi istanziabili sono presenti nelle rispettive librerie. Solo due item essenziali allo svolgimento del gioco sono definiti in structures: key e door.

Si è rivelato necessario identificare ogni classe con un valore all'interno di un enumerazione il cui nome è Class.

Infatti la struttura level_square, che rappresenta una casella del gioco aMAZEing, contiene un void* che punta ad un giocatore e un altro void* che punta ad un oggetto collezionabile, di tipo trigger o item. Essi sono riconoscibili e castabili correttamente solo perché affiancati da una variabile di tipo Class che li identifica.

Il campo square_info della struttura square nella libreria generica maze, viene fatta puntare ad una struttura di tipo level_square: in questo modo si specializza la libreria generica e la si può usare facilmente.

Quasi tutti i prototipi di funzioni che operano su queste "classi" prendono in input un void* che punta all'oggetto ed un valore di tipo Class che li identifica.

Per i dettagli implementativi, si rimanda all'analisi del codice della libreria ed ai commenti.

Per le caratteristiche funzionali delle classi istanziabili, si veda il capitolo sulle componenti di gioco.

Viene qui definita la struct `level` che contiene numerosi campi: in questo paragrafo però discuteremo solo di alcuni campi notevoli la cui presenza permette una gestione efficiente del gameplay.

Per una descrizione esaustiva di ogni singolo campo della struct, si faccia riferimento al codice sorgente e ai commenti.

In primo luogo, abbiamo un puntatore al giocatore: ciò permetterà a tutte le guardie di verificare la posizione del giocatore a tempo costante. Vi sono due liste separate di trigger, una per i bonus ed un'altra per i malus: questa distinzione permetterà di realizzare efficientemente algoritmi di movimento delle guardie che effettuano azioni più articolate del semplice pattugliamento e/o inseguimento: si veda a tal proposito la guardia di tipo C. Le due liste permetteranno, inoltre, di gestire efficientemente la scadenza dei trigger presenti sul campo da gioco: ogni secondo, infatti, si esegue uno scan di queste due liste e si eliminano dal labirinto i trigger scaduti. Con questo approccio, per gestire il problema della scadenza, si ha una complessità lineare sul numero di trigger in campo. Per ulteriori dettagli sulla gestione dei trigger, si veda il capitolo sul gameplay, in particolare Livello e routine di gioco.

item

Questa libreria contiene le funzioni di gestione degli item e tutte le istanze realizzate, eccezion fatta per `key` e `door`.

guard

Questa libreria contiene le funzioni di gestione delle guardie e tutti gli algoritmi di movimento realizzati.

trigger

Questa libreria contiene le funzioni di gestione degli trigger, sia bonus che malus, e tutte le istanze realizzate.

Si è ritenuto opportuno far appartenere bonus e malus alla stessa categoria poichè in pratica effettuano modifiche strutturalmente identiche, seppur opposte ai fini di gameplay.

player

Questa libreria contiene le funzioni di gestione del giocatore e tutte le skill realizzate.

menu

Questa libreria contiene le funzioni di gestione di menu ncurses: è stata separata dal resto del codice per poterla riutilizzare con facilità: crea semplici menu di selezione con la possibilità di dare funzioni parametriche di stampa di loghi ASCII.

loading

Questa libreria contiene le funzioni per la generazione di livelli, siano essi letti da file o generati in maniera casuale.

Per i livelli letti da file, verranno eseguite le funzioni di parsing sia dei labirinti che dei livelli e verranno eseguiti controlli di coerenza: per esempio, se il file contenente le informazioni sul livello riporta una guardia in una posizione non presente nel labirinto, verrà stampato a video la causa dell'errore ed il livello non verrà caricato. Sia i parser che questa libreria tentano di indicare quanto più possibile all'utente gli errori commessi nella creazione dei file di testo.

gameplay

Questa libreria contiene tutte le routine necessarie al funzionamento del gioco, prima su tutte il loop principale che calcola il tempo passato, gestisce le scadenze e organizza i turni d'azione dei personaggi. Inoltre vengono gestite tre finestre grafiche: una contenente il labirinto di gioco, una contenente le statistiche del giocatore ed un'altra contenente il log con gli ultimi avvenimenti del gioco.

main

Il main si occupa semplicemente di inizializzare il terminale in modalità ncurses e di lanciare il menu principale del gioco aMAZEing: qualora vengano implementati nuovi giochi basati sui labirinti, è opportuno inserire qui un selettore di menu di giochi.

Componenti di gioco

Di seguito verrà descritto nel dettaglio il comportamento e le funzionalità realizzate relative ad ogni componente del gioco.

Giocatore(@)

Il giocatore viene mosso mediante comandi tastiera: alla pressione di un tasto direzionale, il giocatore si muoverà in quella direzione finché possibile. Questa funzionalità è realizzata mediante un campo direzione memorizzato nella superclasse `character`. Quando viene intercettato da una guardia, perde una vita e viene teletrasportato nella locazione iniziale: sia quando inizia il livello che quando ha perso una vita, il giocatore gode di 3 secondi di invincibilità, segnalata all'utente dai colori invertiti. L'invincibilità rende il giocatore invisibile alle guardie.

Per migliorare la giocabilità, è presente una variabile `turn` all'interno della struct `player` che permette all'utente di segnalare l'intenzione di svoltare appena possibile in una certa direzione. Tutti questi accorgimenti sono effettuati anche per emulare il comportamento di famosi cabinati arcade quali Pacman.

Il giocatore possiede inoltre un array detto `skill_set`: in esso vi sono memorizzati i puntatori a funzione delle sue abilità e il costo in punti ad esse associato. Infatti, il giocatore potrà acquisire punti in molti modi, come per'esempio la raccolta di un item; essi saranno poi spendibili per chiamare queste funzioni ed interagire con l'ambiente di gioco. Si è scelto un array perchè in questo modo è possibile associare ad un tasto un indice e richiamare a tempo costante la funzionalità desiderata. Alla pressione del tasto corrispondente, se il giocatore ha abbastanza punti e la skill ha effetto, i suoi punti verranno decrementati del valore corrispondente e si realizzerà uno degli effetti descritti qui sotto.

Skill Set

Skill	Costo	Descrizione
Time Shot	40	Rallenta tutte le guardie entro il range del giocatore: -5 all'attributo speed per 15 secondi.
Set Trap	40	Posiziona un malus di tipo `slow` dietro al giocatore. Funziona solo se il giocatore ha una direzione.
Drill	60	Scava un muro davanti a al giocatore: funziona solo se davanti un muro.
Escape	80	Teletrasporta il giocatore nella posizione iniziale del labirinto.

Statistiche del giocatore

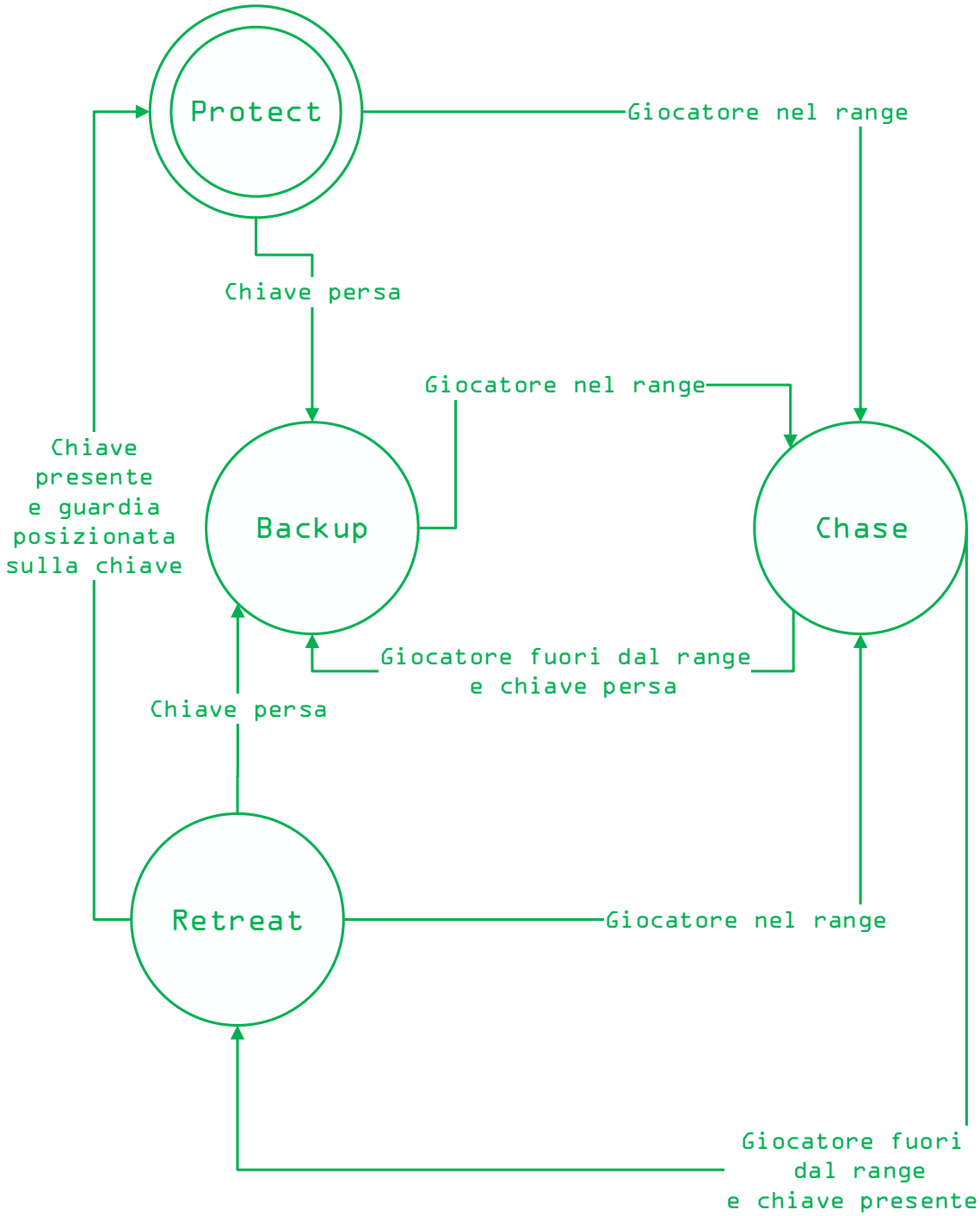
Speed	Range
15	4

Linee guida per l'implementazione di nuove abilità

Per implementare una nuova abilità basta effettuare una serie di piccole modifiche nella libreria player: scrivere il codice della nuova skill, aumentare il define `n_skills`, modificare la funzione `use_skill()` e aggiungere la skill corrispondente all'array in `create_skill_set()` con il suo costo. È consigliabile rifinire le funzioni di interfccia grafica e la creazione di menu per rendere evidente il cambiamento all'utente finale.

Guardia

Si è scelto di modellare una guardia come un automa a stati finiti: ciò permette di esprimere comportamenti della guardia che dipendono non solo dalle condizioni di gioco, ma anche dal suo stato attuale. Di seguito viene illustrato il funzionamento dell'automa guardia:



Lo stato è rappresentato da una variabile interna alla struct `guard` e varia in funzione dello stato precedente e della situazione di gioco. Questa potenza espressiva, si rivela particolarmente utile nella modalità di ritirata: infatti la traccia fornita dal docente richiede che una guardia, una volta uscita dalla modalità di inseguimento *"[...] torna, seguendo il percorso minimo, nel riquadro del suo oggetto e rientra in modalità pattugliamento."*. Volendo prendere alla lettera questa richiesta, sarebbe bastato far utilizzare alla guardia un algoritmo di ritorno fin quando essa non fosse pervenuta nel riquadro di guardia: ciò però avrebbe portato a situazioni paradossali. Se per esempio, la distanza in linea d'area della guardia dalla propria chiave fosse molto piccola, ma la distanza effettiva molto grande, essa potrebbe ritornare comunque in modalità pattugliamento, rendendo estremamente facile per il giocatore impossessarsi della chiave. Anzichè quindi utilizzare costosi algoritmi per la verifica della distanza effettiva, la modalità ritirata rende possibile distinguere il ritorno dal pattugliamento consueto. In questo modo, tutte le volte che una guardia esce dalla modalità d'inseguimento, ritorna sulla sua chiave e, una volta tornata, imposta la modalità di pattugliamento. La scelta implementativa rende quindi il pattugliamento di una guardia sempre coerente senza dover verificare la distanza effettiva dalla chiave.

Continuando l'analisi della traccia, risulta doveroso un approfondimento sul comportamento delle guardie conseguente alla seguente richiesta: "*Non appena il personaggio entra in un riquadro di guardia, la guardia coinvolta entra in modalità inseguimento e inizia a inseguire il personaggio.*". Ciò comporta, talvolta, uno sgradevole effetto: quando il personaggio è nel riquadro di guardia ma il percorso minimo per raggiungerlo porta la guardia al di fuori di questo riquadro, la guardia stessa oscillerà con una grande frequenza fra modalità d'inseguimento e modalità di ritirata. Si è deciso di non intervenire poichè, in questo caso, il comportamento non inficia il gameplay: qualora si volesse evitare, sarebbe necessario un controllo sulla distanza effettiva, rendendo però il metodo di verifica molto più costoso di quello richiesto.
Sempre sul tema della complessità di tempo, si è prestata particolare attenzione al riuso di eventuali percorsi già calcolati per ottimizzare il più possibile l'onere computazionale del comportamento delle guardie: a tal scopo, è presente all'interno della struct guard, una sorta di cache per il percorso verso un obiettivo. Ogniqualvolta una guardia è in modalità ritirata e sta tornando verso la propria chiave, non esegue sempre il proprio algoritmo di pathfinding per calcolare il proprio percorso. Ogni volta, verifica la propria cache: se essa è `NULL`, esegue un algoritmo di pathfinding e la aggiorna, se essa è

diversa da NULL non la ricalcola e la utilizza direttamente per ricavarne la direzione. Nell'esempio della modalità di ritirata, il percorso minimo verso la chiave viene calcolato solo una volta e riutilizzato in tutti i passi successivi verso la chiave.
Lo stesso approccio risulta conveniente in altre situazioni: si veda a tal proposito il codice dell'algoritmo steal_bonus() della guardia C. Per gli algoritmi di inseguimento, poichè la posizione del giocatore è estremamente variabile, si è preferito ricalcolare ad ogni passo il percorso minimo.

Ogni guardia ha un riquadro di inseguimento e la grandezza di quest'ultimo è definita dal suo attributo range presente nella superclasse character: quando il giocatore si trova in questo riquadro, la guardia inverte i propri colori e muta il proprio stato in Chase, utilizzando il proprio algoritmo di inseguimento per intercettarlo. Qualora il personaggio esca dal riquadro di inseguimento, la guardia muta il proprio stato in Retreat o Backup, coerentemente al diagramma dell'automa sopra presentato.

La grandezza del riquadro di guardia è definita dalla variabile key_range, definita nella struct guard: questo riquadro definisce l'area all'interno della quale la guardia può muoversi attorno alla chiave quando il giocatore non è nelle vicinanze.

Ogni guardia ha quattro attributi di tipo puntatori a funzione, che esprimono ognuna gli algoritmi legati ad ogni stato dell'automa.

Questa scelta implementativa permette con facilità di aggiungere nuove guardie, nuovi algoritmi e diversificare ogni attributo di una guardia (range, key_range, speed), offrendo un enorme potenzialità espressiva allo sviluppatore: è persino possibile definire una guardia diversa miscelando algoritmi già esistenti.

La funzione create_custom_guard() permette infatti di creare una guardia specificandone tutti gli attributi, garantendo la massima personalizzazione.

Per comodità, vengono fornite delle guardie standard, le quali hanno l'enum guard_type che le identifica e che permette durante il gioco di indentificarne il tipo. Questo enum rispecchia anche le regole della grammatica del parsing dei livelli, in modo tale da rendere possibile da file la lettura di questi tipi di guardie. La funzione create_guard(), infatti, prende in input una variabile di tipo guard_type e crea una guardia standard del tipo corrispondente. Si descrivono qui le strategie e le caratteristiche delle quattro guardie standard. Per i dettagli

implementativi degli algoritmi qui nominati, si faccia riferimento al codice sorgente.

Guardia A "Stupida" (A)

Speed	Range	Key Range
10	3	5

Backup	Chase
panic()	chase_weak()
La guardia si muove senza senso, simulando un comportamento dettato dal panico, come suggerisce il nome della funzione.	La guardia tenta di intercettare il giocatore seguendo il percorso minimo verso la sua posizione con qualche errore(per ogni passo, c'è una possibilità su 3 che sia random).

Patrol	Retreat
protect_random()	retreat_weak()
La guardia vigila intorno alla chiave senza un pattern preciso.	La guardia segue il percorso minimo verso la propria chiave con qualche errore (per ogni passo, c'è una possibilità su 3 che sia random).

Guardia B "Cauta"(B)

Speed	Range	Key Range
12	4	2

Backup	Chase
help_another_guard()	chase_careful()
La guardia tratta un'altra chiave in gioco come se fosse la propria. Se non ci sono più chiavi, esegue la strategia di backup della guardia A.	La guardia tenta di intercettare il giocatore seguendo il percorso minimo verso la sua posizione evitando però le trappole: se delle trappole ostruiscono ogni percorso, rimane ferma. Ciò viene realizzato richiamando a_star_custom() con una funzione di adiacenza apposita, che considera le caselle con trappole come se fossero muri.

Patrol	Retreat
protect_careful()	retreat_careful()
La guardia vigila intorno alla chiave **seguendo linee dritte ed evitando le trappole.**	La guardia segue il percorso minimo verso la propria chiave evitando però le trappole.

Guardia C "Avida"(C)

Speed	Range	Key Range
12	5	3

Backup	Chase
steal_bonus()	intercept()
La guardia si muove per rubare tutti i bonus in gioco e accumularne gli effetti benefici. Sfrutta il meccanismo di cache per il percorso minimo.	La guardia tenta di intercettare il giocatore prevedendone gli spostamenti in base alla sua direzione.

Patrol	Retreat
protect_random()	retreat_weak()
La guardia vigila intorno alla chiave senza un pattern preciso.	La guardia segue il percorso minimo verso la propria chiave con qualche errore (per ogni passo, c'è una possibilità su 3 che sia random).

Guardia D "Coraggiosa"(D)

Speed	Range	Key Range
14	6	2

Backup	Chase
help_another_guard()	chase_a_star()
La guardia tratta un'altra chiave in gioco come se fosse la propria. Se non ci sono più chiavi, esegue la strategia di backup della guardia A.	La guardia tenta di intercettare il giocatore seguendo il percorso minimo verso la sua posizione senza errori.

Patrol	Retreat
protect_straight()	retreat_a_star()
La guardia vigila intorno alla chiave **seguendo linee dritte.**	La guardia segue il percorso minimo verso la propria chiave.

Linee guida per l'implementazione di nuove guardie

Per implementare una nuova guardia basta effettuare una serie di modifiche nella libreria `guard`: qualora si vogliano implementare nuove strategie, esse vanno ovviamente scritte rispettando il prototipo previsto e poi basterà richiamare la funzione `create_custom_guard()` con i valori e le strategie desiderate. Qualora si voglia rendere la nuova guardia una guardia standard, vanno fatte anche delle modifiche nella grammatica del parser, e nella funzione `char_to_guard_type()`, che converte i caratteri letti da file nel corrispondente enum. Va inoltre modificata la funzione `create_guard()` e aggiunto il valore corrispondente all enum `guard_type`.

Qualora si voglia che la nuova guardia compaia nei livelli random va aumentato il define `n_guards_types`.

Item

Tutti gli oggetti appartenenti a questa categoria risiedono nel labirinto di gioco fino a quando il giocatore non li collezionerà. Non possono essere presi dalle guardie poiché sono oggetti rivolti unicamente al giocatore e quando quest'ultimo va su una casella con un item lo colleziona: ciò ne fa scaturire l'effetto. Sono rappresentati nel gioco mediante un pallino giallo(°), ad eccezione della chiave che ha il simbolo della sterlina(£) e della porta che è un mattoncino giallo( ). Non avendo scadenze, non c'è il bisogno di inserire gli oggetti in gioco in strutture dati ausiliarie e sono presenti unicamente nel labirinto di gioco: inoltre ogni guardia ha come attributo le coordinate della propria chiave e ogni livello ha le coordinate della propria porta.

Item implementati

Item	Effetto
Key	Aggiunge una chiave al giocatore e aumenta di 60 i punti del giocatore: se è l'ultima chiave del livello, fa apparire un oggetto di tipo door all'uscita del labirinto e converte i secondi rimasti in punti.
Door	Modifica lo stato del livello in LEVEL_CLEARED e permette al giocatore di passare al livello successivo.
Life	Aggiunge una vita al giocatore
Points	Aumenta di 40 i punti del giocatore
Few Points	Aumenta di 5 i punti del giocatore
Seconds	Aumenta di 30 secondi il tempo a disposizione del giocatore
Golden Path	Se è presente la porta, lastrica il percorso verso la porta di items del tipo `few_points`. Se la porta non dovesse essere ancora comparsa, sceglie un bonus come fine del percorso. Se non c'è alcun bonus, si comporta come l'item `points`.

Linee guida per l'implementazione di nuovi item

Per implementare un nuovo item basta scrivere la funzione che ne descrive l'effetto nella libreria `item` e darla in input a `create_item()`. Qualora lo si voglia rendere generabile dalla routine di generazione di oggetti random, va aumentato il define `n_item` e va modificata la funzione `init_all_item()`.

Trigger

Tutti gli oggetti appartenenti a questa categoria influenzano sia il giocatore che le guardie che vi finiscono sopra: ciò è possibile perché influenzano attributi della loro superclasse comune `character`. Un loro riferimento, oltre a quello nella casella dove risiedono, è presente in una lista attributo della struct `level`: tutti i malus sul terreno di gioco sono nella lista `malus_tot` e tutti i bonus sul terreno di gioco sono nella lista `bonus_tot`. Si presentano sul terreno di gioco come scacchiere colorate di verde nel caso di bonus(▒) o di rosso nel caso di malus(▒).

Tutti i trigger rimangono sul terreno di gioco per `vanish_time (15)` secondi, dopo i quali, se nessuno li ha raccolti, svaniscono.

La classe `character` ha due liste, una per i bonus attivi ed una per i malus attivi.

La struttura di lista permette di rendere molti effetti, anche uguali, cumulativi fra loro, siano essi positivi o negativi.

Si è deciso di tenerle separate per poter selettivamente influenzare una o l'altra lista: si vedano a tal proposito il bonus `cure` e il malus `curse`.

Le funzioni dei trigger possono essere chiamate in due modalità: `ON` e `OFF`.

Quando un personaggio attiva un trigger, si richiama l'effetto del trigger in modalità `ON` e si aggiunge alla lista corrispondente una struct di tipo `mod` contenente un puntatore alla funzione dell'effetto e un numero di secondi di durata dell'effetto.

Alla scadenza dell'effetto, la funzione verrà richiamata da un'apposita routine in modalità `OFF` eliminando quel singolo effetto.

Bonus implementati

Bonus	Effetto
Haste	Aumenta la velocitá di un personaggio (+5)
Range Bonus	Aumenta il range di un personaggio (+3)
Cure	Rimuove tutti i malus da un personaggio

Malus implementati

Malus	Effetto
Slow	Diminuisce la velocitá di un personaggio (-5)
Range Malus	Diminuisce il range di un personaggio (-3)
Curse	Rimuove tutti i bonus da un personaggio
Teleport	Muove il personaggio in una posizione random del labirinto

Linee guida per l'implementazione di nuovi trigger

Per implementare un nuovo trigger basta scrivere la funzione che descrive il nuovo effetto nella libreria `trigger` e chiamare la funzione `create_trigger()` con il nuovo effetto in input e il tipo di trigger. Qualora lo si voglia rendere generabile dalla routine di generazione di oggetti random, va aumentato il define `n_bonus` o `n_malus` e va modificata la funzione `init_all_bonus()` o `init_all_malus()`.

Gameplay

Di seguito verranno illustrate tutte le modalità di gioco e il funzionamento delle routine di gioco. Quando un livello è terminato, a seconda dell'esito, verrà visualizzata una schermata di vittoria o di gameover, mostrando il livello terminato è il punteggio del giocatore.

Modalità di gioco

Modalità	Descrizione
Single Game	È possibile scegliere singolarmente uno dei livelli standard codificati nei file di testo forniti e giocarli in modalità singola: una volta terminato quel labirinto, si viene riportati al menu principale.
Adventure	Il giocatore parte dal primo livello e tenta di completare tutti i livelli con un numero fissato di vite: da un livello all'altro, però, manterrà i punti accumulati e spendibili per l'uso delle skill che gli faciliteranno l'avanzamento.
Random Survival	Nella modalità Random Survival, il giocatore continua a giocare potenzialmente all'infinito in una serie di livelli generati casualmente. Come in Adventure, il giocatore mantiene i punti accumulati passando da un livello al successivo.
Custom Level	L'utente può fornire un proprio file contenente il labirinto ed il livello e giocarci. Nel caso siano presenti errori di parsing o di coerenza dei file di testo, essi verranno opportunamente segnalati a video e l'utente avrà indicazioni sulla loro origine.

Livello e routine di gioco

Una volta caricato il livello ed inizializzate le strutture necessarie, parte un ciclo di polling, che da qui in poi chiameremo main loop.
Dal main loop si esce in caso di vittoria, gameover o a causa della pressione del tasto ESC durante il gioco.
Ad ogni iterazione del main loop si eseguono le seguenti operazioni:

- Refresh delle finestre virtuali di interfaccia
- Se è passato più di un secondo dall'iterazione precedente, verifica la scadenza di tutti i trigger in campo scorrendo le apposite liste. Verranno inoltre esaminati tutti gli effetti attivi di tutti i personaggi ed eventualmente generati oggetti random.
- Mediante la funzione `delta_time()` si verifica se è passato abbastanza tempo per far agire, prima il giocatore, poi tutte le guardie, scorrendo opportunamente la lista che ne contiene i riferimenti. Nelle funzioni di movimento `move_player()` e `move_guard()` si gestiscono tutti gli eventi di raccolta di item e/o trigger e collisioni fra personaggi. Nella routine `move_player()` il giocatore usa i tasti direzionali per muoversi, la barra spaziatrice per fermarsi e i vari tasti per usare le sua abilità. Nella routine `move_guard()`, le guardie utilizzeranno l'algoritmo appropriato a seconda della situazione di gioco.

Caricamento dei livelli gioco

Verranno ora esaminate nel dettaglio le varie possibilità di caricamento dei livelli.

Caricamento dei livelli da file

Il caricamento da file si compone di tre parti: in primo luogo viene eseguito il parsing su un file di testo contenente il labirinto. Quest'ultimo rispetta una grammatica molto semplice e permette di creare labirinti generici.
La grammatica è molto semplice: '#' indica un muro ed uno spazio ' ' indica un corridoio. Gli unici vincoli da rispettare sono quelli per cui ogni colonna deve avere la stessa altezza ed ogni riga deve avere la stessa lunghezza, poiché sono accettati solo labirinti rettangolari.
Nella seconda fase, dopo aver eseguito il parsing del labirinto, si procede al parsing del livello che ha la seguente grammatica:

```
<level>      ::= <time> <source> <end> <n_guards> <guard_list>
<time>       ::= '['int']'
<start>      ::=  S <pos>
<exit>       ::=  E <pos>
<pos>        ::= '('int,int')'
<n_guards>   ::= '('int')'
<guard>      ::= [A,B,C,D] <pos>
<guard_list> ::= {guard}
```

Vengono esaminate alcune regole:

- Due guardie non possono partire nella stessa locazione.
- Una guardia non può partire sull'entrata.
- Una guardia non può partire sull'uscita.
- Ci deve essere almeno una guardia nel livello.
- Tutti gli interi devono essere numeri maggiori o uguali a 0

Nella terza fase, dopo aver eseguito il parsing del livello, la libreria `loading` provvede a verificare la coerenza fra i due file di testo e alcuni controlli aggiuntivi relativi al gioco aMAZEing:

- Il labirinto minimo è 10x45 e quello massimo è 20x98
- Tutte le posizioni indicate nel file del livello devono essere disponibili nel labirinto
- Il labirinto deve essere una componente connessa

In presenza di un qualsiasi tipo di errore, viene mostrata all'utente la causa dell'errore, si deallocano le strutture parzialmente costruite fino a quel momento e si ritorna al menu principale.

Caricamento dei livelli random

Il caricamento random prende in input una funzione di generazione di labirinti: nel gioco aMAZEing viene sempre usata la funzione random_maze() presente nella libreria maze, ma con questa impostazione è facile far funzionare il caricamento con futuri algoritmi di generazione di labirinti. random_maze() genera un labirinto effettuando una DFS Visit con ordine di visita delle adiacenze random: ad ogni passo, però, un'adiacenza casuale viene colorate di nero senza essere visitata. In questo modo si ha la garanzia che il labirinto generato sia una componente connessa ed aggiunge un certo grado di casualità.
Questo labirinto generico, viene inizializzato con le strutture del gioco e vengono aggiunte in maniera random le posizioni di partenza, uscita e quelle di ogni guardia.
Viene prestata particolare attenzione, ai fini della giocabilità, che le guardie siano tutte distanti almeno il proprio range+1 dal giocatore, impedendo al gioco di iniziare in modalità inseguimento. Come ulteriore tutela al giocatore, quest'ultimo viene fatto partire con due secondi di invincibilità.
Ogni nuovo oggetto da aggiungere al livello, viene aggiunto a delle coordinate precise: se viene trovato un corridoio, esso viene inserito, altrimenti si scava un nuovo nodo e lo si connette alla componente connessa. Un ulteriore accorgimento è l'aggiunta di "stanze" attorno alle guardie: viene effettuata una BFS in un range che varia da 2 a 4 e si scavano tutti i muri attorno ad ogni guardia, creando di fatto delle stanze dove vengono custodite le chiavi.

Linee guida per la creazione di nuovi livelli

Per creare nuovi livelli e giocarli nella modalità Custom Level, basterà seguire le grammatiche e le indicazioni sopra specificate. Qualora si vogliano aggiungere nuovi livelli standard, bisognerà creare i file di testo con la seguente dicitura: *_level_i.txt* e *_maze_i.txt* con *i* il numero del nuovo livello. Va aumentato il define n_levels e il define n_choices nella libreria gameplay, dove bisogna modificare le chiamate alle funzioni di creazione di menu con le nuove stringhe corrispondenti all'opzione di selezione del nuovo livello. Nel codice sorgente sono opportunamente commentate le zone specifiche da modificare, per agevolare eventuali programmatori futuri nel loro compito.