**CROWD SIMULATOR**

Documentazione

Sommario

[1 Introduzione 3](#_Toc124584257)

[2 Crowd Simulator: esecuzione, funzionamento e descrizione 4](#_Toc124584258)

[2.1 Il primo avvio 4](#_Toc124584259)

[2.2 Classi e framework non nativi 4](#_Toc124584260)

[2.3 Specifiche del simulatore 4](#_Toc124584261)

[2.3.1 Specifiche: edificio ed entità statiche 5](#_Toc124584262)

[2.3.2 Specifiche: Folla, gruppi e pedoni 5](#_Toc124584263)

[2.4 *Swing*: Scelta del framework e peculiarità 6](#_Toc124584264)

[2.4.1 Specifiche: Funzionalità dell’interfaccia 8](#_Toc124584265)

[3 Organizzazione delle classi 11](#_Toc124584266)

[3.1 Struttura logica di *Crowd Simulator* 11](#_Toc124584267)

[3.2 Package Models: le entità 11](#_Toc124584268)

[3.3 Problemi di compatibilità con l’algoritmo di movimento innovativo 12](#_Toc124584269)

[3.4 Package *src*: l’interfaccia 12](#_Toc124584270)

[3.5 Raccolta dati a fine simulazione 13](#_Toc124584271)

[3.5.1 Funzionamento e riuso 13](#_Toc124584272)

[4 L’algoritmo di movimento innovativo 15](#_Toc124584273)

[4.1 Introduzione agli algoritmi di movimento 15](#_Toc124584274)

[4.2 Il flocking 15](#_Toc124584275)

[4.3 Funzionamento dell’algoritmo innovativo 16](#_Toc124584276)

# Introduzione

La simulazione del movimento delle folle di pedoni è un argomento che ha destato l'interesse di diversi campi di ricerca. Negli ultimi anni, sono state sviluppate molte tecniche per la rappresentazione verosimile dei movimenti delle folle, sono stati prodotti software di simulazione grafica, sistemi di validazione e valutazione di tali software, nonché indagini e libri sugli approcci e sull’evoluzione di questo ambito di studi. Con il passare del tempo, tale ricerca ha trovato applicazione in svariati campi, come la computer grafica, videogiochi, fisica statistica, robotica, ma anche realtà virtuale, scienze sociali e dinamiche di evacuazione in situazioni di emergenza. Il presente documento propone un resoconto su funzionamento e scopi di *Crowd Simulator*, una GUI (Graphic User Interface) che consente la simulazione di una folla di pedoni composta da singoli individui organizzati in gruppi autonomi, ognuno dei quali ha una serie di obiettivi da raggiungere all’interno dell’ambiente virtuale prima di poterlo abbandonare. La versione del programma presentata in questo documento è una *beta* e contiene al suo interno un solo algoritmo di movimento dei pedoni, sebbene nasca con l’intento di contenerne due, tra i quali sia possibile scegliere in fase di inserimento dei parametri della simulazione. Il primo algoritmo, quello attualmente presente nel simulatore, è una versione rivisitata del noto sistema di *flocking* di Craig Reynolds; il secondo è un algoritmo innovativo sviluppato in parallelo al simulatore. Lo scopo del simulatore è la raccolta di dati inerenti allo spostamento dei pedoni rappresentati. A prescindere dall’algoritmo di movimento utilizzato, a fine simulazione il programma genera un file in sintassi JSON (JavaScript Object Notation) contenente tutte le informazioni necessarie alla ricostruzione dello spostamento dei pedoni da inizio a fine simulazione, nonché quelle necessarie alla rappresentazione degli altri elementi presenti nell’ambiente virtuale, come ostacoli o luoghi di interesse. Grazie alle informazioni che il simulatore fornisce si possono effettuare dei test comparativi o qualitativi, eventualmente utilizzando un sistema o un framework di convalida e valutazione dell’accuratezza dei movimenti.

Il presente elaborato si divide in tre capitoli, all’interno del primo si danno informazioni sul primo avvio del programma e sui package esterni utilizzati, per poi fornire le informazioni necessarie alla comprensione delle mansioni del simulatore. Il secondo capitolo descrive in maniera sommaria la logica dietro alle classi che costituiscono l’interfaccia del programma (con qualche cenno anche alle funzionalità), eventuali dettagli sono presenti all’interno dei commenti del codice. Infine, il terzo capitolo contiene una breve descrizione dell’algoritmo di movimento innovativo, anche in questo caso, per maggiori dettagli si rimanda alla tesi di dottorato in cui questo viene descritto: <https://hdl.handle.net/11584/312918>.

# Crowd Simulator: esecuzione, funzionamento e descrizione

In questo capitolo si descrive come eseguire il programma per la prima volta, alcune caratteristiche tecniche rilevanti, quali compilatore, versione di Java e framework utilizzati. In ultimo, si dà una panoramica dell’attuale funzionamento del simulatore (la versione aggiornata al 10/12/2022), includendo sia le specifiche che il programma deve necessariamente implementare, sia il modo con cui vengono effettivamente implementate nell’attuale versione beta del simulatore.

## Il primo avvio

Il software è stato costruito su sistema operativo Windows 11, utilizzando *Java* 18, compilatore *Javac* e l’IDE *IntelliJ*. Per la sua prima esecuzione è sufficiente compilare ed eseguire la classe *Main.java* (contenuta nella cartella *src*). Una volta avviato il programma si presenta come interfaccia su un’unica pagina, per l’avvio della simulazione è necessario inserire i parametri della simulazione nella sezione sinistra della GUI, poi premere il tasto *confirm* (in basso) e poi il tasto *play* (in alto).

## Classi e framework non nativi

Il programma è scritto in linguaggio Java, ma utilizza una serie di elementi esterni, tra cui:

* Il framework Swing, sul quale si basa l’intera applicazione. Viene utilizzato per la creazione dell’interfaccia grafica sotto ogni suo aspetto, sia visivo che logico-funzionale.
* Il framework contenuto in *external libraries*\*core-3.3.7.jar*, fondamentale per la gestione del movimento dei pedoni (solo quella dell’algoritmo provvisorio). In particolare, si usa la sua classe *PVector*, che rappresenta vettori fisici (con modulo, verso e direzione). Documentazione: <https://processing.org/reference/PVector.html>
* Le classi contenute in *external libraries\gson-2.9.0.jar*, in particolare la libreria GSON, utilizzata per creare il file in sintassi JSON contenente tutte le informazioni sull’ambiente di simulazione (posizione di entità statiche e spostamento dei pedoni). Documentazione: <https://github.com/google/gson>.
* Le classi *RangeSlider.java* e *RangeSliderUI.java*, contenute in *src\support*, le quali sovrascrivono il codice sorgente dell’oggetto *JSlider* di *Swing*, così da creare uno slider doppio, utilizzabile per indicare un range piuttosto che un valore singolo (come avviene nello slider classico). Documentazione: <https://github.com/ernieyu/Swing-range-slider>.

## Specifiche del simulatore

In breve, il programma *Crowd Simulator* propone un’interfaccia divisa in sezioni tramite le quali deve essere possibile:

1. Chiedere e permettere all’utente di inserire i parametri della simulazione quali la grandezza della folla, il numero di ostacoli, il numero di punti di interesse e un intervallo all’interno del quale siano definite le grandezze dei singoli gruppi di pedoni.
2. Visualizzare la simulazione animata e tutte le sue entità.
3. Mettere in pausa, avviare e stoppare la simulazione corrente.
4. Visualizzare i dati in tempo reale delle singole entità presenti nella simulazione.

Nell’ambiente di simulazione è **sempre presente** la **folla**, un **edificio** e delle entità statiche: **ostacoli** e **obiettivi**. I pedoni attraversano l’edificio e interagiscono con gli elementi in esso contenuti, per poi abbandonarlo.

### Specifiche: edificio ed entità statiche

La struttura dell’edificio nella versione beta del programma è realizzata prendendo come riferimento le dimensioni dello schermo sul quale viene visualizzato, così come mostrato in Fig. 1.1. La struttura dell’edificio non deve essere necessariamente quella descritta in figura, in quanto l’algoritmo deve essere in grado di generalizzare su ogni tipo di mappa, tuttavia è necessario che l’edificio sia composto da un’entrata, un’uscita e una serie di stanze.

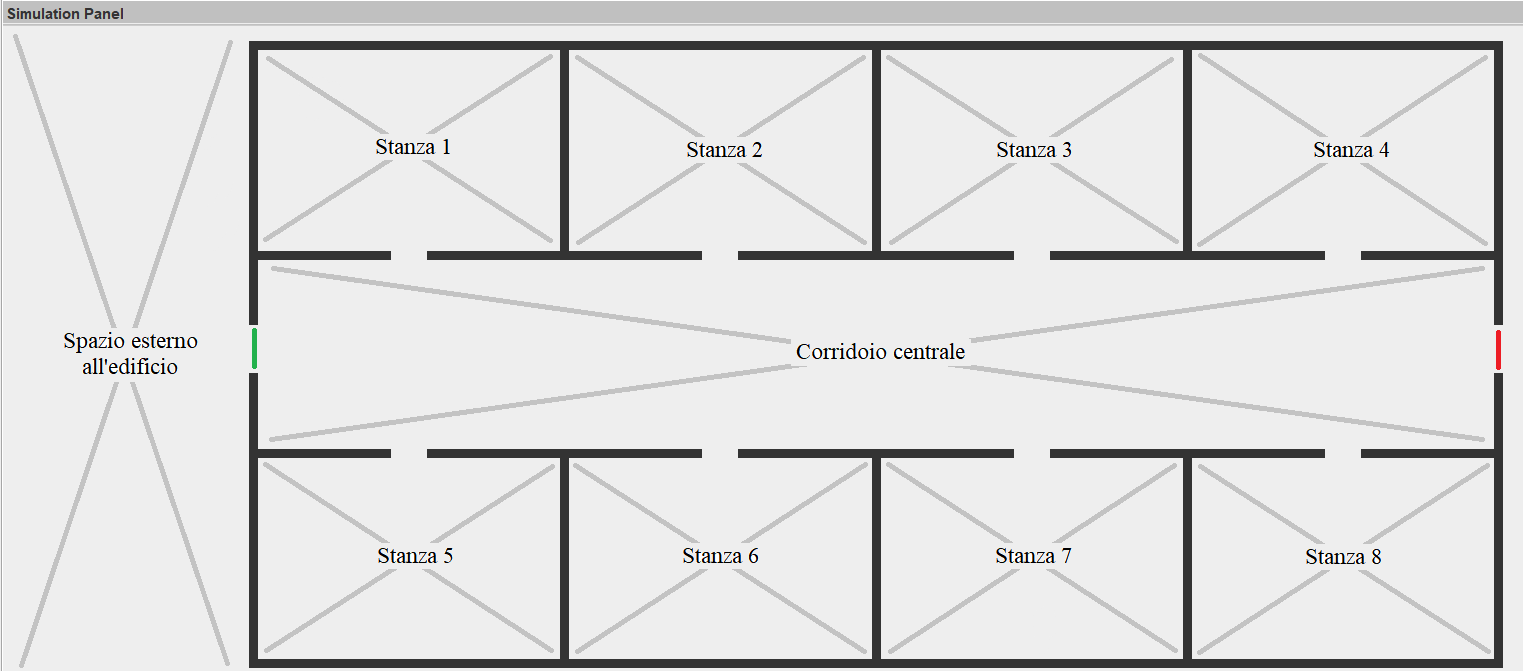


Fig. 2.1: la struttura in nero costituisce l’edificio, in verde la sua porta di entrata e in rosso la sua porta di uscita

All’interno dell’edificio sono collocate due tipologie di entità statiche: gli ostacoli, ovvero gli oggetti che il pedone deve evitare ogni qualvolta vengano incrociati lungo il cammino, in Fig. 2.2 sono rappresentate da quadrati e cerchi neri; gli obiettivi (o punti di interesse), che costituiscono luoghi di interesse e di passaggio per i gruppi di pedoni, che vi effettuano la sosta per un intervallo di tempo arbitrario, per poi andare via e passare al prossimo obiettivo, fino a non aver terminato i punti di interesse da visitare, a quel punto il gruppo può recarsi verso l’uscita dell’edificio e lasciare l’ambiente di simulazione. Gli obiettivi, a loro volta, si distinguono in due tipologie:

* **Obiettivi generici**: non hanno nessuna proprietà aggiuntiva e nella versione beta del simulatore sono rappresentati da un quadratino rosso.
* **Punti di ristoro**, sono il luogo in cui i pedoni possono recuperare l’energia spesa durante l’attraversamento dell’edificio e nella versione beta del simulatore sono rappresentati da un quadratino blu.

Le entità statiche (sia ostacoli che obiettivi) hanno un ID identificativo per renderne più semplice l’identificazione all’interno del pannello delle entità attive[[1]](#footnote-1) e possono essere situate in qualsiasi punto interno all’edificio, ma non in prossimità delle porte, così come mostrato in Fig. 2.2.

### Specifiche: Folla, gruppi e pedoni

La folla è eterogenea: è composta da un insieme di pedoni con caratteristiche personali e differenti da quelle delle altre entità. Ogni pedone ha un gruppo di appartenenza, una lista di obiettivi da raggiungere prima di uscire dall’edificio, un sesso (maschio o femmina), un’età (bambino, giovane, anziano), un certo livello di energia, una velocità massima.[[2]](#footnote-2) Ogni pedone si sposta lungo l’edificio raggiungendo gli obiettivi assieme al suo gruppo, con il quale deve rimanere coeso, senza però collidere con altri membri. Mentre i pedoni camminano o visitano un obiettivo generico (quadratino rosso in figura) consumano energia a intervalli regolari. Una volta che un pedone del gruppo scende sotto una certa soglia di energia, tutto il suo gruppo si dirige presso un punto di ristoro, dove sosta fino a quando ogni componente non ha riacquistato tutta l’energia spesa.

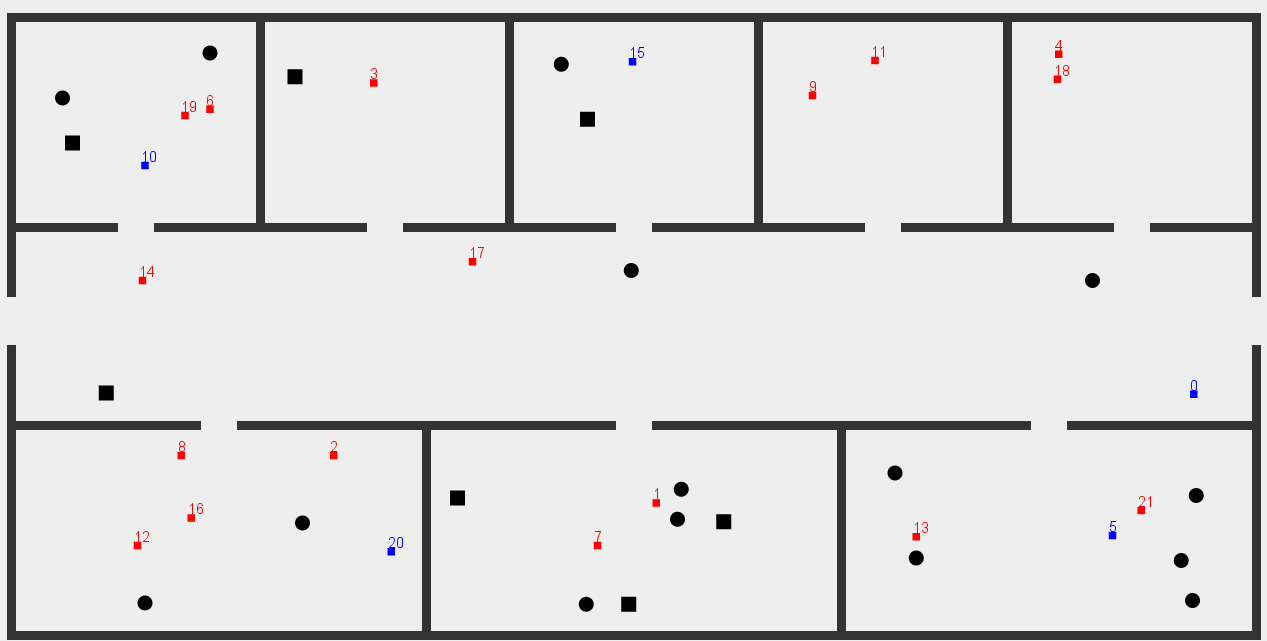


Fig. 2.2: Edificio contenente ostacoli e obiettivi. In nero gli ostacoli, in rosso gli obiettivi generici e in

blu i punti di ristoro

Ogni gruppo ha una grandezza casuale all’interno di un determinato intervallo stabilito dall’utente tramite l’interfaccia della GUI, i componenti del gruppo una volta disposti al di fuori dell’edificio inizieranno a muoversi solamente dopo un certo intervallo di tempo, al termine del quale potranno cominciare la loro ricerca dei punti di interesse.[[3]](#footnote-3) La lista di obiettivi deve essere la stessa per ogni membro del gruppo e ogni gruppo deve avere la propria lista di obiettivi, indipendente da quella degli altri.

## *Swing*: Scelta del framework e peculiarità

*Crowd Simulator* è stato codificato utilizzando il linguaggio Java e il framework *Swing*. Sulla base delle specifiche del programma da implementare, il linguaggio e il framework sono stati scelti sulla base del concetto di semplicità. La scelta del linguaggio è stata influenzata principalmente dalla familiarità con lo stesso e dalla semplicità di fruizione e utilizzo del framework, *Swing* e Java infatti possiedono una buona documentazione ufficiale, completa di esempi di codice eseguibile, casi d’uso, analisi delle criticità e descrizione dei problemi comuni.

*Swing* è un framework orientato allo sviluppo di interfacce grafiche. Fa parte delle JFC (Java Foundation Classes)[[4]](#footnote-4) e si sviluppa sopra il precedente AWT (Abstract Windowing Toolkit)[[5]](#footnote-5), estendendone i componenti e rendendoli più potenti e flessibili. I programmi scritti utilizzando *Swing* sono completamente indipendenti dalla piattaforma su cui vengono eseguiti e si basano sul paradigma MVC (Model View Controller), che separa la logica di presentazione dei dati dalla logica di gestione degli stessi. I componenti di *Swing*, inoltre, si snodano seguendo una precisa gerarchia, così come mostrato in Fig. 2.3: gerarchia dell'API Java Swing., che ha come biforcazione principale la divisione tra *Container* e *JComponent.*

Un *Container* è un componente che ha l’importante proprietà di poter racchiudere altri elementi al suo interno (che a loro volta possono essere *Container* o *JComponent*). In particolare, in *Crowd Simulator* si è fatto utilizzo di un unico *Frame* e di molteplici *Panel*, i 3 fondamentali sono quelli che costituiscono le tre sezioni dell’interfaccia descritte nel paragrafo 162.3.

Un *JComponent* invece è un qualunque altro widget del framework. Questo insieme include pulsanti (*JButton*), etichette e stringhe (*JLabel*), menù di diverso tipo (*JMenu*) e, in generale, tutti quegli elementi dell’interfaccia con cui è possibile interagire allo scopo di ricevere una risposta da parte del sistema.

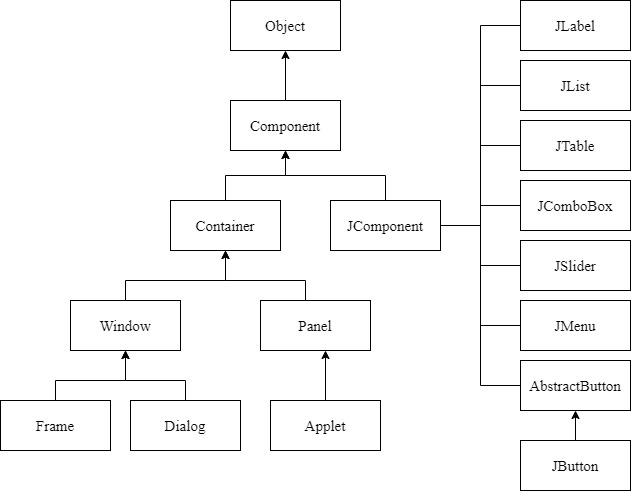


Fig. 2.3: gerarchia dell'API Java Swing.

(Fonte: <https://www.javatpoint.com/java-swing>)

Le componenti appena descritte costituiscono gli elementi dell’interfaccia grafica con la quale è possibile interagire, tuttavia, la simulazione animata vera e propria fa uso della già citata libreria AWT, in particolare della sua classe java.awt.Graphics2D, la quale fornisce la possibilità di disegnare immagini, forme o stringhe di testo sui pannelli di *Swing*, nonché di gestirne il colore, eventuali trasformazioni, o spostamenti. Il processo di animazione è coadiuvato da un Timer ciclico fornito da *Swing,* al termine del quale vengono cancellati tutti gli elementi della schermata di simulazione e vengono ridisegnati con le posizioni aggiornate, dando così l’illusione del movimento.

### Specifiche: Funzionalità dell’interfaccia

La GUI nella versione beta è interamente contenuta all’interno di un *JFrame*, che a sua volta si compone di tre *JPanel* all’interno dei quali sono situate le tre sezioni principali dell’interfaccia. Il primo di questi pannelli è quello che mostra il già descritto ambiente di simulazione, mentre quelli adiacenti contengono tutte le funzionalità dell’interfaccia.

Il secondo pannello è denominato *Settings Menu* e permette di impostare i parametri della simulazione, come la grandezza della folla, il numero di ostacoli e di obiettivi, il numero di pedoni appartenenti a ogni gruppo. Tramite questa sezione è anche possibile dare il via alla simulazione, metterla in pausa e bloccarla, resettando i parametri di quella corrente, così da poterne avviare una nuova. All’avvio del programma il pannello in questione si presenta come in Fig. 2.4, per avviare la simulazione l’utente deve inserire i parametri negli appositi campi, cliccare sul pulsante “conferma” e poi sul pulsante “play”, ossia quello centrale nella parte alta dello schermo. Il valore di ogni parametro inserito deve essere maggiore di 0, in caso contrario compare una finestra di avviso che avvisa l’utente dell’errore commesso (quest’ultima clausola vieta, ad esempio, di eseguire una simulazione in cui non ci siano ostacoli, dunque può essere rivista all’occorrenza). Una volta che la simulazione è avviata, verranno sbloccati anche i pulsanti “pause” (a sinistra di “play”) e “stop” (a destra di “play”), che permettono rispettivamente di sospendere e di terminare la simulazione in corso.

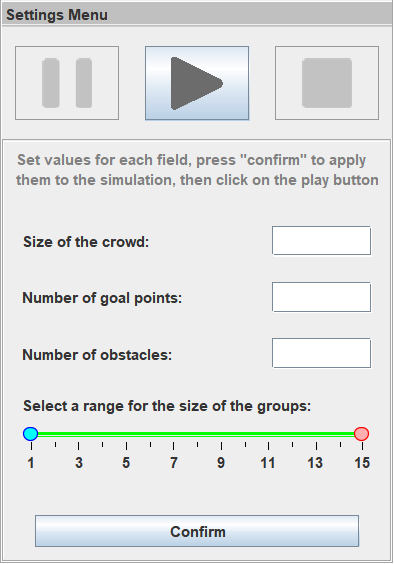


Fig. 2.4: Aspetto del pannello Settings Menu prima dell'inizio della simulazione

Infine, il terzo dei tre pannelli è diviso a sua volta in tre *tab*, ossia in tre pannelli sovrapposti ed intercambiabili, il primo dei quali mostra le informazioni relative ai pedoni, il secondo contiene le informazioni relative agli ostacoli e il terzo quelle relative agli obiettivi. In Fig. 2.5 viene mostrato l’aspetto di tale pannello prima dell’inizio della simulazione, le opzioni di ordinamento e filtraggio sono disattivate e viene mostrato un messaggio che indica che non ci sono pedoni da mostrare. Prima dell’inizio della simulazione i due pannelli che riportano informazioni su ostacoli e obiettivi contengono solamente la dicitura “There are no pedestrians to show”, in essi non vi è la possibilità di applicare funzioni di ordinamento o filtraggio dei risultati.

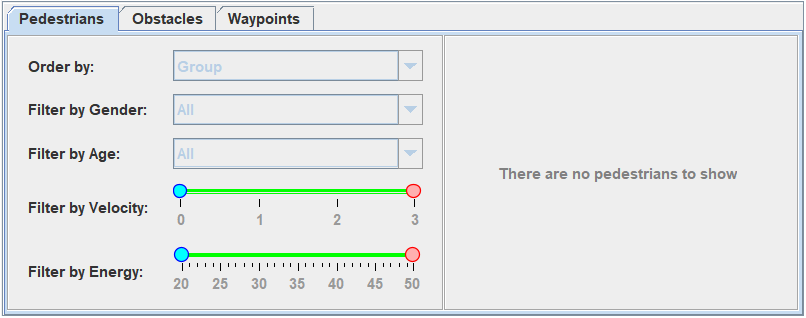


Fig. 2.5: pannello delle entità attive prima dell'inizio della simulazione

Una volta avviata la simulazione, le opzioni di filtraggio e ordinamento vengono sbloccate e i pedoni sono mostrati suddivisi per gruppo di appartenenza, come appare come in Fig. 2.6.

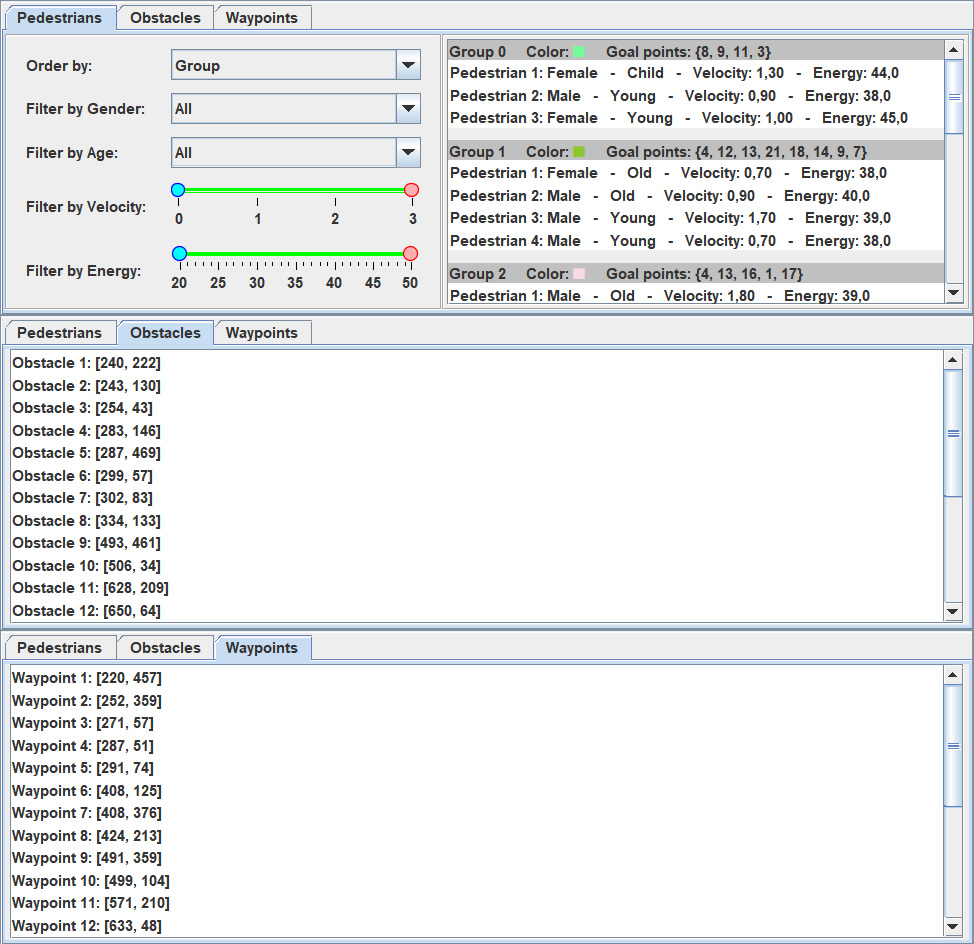


Fig. 2.6: aspetto del pannello dei pedoni attivi dopo aver avviato la simulazione

Diventa possibile vedere gli obiettivi del gruppo, ordinare e filtrare i pedoni in base ai loro parametri principali (sesso, età, velocità ed energia). Inoltre, il pannello si aggiorna man mano che i dati della simulazione variano, mostrando il consumo e il recupero dell’energia dei pedoni, così come l’obiettivo corrente e quelli non ancora raggiunti. Alle entità statiche non sono associate funzioni di ordinamento o filtraggio, ma solamente un ID univoco che viene riportato sia sul pannello delle entità attive sia nella schermata che mostra l’ambiente di simulazione, come già mostrato in Fig. 2.2. Nella suddivisione di pedoni per gruppi mostrata in Fig 2.6, a ognuno di loro viene associato un colore, così da renderne più facilmente individuabili i membri all’interno dell’ambiente di simulazione. Inoltre, per ogni gruppo si mostrano gli obiettivi che restano ancora da raggiungere, inserendone gli identificativi tra parentesi graffe e man mano che i pedoni visitano gli obiettivi questi ultimi vengono rimossi dalla lista.

Dopo aver avviato la simulazione i due pannelli delle entità attive rimanenti racchiudono le informazioni che riguardano ostacoli e obiettivi. Essendo elementi statici se ne riporta solamente la posizione e il numero identificativo, così come mostrato in Fig. 2.7.

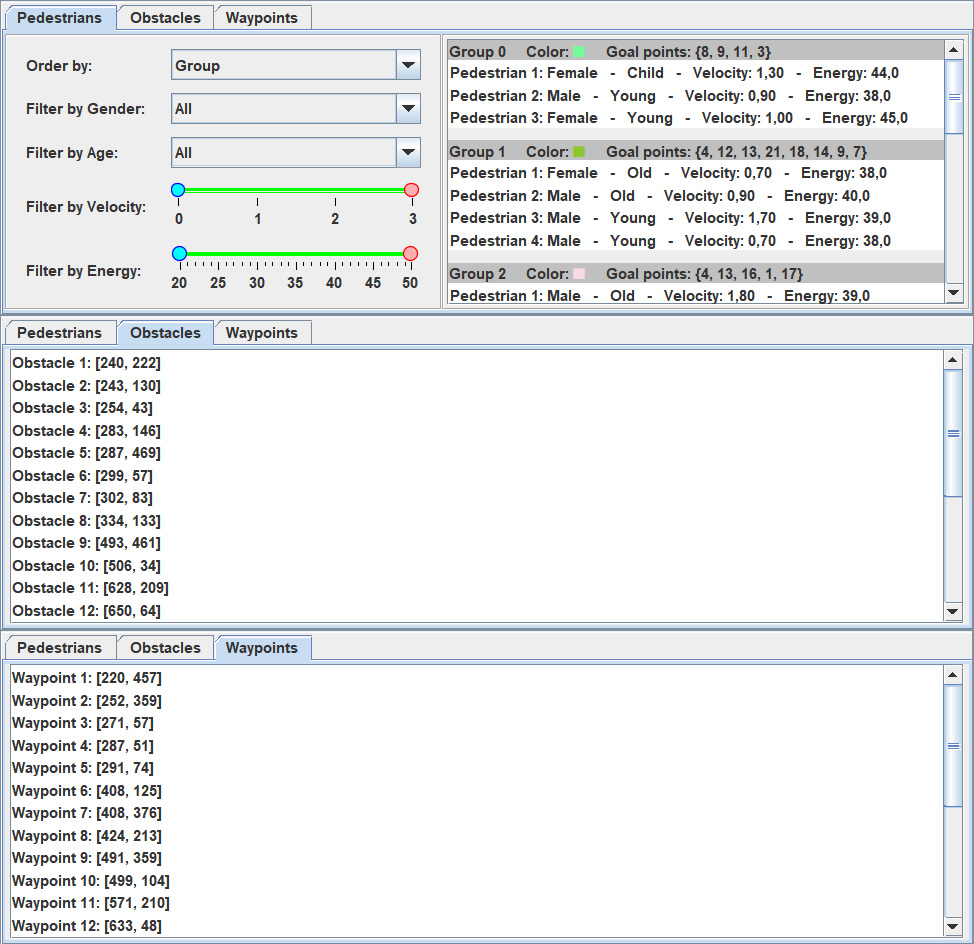


Fig. 2.7: aspetto dei pannelli di ostacoli e obiettivi attivi dopo aver avviato la simulazione

# Organizzazione delle classi

In questo capitolo viene dato un breve resoconto dell’organizzazione logica delle componenti del simulatore, senza scendere nei dettagli, che è comunque possibile trovare all’interno dei commenti del codice.

## Struttura logica di *Crowd Simulator*

La logica dell’intero programma è organizzata secondo la struttura che può essere riassunta e semplificata in uno schema contenente le sole classi di alto livello come quello in Fig. 3.1, dove ogni freccia sottintende una collaborazione tra le due classi. L’elemento *Models* in figuranon è una classe, bensì un package che racchiude a sua volta tutte le classi delle entità utilizzate nel programma: pedoni, gruppi di pedoni, ostacoli e obiettivi, ma anche l’edificio, le porte e le stanze.

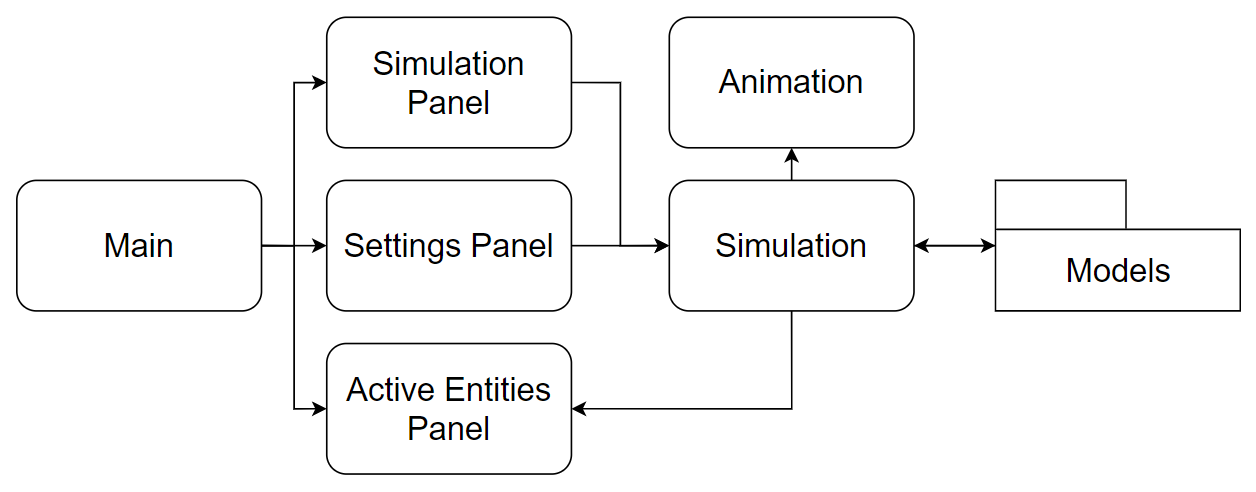


Fig. 3.1: organizzazione logica delle classi di alto livello e loro collegamenti

Come si può vedere, i 3 pannelli principali (Simulation Panel, Settings Panel e Active Entities Panel) vengono inizializzati nella classe *Main.java*. Una volta creata l’istanza della classe, ogni pannello gestisce in maniera autonoma le proprie funzionalità, inviando o prelevando dati dalla simulazione. A sua volta la simulazione preleva e sovrascrive informazioni in merito alle entità contenute nel package *Models*, per poi inviare informazioni alla classe Animation, che si occupa della rappresentazione grafica.[[6]](#footnote-6)

## Package Models: le entità

Il package *Models* contiene tutte le entità presenti nella simulazione, sia statiche che dinamiche, secondo uno schema logico di relazioni come quello in figura Fig. 3.2. Pedoni, ostacoli ed entità ereditano da un’unica superclasse *Entity* che contiene attributi e metodi comuni. La classe Pedestrian (che indica il pedone) collabora con la classe Group (che indica il gruppo di pedoni) in quanto ogni pedone conosce il proprio gruppo. In realtà, seppur indirettamente, la classe *Pedestrian* collabora con ogni classe del package in quanto, ogni pedone, per calcolare la sua prossima posizione ha bisogno di tutte le informazioni sull’ambiente che lo circonda, e quindi prende informazioni da tutte le classi del package.

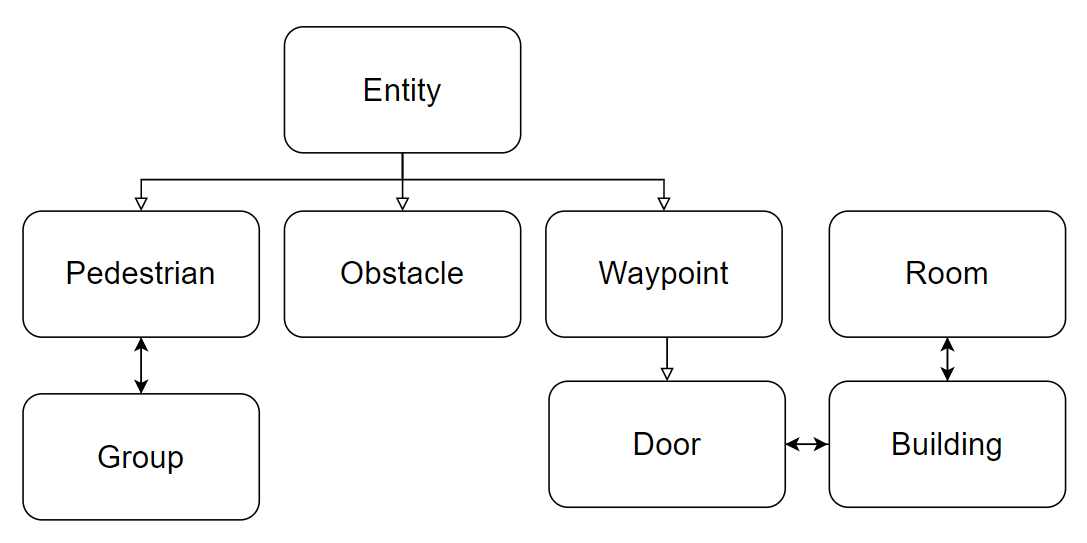


Fig. 3.2: schema logico classi del package Models. Le frecce vuote indicano ereditarietà, le frecce piene indicano una generica collaborazione

Inoltre, la classe *Door* rappresenta una porta dell’edificio e costituisce un *waypoint* in quanto ogni porta viene vista come meta-obiettivo da raggiungere durante l’esplorazione dei pedoni. La classe *Door* dunque collabora con *Building*, che rappresenta l’edificio, che a sua volta collabora con *Room*, la classe che rappresenta una delle sue stanze.

## Problemi di compatibilità con l’algoritmo di movimento innovativo

In questa trattazione non si entrerà nei dettagli del codice che costituisce le classi del package Models, così come non si tratteranno in maniera approfondita le caratteristiche delle classi che non riguardino direttamente l’interfaccia grafica e il suo funzionamento. Il motivo è che *Crowd Simulator* nasce con l’intento di raccogliere informazioni riguardo un algoritmo di movimento innovativo, il quale necessita di un sistema sottostante differente dall’attuale struttura della versione beta del simulatore. Per questo motivo, il programma andrà radicalmente ampliato oppure riscritto in molte delle sue parti, rendendo superflua la trattazione approfondita delle caratteristiche delle singole classi.

Il problema fondamentale di incompatibilità tra i due algoritmi riguarda il fatto che l’attuale sistema di movimento dei pedoni utilizza uno spazio definito per coordinate reali, mentre l’algoritmo di gestione del movimento innovativo necessita di una “griglia” di posizioni, in quanto fa utilizzo di tecniche di *path finding*.

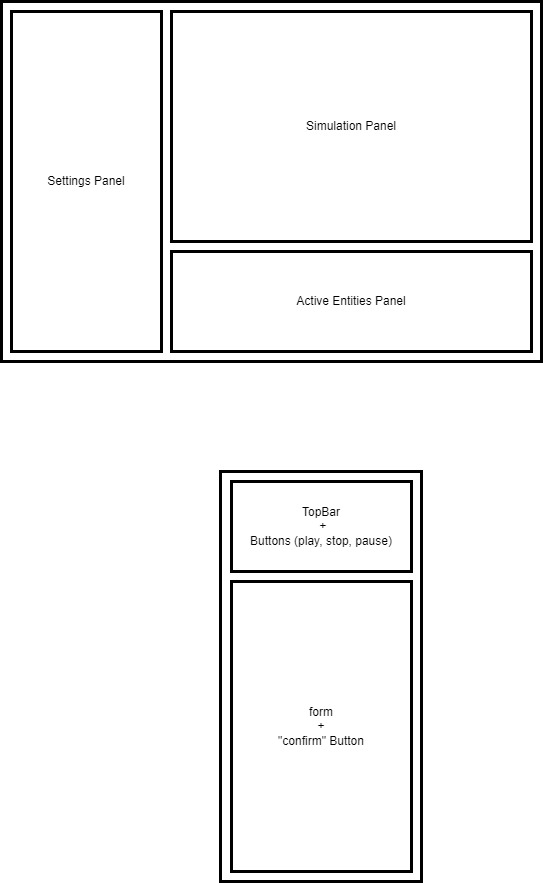
Nel capitolo 4 si tratterà in maniera più accurata delle caratteristiche dell’algoritmo di movimento innovativo e dei suoi requisiti.

## Package *src*: l’interfaccia

Come già descritto, l’interfaccia viene suddivisa inizialmente in tre macro-sezioni, ognuna delle quasi ha una classe Java corrispondente, la quale ne gestisce l’aspetto e le funzionalità:

* Setting panel (*src\SettingPanel.java*);
* Simulation panel (*src\SimuationPanel.java*);
* Active entities panel (*src\ActiveEntitiesPanel.java*).

I tre pannelli appena elencati sono disposti all’interno del JFrame principale così come mostrato in Fig. 1.1.



Il loro funzionamento viene descritto più nello specifico nei commenti presenti nel codice in sintassi Javadoc.

## Raccolta dati a fine simulazione

La raccolta dei dati in merito alla simulazione avviene grazie all’utilizzo della classe Gson, che permette di trasporre un oggetto nella sua rappresentazione JSON e viceversa. A questo proposito, la raccolta di dati è gestita dalle classi nel package *src/support/dataHandler*.

Il file JSON prodotto a fine simulazione contiene, in ordine:

* l’elenco di tutti gli ostacoli, con i valori dei relativi attributi;
* l’elenco di tutti gli obiettivi, con i valori dei relativi attributi;
* l’elenco di tutti i gruppi di pedoni, per ogni gruppo sono riportati prima i suoi dati statici e poi tutti i *timestamp,* con i suoi attributi aggiornati a intervalli regolari;
* l’elenco di tutti i pedoni, per ognuno sono riportati prima i suoi dati statici e poi tutti i *timestamp,* con i suoi attributi aggiornati a intervalli regolari.

I dati statici di ogni entità sono prelevati direttamente dalle classi interessate, mentre per la gestione dei *timestamp* sono state create delle classi ad hoc la cui istanza è una fotografia di un’entità dinamica in un dato momento della simulazione. Il costruttore di queste classi prende in input un’entità dinamica, ossia:

* un gruppo di pedoni nel caso di *src/support/dataHandler/GroupTimestamp;*
* un pedone nel caso di *src/support/dataHandler/PedestrianTimestamp*.

Dato questo input, si prelevano le informazioni interessanti dell’entità e le si salvano negli attributi della classe, che a questo punto è pronta per essere convertita in JSON.

### Funzionamento e riuso

Questo sistema d raccolta dati è costruito su misura delle caratteristiche attuali della versione beta del simulatore, potrebbe essere necessario modificarlo per renderlo compatibile con l’algoritmo di movimento innovativo.

In caso contrario, il sistema di raccolta dati funziona seguendo i seguenti step:

* Nel momento in cui la simulazione ha inizio si crea un’istanza della classe *dataHandler* e si collezionano i dati statici di tutte le entità, ossia i valori degli attributi delle entità che non varieranno nel corso della simulazione, questo è fatto tramite l’uso delle funzioni
  + *dataHandler.setObstaclesData(obstacles);*
  + *dataHandler.setWaypointsData(wayPoints);*
  + *dataHandler.setGroupsStaticData(groups);*
  + *dataHandler.setPedestriansStaticData(crowd);*
* Si fa partire il timer ciclico al termine del quale di volta in volta si collezionano i dati dinamici delle entità che ne possiedono (gruppi e pedoni), questo viene fatto tramite l’uso delle funzioni:
  + *dataHandler.setPedestriansTimestamp(crowd, timestampNum);*
  + *dataHandler.setGroupsTimestamp(groups, timestampNum);*

Il parametro formale *timestampNum* indica il tempo in cui è stata registrato lo stato dell’entità dinamica in questione, se il timer scade ogni secondo allora *timestampNum* va incrementato di 1 ogni volta che vengono chiamate queste due funzioni (se il timer scade ogni 0.3 secondi andrà incrementato di 0.3 e così via)

* Questi due primi step hanno permesso alla classe *dataHandler* di collezionare tutti gli attributi alla creazione del file JSON. Una volta terminata la simulazione è necessario richiamare la funzione *dataHandler.simulationDataToJSON()*, che creerà automaticamente il file.

# L’algoritmo di movimento innovativo

Questo capitolo propone le basi per la comprensione del funzionamento generale di un algoritmo di movimento. Queste informazioni vengono fornite nel paragrafo 4.1 e 4.2. In seguito, nei paragrafi successivi viene presentato l’effettivo algoritmo di movimento innovativo che sarà necessario implementare all’interno del simulatore. Per dettagli più accurati si consiglia anche la lettura della tesi di dottorato in cui viene descritto questo algoritmo: <https://hdl.handle.net/11584/312918>.

## Introduzione agli algoritmi di movimento

Nell’ambito della simulazione di una folla di pedoni, le tecniche che ne definiscono il movimento solitamente si dividono in due grandi famiglie:

* la prima è quella delle tecniche **macroscopiche**. Queste ultime hanno un approccio *flow-based*, ossia si concentrano sui comportamenti dell’intera folla come se fosse un’unica entità, sono particolarmente efficienti per simulare in maniera poco dettagliata gli spostamenti di un gran numero di entità; quindi, vengono utilizzate soprattutto negli studi in cui si cerca di fare una stima dello stato della folla in tempo reale senza soffermarsi sull’esatto movimento del singolo.
* Le tecniche **microscopiche** si concentrano invece sulla gestione del singolo pedone, sono utili per calcolare il percorso che la singola entità deve seguire evitando eventuali ostacoli statici o dinamici presenti nell’ambiente. Questa tecnica implementa un approccio detto *agent-based*, caratterizzato da individui autonomi con un certo grado di intelligenza; ognuno di loro può reagire a ogni situazione in modo autonomo sulla base di un insieme di regole decisionali. Le informazioni utilizzate per decidere un'azione sono ottenute localmente dall'ambiente circostante. Il più delle volte questo approccio viene utilizzato per simulare un comportamento realistico della folla, in quanto il programmatore ha piena libertà di implementare qualsiasi comportamento.

Le folle di pedoni possono avere determinate caratteristiche. Solitamente si parla di folle **omogenee** se ogni pedone ha comportamenti e obiettivi molto simili, oppure di folle **eterogenee** se ogni pedone mantiene un’identità distinta e osservabile. A prescindere dalle sue caratteristiche intrinseche, il calcolo del movimento di un pedone è suddiviso in due approcci distinti: quello che gestisce la **navigazione globale** e quello che gestisce la **navigazione locale**. La navigazione globale mira a calcolare un percorso a lungo termine senza collisioni verso un obiettivo, considerando solamente gli ostacoli statici. Al contrario, le tecniche di navigazione locale tengono conto anche del movimento di eventuali ostacoli dinamici e delle altre entità nell'ambiente.

La pianificazione del movimento può a sua volta essere divisa in due ulteriori approcci: **centralizzato** e **disaccoppiato**. L’approccio centralizzato considera tutti i pedoni e tratta il sistema risultante come un unico sistema composito dove i possibili movimenti dei singoli pedoni sono combinati in uno spazio composito in modo che l'algoritmo risultante cerchi una soluzione in questo spazio di configurazione, come se stesse risolvendo un labirinto per ognuno dei pedoni contemporaneamente. Al contrario, l’approccio disaccoppiato procede in modo distribuito tra i vari pedoni, calcolando il percorso della singola entità; perciò, l’algoritmo di calcolo del movimento del singolo pedone ha necessità di essere coordinato con quello degli altri pedoni presenti nella simulazione. Il primo di questi due approcci, quello centralizzato è tendenzialmente meno efficiente rispetto a quello disaccoppiato poiché si basa su un principio molto simile a quello degli algoritmi di *path finding*, ossia di risoluzione dei labirinti, con l’aggravante di dover tenere conto non solo della struttura all’interno della quale trovare il percorso ottimale, ma anche della presenza delle altre entità.

## Il flocking

Uno dei pilastri fondamentali del movimento di entità autonome è quello basato sul comportamento di *flocking*. Il primo a mostrare uno studio importante in merito agli algoritmi di *flocking* fu il professor Craig Reynolds nel 1987, nel suo articolo intitolato *Flocks, herds and schools: A distributed behavioral model*. L’intento iniziale non era la simulazione di folle di pedoni, bensì cercare un modo per simulare il comportamento di stormi di uccelli; ogni entità nel lavoro di Reynolds prende il nome di *boid*. Ogni *boid* è implementato come un’entità autonoma che naviga in base alla sua percezione locale dell'ambiente circostante, alle leggi della fisica simulata che ne regolano il movimento e a un insieme di comportamenti. Il movimento aggregato dello stormo è il risultato della fitta interazione dei comportamenti relativamente semplici delle singole entità simulate. Per determinare i loro spostamenti, i *boid* devono costantemente monitorare i loro vicini per assumere un comportamento verosimile.

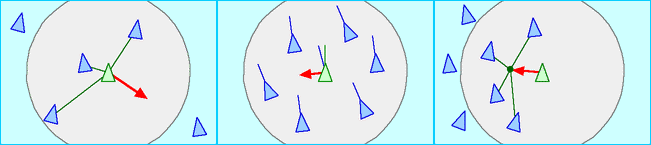


Fig. 4.1:Da sinistra a destra, il vettore rosso rappresenta rispettivamente le forze di separazione, allineamento e coesione dell’entità verde. Il cerchio bianco indica l’area locale del boid verde, le entità all’interno di questo cerchio sono considerate ai fini del calcolo delle forze di coesione, allineamento e separazione.

Nell’articolo, Reynolds parla di un approccio basato su tre regole che governano i movimenti delle singole entità simulate. Queste tre regole costituiscono il *flocking* e sono:

* una forza di **separazione** che allontana le entità dai propri vicini per evitare casi di affollamento locale e collassamento di tutte le entità in un unico punto;
* una forza di **allineamento**, che ha la funzione di regolare la velocità del *boid* rispetto alle entità vicine e allinearlo secondo la direzione e il verso di questi stessi vicini;
* una forza di **coesione** che spinge l’entità verso la posizione media (baricentro) dei suoi vicini. Queste forze sono rappresentate in Fig. 4.1.

Grazie al solo utilizzo di queste tre forze, lo stormo di entità può spostarsi in un ambiente virtuale semplice, si veda l’esempio interattivo nel [sito del professor Shiffman](https://processing.org/examples/flocking.html). Chiaramente poi questo comportamento può essere reso maggiormente complesso a seconda del risultato che si vuole ottenere, si possono includere tecniche di prevenzione delle collisioni, dinamiche di gruppo e molto altro.

## Funzionamento dell’algoritmo innovativo

L’algoritmo di movimento innovativo è un metodo per la simulazione di folle di pedoni eterogenee con gestione disaccoppiata del movimento. Le entità su cui lavora sono le stesse già descritte nel paragrafo 2.3, cambia solamente l’approccio con il quale viene gestito lo spostamento dei pedoni. Il metodo proposto utilizza un modello microscopico multi-gruppo per generare una traiettoria in tempo reale per ogni pedone durante la navigazione nell'area dell’edificio. Inoltre, viene introdotto un modello *agent-based* per modellare i comportamenti dei singoli pedoni. Grazie a questo duplice approccio, i pedoni utilizzano la conoscenza locale per evitare le collisioni e interagire con gli altri membri della folla, mentre utilizzano la conoscenza globale per la pianificazione a lungo termine e per essere in grado di orientarsi verso il loro obiettivo corrente. L’insieme degli *steering behavior* che gestiscono il movimento racchiudono le forze di *flocking*, ossia separazione, coesione e allineamento, così come discusse nel paragrafo precedente e una forza atta a indirizzare il pedone verso il suo obiettivo. Le forze appena descritte vengono unite all’interno di un’unica forza risultate, determinando il movimento del pedone. Durante questa fase l’influenza e il peso di ciascuna delle forze dipendono dal contesto in cui il pedone si trova e avviene tramite due tecniche: ***switching*** e ***blending***. Nello *switching*, man mano che le circostanze nell'ambiente simulato lo richiedono, il pedone può "passare" da una modalità di comportamento all'altra. Invece, nel ***blending*** questi comportamenti, che agiscono in parallelo, sono "mescolati" insieme in un’unica forza risultante. Ad esempio, in situazioni normali, i pedoni che si muovono nell'ambiente simulato verso la loro destinazione fondono i comportamenti di *flocking* e con quelli di orientamento verso l'obiettivo in un unico vettore direzionale, per consentire al gruppo di camminare verso la meta e rimanere coesi allo stesso tempo. Inoltre, supponiamo che un gruppo di pedoni si muova in un determinato ambiente e si avvicinino a un ostacolo o ad altri pedoni dello stesso gruppo o di gruppi diversi. Questa situazione porta a passare da una fase in cui la priorità è il movimento verso l’ostacolo, a una in cui la priorità è l'evitamento delle collisioni, anche a costo di spezzare la coesione del gruppo.

Per quanto riguarda la prevenzione delle collisioni si implementa un sistema basato su un BNM (Boundary Node Model) esteso, che verrà usato per generare un percorso senza collisioni per ogni pedone quando si avvicina agli ostacoli in tempo reale. L'utilizzo della pianificazione globale del percorso in contesti real-time è difficile da modellare utilizzando un modello *agent-based*, perché diventa computazionalmente costoso. Pertanto, i modelli basati su agenti utilizzati in questo algortimo separano la prevenzione delle collisioni locali dalla pianificazione globale del percorso. Sulla base del metodo BNM esteso, ogni pedone nella folla è simulato da un quadrilatero a nove nodi, come descritto in Fig. 4.2.

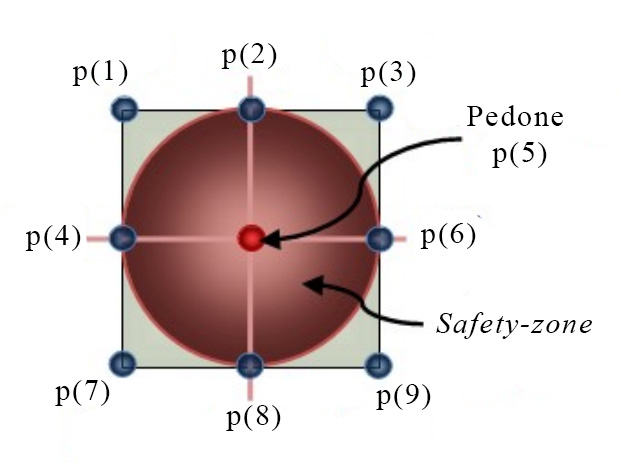


Fig. 4.2: safety-zone del pedone e bordi che ne delimitano lo spazio di collisione

A ogni fotogramma chiave (*keyframe*) i pedoni si muovono in avanti rispetto alla loro posizione corrente alla nuova posizione aggiornata (*Positionpe*) con la velocità di camminata *Velocitype*. L'algoritmo a.6 spiega il movimento dei pedoni nell'ambiente virtuale dal punto di partenza al punto di arrivo e il processo è illustrato nelle seguenti fasi.

1. L’algoritmo prende in input i dati inerenti all’ambiente di simulazione e alle sue entità: il numero di frames che andranno a comporre l’intera animazione *nFrames*; il numero di gruppi di pedoni *nGroups*; la lista di attivazione dei gruppi *Activategroups*; la lista di terminazione dei gruppi *Terminategroups*; la lista di pedoni nella folla *Pedestrianscrowd*; il numero di pedoni nella folla *nPedestrianscrowd*; la lista di obiettivi per ogni gruppo di pedoni *Goalscrowd*; la lista delle posizioni di ogni membro della folla *Positionscrowd*; la costante che stabilisce di quanto deve diminuire l’energia dei pedoni ad ogni keyframe cs1.
2. Ad ogni *keyframe* (riga 1*,* algoritmo a.6), il valore di attivazione di ogni gruppo *ActivateGroups[gr]* è impostato su “on” quando keyframe raggiunge il valore *kFrameactivate[gr]* (riga 2-3*,* algoritmo a.6). Questo valore consente ai membri, inizialmente fermi, di iniziare a camminare, dunque, per ogni gruppo attivo si definisce:
   1. l’indice dell’obiettivo *Goalscrowd[gr]* (riga 7*,* algoritmo a.6);
   2. i pedoni nel gruppo *Pedestrianscrowd[gr]* (riga 9*,* algoritmo a.6);
   3. numero di pedoni nel gruppo n*Pedestrianscrowd[gr]* (riga 10*,* algoritmo a.6);
   4. gli obiettivi *Goalscrowd[gr]* per il gruppo n*Pedestrianscrowd[gr]* (riga 11*,* algoritmo a.6).
3. Per ogni pedone (riga 12*,* algoritmo a.6) la sua posizione corrente *Positionpe* è ottenuta in tempo reale dalla scena, mentre la sua velocità *Velocitype* e la sua energia *Energype* sono ottenute dagli attributi del pedone contenuto in *Pedestriansgroup[pe]* (riga 14*,* algoritmo a.6).
4. Ad ogni keyframe viene anche calcolata la nuova *Velocitype*  per ogni pedone in *Pedestriansgroup[pe]* nei gruppi attivi. La velocità è limitata ad un certo valore che viene influenzato da cs1. Allo stesso tempo viene anche calcolata la nuova *Positionpe*  sulla base degli *steering behavior* (riga 26*,* algoritmo a.6).
5. Si verifica che non avvengano collisioni con altri pedoni (riga 27*,* algoritmo a.6), con i quali deve essere mantenuta sempre una determinata distanza minima, in caso di avvenuta collisione si procede subito con il calcolo della nuova posizione del pedone (riga 28*,* algoritmo a.6). Si verifica anche se avvengono collisioni con ostacoli statici (riga 30*,* algoritmo a.6), in caso affermativo si utilizza il metodo BNM per calcolare una nuova *Positionpe*  per il pedone interessato.
6. Ad ogni keyframe il livello di energia di ogni pedone viene decrementato (riga 15*,* algoritmo a.6). Quando il valore di *Positionpe*  scende sotto a un certo valore soglia tutti i pedoni del gruppo si dirigeranno verso un punto di ristoro dopo aver concluso il compito che stavano svolgendo (riga 33-34*,* algoritmo a.6). Il gruppo rimane presso il punto di ristoro fino a quando ogni componente ha di nuovo raggiunto il livello massimo di energia. A questo punto continueranno a dirigersi verso i loro obiettivi (riga 36-41*,* algoritmo a.6).
7. Ad ogni keyframe, se la distanza tra un pedone del gruppo *Pedestriansgroup* situato in *Positionpe*  e un obiettivo *Goalsgroup[Goali]* è minore di cs2, significa che quel gruppo è arrivato nei pressi dell’obiettivo corrente. Dunque, questo gruppo deve cercare un nuovo obiettivo dalla sua lista e aggiornare l’*index(Goalindex)* degli obiettivi (riga 18-25*,* algoritmo a.6).
8. Ad ogni keyframe i pedoni cambiano la loro posizione attuale in *Positionpe*  in tempo reale, subito dopo, vengono aggiornati anche gli attributi del pedone, includendo *Velocitype*  ed *Energype*  in *Pedestriansgroup[pe]* (riga 42-44*,* algoritmo a.6).
9. Quando il gruppo *Pedestriansgroup[gr]* raggiunge l’ultimo obiettivo *Goalsgroup[last]*, il valore di *Terminategroups[gr]* viene impostato su “on” e il valore di *Activategroups[gr]*  viene impostato su “off” (riga 18-25*,* algoritmo a.6).

|  |  |
| --- | --- |
| Input: nFrames, nGroups, Activategroups, Terminategroups, Pedestrianscrowd, nPedestrianscrowd, Goalscrowd, Positionscrowd, cs1 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | **for** keyframe ← 1 **to** nFrames **do**  **if** keyframe = kFrameactivate **then**  Activategroups[gr] ← “on”  **end if**  pc ← 0, Goalsindex([1] \* nGroups)  **for** gr ← 1 **to** nGroups **do**  pc ← pc + 1, Goali ← Goalsindex[gr]  **if** Terminategroups[gr] = “off” **AND** Activategroups[gr] ← “on” **then**  Pedestriansgroup ← Pedestrianscrowd [gr]  nPedestriansgroup ← nPedestrianscrowd[gr]  Goalsgroup ← Goalscrowd[gr]  **for** pe ← 1 **to** nPedestriansgroup **do**  Compute Separation, Cohesion, Allignment  (Positionpe, Velocitype, Energype) ← Pedestriansgroup[pe]  Energype ← Energype – Cs1  Pedestriansgroup[pe] ← Energype  **if** Energype > minimumEnergype **then**  **if** dist(Positionpe, Goalsgroup[Goali]) < cs2 **then**  Goali ← Goali + 1  updateGoalsindex[gr] ← Goali  **if** Goali = len(Goalsgroup) **then**  Terminategroups[gr] ← “on”  Activategroups[gr] ← “off”  **end if**  **end if**  Compute new Velocitype and Positionpe  **if** indCollision = “true” **then**  Positionpe ← pedestrian collision avoidance  **end if**  **if** obstacleCollision = “true” **then** Positionpe ← BNM  **end if**  **else**  stay in service point, modify Goalsgroup & Energype  update Pedestriansgroup[pe] with new Energype  **if** staying time is finished **then**  **for** pe ← 1 **to** nPedestriansgroup **do**  new Energype ← maximum Energype  update Pedestriansgroup[pe] ← new Energype  **end for**  **end if**  **end if**  Positioncrowd[pc] ← Positionpe  update keyframe and Positionpe  update Pedestriansgroup[pe] ← new(Positionpe, Velocitype)  **end for**  **end if**  **end for** **end for** |

Fig. 4.3: Gestione del movimento nell'algoritmo innovativo

1. Il pannello delle entità attive mostra un resoconto di tutti gli elementi presenti nella simulazione, il suo funzionamento viene descritto più nel dettaglio nel paragrafo 2.3. [↑](#footnote-ref-1)
2. I valori di energia e velocità massima del pedone sono generati casualmente all’interno di un certo intervallo casuale, che varia in funzione dell’età. Tendenzialmente i pedoni più anziani saranno più lenti e avranno meno energia. [↑](#footnote-ref-2)
3. Questo aspetto rappresenta il fatto che normalmente una folla di persone non arriva in un posto tutta nello stesso tempo, ma ogni gruppo vi si reca autonomamente in tempi indipendenti dagli altri [↑](#footnote-ref-3)
4. JFC è un set di componenti (finestre, pannelli e widget in generale) per lo sviluppo di applicazioni desktop [↑](#footnote-ref-4)
5. AWT è la libreria Java contenente le classi e le interfacce fondamentali per il rendering grafico. [↑](#footnote-ref-5)
6. Attenzione! Nella classe Animation, sebbene non sia specificato in Fig 3.1, viene anche richiamata la funzione interna alla classe *Models\Pedestrian.java* che permette al pedone di calcolare la sua nuova posizione. Dunque, tecnicamente Animation collabora anche con elementi interni a *Models* [↑](#footnote-ref-6)