Modelli comportamentali durante la visita di un museo e la loro implementazione in ambiente 3D con Unity.

Domenico Esposito
ISISLab, UNISA

Obiettivi della tesi

In questa presentazione verrà illustrata l'implementazione in ambiente Unity 3D di alcuni comportamenti che solitamente gli utenti tendono a seguire durante le visite all'interno dei musei.

I *contenuti* principali della presentazione:

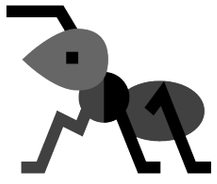
1. Introduzione ai 4 pattern di movimento: Poisson, Fourmi, Papillon e Sauterelle
2. Panoramica della struttura del museo di riferimento in ambiente Unity
3. Esposizione degli algoritmi usati per generare e gestiti: i percorsi, i movimenti e le animazioni dei bot
4. Implementazione dei pattern in Unity con l’uso di C#

Informazioni di partenza

Le informazioni di partenza su cui è basata questa presentazione sono state recuperate dal paper “Etnografia di un’esposizione. Storia dell’expo, popoli e culture” del 1983.

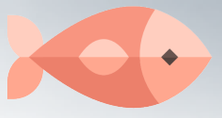
In questo paper, le diverse osservazioni del pubblico all’interno di un museo hanno portato alla definizione di quattro modelli comportamentali di base.

I quattro comportamenti



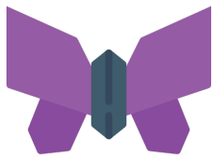
Formica *(Fourmi)*

- Rimangono sempre vicino lo stesso muro, fin quando possibile.



Pesce *(Poisson)*

- Si fermano all’interno di spazi vuoti e visitano pochi quadri.



Farfalla *(Papillon)*

- Si muovono a zig-zag, da una parete a quella opposta.



Cavalletta *(Sauterelle)*

- Passano da un quadro all’altro saltando quelli che sono nel mezzo.

Implementazione in Unity

Cos’è Unity?

Piattaforma di sviluppo cross-platform rilasciata nel 2005 e costantemente aggiornata d’allora.

Permette di progettare ambienti 3D/2D, sia che siano interfacce utente oppure esperienze interattive come videogame.

Perché Unity?

Unity è usato:

- Nel 70% dei contenuti interattivi
- Nel 50% dei videogiochi mobile
- Nel 60% dei contenuti in VR/AR
- Su 3 miliardi di dispositivi nel mondo.

source: unity.com

La struttura del museo

È possibile suddividere la struttura del museo 3D in tre parti principali:

Plane Picture Wall

Plane

Sono la base su cui si basa l’implementazione di tutti i pattern di movimento.

Ogni plane ha al suo interno una griglia che definisce i punti di arrivo per i vari bot; la griglia è creata tramite lo script GridSystem.

Implementazione della griglia (GridSystem)



1. Lo script recupera le dimensioni della Mesh del plane.
2. Tramite i due cicli for sceglie riga e colonna dove posizionare il “point”.
3. Successivamente posiziona il point calcolando le coordinate X e Z

```
//...

Vector3 size = GetComponent<MeshFilter>().mesh.bounds.size;

for ( float x //... )
{
   for ( float z //... )
   {
       GameObject point = Instantiate( gridPoint, //... );
       point.AddComponent<DestinationPoint>();
       point.transform.localPosition = new Vector3((size.x/columns) * x, 0f, (size.z/rows) * z );
       gridPoints.Add( point );
    }
}
```

La classe mette poi a disposizione altri metodi per il recupero di punti di destinazione liberi presenti nella griglia.

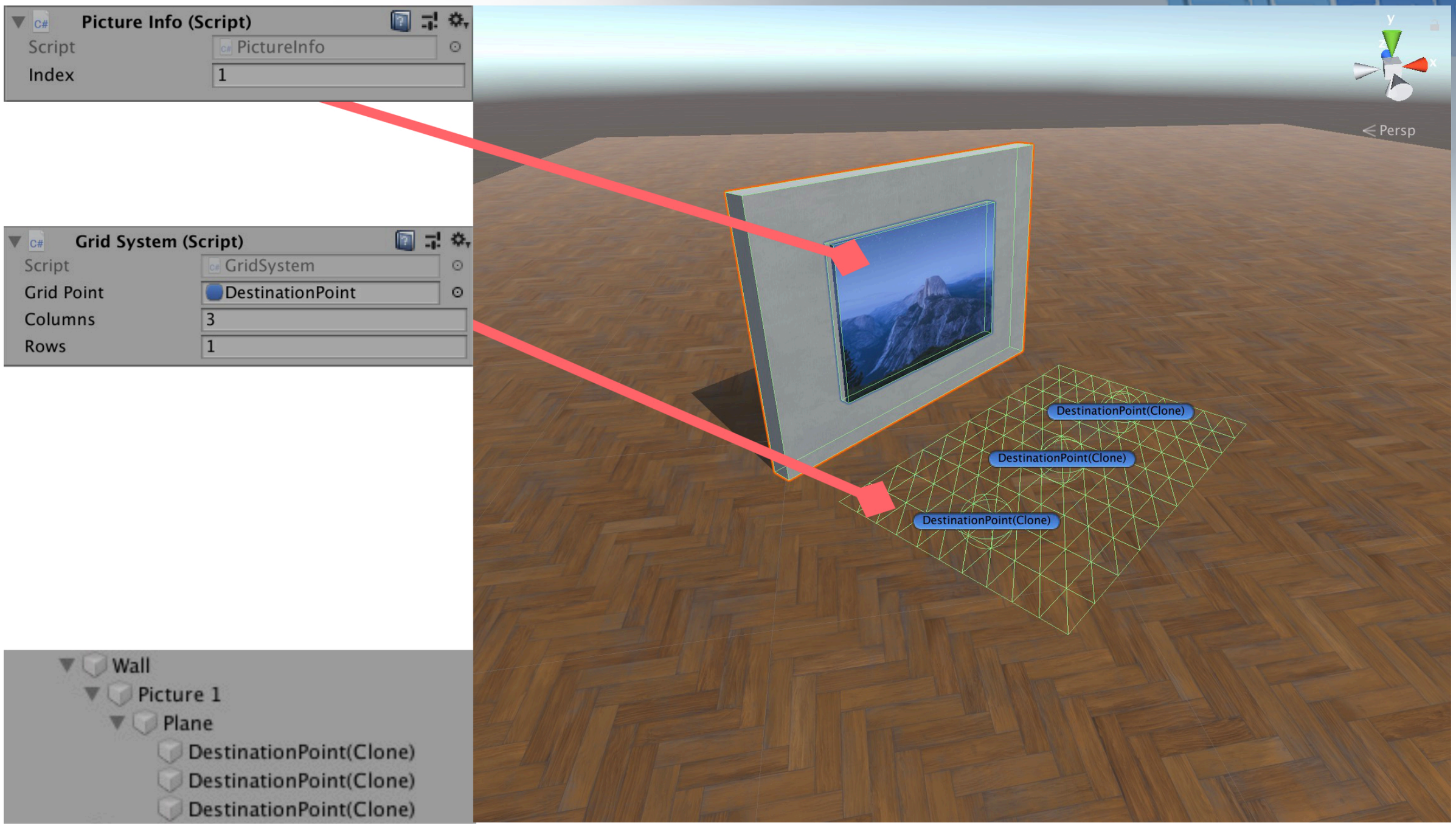
Pictures

Ogni quadro è composto da due GameObject:

- Il primo, “*Picture X*”, ha sempre una componente
  - ***PictureInfo***: definisce l’indice del quadro secondo l’idea dell’espositore

- Il secondo, “*Plane*” (*plane per i quadri), ha sempre una componente*
  - ***GridSystem***: posiziona all’interno del plane un insieme di punti di destinazione in fase di runtime

Risultato finale: wall, quadro e plane con griglia



Animazioni e Modelli 3D

Le animazioni e i modelli 3D dei personaggi sono stati scaricati da Adobe Mixamo.





Animazioni: tolleranze e angoli

La scelta dell'animazione da avviare tiene presente due tipi di parametri:

1. ***Tolleranze***: *tolleranceLeft*, *tolleranceRight*, …
   - Definiscono quando una destinazione è considerata alla sinistra/destra del bot.
2. ***Angoli* di rotazione delle animazioni:**
   - Usati come paragone con l'angolo che la destinazione del bot forma con la direzione (α). Se l'angolo di rotazione è inferiore ad α, allora si attiva l'animazione.

Scelta delle animazioni: esempi

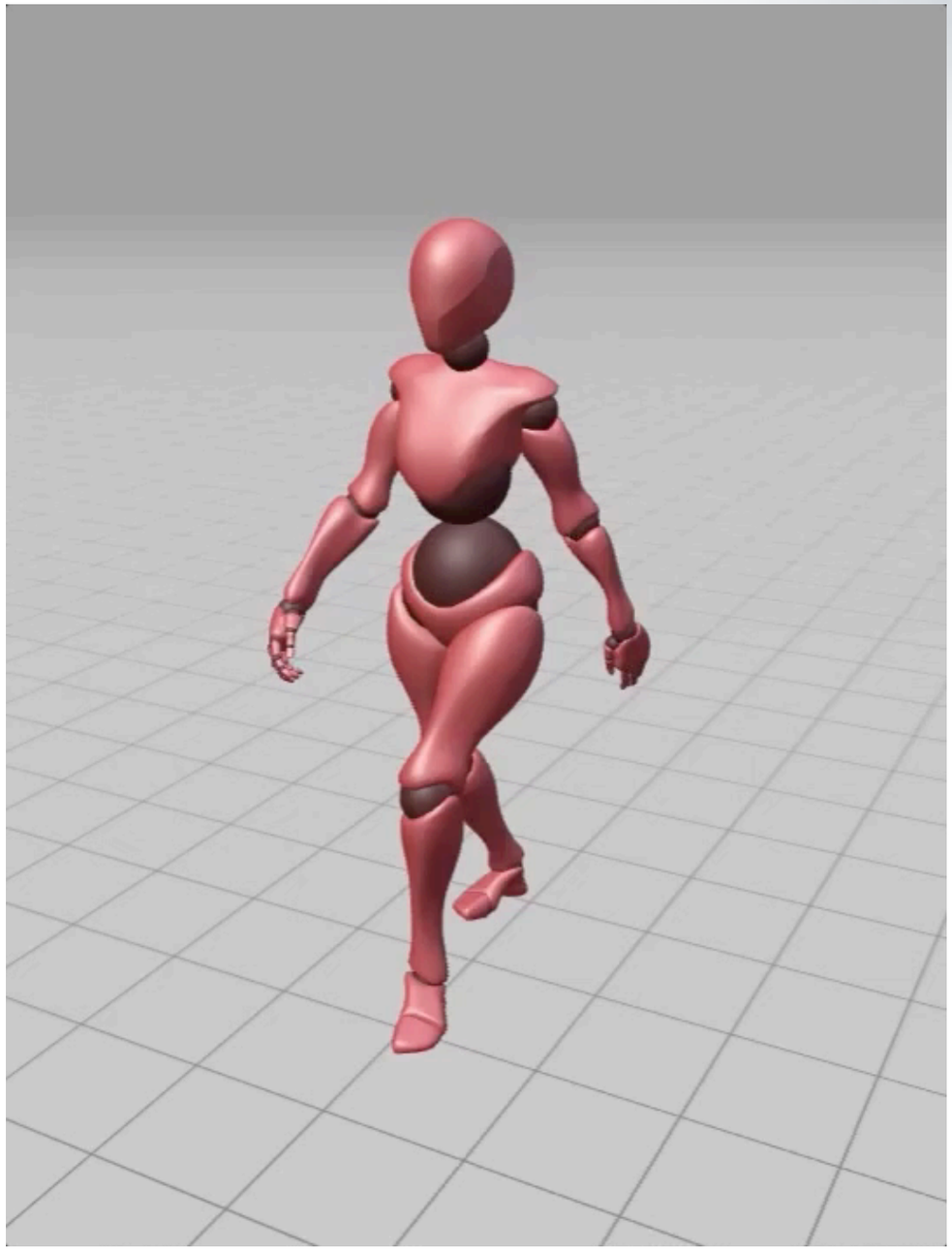
Rotazione verso una destinazione:

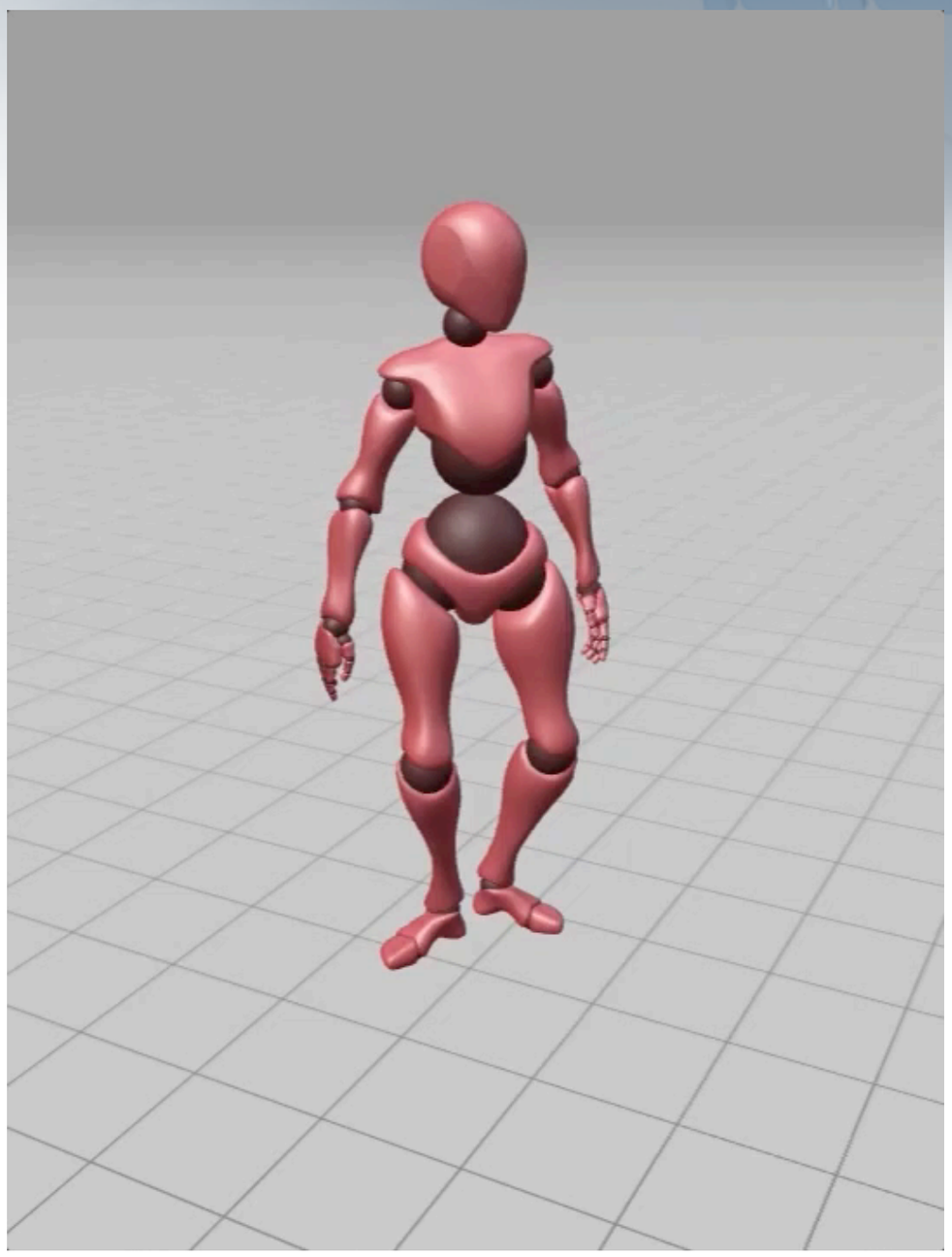
- Viene calcolata la posizione locale (*localPos*) della destinazione relativa a quella del bot.
- localPos viene utilizzata, assieme alle tolleranze, per capire se la destinazione è situata a destra, sinistra o alle spalle del bot e attivare la relativa animazione in base ai vari angoli.

Walk:

- Per il movimento in avanti, viene modificato il valore “speed” della classe CharacterAnimator. Il cambio del valore da 0 a 1 provoca l’avvio dell’animazione di Walk.

Animazioni





Movimento dei personaggi

Per il calcolo del miglior percorso da A a B, si è scelto di utilizzare il metodo NavMesh.CalculatePath, di UnityEngine.AI. Tale metodo genera un **NavMeshPath**, che dispone di un attributo chiamato **Corners.**

L'idea è creare una classe principale che si occupi di:

- generare un nuovo percorso,
- spostare il personaggio attraverso i vari corners,
- attivare le animazioni necessarie (Walk, Turn)

Implementazione dei movimenti: PathManager

Tra le proprietà importanti:

- *pauseTime*: tempo medio da dedicare ad ogni punto di destinazione

- *rigidBodySpeed*: velocità con cui si muoverà il bot da un punto all'altro del percorso

- *isHasty*: definisce se un bot è "frettoloso", ovvero se rimane fermo in attesa che si liberi un posto nella sua prossima destinazione oppure passa alla successiva

Implementazione dei movimenti: PathManager

Il metodo più importante:

- ***GetNextDestination*** (abstract): restituisce il prossimo plane di destinazione, ogni pattern implementerà strategie differenti per tale scelta

Altri metodi:

- UpdatePathAndDestination: pulisce il percorso e recupera la prossima destinazione tramite GetNextDestination.

```
private void UpdatePathAndDestination ( )
{
    path = new NavMeshPath();
    destination = GetNextDestination();
    //...
}
```

Implementazione dei movimenti: PathManager



- *CheckNextDestination*: verifica se nel plane di destinazione sono presenti punti disponibili, quindi calcola il percorso con GeneratePath (utilizza NavMesh.CalculatePath).

```
private void CheckNextDestination ()
{
    //...

    if ( destination.GetComponent<GridSystem>().HaveAvailablePoint() )
    {
        GeneratePath();
        //...
    }
    else
    {
        if( !isHasty )
            inPausa = true;
        else
            UpdatePathAndDestination();
    }
}
```

Implementazione dei movimenti: PathManager



- Update:
  - Se il bot è in pausa, controlla ripetutamente se la destinazione si è liberata.
  - Altrimenti, se il tempo di visita per il quadro attuale è terminato aggiorna destinazione e percorso

```
private void Update ()
{
    if ( inPausa )
        CheckNextDestination();
    else
    {
        if ( timedelta > pauseTime )
        {
            UpdatePathAndDestination();
            timedelta = 0f;
        }

        Walk();
    }
}
```

La classe PathManager



- FollowPath: sposta il bot attraverso i vari corner del percorso attivando l'animazione di camminata

```
private void FollowPath ()
{
    //...

    rigidBody.MovePosition( Vector3.MoveTowards( transform.position, path.corners[ indexCornerPath ],
                                                 Time.deltaTime * rigidBodySpeed ) );
    animationManager.speed = 1f;
    //...

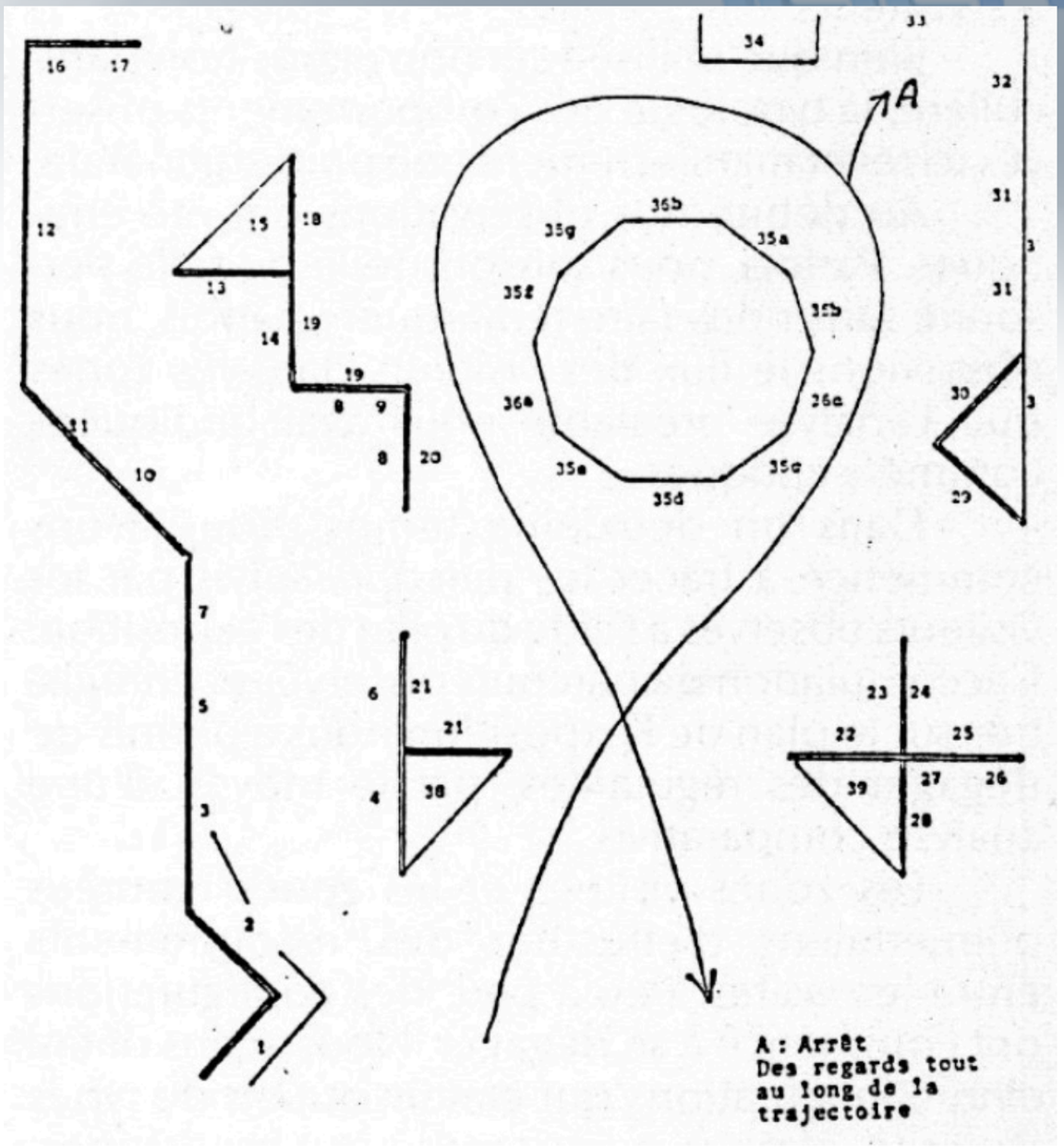
    float distanceFromCorner = Vector3.Distance( transform.position, path.corners[ indexCornerPath ]);

    // Passa al corner successivo
    if ( distanceFromCorner < 0.5f )
    {
       if ( !PathHaveNextCorner() )
       {
          //...
          animationManager.speed = 0f;
        }

    //...
```

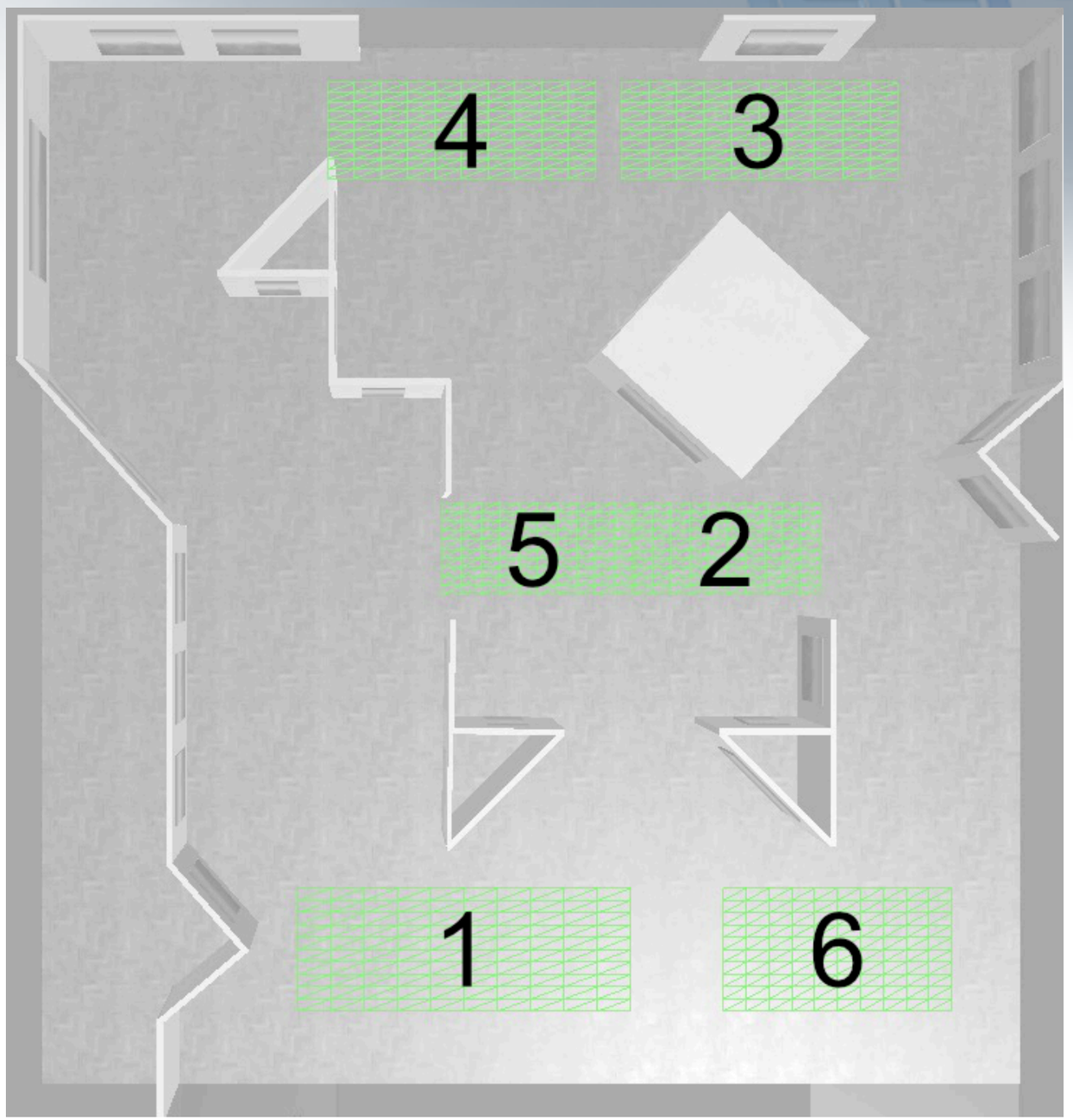
Il pattern Pesce

- Preferisce passare al centro tra due muri.
- Si posiziona spesso in spazi vuoti.
- Media di 5-10 minuti per fermata.
- Osserva direttamente pochi quadri, senza un ordine specifico.



Il pattern Pesce

Per l'implementazione delle prime due scelte, sono stati posizionati sei plane numerati.



Il pattern Pesce

Il comportamento del bot Fish è definito come segue:

A. Il bot sceglie inizialmente se visitare il museo muovendosi dal plane 1 al 6 o viceversa.

B. In maniera random sceglie se visitare un quadro oppure passare al prossimo “Fish Plane”

C. Si dirige verso la destinazione, attende il pauseTime e riparte dal punto B.

Il posizionamento random del bot in ogni griglia e la scelta random dei quadri da visitare permette di generare percorsi sempre differenti.

Il pattern Pesce: classe PoissonPattern

La classe **PoissonPattern estende la classe PathManager** precedentemente descritta.

Abbiamo una proprietà principale:

- *pathPart:* enumeratore della lista contenente i “Fish Plane” ordinati (da 1 a 6 o da 6 a 1)

La classe PoissonPattern

I metodi principali:

- *GetNextDestination*: sceglie in maniera random se restituire un fish plane oppure un quadro.
- *GetPictureDestination:* restituisce il plane del quadro più vicino al bot (lunghezza del percorso, no distanza spaziale)

```
public override GameObject GetNextDestination ()
{
    bool viewPicture = Random.Range( 0, 10 ) > 7 ? true : false;

    if( viewPicture )
        return GetPictureDestination();

    return GetFishPlaneDestination();
}
```

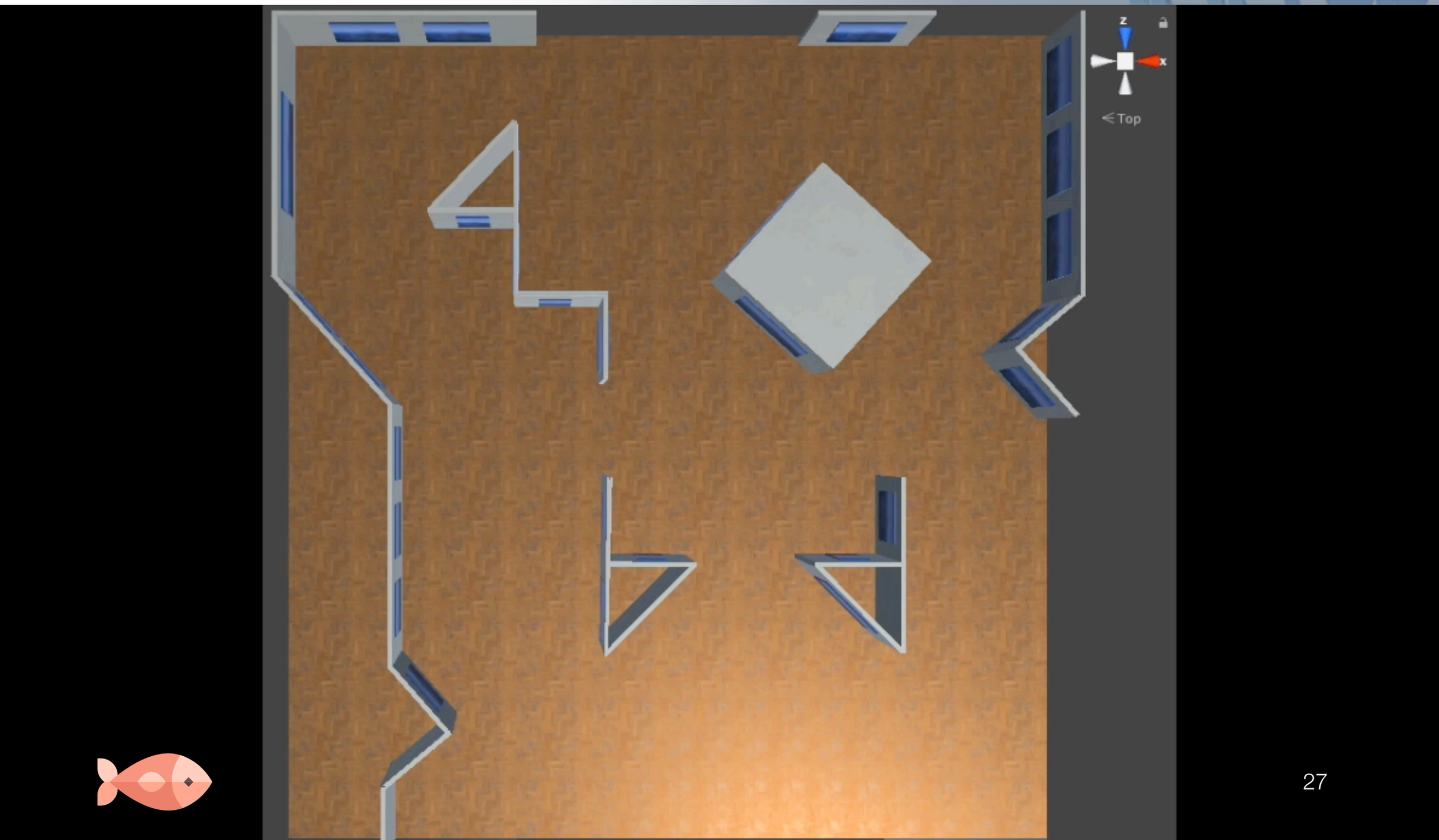
La classe PoissonPattern

- *GetFishPlaneDestination: scorre l'enumeratore pathPart e* restituisce il prossimo fish plane oppure l'uscita
- *GetPositionInFloorPicture*: recupera un punto random disponibile all'interno del plane di destinazione, lo occupa e ne restituisce la posizione

```
public override Vector3 GetPositionInFloorPicture ()
{
    destinationPoint = destination.GetComponent<GridSystem>().GetAvailableRandomPoint();
    destinationPoint.GetComponent<DestinationPoint>().Occupa();

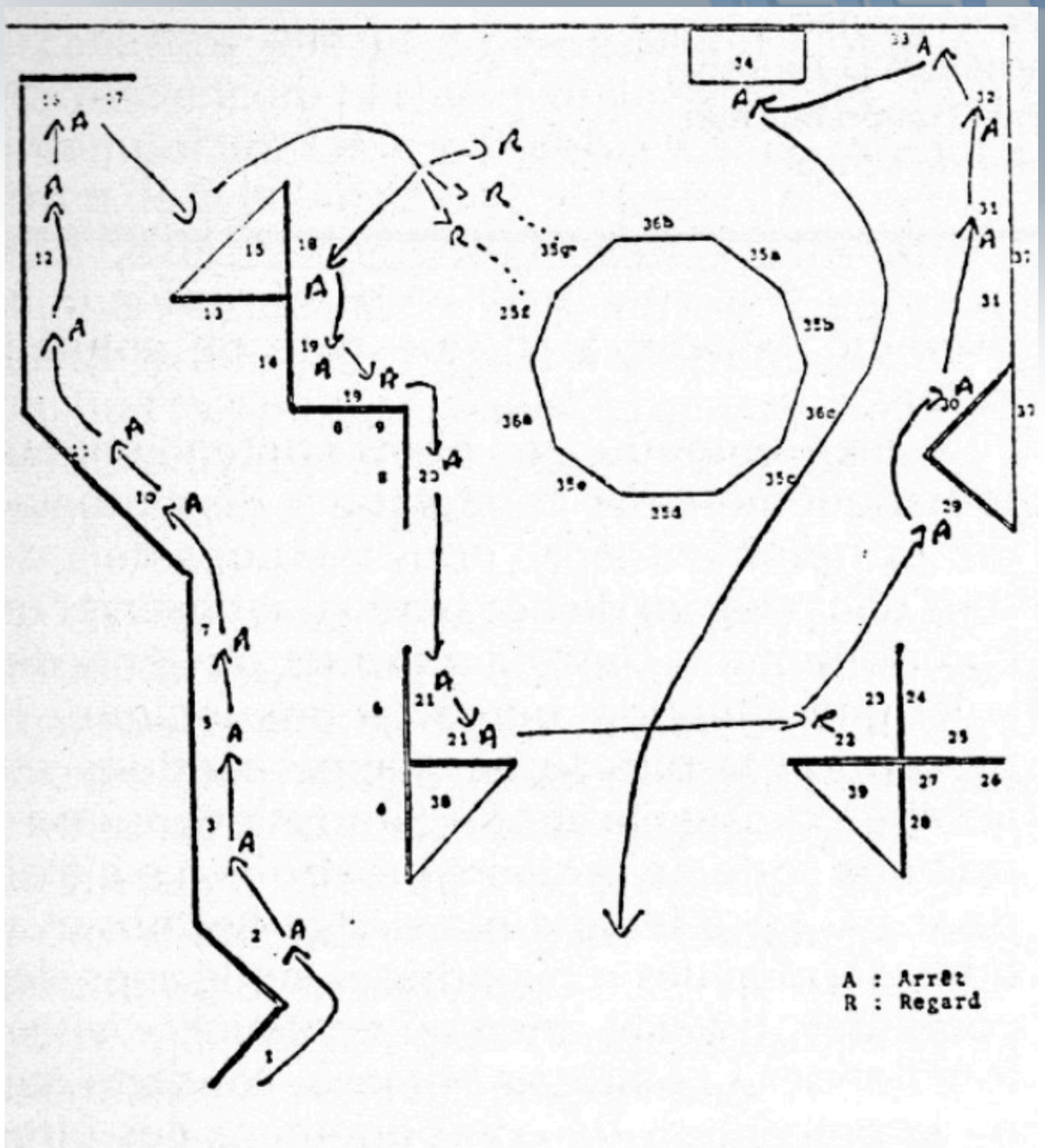
    return destinationPoint.transform.position;
}
```

Risultato finale PoissonPattern



Il pattern Formica

- Resta “attaccato” allo stesso muro fin quando ne ha la possibilità.
- Media di 20 minuti per quadro.



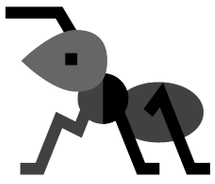
Il pattern Formica

L’idea per l’implementazione di questo pattern è:

A. Continua a visitare i quadri sul muro attuale.

B. Se sono terminati, seguendo l’ordine cronologico, visita i quadri con indice superiore a quello corrente presenti sul muro che tocca il muro attuale.

C. Altrimenti, dirigiti verso il quadro successivo cronologicamente e poi prosegui da A.



Il pattern Formica: classe FourmiPattern

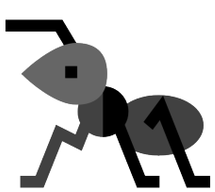
Tra gli attributi rilevanti nell'implementazione, troviamo:

- *walls*: lista con tutti i muri del museo che hanno almeno un quadro
- *picturesOnWalls*: dizionario (K=Wall, V=Pictures non visitate)

Il pattern Formica: classe FourmiPattern

I metodi più importanti:

- *GetNextDestination*:
  - Controlla se sul muro corrente è presente un quadro successivo a quello attuale e lo restituisce.
  - Altrimenti, *verifica se esiste un muro con quadro successivo a quello attuale, dando priorità ai muri vicini. In caso positivo* restituisce il quadro con indice più piccolo sul muro scelto.
  - Altrimenti restituisce l'uscita.



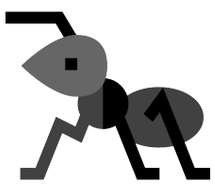
La classe FourmiPattern




```
public override GameObject GetNextDestination ()
{
    if( numberOfStop <= 0 )
        return GetPlaneOfExit();

    if ( MoveToNextPicOnCurrentWall() )
    {
        //...
        numberOfStop -= 1;
    }
    else if ( MoveToNextPicOnAnotherWall() )
    {
        //...
        numberOfStop -= 1;
    }
    else{
        return GetPlaneOfExit();
    }

    return GetPlaneOfCurrentPicture();
}
```



La classe FourmiPattern



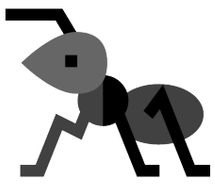
- *NextPictureIsInClosestWall: controlla* se esiste o meno un muro che si interseca con currentWall e che abbia quadro più piccolo successivo all'attuale. Imposta poi questa parete come corrente.

```
private bool NextPictureIsInClosestWall ()
{
    List<GameObject> wallsIntersectCurrentWall = GetWallsClosestToCurrent();

    if ( wallsIntersectCurrentWall.Count > 0 )
    {
        GameObject wallWithMinPicIndex = wallsIntersectCurrentWall[ 0 ];
        GameObject picWithMinIndex = picturesOnWalls[ wallWithMinPicIndex ][ 0 ];

        if ( picWithMinIndex.GetComponent<PictureInfo>().index >= currentPictureIndex )
        {
            currentWall = wallWithMinPicIndex;
            return true;
        }
    }

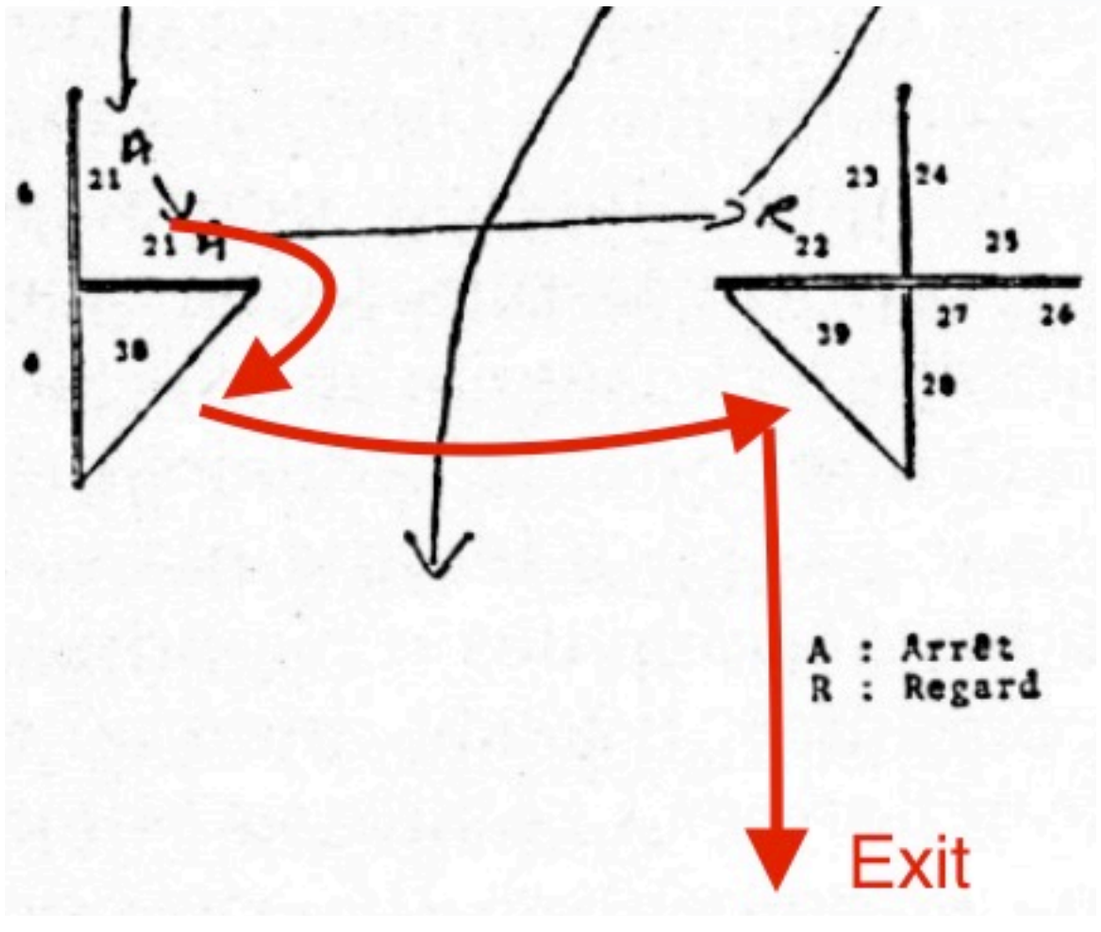
    return false;
}
```



Il pattern Formica: classe FourmiPattern

- *GetWallsClosestToCurrent*, restituisce la lista dei muri che toccano quello corrente e che hanno indice del più piccolo quadro inferiore a currentPictureIndex + 10.

+10 serve per evitare di considerare quadri che sono cronologicamente molto distanti rispetto all'attuale



La classe FourmiPattern




```
private List<GameObject> GetWallsClosestToCurrent ()
{
    List<GameObject> intersectsWalls = new List<GameObject>();

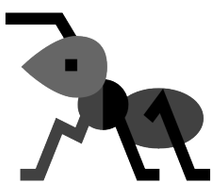
    foreach ( GameObject wall in walls )
    {
        //...

        Bounds currentWallBounds = currentWall.GetComponent<MeshRenderer>().bounds;
        Bounds wallBounds = wall.GetComponent<MeshRenderer>().bounds;

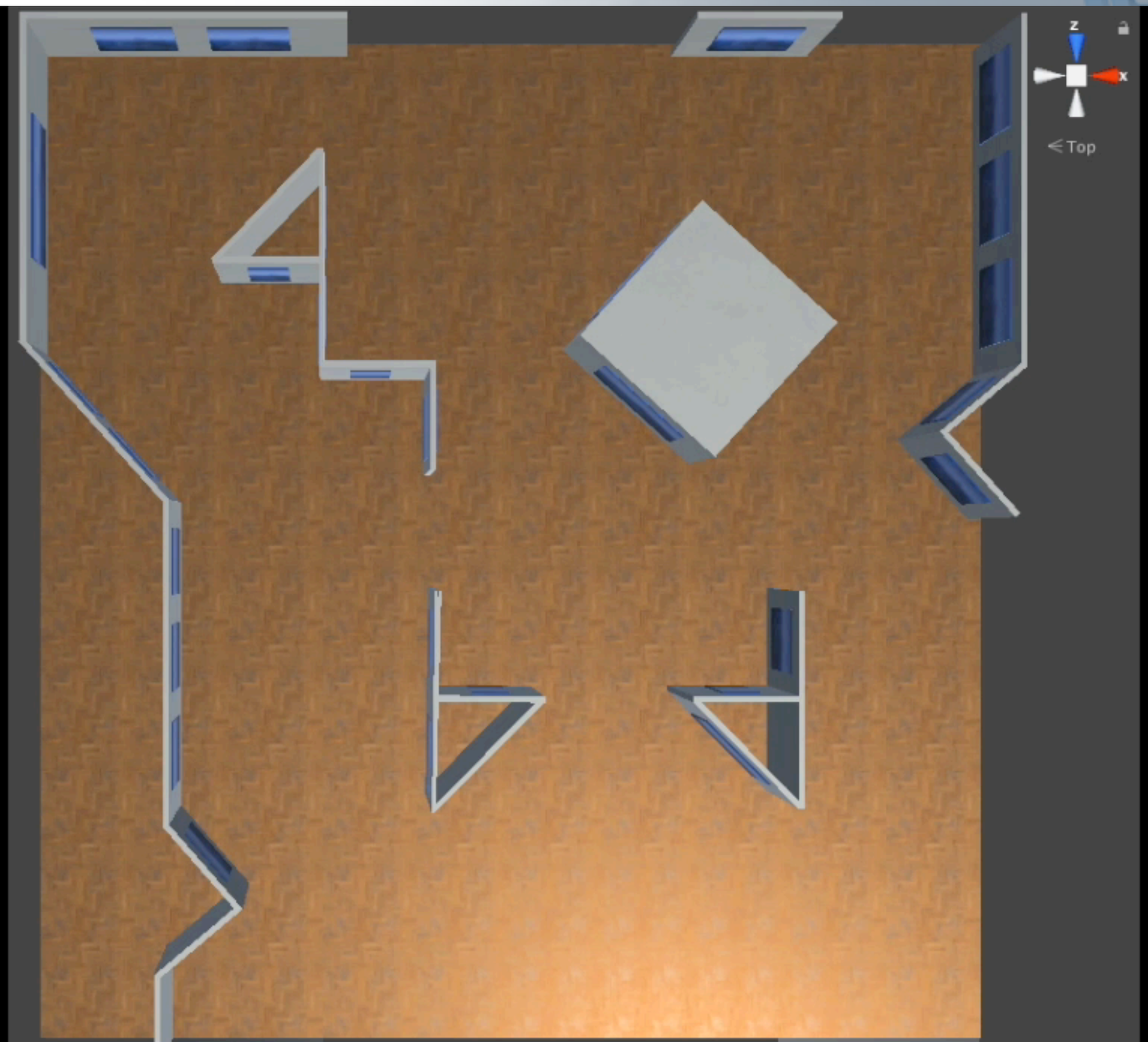
        if ( currentWallBounds.Intersects( wallBounds ) )
            if( picWithMinIndexOnWall.GetComponent<PictureInfo>().index < currentPictureIndex + 10 )
                intersectsWalls.Add( wall );
    }

    intersectsWalls.Sort( SortByIndexPictureInWalls );

    return intersectsWalls;
}
```

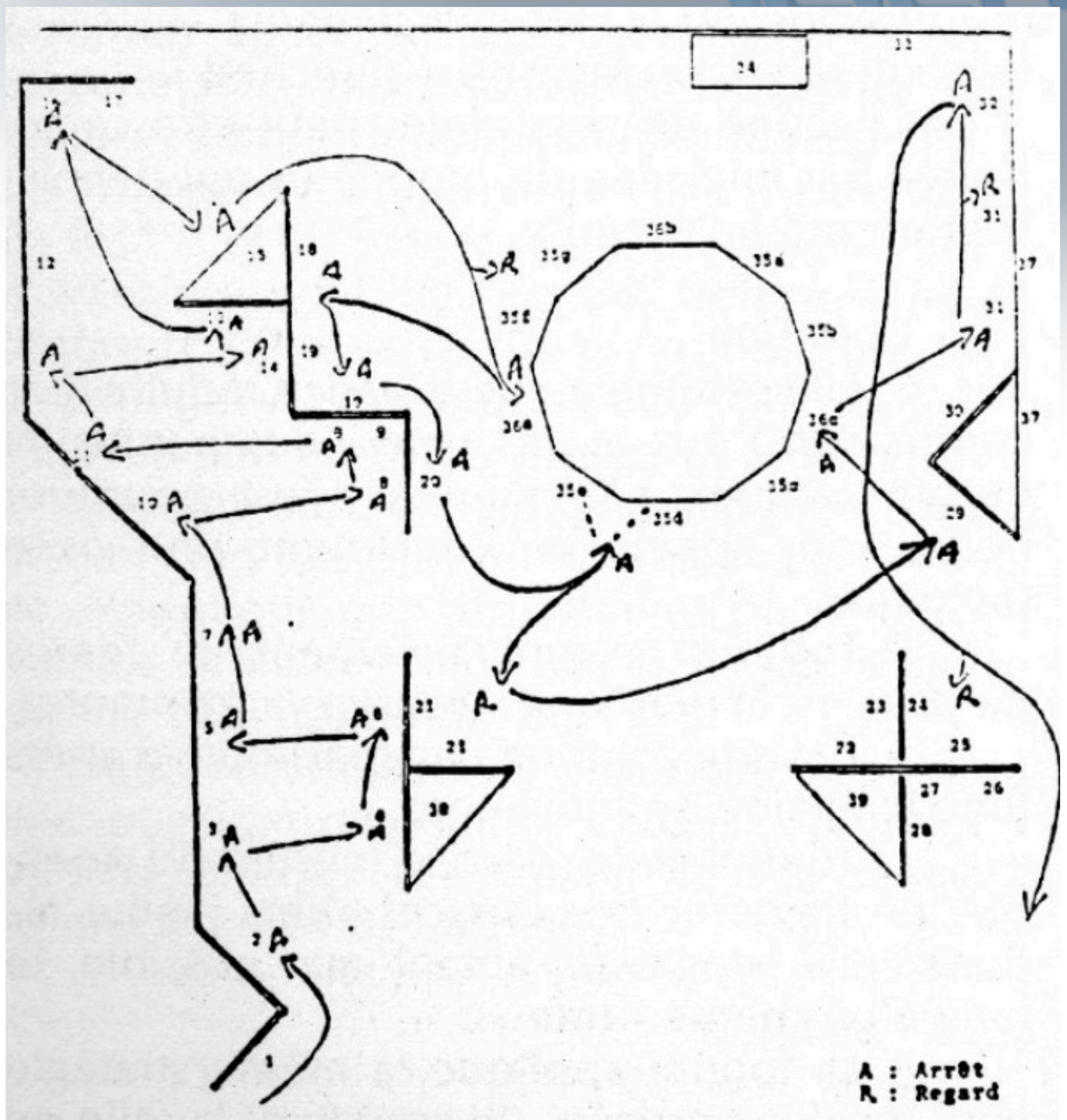


Risultato finale FourmiPattern



Il pattern Farfalla

- La sua visita è contraddistinta da un movimento a zig-zag.
- Cerca sempre di seguire l'ordine cronologico, per quanto possibile.
- Media di 5-10 minuti per quadro.

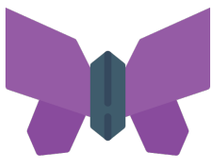


Il pattern farfalla

Per permettere al bot di verificare se alle sue spalle sia o meno presente un quadro si sono usati tre **Physics.RayCast**.

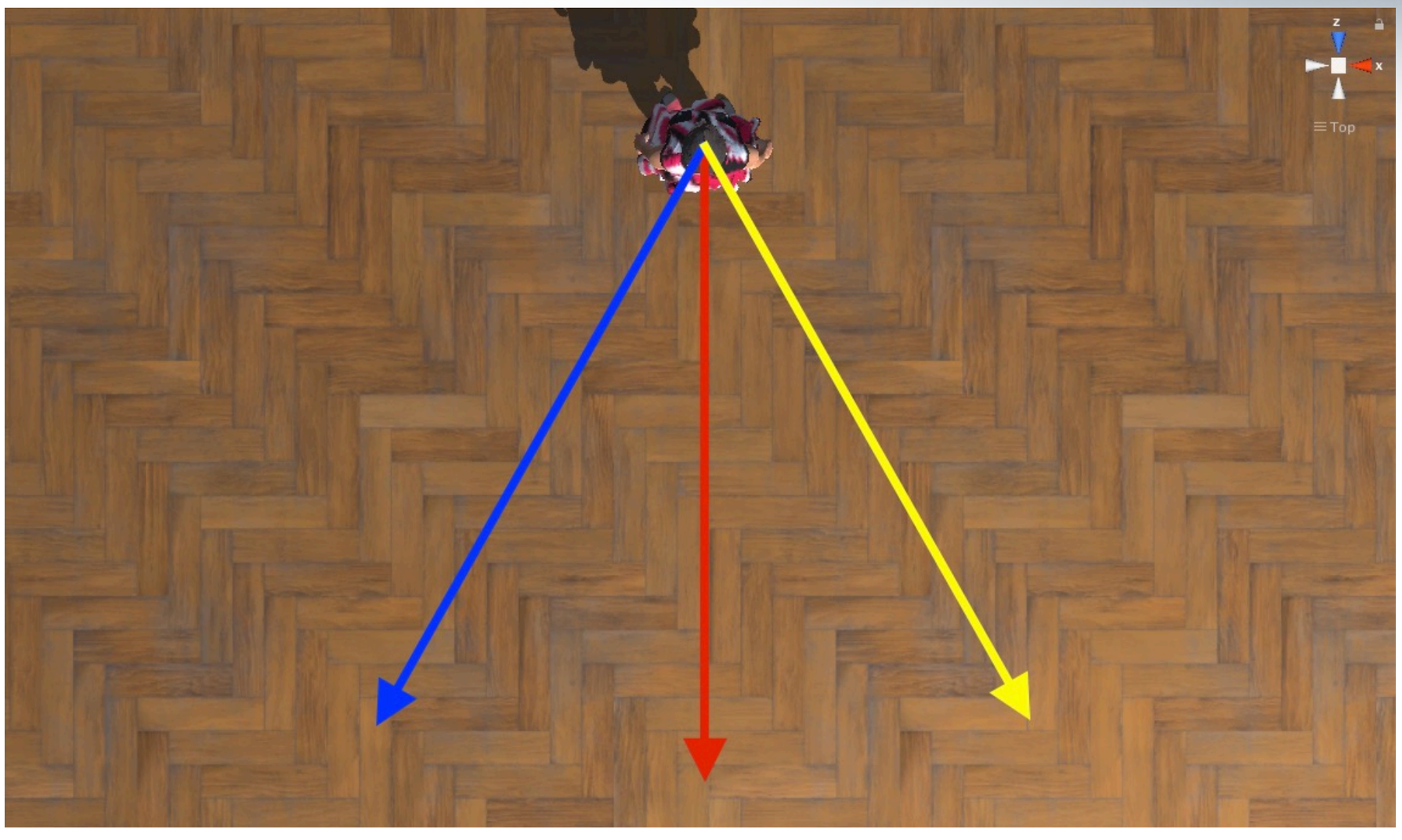
Se uno di questi colpisce un muro avente un quadro con indice superiore a quello attuale, allora il bot si dirigerà verso il quadro non visitato più vicino presente su tale muro.

Altrimenti visiterà il quadro successivo a quello attuale.



Il pattern farfalla

Direzioni RayCast:



Il pattern farfalla: la classe PapillonPattern

Tra gli attributi più importanti della classe **PapillonPattern**, troviamo:

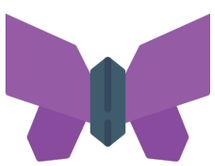
- *walls:* lista con tutti i muri del museo aventi almeno un quadro

- picturesOnWall: dizionario del tipo (K=Wall, V=Pictures non visitate)

Il pattern farfalla: la classe PapillonPattern



- GetNextDestination, effettua le seguenti operazioni:
  - Restituisce il plane di un quadro alle spalle del bot tramite l'uso di LookInBackward.
  - Se LookInBackward fallisce, restituisce il plane del prossimo quadro nel museo tramite l'uso di LookNextIndex.
  - Restituisce l'uscita se tutti i precedenti metodi di ricerca falliscono oppure è terminato il numero di quadri.



La classe PapillonPattern

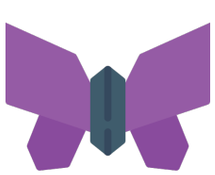



```
public override GameObject GetNextDestination ()
{
    if ( numberOfStop <= 0 )
        return GetPlaneOfExit();

    if ( LookInBackward() )
    {
        //...
        return nextDestination;
    }

    if ( LookNextIndex() || //... )
    {
        //...
        return nextDestination;
    }

    return GetPlaneOfExit();
}
```



La classe PapillonPattern



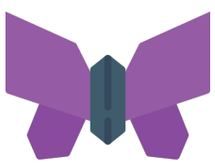
- LookInBackward:
  - cerca, utilizzando i tre RayCast, i muri alle spalle del bot.
  - Delega poi la scelta del quadro a SelectNextPicInBackwardWalls

```
private bool LookInBackward ()
{
    Vector3[ ] directions =
    {   -transform.forward,
        Quaternion.AngleAxis( 150, transform.up ) * transform.forward,
        Quaternion.AngleAxis( 210, transform.up ) * transform.forward,
    };

    //...

    if ( Physics.Raycast( transform.position, direction, ...) )
    {
        //...
        considerateWall.Add( hit.collider.gameObject );
    }

    return SelectNextPicInBackwardWalls(considerateWall);
}
```



La classe PapillonPattern



- SelectNextPicInBackwardWalls
  - Sceglie come prossimo quadro quello più vicino presente sul muro che tra tutti quelli considerati ha il quadro con indice più piccolo.

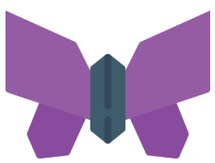
```
private bool SelectNextPicInBackwardWalls(List<GameObject> considerateWall)
{
    considerateWall.Sort( SortByIndexPictureInWalls );

    //...

    List<GameObject> consideratePics = picturesOnWall[ considerateWall[ 0 ] ];
    consideratePics.Sort( Distanza );

    nextDestination = mostClosePicture.transform.GetChild( 0 ).gameObject;

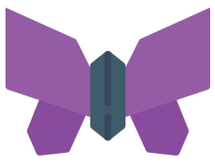
    //...
}
```



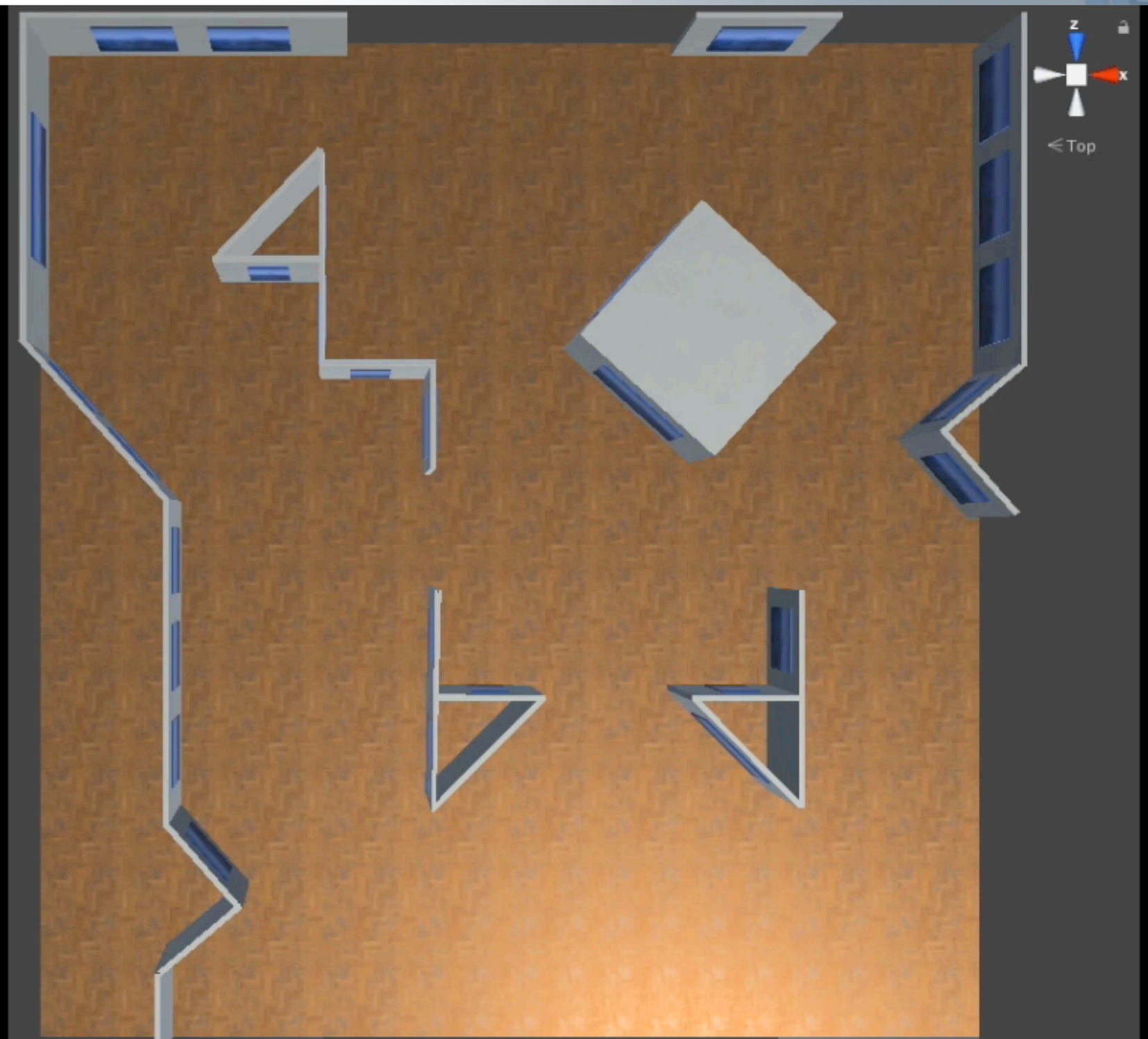
La classe PapillonPattern

- LookNextIndex, verifica la presenza di un quadro visitabile
  - Sceglie come quadro quello con index superiore all'attuale ma allo stesso tempo minore della somma (currentPictureIndex+maxJump)

```
private bool LookNextIndex (int maxJump = 5)
{
    //...
    foreach(GameObject pic in pics )
    {
        if( pic.GetComponent<PictureInfo>().index > currentPictureIndex &&
            pic.GetComponent<PictureInfo>().index < currentPictureIndex + maxJump )
        {
             nextDestination = pic.transform.GetChild( 0 ).gameObject;
             currentPictureIndex = pic.GetComponent<PictureInfo>().index;
             //...
             return true;
        }
     //...
```

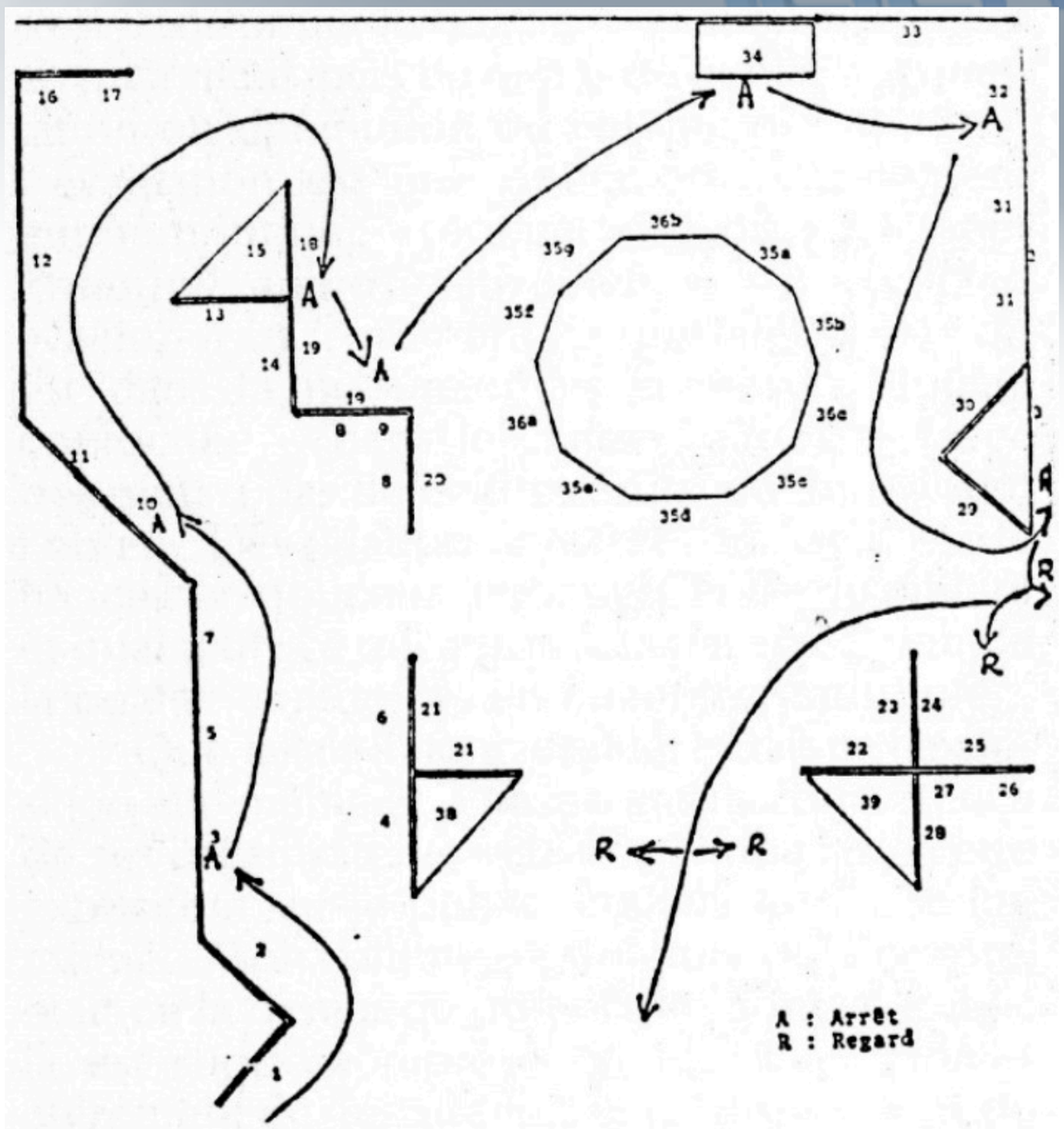


Risultato finale PapillonPattern



Il pattern Cavalletta

- Avanza da un’attrazione all’altra ignorando tutto ciò che è nel mezzo.
- Media di 5 minuti per quadro.
- Visita poche esposizioni e non tiene conto del loro ordine.



Il pattern Cavalletta: classe SauterellePattern



La classe **SauterellePattern** ha solo due attributi degni di nota:

- *picturesToWatch:* enumeratore di tutte le esposizioni che il bot visiterà
- *maxJump*: numero massimo di quadri che il bot può ignorare passando da un quadro all'altro



Il pattern Cavalletta: classe SauterellePattern

I metodi più importanti sono:

- *SetPictureToWatch*: sceglie le attrazioni che il bot visiterà.
  Un'attrazione viene scelta se:
  - Un valore random > di un valore definito, OR
  - Il quadro preso in considerazione ha un indice superiore alla somma dell'indice dell'ultimo quadro aggiunto alla lista più maxJump. Evita di saltare troppi quadri.

- *GetNextDestination:* restituisce la prossima destinazione,
  - Scorre picturesToWatch e restituisce il valore corrente
  - Se sono terminati gli elementi restituisce l'uscita del museo



La classe SauterellePattern




```
private void SetPictureToWatch ()
{
    //...

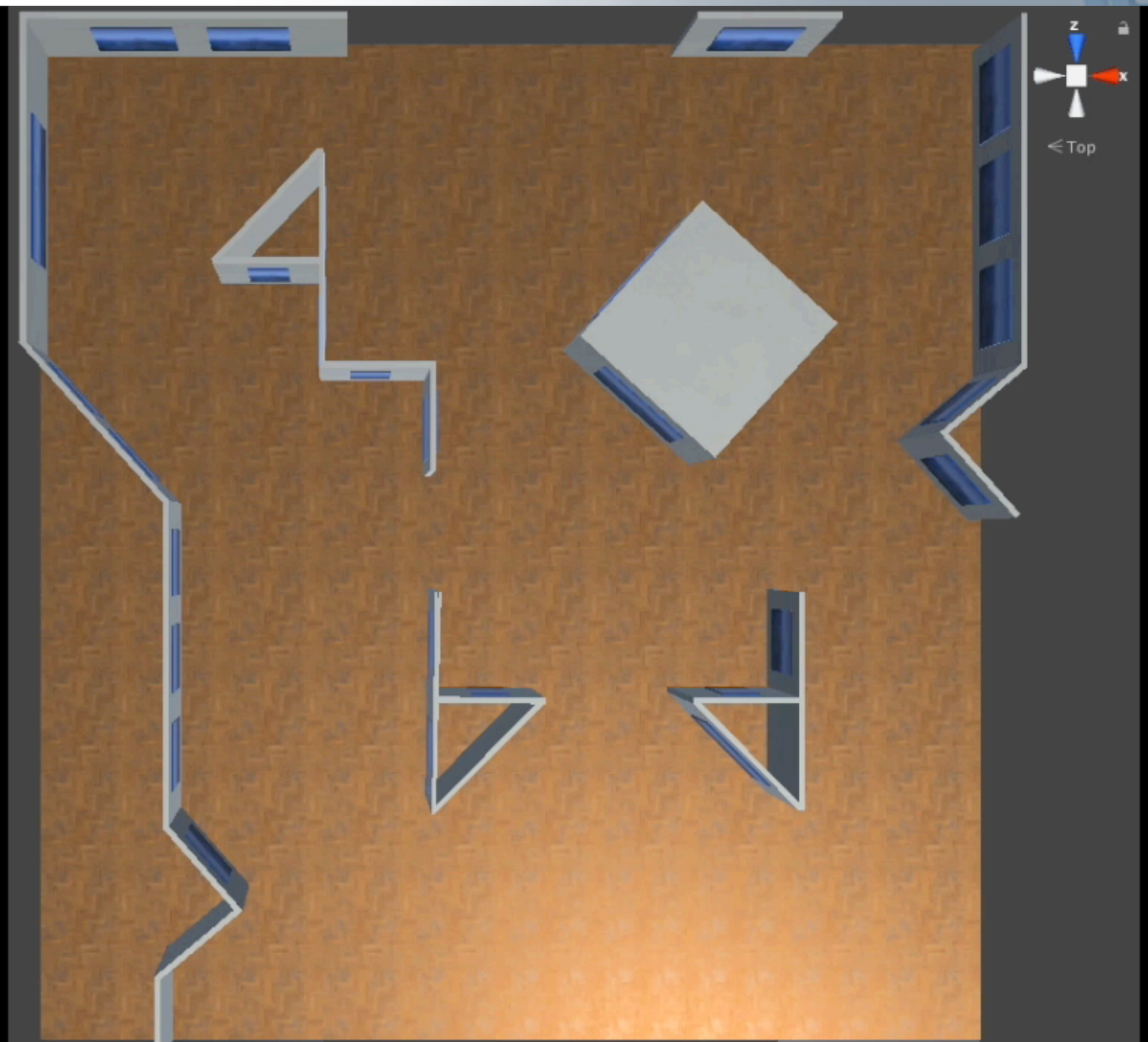
    foreach ( GameObject picture in pictures )
    {
        int pictureIndex = picture.GetComponent<PictureInfo>().index;

        if ( Random.Range( 0, 10 ) > 6 || IsMaxJump( pictureIndex, lastPictureIndexAdded ) )
        {
            lastPictureIndexAdded = picture.GetComponent<PictureInfo>().index;
            picturesToWatch_list.Add( picture );
        }
    }

    picturesToWatch = picturesToWatch_list.GetEnumerator();
}
```



Risultato finale SauterellePattern



Conclusioni

Si conclude quindi la lista dei pattern presenti nel paper.

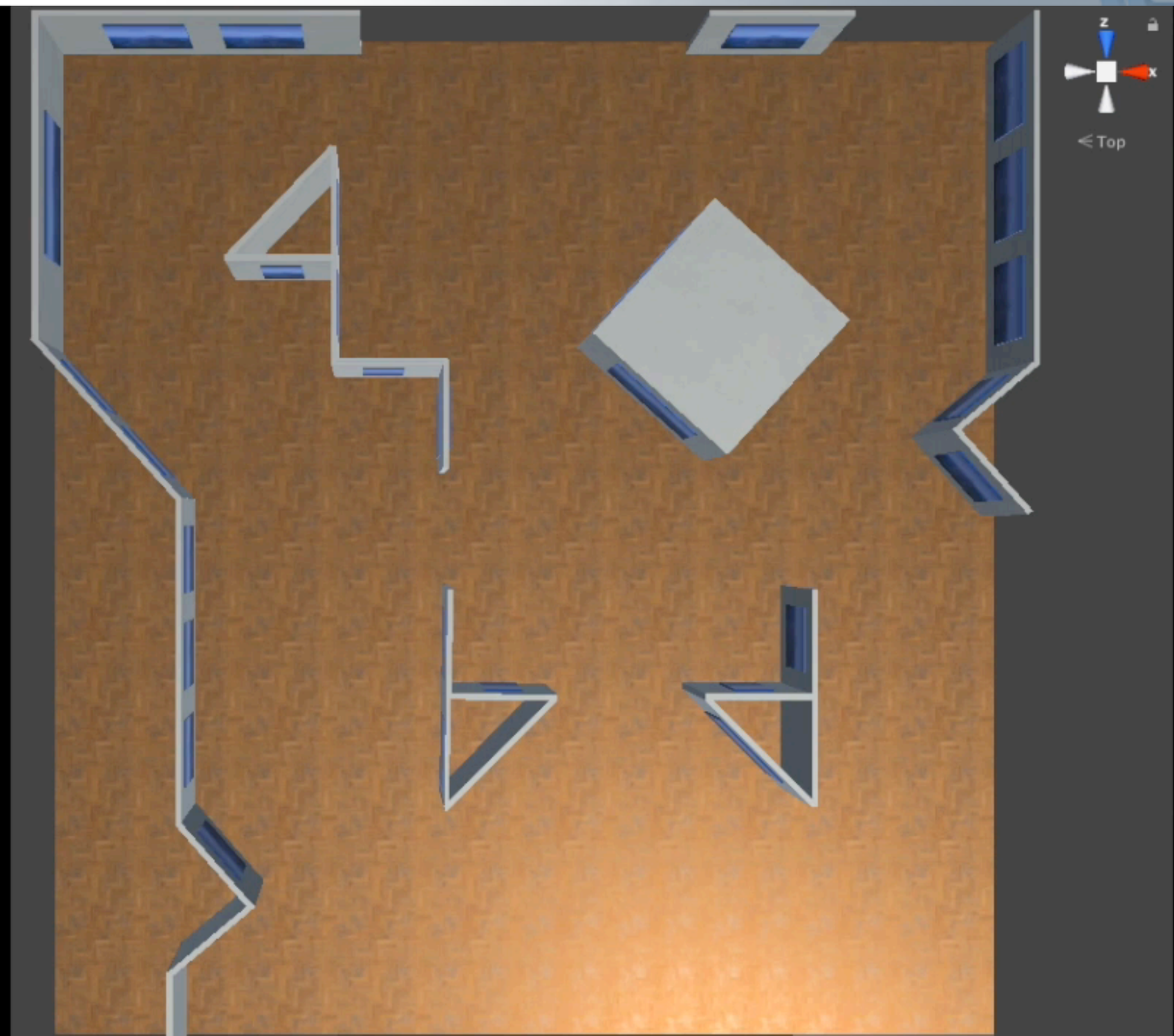
Abbiamo visto come tutti i concetti chiave, come la gestione delle animazioni, dei movimenti di base ed infine dei punti di destinazione, siano separati dalle implementazioni dei singoli pattern.

Questo tipo di gestione offre una maggiore flessibilità al progetto che in futuro potrebbe:

- Ampliare il numero di pattern comportamentali
- Aggiungere nuove animazioni più precise e studiate ad-hoc
- ...

Tutto ciò senza modificare le classi dei pattern principali che continueranno comunque a funzionare.

Risultato finale



Sviluppi futuri?

La stanza 3D implementata in questo progetto fa riferimento ad un’esposizione tenuta all’interno della Biblioteca Pubblica di Informazioni (BPI) in Francia.

Nulla vieta, per sviluppi futuri, di sostituire il museo di base con un ambiente 3D raffigurante un diverso museo/stanza e analizzare i comportamenti dei vari bot a fini statistici o per la gestione delle emergenze.



Grazie per l’attenzione.

Domenico Esposito
mail@domenicoesposito.it

LinkedIn: linkedin.com/in/domenicoesposito