Lezione 12 07/05/2025

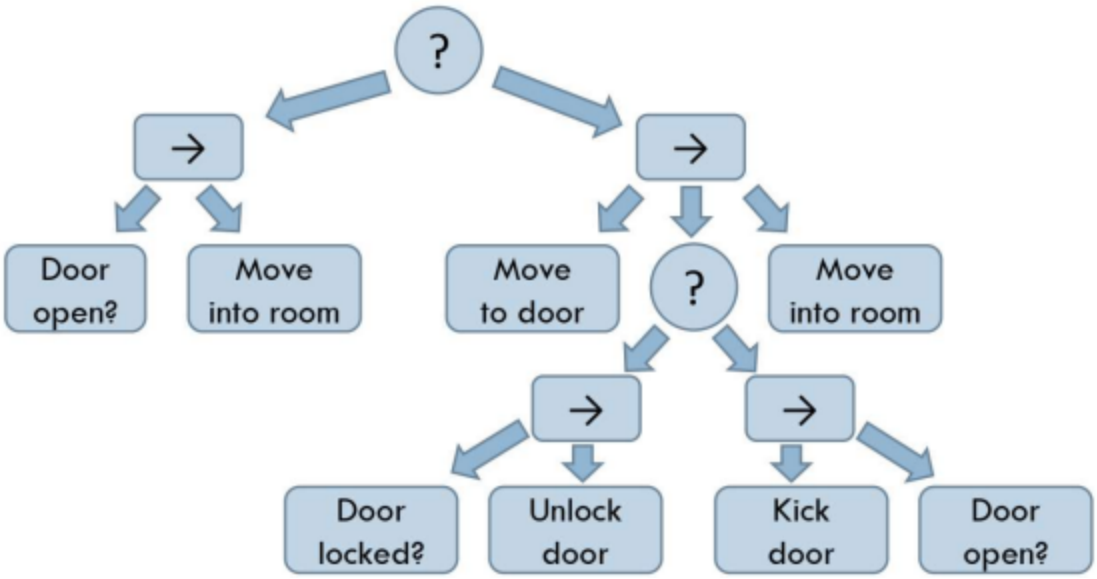
AI parte 2

Il problema dei metodi visti è che bisogna andare a definire tutte le possibilità, di modo che l'algoritmo può andare a scegliere il path migliore.

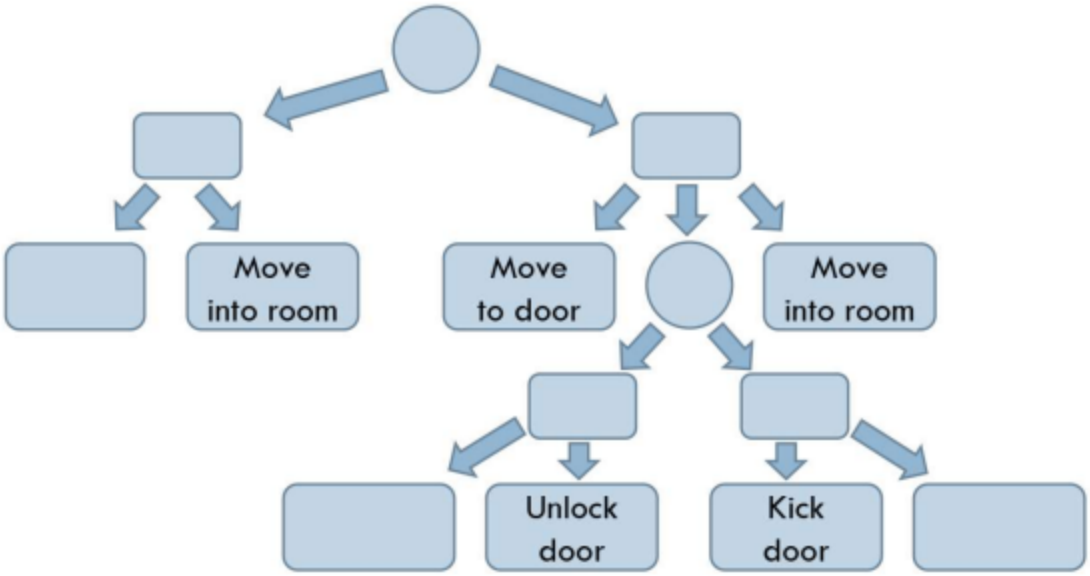
Goal Oriented Action Plan (GOAP)

L'idea è quella di svincolarsi da quest'enumerazione esaustiva di tutte le possibili azioni e cammini.

Questo è un behavioral tree che abbiamo visto, dove abbiamo in cima un certo obiettivo e sotto i diversi cammini che possono essere intrapresi in base alle azioni:

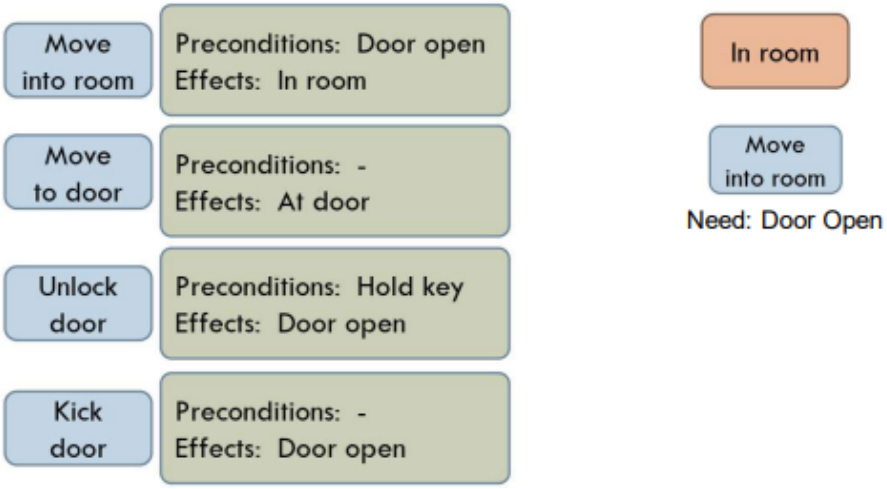


Vogliamo ignorare questi cammini, e concentrarsi solo sulle azioni che sarebbe possibile fare in quel particolare contesto del gioco:

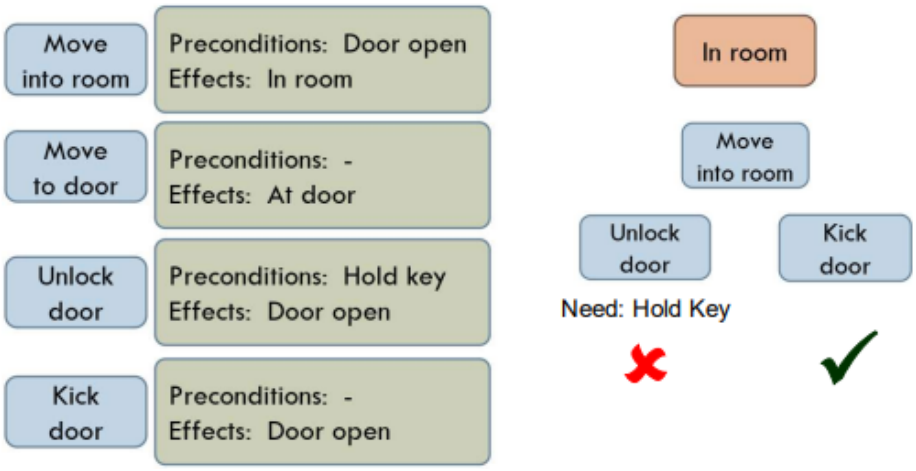


Quindi praticamente si ignora tutto tranne le foglie con le azioni.

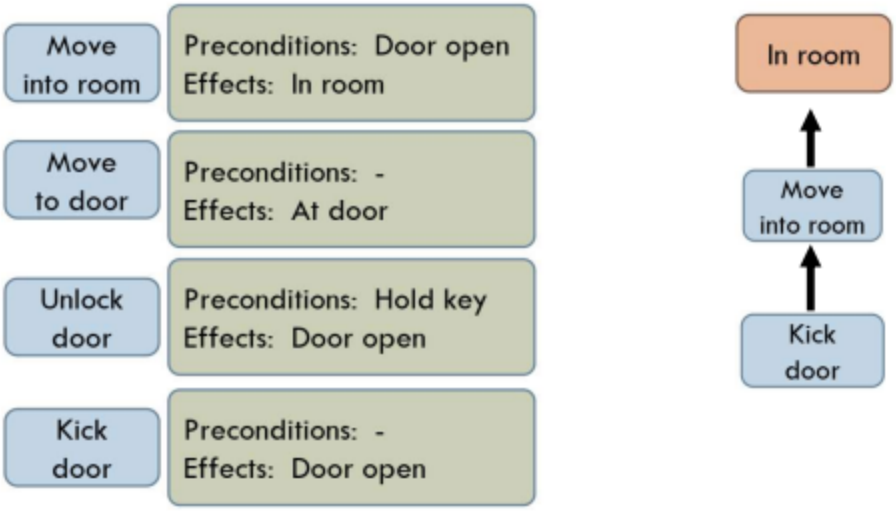
Queste azioni vengono considerate fattibili se ci sono delle precondizioni soddisfatte. Quindi ad ogni azione vengono associate delle precondizioni che sono delle altre azioni che vengono prima, e che potrebbero a loro volta generare delle post condizioni.



- Repeat the previous step until all preconditions satisfied



- If a viable plan exists, execute it
- On failure, check the next goal



è una algoritmo adattivo, perchè basta di dare queste azioni potenziali e lui decide la strategia da prendere in base alle condizioni.

Altro esempio:

se il nostro personaggio ha 3 obiettivi, con i 3 stati che vuole raggiungere.

Goals	Desired World State
Kill Enemy	Attacking Target X
Use Work Node	Using Node Y
Idle	Idling

abbiamo tutte le azioni ammissibili in quel momento del gioco, con le post condizioni e le pre condizioni.

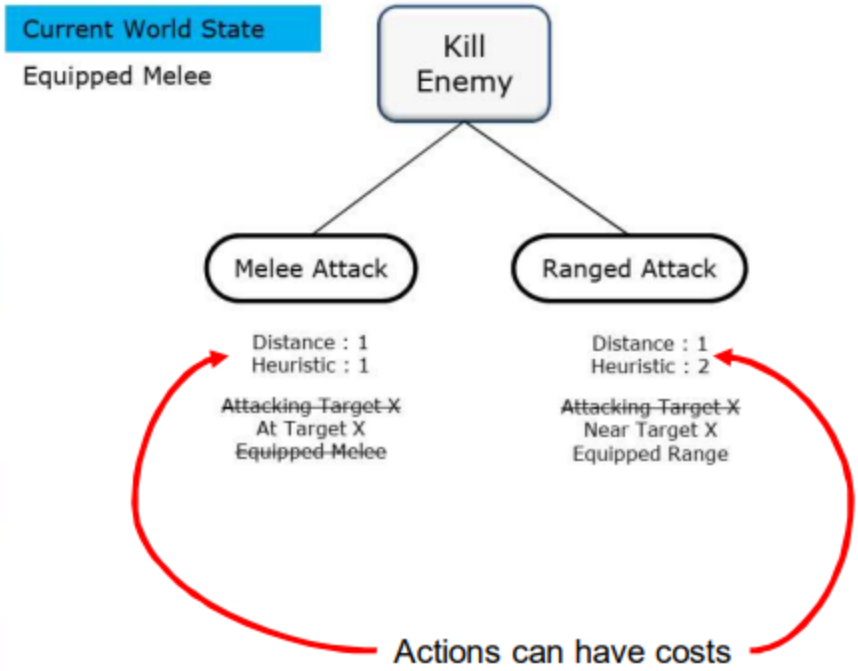
Actions	Satisfies World State	Requires World State
Melee Attack	Attacking Target X	At Target X Equipped Melee
Ranged Attack	Attacking Target X	Near Target X Equipped Ranged
Goto Target	At Target X Near Target X	
Switch Weapon	Equipped Z	
Play Node Animation	Using Node Y	At Node Y
Goto Node	At Node Y	
Idle	Idling	

quindi supponiamo che lo stato del nostro personaggio sia questo:

se il nostro obiettivo è:

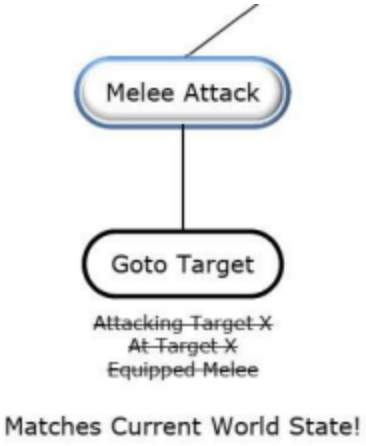
allora andiamo a vedere quali sono le azioni che generano questo effetto:

Actions	Satisfies World State	Requires World State
Melee Attack	Attacking Target X	At Target X Equipped Melee
Ranged Attack	Attacking Target X	Near Target X Equipped Range



si vanno quindi a vedere le altre condizioni che devono verificarsi. Poi abbiamo anche dei costi che possiamo associare al prendere una decisione rispetto ad un altra, per esempio dati da quanto mi costa raggiungere il nemico per attaccarlo.

Quindi il cammino finale è questo:



Questa tecnica è molto flessibile perchè non c'è nulla di predefinito, stiamo soltanto indicando per ciascuna azione le precondizioni e postcondizioni. L'algoritmo andrà a scegliere in base a queste e agli stati attuali. Bisogna definire i costi, di modo che l'algoritmo possa scegliere quelle con costo minore.

Pro:

- Facile da gestire un gran numero di comportamenti generati
- Pianificazione dinamica: nessuna strategia predefinita
- Comportamenti emergenti
- Facile da adattare a nuove azioni/stati

Contro:

- Rischio di effetti non desiderati
- Il debugging è più complicato
- Richiede più strumenti (tooling)
- Necessità di risolvere piani diversi in tempo reale

Interactive Agent (believably intelligent)

Una volta che ho definito il mio piano, lo devo mettere in azione.

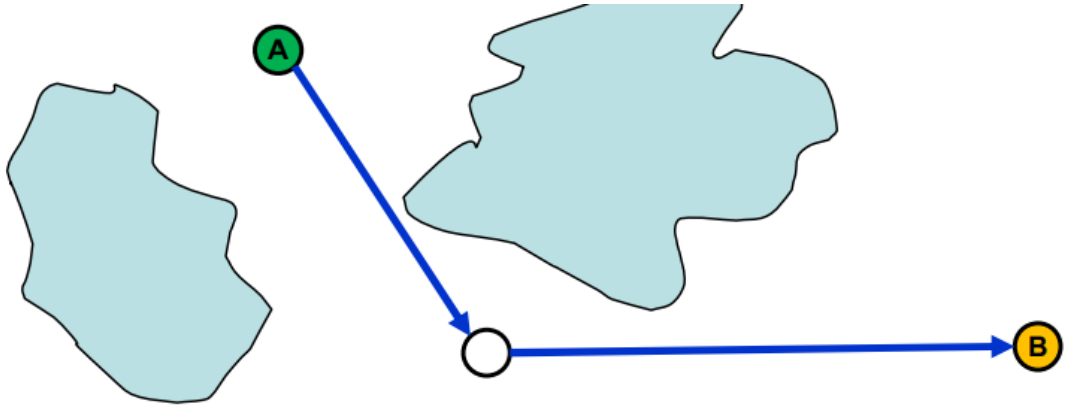
Bisogna definire come il personaggio reagisce al piano stabilito, quindi bisogna muoverlo, animarlo, associargli dei suoni, magari deve interagire con altri personaggi... di modo che il personaggio si comporti in modo plausibile e più o meno intelligente.

Come possiamo muoverlo di modo che sembri si muova in modo naturale?

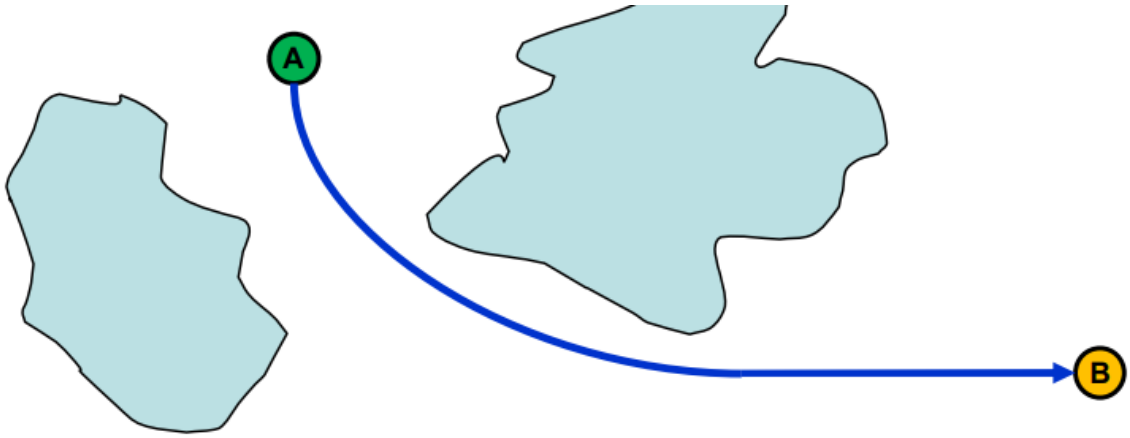
Movimento

Se devo spostare il personaggio da una posizione all'altra, devo fare pathfinding per trovare il cammino. Questo l'abbiamo già visto.

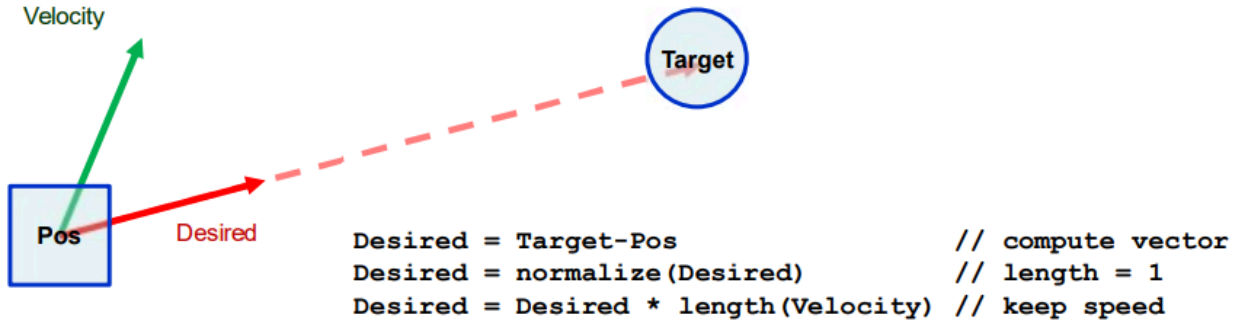
Qui non dovrebbe muoversi in 2 linee rette:



vogliamo che si muova in un modo curvo, tramite **steering behavior**, di modo che sembri più naturale, senza fare movimenti bruschi.

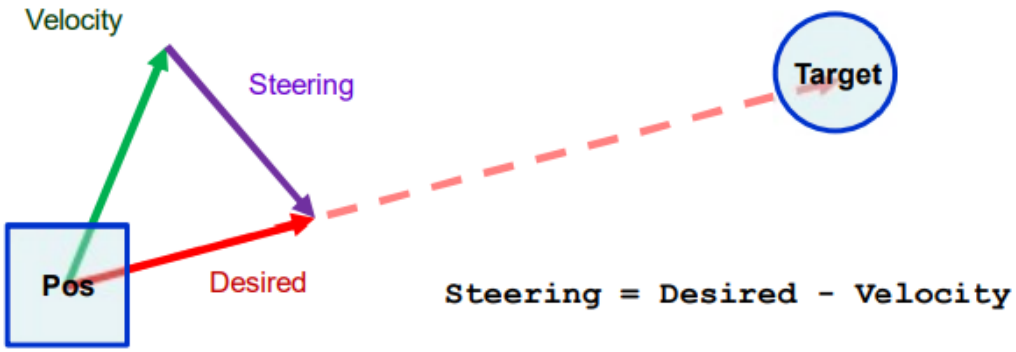


Il personaggio deve ruotare gradualmente dalla direzione verde dove sta andando, alla direzione rossa dove deve andare.



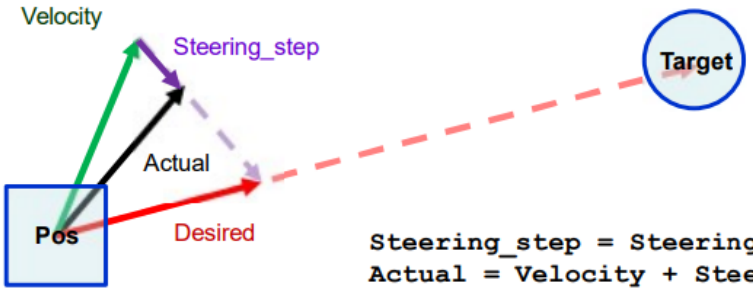
Target e pos sono 2 punti, la loro differenza è un vettore, lo normalizziamo di modo che ha lunghezza 1 perchè ci interessa la direzione e non la velocità. Poi possiamo definire un nuovo vettore, ri-scalandolo alla lunghezza della velocità iniziale.

Ora come passiamo dal vettore verde a quello rosso? Andando a calcolare il vettore di scostamento, il vettore viola, che è la differenza di questi due:



Questo vettore ci dice lungo quale direzione dovremmo ruotare il vettore verde per farlo combaciare con quello rosso.

Quindi, per non avere un cambiamento di direzione istantaneo, scalo il vettore viola di una certa quantità, per avvicinarmi alla direzione rossa passo passo. Questa quantità mi dice quanto velocemente cambio la direzione.



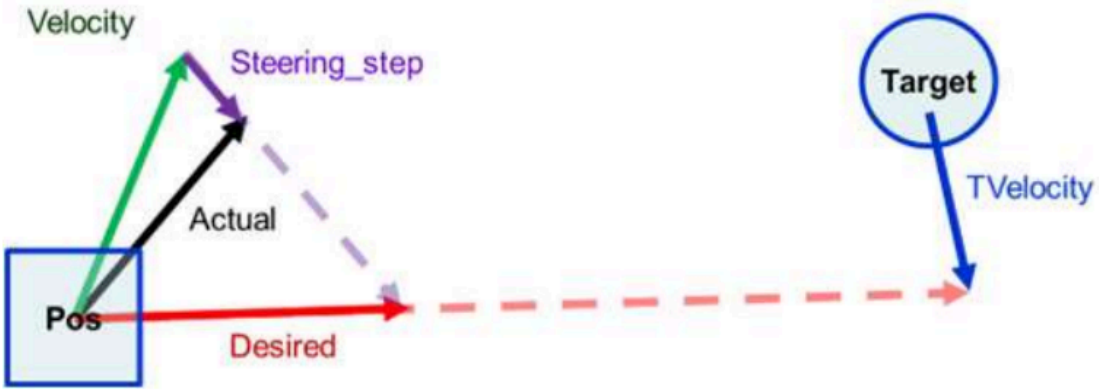

```
Steering_step = Steering * force  // 0 <= force <= 1
Actual = Velocity + Steering_step
```

Questo passo devo farlo tante volte fino a quando arrivo a questo orientamento, ma nel frattempo la direzione si sta spostando sul vettore nero perchè sto curvando ma mi sto anche muovendo in quella direzione.

Questo è il **“seek” behavior**, perchè il personaggio sta tendendo al target. Giocando con questi vettori posso dotare il personaggio di comportamenti diversi.

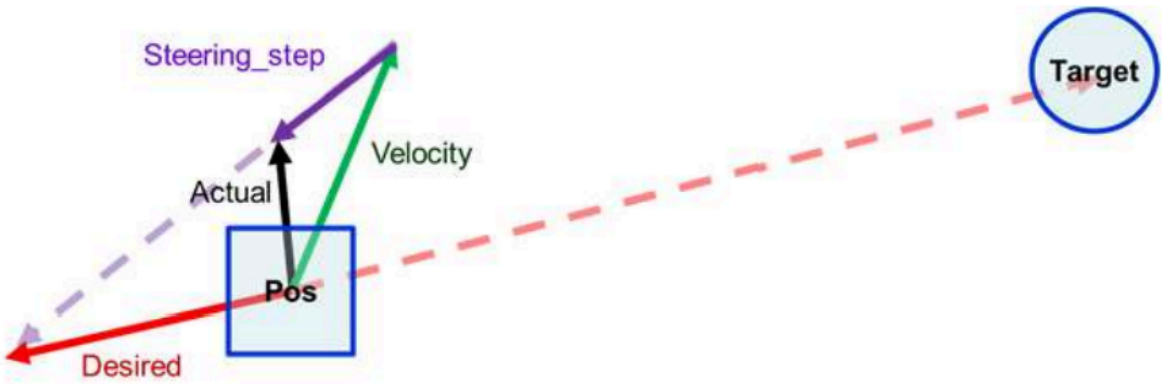
Il **“pursue” behavior** è quando non voglio raggiungere esattamente la direzione target, ma voglio raggiungere una posizione adiacente perchè per esempio è il nemico. Magari sapendo la velocità a cui si muove il target. Quindi la direzione desiderata è la differenza tra dove si potrebbe trovare il target in un certo periodo di tempo e il mio target.

◆ «Pursue» Behavior



Il “flee” behavior è quando voglio evitare il mio nemico, quindi vado nella direzione opposta.

◆ «Flee» Behavior

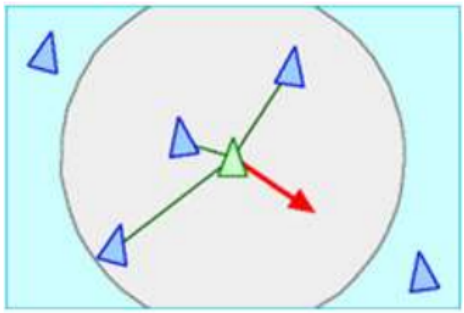


Questi meccanismi possono essere applicati anche con più personaggi contemporaneamente. Qui c'è il problema che non devono darsi fastidio, non devono andare uno sopra all'altro.

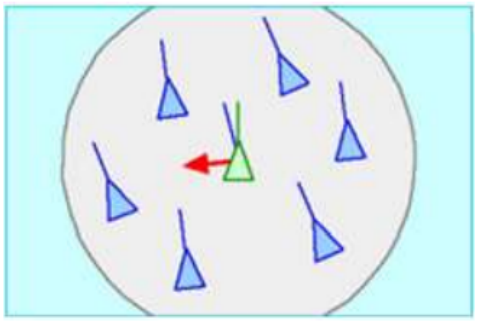
I **flocking algorithms** definiscono questi comportamenti, sono algoritmi applicati ai singoli agenti. Tipicamente, il comportamento che devono avere questi agenti è un movimento simile tra di loro, evitandosi (non collidere) e devono disperdersi.

L'algoritmo classico (craig w reynolds) ha delle regole semplici (usato per modellare il comportamento degli uccelli):

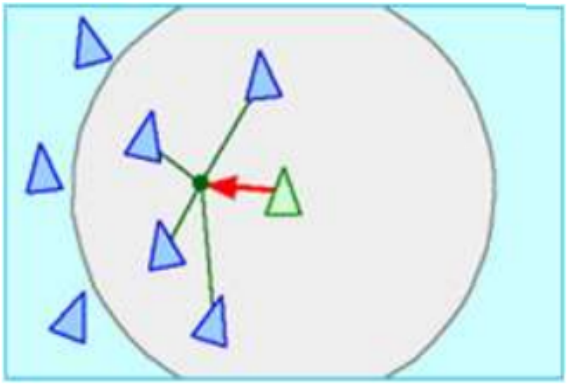
- Tutti gli agenti coinvolti (triangoli) non devono collidere tra loro (**separazione**).



- Devono muoversi tutti nella stessa direzione (**allineamento**).



- Se ho tanti agenti dispersi nello spazio, e gli dico a tutti di andare in un certo target, l'ideale è che si muovano in modo compatto al posto di andare in modo randomico. Questo viene fatto forzando gli agenti a muoversi non direttamente verso il target, ma verso il centro di massa dello stormo degli agenti che si stanno muovendo verso il target. Abbiamo quindi degli agenti separati che ad un certo punto si muoveranno in modo compatto. (**coesione**)



Questa coerenza è importante anche per questioni pratiche.

Procedural Content Generation

Si può usare AI per la generazione di asset. Per esempio per generare direttamente i livelli di gioco, modificare la difficoltà di gioco durante il gameplay.

Per esempio potremmo generare degli alberi, oppure il terreno, oppure dungeons.

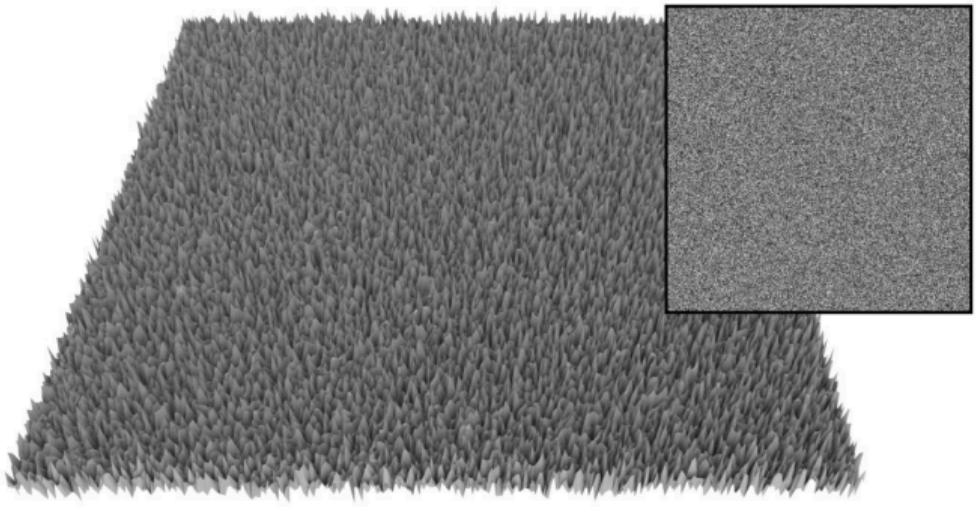
Nel caso degli alberi come funziona? Per esempio con gli **L-systems** si seguono delle regole per generare i rami in modo ricorsivo. Esempio semplice:

```
root              → branch (1m long)
branch (x long)   → branch ( 2x/3 long, -25°), branch ( x/2 long, +55°)
```

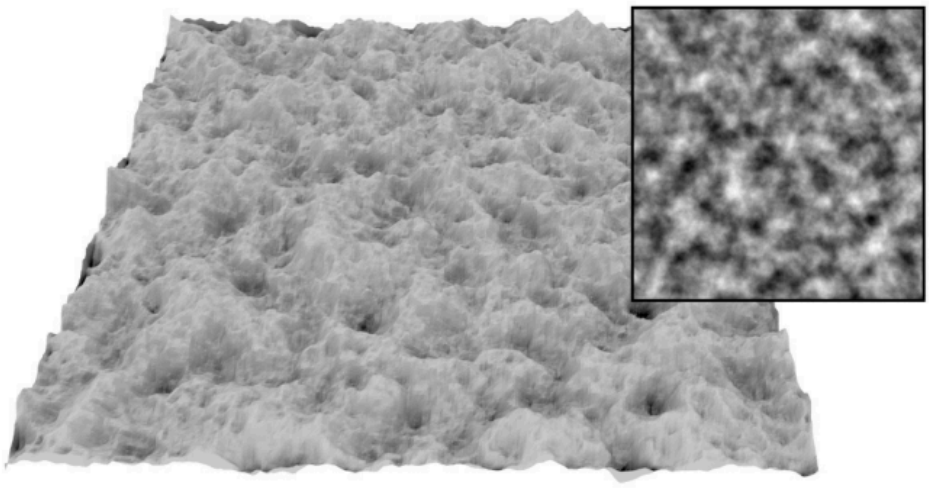
Gli algoritmi di solito generano i rami, e poi si aggiungono le foglie dopo.

Nel caso della generazione di terreni abbiamo diverse possibilità. Di solito di generano le **height maps**. Queste possono contenere anche informazioni come il tipo di materiale (roccioso, sabbia, acqua...) o anche quanto costa per il personaggio attraversarlo.

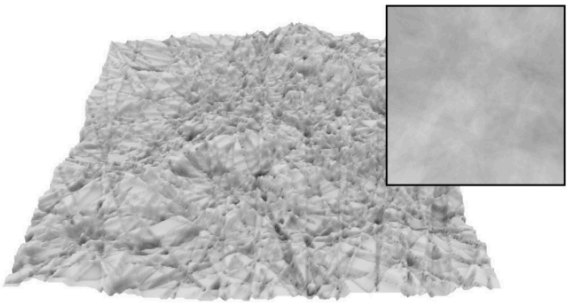
Esempio di generazione da immagine di noise (potrebbe ricordare un prato):

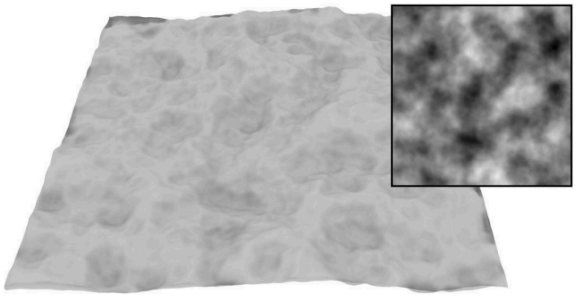


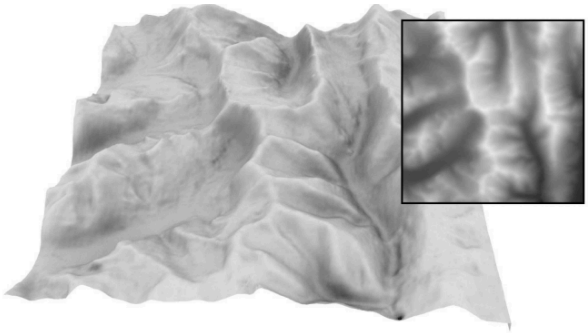
Oppure (questo potrebbe essere un ambiente roccioso):

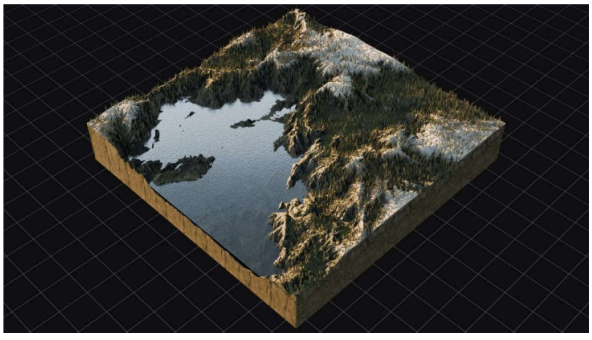


Oppure queste sono una serie di faglie, increspature, per esempio per un effetto di erosione:

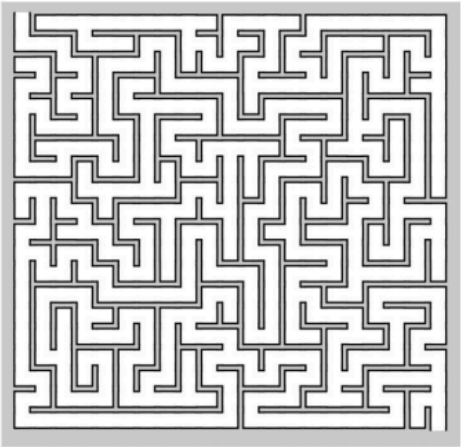








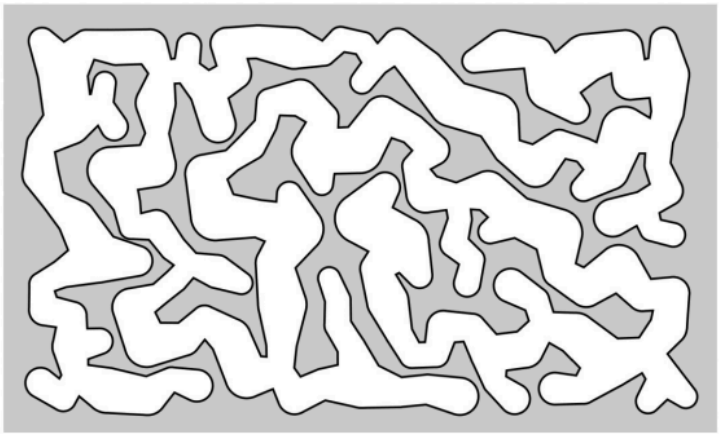
Dungeon and maze generation:



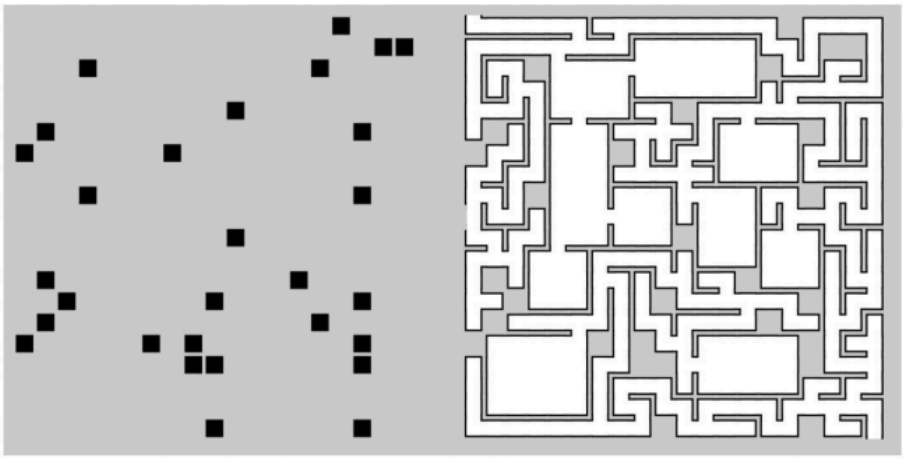
Prim's Algorithm

1. Start with a grid full of walls.
2. Pick a cell, mark it as part of the maze.
   Add the walls of the cell to the wall list.
3. While there are walls in the list:
   1. Pick a random wall from the list. If only one of the cells that the wall divides is visited, then:
      1. Make the wall a passage and mark the unvisited cell as part of the maze.
      2. Add the neighboring walls of the cell to the wall list.
   2. Remove the wall from the list.

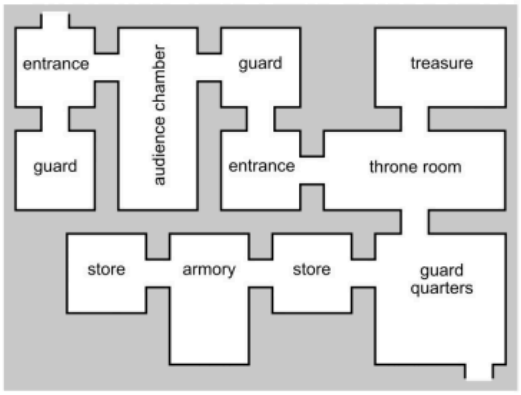
◆ Cave Generation

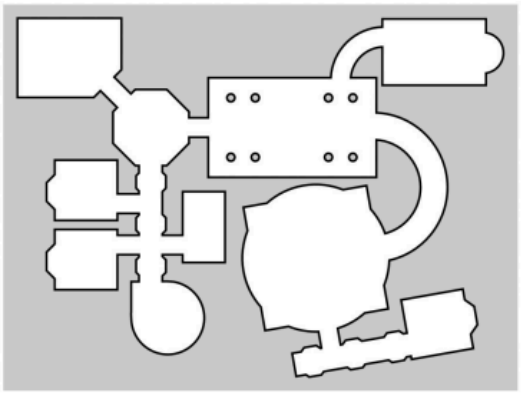


◆ Maze with Rooms and obstructions

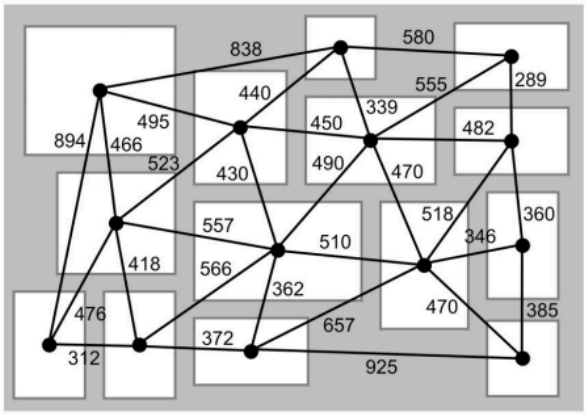


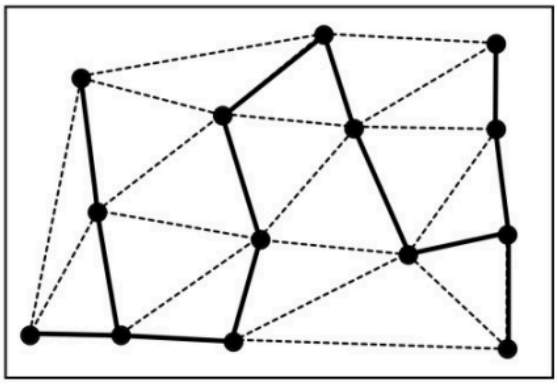
◆ Rooms



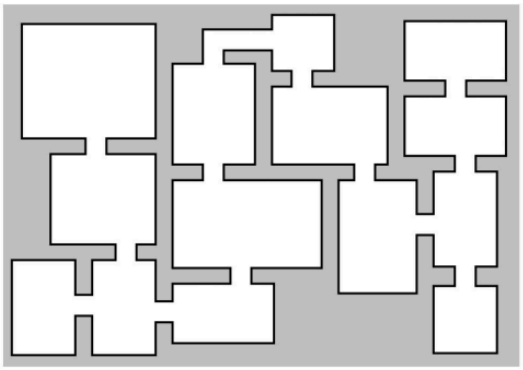


◆ Minimum Spanning Tree

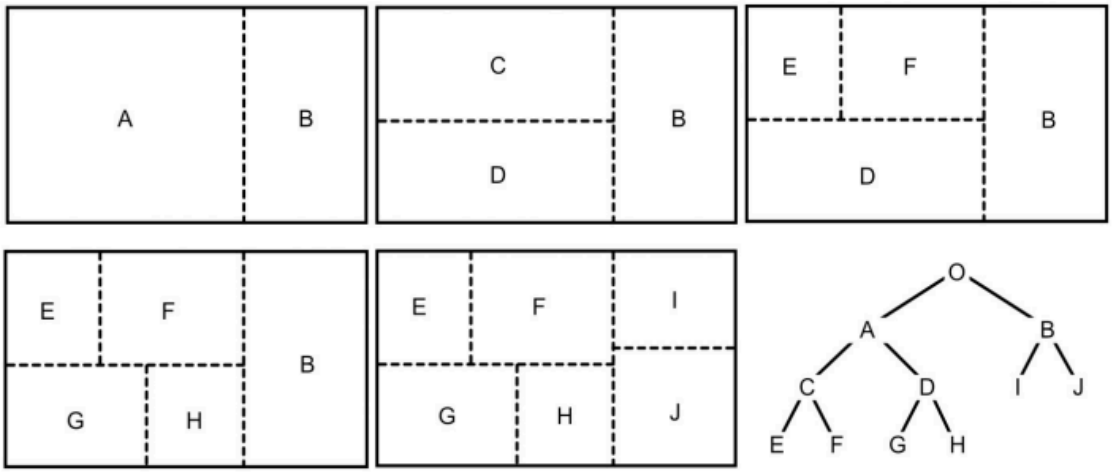




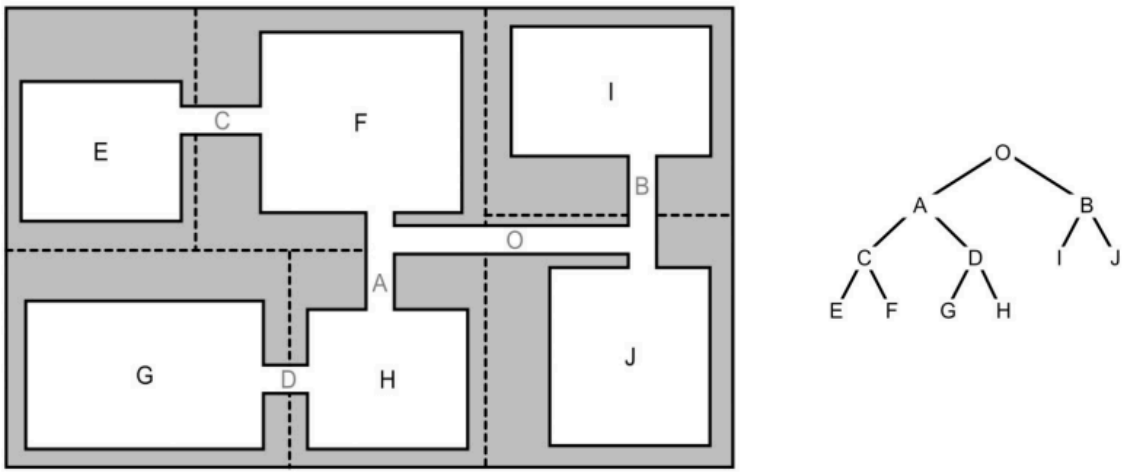
◆ Minimum Spanning Tree



◆ Space Subdivision



◆ Space Subdivision



le foglie sono le stanze, i rami sono i corridoi

AI support in a game engine: a summary

- Assets for (NPC) AI:
  - for behavior modelling:
    - Scripts (can well be the only one)
    - FSM, HSFM, BT, …
  - for navigation:
    - nav-meshes (aka AI-meshes)
  - for sensing / queries:
    - hit-boxes, bounding volumes, spatial indexing
- Game assets
  - To assist their construction (by AI designer)
- Support for a few hard-wired functions
  - To solve lowest level tasks on a 3D environment