

GARA GRA i LABIRINTI

Funzionalità, algoritmi e scelte implementative

Indice

1 Funzionalità previste (breve panoramica)	1
Funzionalità previste	1
2 Strutture dati	3
Strutture dati	3
Generale	3
Librerie standard	3
Lista	3
Stack	3
Coda	4
Coda a priorità	4
Hashtable	4
Parser	5
Parser generico	5
Parsing di files e stringhe	6
Parsing dei files di configurazione	6
Interfacciamento con la libreria grafica e funzioni di base	6
Inizializzazione libreria Allegro	6
Timers	6
Immagini	7
Tiling	7
Audio	8
Tastiera	8
Funzioni varie	9
Random	9
Entità geometriche: punti, dimensioni, rettangoli	9
Gioco	10
Contesto di gioco	10
Eventi	10
Livello	11
Mappa	11
Cella	12
Personaggio	12
Bonus	13
3 Algoritmi	15
Algoritmi	15
Personaggi	15
Collisioni tra personaggi	15
Comportamento degli avversari	15
Strategie di inseguimento	16
Mappe	16
Conversione di una mappa dalla rappresentazione testuale	16
Generazione di un labirinto perfetto	17
Generazione di un labirinto intracciato	18
Posizionamento degli avversari e dei bonus	18

Calcolo della velocità di un personaggio in base alla cella	18
---	----

1 Funzionalità previste (breve panoramica)

- Generale**
- Cross-platform
 - Gran parte del gioco è personalizzabile tramite dei files di configurazione:
 - Personaggi
 - Livelli
 - Mappe
 - Bonus
 - Effetti sonori
- Livelli**
- Numero e tipo di avversari personalizzabile.
 - Numero e tipo di mappe personalizzabile.
 - Aspetto delle mura e dei corridoi personalizzabile.
 - Grado di complessità delle mappe generate dipendente dal livello (mappe più complesse hanno più zone con costo diverso e meno bonus).
- Mappe**
- Descrizione tramite un file di configurazione.
 - Generazione casuale di labirinti perfetti o meno.
 - Toroidali, due celle sullo stesso asse e agli estremi opposti della mappa sono collegate.
 - Possibilità di associare ad ogni cella della mappa un diverso *costo* che ne determina la velocità di percorrenza.
- Utente**
- Proprietà configurabili¹:
 - velocità
 - possibilità di ignorare le collisioni
 - possibilità di abbattere le mura
 - dimensione dell'area di mappa visibile
- Avversario**
- Proprietà configurabili¹:
 - velocità
 - possibilità di ignorare le collisioni
 - dimensione dei riquadri di ricerca dell'uscita e di inseguimento
 - preferenza per l'inseguimento o la ricerca dell'uscita
 - metodo di ricerca del percorso minimo
 - algoritmo per l'inseguimento dell'utente:
 - ◊ inseguimento classico
 - ◊ predizione basilare della posizione
 - ◊ imboscata
- Bonus**
- Comparsa in posizioni casuali ad intervalli di tempo casuali.
 - Utilizzabili sia dall'utente che dagli avversari.

¹Proprietà configurabili ai fini del comportamento, per tutte le proprietà fare riferimento alla guida sul funzionamento, sezione *Configurazione*.

- Possibilità da parte dell'utente di accumulare bonus e utilizzarli in un secondo momento tramite pressione di un tasto.
- Possibilità di favorire/penalizzare uno o più personaggi. Dunque un bonus può funzionare anche da trappola.
- Descritti dalle proprietà che andranno a modificare sui personaggi. Qualsiasi proprietà di un personaggio può subire effetti da parte di un bonus.

2 Strutture dati

Generale

L'applicazione è composta da:

1. Codice specifico del gioco, indipendente dalla libreria grafica sottostante.
2. Funzioni varie: generazione di numeri random, geometria
3. Una interfaccia tra la libreria grafica e il gioco.
4. Libreria per il parsing di file e in particolare per i file di configurazione.
5. Librerie standard: lista, coda, stack, lista a priorità, hashtable.

Librerie standard

Lista doppiamente linkata (**std/list.c**)

La libreria per le liste consente di rappresentare e gestire liste doppiamente linkate.

list_s: lista

head puntatore al primo nodo della lista

tail puntatore all'ultimo nodo della lista

element_type funzioni per gestire il tipo di dato contenuto nei nodi

length numero di nodi contenuti nella lista

list_node_s: nodo

value puntatore al dato contenuto nel nodo

next puntatore al nodo successivo

prev puntatore al nodo precedente

free se deallocare o meno il valore quando si dealloca il nodo

Operazioni eseguibili su una lista:

- Creazione/distruzione
- Inserimento di un elemento in testa o in coda
- Estrazione di un elemento
- Estrazione dell'elemento in testa
- Estrazione di un elemento casuale

Stack (**std/stack.h**)

sstack_s non è altro che una ri-definizione di **list_s** accompagnata da una serie di macro per l'inserimento e la rimozione in e dalla testa della lista.

Coda (std/queue.h)

queue_s, come per lo stack, non è altro che una ri-definizione di list_s accompagnata da una serie di macro per l'inserimento e la rimozione rispettivamente in coda e dalla testa della lista.

Coda a priorità (std/priority_queue.c)

La coda a priorità è implementata, nelle varianti con min e max heap, utilizzando un array per rappresentare lo heap.

priority_queue_s: lista a priorità

nodes array dei nodi

size dimensione dell'array

type tipo di coda (PRIORITY_QUEUE_MIN/PRIORITY_QUEUE_MAX)

length numero di nodi contenuti nella lista

priority_queue_node_s: nodo

key chiave dell'elemento, determinante la posizione nello heap

value puntatore al valore dell'elemento

index posizione del nodo nell'array dei nodi

Operazioni eseguibili su una coda a priorità:

- Creazione/distruzione
- Estrazione del massimo/minimo
- Lettura del valore dell'elemento in testa
- Incremento/decremento valore di una chiave
- Inserimento di un nuovo elemento (nei limiti della dimensione pre-allocata)

Hashtable (std/hashtable.c)

Implementazione di hashtable con closed addressing.

L'hashing delle chiavi, di default, è effettuato utilizzando la funzione FNV1a².

hashtable_s: hashtable

size numero di locazioni disponibili

hash puntatore alla funzione di hashing

table locazioni della hashtable, ognuna contenente una lista di nodi per gestire le collisioni

key_type funzioni per la gestione del tipo di dato delle chiavi

²<http://www.isthe.com/chongo/tech/comp/fnv/>

hashtable_node_s: nodo della tabella

hash hash della chiave del nodo

key puntatore al valore della chiave

value puntatore al valore del nodo

value_type funzioni per gestire il tipo di dato del valore del nodo

free se deallocare o meno il valore in fase di deallocazione del nodo

table puntatore alla hashtable di appartenenza

Operazioni eseguibili su una hashtable:

- Creazione/distruzione
- Inserimento di una coppia chiave/valore
- Sostituzione di un valore tramite ricerca della chiave
- Ricerca di un valore
- Rimozione di un valore
- Iterazione sugli elementi contenuti nella hashtable

Parser

Parser generico: parser/parser.c

Un parser è una struttura utilizzata come supporto per analizzare stringhe (contenute in file, memoria, altro...).

In fase di creazione di un nuovo parser gli si associano: un buffer, una destinazione e una serie di funzioni per operare sul buffer.

parser_s: parser

buffer sorgente dei dati da analizzare

destination destinazione dei dati analizzati

position offset corrente nel buffer

length lunghezza del buffer

methods funzioni per la gestione del buffer:

is_eof verifica se ci si trova alla fine del buffer

current ritorna il carattere alla posizione corrente

next sposta in avanti di un carattere la posizione corrente

seek sposta la posizione corrente in un punto qualsiasi del buffer

Al parser sono associate una serie di funzioni per:

- Leggere id numerici
- Leggere id alfanumerici

- Leggere id di soli caratteri
- Leggere interi
- Leggere floats
- Leggere stringhe (tra doppi apici)

Parsing di files e stringhe: parser/buffer|file_parser.c

Per l'utilizzo nel contesto dell'applicazione sono definite delle funzioni per gestire il parsing da file e da stringhe in memoria.

Parsing dei files di configurazione (.cfg): config/config.c

I files di configurazione dell'applicazione (come descritti nel documento riguardante il funzionamento), sono letti ed analizzati sfruttando il parser e le funzioni per l'input da file.

Un file di configurazione può contenere diversi tipi di dato:

```
interi int variabile = 1;

floats float variabile = 1.5;

stringhe string variabile = "Ciao";

liste list[int] variabile = 1, 2, 3;

dimensioni size variabile = [640, 480];

rettangoli rectangle variabile = [x, y, 640, 480];

dizionari dictionary variabile = {
    int chiave1 = 2,
    dictionary chiave2 = {
        string chiave3 = "prova";
    }
}
```

Interfacciamento con la libreria grafica e funzioni di base

Inizializzazione libreria Allegro (main/graphics.c)

L'inizializzazione della libreria Allegro è eseguita in fase iniziale e si svolge in 2 parti:

1. `graphics_initialize_library()` si occupa di inizializzare la libreria grafica e i suoi moduli principali.
2. `graphics_initialize(configurazione principale)` si occupa di creare la finestra principale, inizializzare il gestore degli eventi, i timer e caricare il font per la scrittura.

Timers (main/timer.c)

Un timer (`ttimer_s`) non è altro che generatore di eventi che si attiva con una certa frequenza. Il gioco prevede l'utilizzo di 3 timers principali:

Timer globale — **Frequenza:** $\frac{1}{60}$ **di secondo** è utilizzato per aggiornare il contenuto dello schermo, quindi la mappa e i suoi contenuti.

Timer dei bonus — **Frequenza:** **1 secondo** è utilizzato per aggiornare lo stato dei bonus in uso da parte dei personaggi.

Timer dei personaggi — **Frequenza:** $\frac{1}{5}$ **di secondo** è utilizzato per animare i personaggi, cioè alternare i frame che vengono disegnati sullo schermo per simulare il movimento.

Immagini (main/image.c)

Tutti gli elementi grafici del gioco sono contenuti in immagini in formato Bitmap che all'occorrenza sono caricati in memoria e immagazzinati in una struttura di tipo `image_s`.

`image_s`: Immagine

bitmap riferimento all'immagine caricata dalla libreria grafica

size dimensioni dell'immagine

rectangle un'immagine può anche rappresentare una parte di una bitmap più grande, in tal caso **rectangle** descrive l'area da considerare

In fase di caricamento di una Bitmap dal filesystem, utilizzando la funzione

`image_load_new(path, region, mask_alpha)`

è possibile specificare la parte da prendere in considerazione (*region*) e se rendere il colore `#007575` trasparente (*mask_alpha*) quando disegnato sullo schermo.

Tiling (main/tiling.c)

Le immagini che rappresentano i personaggi, il terreno e le mura sono conservate in un unico file immagine per tipo quindi, prima dei poter essere utilizzate, devono essere scomposte e ricomposte nell'ordine esatto (tiling).

A questo scopo sono definite due funzioni di tiling:

- `character_tiling(character, texture, rect)` si occupa del tiling dei personaggi.
Un personaggio è caratterizzato da una bitmap che contiene 12 frame, 3 per il movimento in ogni direzione.
La funzione si occupa di separare questi 12 frame e inserirli in un array per poter essere utilizzati in fase di animazione del personaggio.



Figura 1: Personaggio

- `map_get_tiling_points(map, point, points)` si occupa del tiling di una cella della mappa.
Il tiling delle celle della mappa è più complesso rispetto a quello dei personaggi, in questo caso infatti, ogni cella è composta in modo tale da combaciare con le celle presenti ai 4 lati.
Per ottenere questo risultato ogni cella è divisa in 4 parti: A, B, C, D composte in base al tipo di cella a:
 - Sinistra e in alto rispetto ad A
 - Destra e in alto rispetto a B
 - Sinistra e in basso rispetto a C
 - Destra e in basso rispetto a D

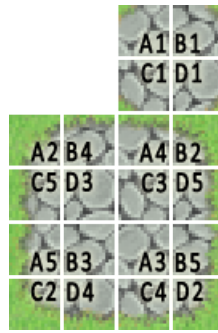


Figura 2: Divisione di una bitmap che rappresenta un percorso o un muro

Audio (main/audio.c)

Se la libreria Allegro è compilata con il supporto all'audio, il gioco consente il caricamento di tracce audio WAV e MOD da utilizzare come sottofondo musicale per un livello o come effetto sonoro per l'utilizzo di un bonus. Ogni traccia audio è contenuta in una struttura `audio_sample_s`.

`audio_sample_s`: **traccia audio**

type tipo di traccia, tra: `AUDIO_SAMPLE_TYPE_MOD` e `AUDIO_SAMPLE_TYPE_WAV`

gain livello di volume della traccia $\in [0.0, 1.0]$

fading indica se la traccia è in fase di fading out

track struttura che contiene i riferimenti alle tracce caricate dalla libreria Allegro.

Sono supportate solamente le funzionalità di base sulle tracce audio:

- Avvio
- Stop
- Muto
- Fade out

Tastiera (io/keyboard.c)





L'unico dispositivo di input supportato è la tastiera.

La gestione è molto semplice e avviene attraverso 2 funzioni.

- `keyboard_get_state()` aggiorna lo stato dei tasti
- `keyboard_is_key_down(key)` controlla se il tasto `key` è premuto

I tasti sono definiti nel modo seguente:

<code>KYB_KEY_0-9</code>	<input type="text" value="0"/> <input type="text" value="1"/> ... <input type="text" value="9"/>
<code>KYB_KEY_A-Z</code>	<input type="text" value="A"/> <input type="text" value="B"/> ... <input type="text" value="Z"/>
<code>KYB_KEY_UP</code>	<input type="text" value="↑"/>
<code>KYB_KEY_DOWN</code>	<input type="text" value="↓"/>

KYB_KEY_LEFT	
KYB_KEY_RIGHT	
KYB_KEY_ESCAPE	
KYB_KEY_ENTER	

I restanti non sono utilizzati.

Funzioni varie

Random (misc/random.c)

Aniché fare affidamento sulla funzione `random` della libreria standard del C, l'applicazione utilizza un migliore generatore di numeri pseudo-casuali: WELL512³.

Il generatore richiede una inizializzazione eseguita con la chiamata alla funzione `well512_initialize()`.

Tramite una serie di macro è possibile generare:

- `random_float()`: float $x \in [0.0, 1.0]$
- `random_int(m, M)`: intero $x \in [m, M]$
- `random_bool(p)`: valore booleano 1 con probabilità $p \in [0.0, 1.0]$, 0 con probabilità $p_1 = 1.0 - p$.
- `random_string(max)`: stringa di lunghezza massima *max*.

Entità geometriche: punti, dimensioni, rettangoli (misc/geometry.c)

Per semplificare le operazioni di disegno, movimento, collisioni sono definite una serie di strutture per rappresentare: punti, dimensioni e rettangoli.

`point_s`: punto

float **x** coordinata sull'asse della ascisse

float **y** coordinata sull'asse delle ordinate

`dimension_s`: dimensioni

float **width** larghezza

float **height** altezza

`rectangle_s`: rettangolo

point_t **origin** origine

dimension_t **size** dimensione

³<http://www.iro.umontreal.ca/panneton/WELLRNG.html>

Gioco

Contesto di gioco (game/game.c)

La struttura principale che contiene tutte le informazioni sullo stato del gioco è `game_s`.

Nel dettaglio, la struttura raccoglie:

state bitmask che rappresenta lo stato del gioco, i bit rappresentano:

0. `GAME_STATE_STARTED` se attivo indica che il gioco è stato avviato
1. `GAME_STATE_RUNNING` se attivo indica che il gioco è in corso (non in pausa)
2. `GAME_STATE_OVER` se attivo indica che il gioco è finito
3. `GAME_STATE_WON` se attivo indica che l'utente ha vinto

mute indica se riprodurre (false) o meno (true) l'audio

user personaggio controllato dall'utente

levels coda dei livelli del gioco, ogni volta che l'utente passa da un livello all'altro, il livello di provenienza viene estratto e deallocato

powerups lista dei bonus caricati

events_table hashtable contenente gli handlers degli eventi

events_queue coda degli eventi da gestire

powerups_timer timer dei bonus

characters_timer timer dei personaggi

transitions animazioni visualizzate in fase di cambio livello

animations animazioni da gestire ad ogni loop

audio_samples tracce audio caricate

Eventi (game/events.c)

Una delle strutture fondamentali alla base del gioco è il gestore degli eventi.

Ad ogni loop è possibile generare un evento, che al prossimo loop sarà gestito chiamando una serie di callbacks precedentemente registrati.

Il gestore degli eventi ha alla base 2 strutture dati:

- `game.events_table`: una hashtable che ad ogni tipo di evento associa la lista dei callback registrati per la gestione di quell'evento
- `game.events_queue`: una coda che contiene gli eventi appena generati e ancora da gestire

Un evento è identificato tramite una costante (definita in `game/events.h`).

Ognuna delle costanti rappresenta un bit di una sequenza di 32 (`uint32_t`), tale rappresentazione permette di:

1. Rappresentare tutti gli eventi verificatisi in un loop attraverso una bitmask
2. Memorizzare gli handler in sole 32 locazioni nella hashtable, una per ogni tipo di evento, utilizzando come funzione di hashing: $\log_2(tipo)$

Al fine di registrare un callback per un certo tipo di evento si utilizza la funzione:

```
event_subscribe(game, event_type, handler)
```

dove *handler* è un puntatore ad una funzione di callback del tipo:

```
void f(game_t * game, event_t event)
```

Un evento può, invece, essere generato chiamando la funzione:

```
event_raise(game, event_type, object)
```

dove *object* può puntare a dei dati da passare al callback e accessibile tramite `event.object`.

Livello (game/level.c)

La struttura `level_s` raccoglie le informazioni che riguardano un livello:

name nome del livello, univoco

maps lista delle mappe che compongono il livello, ordinate in ordine di comparizione

textures textures di mura e corridoi

animations animazioni utilizzate come transizione tra le mappe

enemies lista dei personaggi avversari del livello

complexity livello di complessità, utilizzato nella generazione delle mappe random, più è alto più le mappe saranno complicate

Le informazioni di ogni livello sono contenute nel rispettivo file di configurazione contenuto nella cartella specificata dalla proprietà `levels_path` del file di configurazione principale.

Mappa (game/map.c)

In fase di caricamento di un livello, sono caricate e/o generate le mappe che lo compongono. Ogni mappa è descritta dalla struttura `map_s` composta dalle proprietà:

size dimensione, in celle, della mappa

offset offset, in pixel, rispetto all'origine al quale disegnare la mappa. In questo modo la mappa è disegnata sempre al centro dello schermo

start punto di ingresso

end punto di uscita

grid griglia delle celle, rappresentata da un array `[size.height][size.width]`

powerup_probability probabilità che una cella della mappa possa contenere un bonus

powerups_time tempo di attesa minimo, in secondi, tra la comparsa di due bonus

powerup_cells lista delle celle nelle quali può comparire un bonus

powerup_time istante dell'ultimo inserimento di un bonus

powerups_limit numero massimo di bonus da mostrare contemporaneamente

next prossima mappa

Una mappa può sia essere caricata da un file di configurazione (contenente la rappresentazione testuale della struttura della mappa) sia generata automaticamente creando un labirinto perfetto o meno.

Cella (game/cell.c)

Una mappa è composta da una griglia di celle costituente un grafo dove ogni cella può essere connessa a quattro adiacenti (nord, sud, est, ovest).

Una cella è descritta tramite la struttura `cell_s`:

type tipo di cella, tra:

- `CELL_TYPE_UNKNOWN`
- `CELL_TYPE_PATH`
- `CELL_TYPE_WALL`

location coordinate della cella nella griglia

powerup puntatore al bonus contenuto nella cella, NULL in caso non contenga bonus

struct **adjacency** struttura che rappresenta la lista di adiacenza di un nodo, se presente contiene un puntatore ad ognuna delle 4 celle confinanti:

north cella di coordinate $(x, y - 1)$

south cella di coordinate $(x, y + 1)$

west cella di coordinate $(x - 1, y)$

east cella di coordinate $(x + 1, y)$

struct **node** informazioni della cella vista come nodo di un grafo:

parent nodo genitore

color colore del nodo, in fase di visita

distance distanza dal nodo sorgente della visita

value valore (peso) del nodo, utilizzato per rappresentare celle più veloci/lente (1: lento, 9: veloce)

Personaggio (game/character.c)

La struttura `character_s` definisce un personaggio sia esso controllato dall'utente o dagli avversari. Le informazioni contenute nella struttura sono:

is_user se il personaggio è dell'utente o meno

tiles bitmap che rappresentano il personaggio nel corso dei suoi movimenti (in base a direzione e stato dell'animazione)

ratio percentuale di spazio percorso tra due celle

animations animazioni del personaggio: passi

location posizione sulla griglia

position posizione, in pixel, sullo schermo

last_position ultima posizione, in pixel

map mappa in cui si trova

direction direzione che sta tenendo, tra:

- `DIRECTION_NONE`

- `DIRECTION_NORTH`
- `DIRECTION_EAST`
- `DIRECTION_SOUTH`
- `DIRECTION_WEST`

next_direction direzione verso la quale il personaggio andrà girato appena possibile

path stack contenente i nodi del percorso da seguire nel caso il personaggio non fosse dell'utente. In caso contrario può contenere i nodi del percorso minimo fino all'uscita quando è necessario mostrarlo

path_color colore con il quale evidenziare il percorso sulla mappa

powerups lista dei bonus in possesso del personaggio

lives vite del personaggio, se controllato dall'utente

methods.control puntatore alla funzione utilizzata per controllare i movimenti del personaggio

config hashtable contenente le proprietà variabili del personaggio

default_config copia della hashtable *config* utilizzata per ripristinare i valori di default

Bonus (game/powerup.c)

I bonus che i personaggi possono raccogliere nel corso del gioco sono descritti dalla struttura `powerup_s`:

tile immagine che rappresenta il bonus

name nome del bonus, univoco

appearance_probability probabilità di apparizione

effects_rect_size dimensione del riquadro, centrato nella posizione del personaggio, nel quale si riscontrano gli effetti del bonus

duration durata, in secondi, degli effetti del bonus

trigger tasto utilizzato per attivare il bonus ($A - Z$)

limit numero massimo di bonus di questo tipo da mostrare

placed numero di volte che il bonus è stato posizionato sulla mappa

picker proprietà da applicare al personaggio che raccoglie il bonus

characters proprietà da applicare a tutti gli altri personaggi

Nel momento in cui un bonus è raccolto da un personaggio, viene allocata una nuova struttura `powerup_status_s` che viene inserita nella lista `character->powerups`.

`powerup_status_s`

powerup puntatore al bonus raccolto

enabled stato del bonus, se attivo o meno

counter numero di volte che il bonus è stato raccolto

elapsed tempo trascorso, in secondi, dall'utilizzo del bonus

character puntatore al personaggio che ha raccolto il bonus

Quando gli effetti di un bonus terminano, la struttura `powerup_status_s` associata viene rimossa dalla lista e deallocata.

3 Algoritmi

Personaggi

Collisioni tra personaggi

La strategia di controllo delle collisioni tra personaggi è molto semplice.

Quando un personaggio P_1 si muove:

Algoritmo principale

$\hookrightarrow \forall P_n \in \text{Personaggi} - \{P_1\}$

- \hookrightarrow Se P_1 e P_n non sono nella stessa mappa, si passa al prossimo personaggio
- \hookrightarrow Se P_1 e P_n ignorano le collisioni, si passa al prossimo personaggio
 - Si calcolano i rettangoli R_1 e R_n che i personaggi occupano sulla mappa. Se $\text{Intersezione}(R_1, R_2) \geq 0.6$ della loro area, si è verificata una collisione
- \hookrightarrow Se non si è verificata una collisione, si passa al prossimo personaggio
 - Se P_1 o P_n è un personaggio dell'utente, il personaggio perde una vita
 - P_1 e P_2 , se possono subire gli effetti delle collisioni, sono spostati in due punti (corridoi) casuali della mappa

Intersezione tra rettangoli

- Si calcola la somma delle aree dei rettangoli A e B

$$\text{Area}_{A+B} = (B_A \times H_A) + (B_B \times H_B)$$

dove B_A è la base di A e H_A l'altezza.

- Si calcola l'area di intersezione

$$\begin{aligned} B_I &= \max\{0, \min\{X_A + B_A, X_B + B_B\} - \max\{X_A, X_B\}\} \\ H_I &= \max\{0, \min\{Y_A + H_A, Y_B + H_B\} - \max\{Y_A, Y_B\}\} \\ \text{Area}_I &= B_I \times H_I \end{aligned}$$

dove X_A e Y_A sono le coordinate di origine di un rettangolo.

- A questo punto il livello di intersezione è dato dal rapporto:

$$\frac{\text{Area}_I}{\text{Area}_{A+B} - \text{Area}_I}$$

Comportamento degli avversari

Ad ogni avversario, sono associati 2 riquadri centrati nella sua posizione attuale: un riquadro di inseguimento e un riquadro di ricerca dell'uscita. Possono verificarsi 4 casi:

1. **L'uscita è nel riquadro di ricerca dell'uscita e il personaggio dell'utente non è nel riquadro di inseguimento** \rightarrow Il personaggio calcola, utilizzando l'algoritmo di ricerca del percorso minimo assegnato-gli, il percorso tra la sua posizione e la cella di uscita e comincia a seguirlo.

2. **Il personaggio dell'utente è nel riquadro di inseguimento e l'uscita non è nel riquadro di ricerca dell'uscita** → Il personaggio calcola, utilizzando l'algoritmo di ricerca del percorso minimo assegnatogli, il percorso tra la sua posizione e la posizione attuale dell'utente e comincia a seguirlo.
3. **L'uscita è nel riquadro di ricerca dell'uscita e il personaggio dell'utente è nel riquadro di inseguimento** → In base al valore della proprietà `chase_user` dell'avversario questo decide se dare preferenza all'inseguimento dell'utente o meno.
4. **Nessuno dei casi precedenti** → L'avversario segue il percorso minimo dalla sua posizione corrente alla locazione di tipo corridoio corrispondente (più vicina) all'angolo del riquadro più vicino all'uscita.

Strategie di inseguimento

Inseguimento classico Il metodo di inseguimento più semplice consiste nel seguire il percorso più breve tra la cella dell'inseguitore e la cella dell'inseguito.

Predizione della posizione Il personaggio inseguitore si basa sulla direzione e sulla posizione dell'inseguito per calcolare la cella più lontana che questo può raggiungere tenendo quella direzione. Trovata la cella, l'inseguitore segue il percorso dalla sua posizione alla cella *predetta*.

Imboscata Simula una sorta di basilare collaborazione tra due avversari.

1. Si considera una cella P a distanza 2 dalla cella dell'inseguito U
2. Si traccia un ipotetico vettore tra il personaggio inseguitore C_1
3. Si raddoppia la lunghezza del vettore
4. Il personaggio inseguitore C_2 segue il percorso fino alla cella raggiunta dal vettore

Quindi, l'inseguitore C_1 raggiungerà la cella calcolata dalla sua strategia. C_2 , invece, la cella alle coordinate:

$$C_2.x = 2 \times P.x - C_1.x$$

$$C_2.y = 2 \times P.y - C_1.y$$

se di tipo corridoio, in caso contrario la cella di tipo corridoio più vicina a C_2 .

Mappe

Conversione di una mappa dalla rappresentazione testuale

Algoritmo

- | | |
|-------------------------|---|
| Inizializzazione | <ul style="list-style-type: none"> • Si inizializza un parser per la lettura della stringa rappresentante la mappa • Si inizializza una mappa vuota della dimensione specificata nel file di configurazione • Si inizializza una variabile <i>point</i> = (0, 0) • Si avvia il parsing |
| Parsing | <ul style="list-style-type: none"> ○ Finché non si è arrivati alla fine della stringa, per ogni carattere c ci si riferisce alla cella alla posizione <i>point</i> <ul style="list-style-type: none"> – Se $c \in [0, 9]$, la cella è un percorso con costo $Integer(c)$ – Se $c = spazio$, la cella è un percorso con costo 5 – Se $c = P$, la cella è un percorso con costo 5 e destinato ad un bonus |

- Se $c = V$, la cella è un percorso con costo 5 e destinato ad un avversario
- Se $c = S$, la cella è un percorso con costo 5 e punto di ingresso del labirinto
- Se $c = E$, la cella è un percorso con costo 5 e punto di uscita del labirinto
- Se $c = \#$, la cella è un muro
- Se $c = \text{newline}$, $\text{point}.x = 0$, $\text{point}.y = \text{point}.y + 1$
- Se $c \neq \text{newline}$, $\text{point}.x = 0 = \text{point}.x + 1$

- A questo punto si passa alla fase di connessione delle celle di tipo corridoio

Connessione

○ $\forall \text{ cella} \in \text{mappa}$

↪ Se *cella* non è di tipo percorso non fare nulla

- Se $\text{cella}.x > 0$ e la cella alle coordinate $\text{cella}_1 = (\text{cella}.x - 1, \text{cella}.y)$ è una cella di tipo corridoio, si connettono le due celle:

$$\text{East}(\text{cella}_1) = \text{cella}$$

$$\text{West}(\text{cella}) = \text{cella}_1$$

- Se $\text{cella}.y > 0$ e la cella alle coordinate $\text{cella}_1 = (\text{cella}.x, \text{cella}.y - 1)$ è una cella di tipo corridoio, si connettono le due celle:

$$\text{North}(\text{cella}_1) = \text{cella}$$

$$\text{South}(\text{cella}) = \text{cella}_1$$

Generazione di un labirinto perfetto

Labirinto perfetto Una labirinto è detto *perfetto* se, preso un qualunque punto (di tipo corridoio), esiste uno e un solo percorso da quel punto ad ogni altro punto (di tipo corridoio) del labirinto.

Algoritmo**Inizializzazione**

- Il labirinto è riempito di celle di tipo muro
- Si stabilisce una cella iniziale $(1, 1)$ e si inserisce in una coda Q

Costruzione

○ Finché $Q \neq \emptyset$

- Sia *cell* la cella in testa alla coda
- Siano (x, y) le coordinate di *cell*, si controllano le celle alle coordinate:

$$\begin{array}{ccccc}
 & & (\mathbf{x}, \mathbf{y} - \mathbf{2}) & & \\
 & & (x, y - 1) & & \\
 (\mathbf{x} - \mathbf{2}, \mathbf{y}) & (x - 1, y) & \boxed{\text{cell}} & (x + 1, y) & (\mathbf{x} + \mathbf{2}, \mathbf{y}) \\
 & & (x, y + 1) & & \\
 & & (\mathbf{x}, \mathbf{y} + \mathbf{2}) & &
 \end{array}$$

e si conservano in un array le coppie di coordinate per le quali le celle alle coordinate indicate in grassetto sono di tipo **muro**

- ↪ Se nessuna delle 4 celle soddisfa la condizione, la cella è un vicolo cieco. Quindi si estrae dalla coda

- ↪ In caso contrario, si sceglie casualmente una delle coppie di celle conservate, si costruisce un percorso in quelle celle e si inserisce la cella della coppia che confina con *cell* nella coda *Q*

Generazione di un labirinto *intrecciato*

Labirinto *intracciato* Una labirinto è detto *intracciato* se, preso un qualunque punto (di tipo corridoio), esiste più di un percorso da quel punto ad un altro punto (di tipo corridoio) del labirinto.

Algoritmo

Parameteri	float <i>deadends_probability</i> probabilità da 0.0 a 1.0 che un vicolo cieco venga eliminato.
Inizializzazione	• Si genera un labirinto perfetto conservando i nodi alla fine di un vicolo cieco in una coda <i>deadends</i>
Costruzione	<ul style="list-style-type: none"> ⌚ Finché <i>deadends</i> $\neq \emptyset$ <ul style="list-style-type: none"> – Si estrae dalla testa della coda una cella <i>cell</i> ↪ Se la probabilità è sfavorevole si passa al prossimo ciclo ↪ In caso contrario, si cercano le celle di tipo muro nelle adiacenze di <i>cell</i> – Presa casualmente una delle celle di tipo muro, se ne cambia il tipo e la si trasforma in una cella corridoio

Posizionamento degli avversari e dei bonus

Avversari Nel caso in cui la mappa sia generata automaticamente, lo sono anche le posizioni degli avversari. Siano *P* il personaggio, *L_P* la lista dei personaggi e δ la funzione distanza, ogni avversario è posizionato in una cella casuale tale che:

$$\delta(P, \text{centro}_{\text{mappa}}) \leq |L_P| + 3$$

Bonus Le celle destinate ai bonus sono assegnate in fase di costruzione della mappa tenendo conto della proprietà *powerup_probability* della mappa. Per ogni cella, si controlla se la probabilità è favorevole e, nel caso, si contrassegna come cella per bonus.

Calcolo della velocità di un personaggio in base alla cella

La velocità di un personaggio è rappresentata dal numero di pixel dei quali questo si sposta ad ogni loop.

- Sia *base_speed* la velocità di base dell'utente (come da file di configurazione) e *cell_value* il peso della cella.

$$\hookrightarrow \text{Se } cell_value = peso_di_default \rightarrow speed = base_speed$$

$$\hookrightarrow \text{Se } cell_value < peso_di_default \rightarrow speed = base_speed + (peso_di_default - cell_value) \times 1.5$$

$$\hookrightarrow \text{Se } cell_value > peso_di_default \rightarrow speed = \frac{base_speed}{(cell_value - peso_di_default) \times 1.5}$$