

Rapport : Projet de CRV



Cindy Even

Guillaume Biannic

Simulation de la visite guidée d'un musée virtuel.

Master Recherche en Informatique

c9even@enib.fr

g9bianni@enib.fr

15/01/2015

Table des matières

Introduction :	2
Installation :	2
1) Présentation de Unity3D	2
2) Présentation de l'interface	4
3) Création du musée :	5
a) Objets 3D constituant la scène :	5
b) Déplacements de l'utilisateur	7
c) Graphe de navigation	8
4) Faire se déplacer un visiteur virtuel :	9
a) Visiteur virtuel	9
b) Algorithme Breadth First Search	11
c) Algorithme A*	12
d) Algorithme A* sans lien	12
5) Faire se déplacer un groupe de visiteurs virtuels vers UN tableau :	13
6) Faire se déplacer un groupe de visiteurs virtuels vers DES tableaux :	15
7) Créer une visite (1)	15
8) Créer une visite (2)	16
9) Créer une visite (3)	17
Conclusion	18

Introduction :

Dans le cadre du module de CRV (Comportements Anthropomorphes en Réalité Virtuelle et Augmentée) il nous a été demandé de développer une simulation afin de mettre en pratique les notions étudiées en cours.

Voici l'énoncé du projet :

On considère un musée contenant une quarantaine de tableaux (ou posters, vieilles cartes postales, ...). Il s'agit de guider l'utilisateur dans ce musée en tenant compte de ses préférences. Plutôt que d'utiliser un système de fléchage (dynamique ou non) ou d'incarner un système de guidage par un acteur virtuel, on préfère ici utiliser les autres visiteurs (représentés par des acteurs virtuels) pour inciter l'utilisateur à se diriger vers un tableau plutôt qu'un autre.

L'hypothèse qui est faite ici est la suivante :

Un attroupement de visiteurs devant un tableau doit inciter l'utilisateur à aller vers ce tableau.

Nous avons choisi de travailler sur ce projet à l'aide du logiciel Unity3D. Dans ce rapport nous vous présentons le travail que nous avons effectué ce semestre sur ce projet. Dans un premier temps nous vous présentons le logiciel Unity3D et les différents composants du logiciel que nous avons utilisé dans notre application. Ensuite nous expliquons comment utiliser l'interface de la simulation. Enfin nous consacrons une partie pour chacune des questions du projet qui nous a été posé en détaillant les différentes méthodes et algorithmes que nous avons utilisés pour y répondre.

Installation :

Pour installer l'application, vous devez d'abord extraire au même emplacement le fichier *MuseumVisit_linux_x86_64.x86_64* et le dossier *MuseumVisit_linux_x86_64_Data* de l'archive *Executable_Linux_x86_64.zip*. Il faut ensuite donner les permissions nécessaires.

Pour cela il faut ouvrir un terminal et se placer là où le fichier et le dossier ont été extrait puis saisir la commande `chmod +x ProjetCRV_Even_Biannic.x86_64`. L'application peut alors être lancée avec la commande `./ProjetCRV_Even_Biannic.x86_64`.

1) Présentation de Unity3D

Unity3D est un système multiplateforme de création de jeu vidéo développé par Unity Technologies. Ce système comprend un moteur de jeu et un environnement de développement intégré (IDE) et utilise entre autre le moteur physique PhysX de Nvidia. Afin de faciliter la compréhension du travail que nous avons réalisé, voici un petit glossaire comprenant différents éléments importants utilisés dans Unity3D :

- **GameObjet (GO)** : ce sont les entités les plus importantes dans Unity3D. Tout objet dans l'application est un GO, cependant, ils ont besoin de composants (qui héritent de la classe Component) pour se différencier les uns des autres. On peut voir un GO comme une boîte

vide à laquelle il faut ajouter des composants pour en faire un personnage, une lumière, ou des effets spéciaux par exemple.

- **Component** : les composants (ou Components) sont les pièces fonctionnelles d'un GO. Certains existent déjà dans Unity3D, mais il est possible de créer nos propres composants à l'aide des Scripts.

Voici la liste des composants utilisés dans notre application :

- **Transform** : c'est le seul composant que tout GO possède. Il permet de définir la position, la rotation et l'échelle du GO dans la scène. Ce composant permet aussi le *Parenting*, ce concept permet d'attacher un GO à un autre GO, le premier étant alors enfant du second. Un enfant héritera alors du mouvement et de la rotation de son parent.

- **Camera** : il permet de capturer et d'afficher le monde à l'utilisateur. Il peut y avoir plusieurs caméras dans la scène mais une seule peut être active à la fois. Pour la visite du musée nous avons deux points de vue (et donc deux caméras). L'un est le point de vue du visiteur (First Person View) (Fig. 1) et le second permet de voir le musée en entier (Fig. 2). Pour réaliser un FPV il suffit d'attacher la caméra au visiteur et de la placer au niveau des yeux.

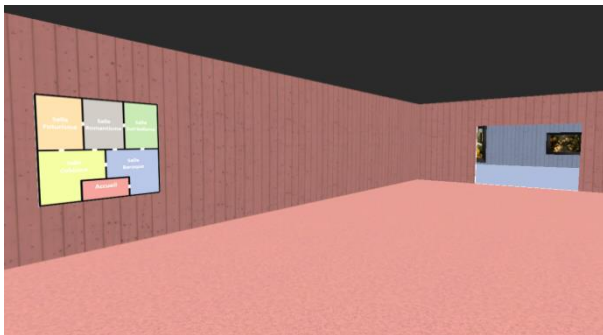


Fig. 1 Caméra en vue Visiteur

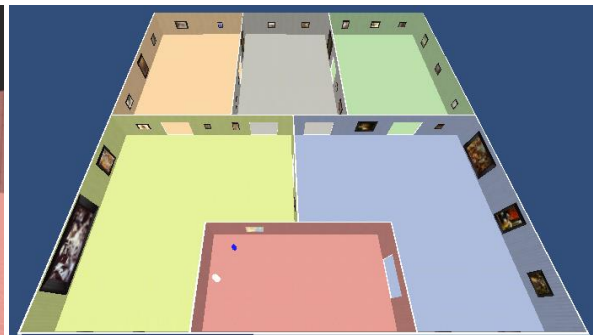


Fig. 2 Caméra en vue Globale

- **Scripts** : comme dit précédemment, Unity nous permet de créer nos propres composants à l'aide de scripts. Pour cela il suffit de créer une nouvelle classe héritant de *MonoBehaviour*. Nous pourrions alors utiliser les méthodes *Start()*, *Update()* et autres pour implémenter les fonctionnalités voulues. Voici quelques fonctions utilisées :

- *Start()* est appelée à la toute première frame.
- *Update()* est appelée toutes les frames.
- *OnGUI()* est appelée pour le rendu et la gestion des événements de l'interface graphique.

Nous pouvons aussi créer des Scripts contenant des classes n'héritant pas de *MonoBehaviour* et ne seront alors pas considérés comme des composants mais comme des classes normales.

- **Rigidbody** : c'est le composant principal qui permet à un objet d'être soumis aux lois de la Physique. En ajoutant ce composant à un objet, ce dernier sera immédiatement répondre à la gravité. Il nous permet aussi d'appliquer des forces sur l'objet.


- **Collider** : ce composant permet de définir la forme utilisée pour les collisions. Plusieurs formes sont possibles comme un cube, une sphère, un cylindre, ou même la forme même de l'objet : le mesh. Plus la forme est simple et plus les calculs de collisions seront rapides.

Il est aussi possible d'utiliser le moteur physique pour détecter quand un Collider pénètre dans un autre Collider sans créer de collision. Pour cela il suffit d'utiliser la propriété Trigger du Collider. L'objet ne se comporte alors pas comme un objet solide et permettra à d'autres Colliders de passer à travers. Quand un Collider entre dans l'espace défini par un autre Collider, ceci appelle la fonction `OnTriggerEnter()`.

- **Tag** : Un tag est un mot qui permet d'accéder à un ou plusieurs GO. Nous pouvons les utiliser dans les Scripts pour trouver un GO qui contient le Tag souhaitée. Ce résultat est obtenu en utilisant la fonction `GameObject.FindWithTag()`.

Ces explications devraient être suffisantes pour comprendre la suite de notre travail, mais pour des informations complémentaires il est possible de consulter la documentation d'Unity3D : <http://docs.unity3d.com/ScriptReference> et <http://docs.unity3d.com/Manual>.

2) Présentation de l'interface

Une fois l'application lancée, une fenêtre s'ouvre et vous invite à choisir la configuration qui vous convient. Une fois la configuration faite, cliquez sur le bouton .

Le menu principal s'affiche alors (Fig. 3). A partir de ce menu vous pouvez accéder aux différentes parties de la simulation et quitter l'application. Afin que vous puissiez voir notre travail étape par étape, nous avons séparé les questions. A chaque question correspond un bouton dans le menu. Pour voir notre travail sur la première question : Faire se déplacer un visiteur virtuel, il suffit de cliquer sur le bouton associé et ainsi de suite.

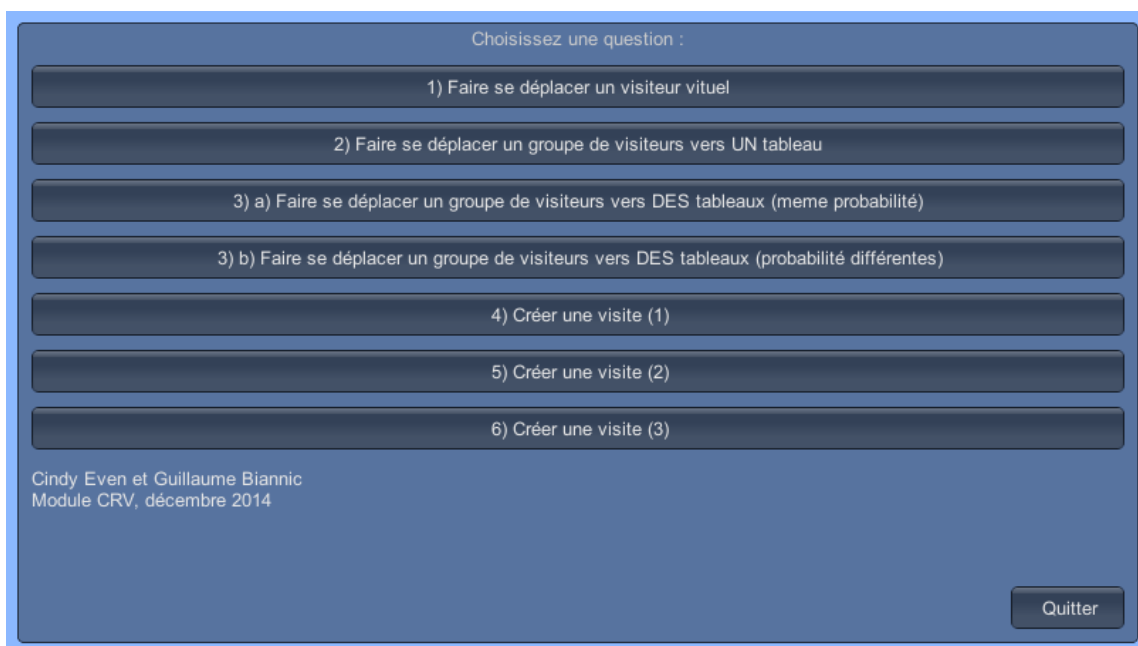


Fig. 3 Menu principal

Après avoir choisi une question, vous allez vous trouver dans le hall du musée dans la peau d'un visiteur. A tout moment vous pouvez afficher ou cacher le panneau des Options (Fig. 4) en cliquant sur la touche Espace.



Fig. 4 Panneau des Options

A partir de ce tableau vous pouvez :

- * Revenir au menu principal
- * Changer la vue de la simulation
- * Modifier des paramètres spécifiques à la question courante.

Notez qu'à tout moment de la simulation vous pouvez revenir au menu principal grâce à la touche M et passer de la vue globale à la vue visiteur et vis-versa avec la touche A. Les options spécifiques seront détaillées dans les parties du rapport suivantes.

3) Création du musée :

a) Objets 3D constituant la scène :

Les modèles 3D du musée et d'un tableau ont été construits à l'aide d'Autodesk 3ds Max qui est un logiciel d'animation, de rendu et de modélisation 3D. Le musée est constitué de 6 sales comme indiqué sur le plan ci-dessous :



Fig. 5 Plan du musée

La salle Accueil est la salle où commence la simulation. C'est à partir de cette salle que tous les agents virtuels et le visiteur commencent la visite du musée. Les cinq autres salles contiennent dix tableaux chacune, répartis de façon uniforme. Chaque salle représente l'un de ces mouvements artistiques : baroque, cubisme, futurisme, romantisme, surréalisme.

Un tableau possède une face avant et une face arrière différente l'une de l'autre.



Fig. 6 Un tableau vu de face et de derrière

Le modèle 3D du tableau possède deux matériaux (Materials dans Unity3D) : un pour le cadre et un autre pour la peinture. Pour créer de nouveaux tableaux dans le musée il nous suffit de remplacer le Material de la peinture et de modifier les dimensions du tableau. Afin d'économiser du temps nous avons choisi d'automatiser ce processus de création et de positionnement des tableaux dans le musée. Pour ce faire, nous avons créé un fichier XML contenant toutes les informations nécessaires pour tous les tableaux. Voici la structure du fichier XML :

```
<SceneObjects>
  <Paintings>
    <Painting name="painting1">
      <Transform>
        <Position>0.203727, 2.000000, -2.734337</Position>
        <Rotation>0.000000, 90.000040, 0.000000</Rotation>
        <Scale>1.770000, 1.220000, 1.000000</Scale>
      </Transform>
      <Material>PBaroque1Mat</Material>
      <Informations>
        <ArtisticMovement>Baroque</ArtisticMovement>
        <Artist>Diego vélasquez</Artist>
        <PaintingName>Vénus à son miroir</PaintingName>
        <Year>1650</Year>
      </Informations>
    </Painting>
    <Painting name="painting2">
      [...]
    </Painting>
  </Paintings>
</SceneObjects>
```

Fig. 7 Structure du fichier XML paintingList.xml

Pour lire et écrire dans ce fichier XML nous avons implémenté une classe (PaintingsAutoInitInspector) qui utilise différentes classes de l'espace de noms System.Xml fournissant une prise en charge standard du traitement XML.

Nous avons modifié l'interface d'Unity3D en ajoutant deux boutons (ces modifications sont possibles grâce à la classe Editor). Le premier bouton «Instanciate Paintings» permet d'instancier tous les tableaux de la scène, instances de la classe Painting (voir Fig. 8), avec la position, rotation, taille et Material définis dans le fichier XML. S'il y a un problème avec un tableau (se trouve au mauvais endroit, a les mauvaises dimensions, ...) nous pouvons corriger le problème directement dans Unity3D et cliquer sur le second bouton «Save modifs» pour rectifier les erreurs dans le fichier XML.

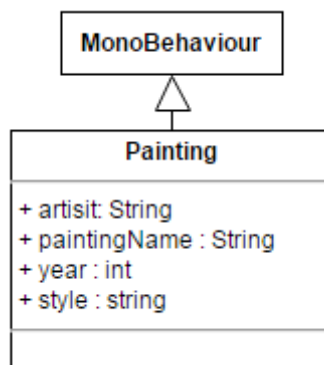


Fig. 8 Diagramme UML de la classe Painting

b) Déplacements de l'utilisateur

L'utilisateur peut se déplacer dans le musée grâce aux touches du clavier :

↑ ou Z pour avancer

↓ ou S pour reculer

← ou Q pour se déplacer latéralement sur la gauche

→ ou D pour se déplacer latéralement sur la droite.

Le visiteur se tourne de façon à suivre les mouvements de la souris sur l'axe horizontale. Et la caméra suit les mouvements de la souris sur l'axe vertical.

Ces comportements sont implémentés dans les classes VisitorMouvement et MouseLook.

Lorsque l'utilisateur est suffisamment proche d'un tableau et que le curseur de la souris est au-dessus de ce tableau, des informations sur ce dernier s'affichent (nom de l'artiste, titre du tableau et année de sa création).

c) Graphe de navigation

Afin de permettre aux visiteurs virtuels de se déplacer dans le musée, nous avons créé un graphe de navigation, composé de nombreux points reliés entre eux (Fig. 9).

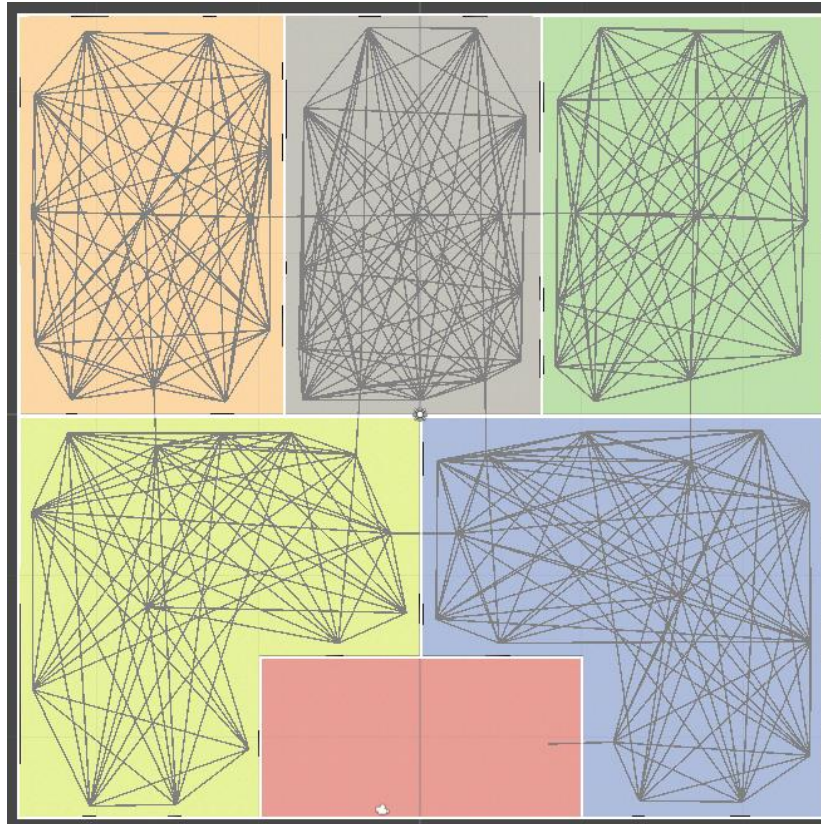


Fig. 9 Graphe de navigation du musée (en gris)

Pour construire les points qui composent le graphe de navigation, nous créons une classe « Point ». Elle contient un GameObject « Vide » qui est donc invisible dans la scène et qui nous permet de les positionner et de définir les liens entre les points (voir Fig. 10). Les attributs *parent*, *score* et *scoreG* sont utilisés pour les algorithmes de recherche de chemin détaillé plus loin dans le rapport.

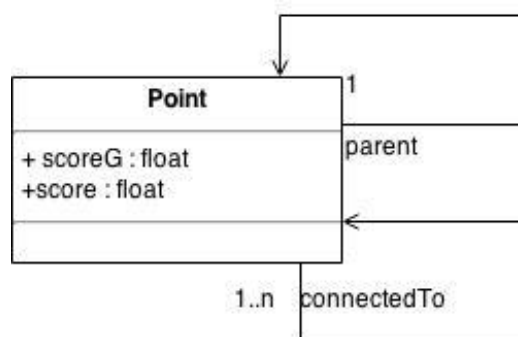


Fig. 10 Diagramme UML de la classe Point

Afin de simplifier la création et la modification du graphe de navigation, nous avons repris la méthode utilisée pour créer les tableaux. Nous avons donc implémenté une classe pouvant

créer les points en utilisant les informations contenus dans un fichier XML et également modifier le contenu du fichier XML lorsque des modifications étaient réalisées dans Unity3D. Voici comment est structurée l'information dans le fichier XML pour un point :

```
<Point name="DoorBA">
  <Transform>
    <Position>11.970990, 0.500000, -20.333140</Position>
  </Transform>
  <Connected>
    <To>Tab6</To>
    <To>Tab5</To>
    <To>Tab4</To>
    <To>Tab3</To>
    <To>Tab2</To>
    <To>Tab1</To>
    <To>CenterB</To>
    <To>DoorBD</To>
    <To>DoorAB</To>
  </Connected>
</Point>
```

Fig. 11 Structure du fichier XML pointList.xml

4) Faire se déplacer un visiteur virtuel :

Pour qu'un visiteur virtuel puisse se déplacer dans le musée à partir du graphe de navigation, il faut un algorithme de recherche de chemin qui, à partir d'un point de départ et d'un point d'arrivée, va retourner une liste des points par lesquelles il faut passer pour se rendre à la destination voulu. Dans ce projet, nous avons implémentés deux algorithmes de recherche de chemin qui peuvent être choisi via l'interface de la simulation : le Breadth First Search (BFS ou recherche en largeur) et l'algorithme A* avec deux implémentations différentes. Dans un premier temps nous allons expliquer comment est défini le visiteur virtuel.

a) Visiteur virtuel

Le comportement d'un visiteur est défini par la classe « AgentBehavior » (voir Fig. 12). La méthode *Start* est appelé à la création de l'agent et permet de faire l'initiation des attributs et la méthode *Update* est appelé à chaque frame. Une fois que l'utilisateur a choisi le tableau à atteindre en cliquant dessus, les points de départ et d'arrivée de l'agent sont définis par la méthode *FindNearestPoint*. Cette méthode permet de trouvé le point du graphe accessible le plus proche du *GameObject* passé en paramètre. Pour obtenir le point de départ, il faut donc appeler cette méthode en passant en paramètre le *GameObject* du visiteur virtuel et pour obtenir le point d'arrivée on passe le *GameObject* du tableau *targetPainting* en paramètre. Une fois que nous avons ces deux points, la méthode *CalculPath* permet de calculer les points de passage pour atteindre l'objectif et sauvegarde le résultat dans l'attribut *path*. L'algorithme utilisé dans la méthode *CalculPath* dépend de l'algorithme coché par l'utilisateur dans l'interface de la simulation.

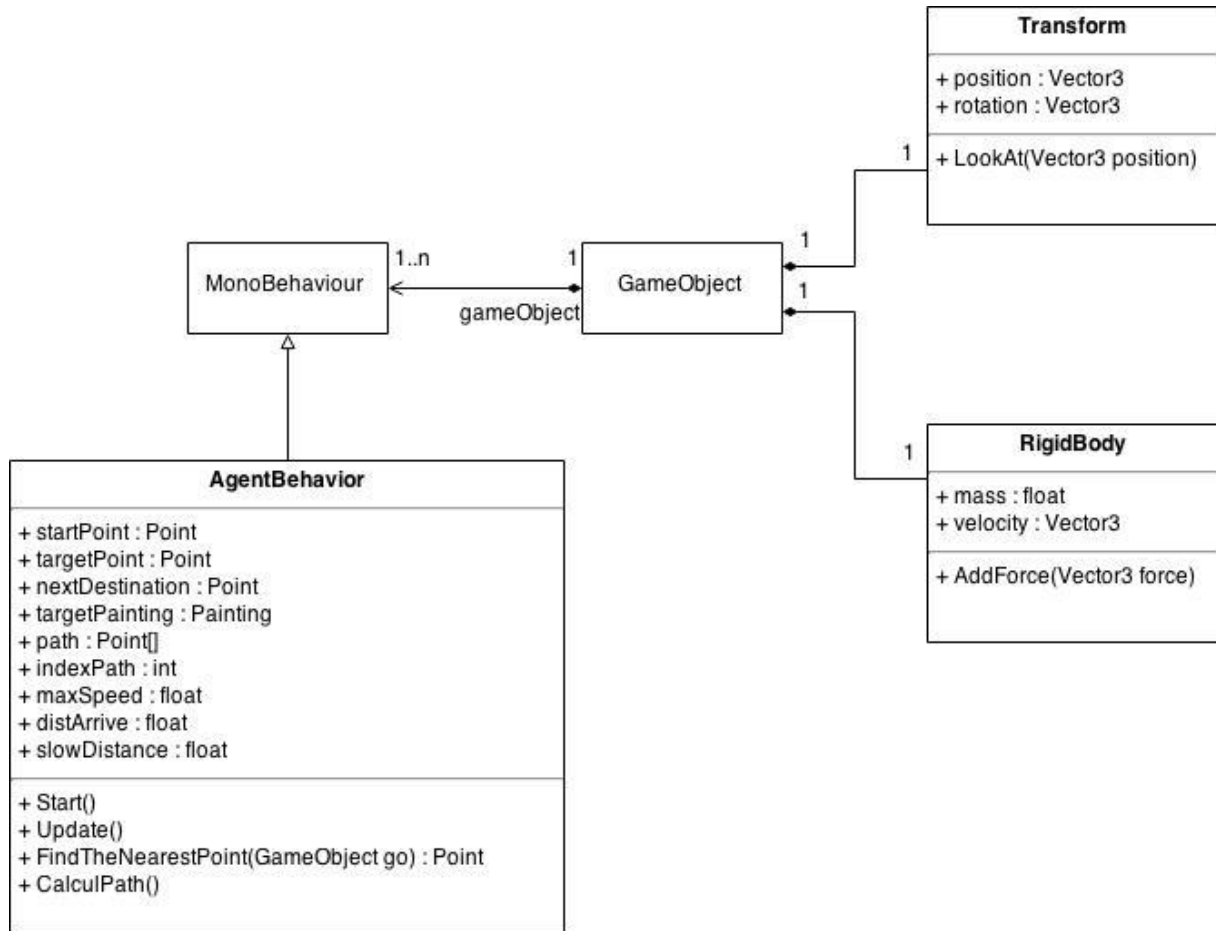


Fig. 12 Diagramme UML de la classe *AgentBehavior*

Le déplacement du visiteur virtuel est réalisé dans la méthode *Update*, qui calcule la force *seek* nécessaire pour qu'il se rende au point du graphe désiré.

La force *seek* est calculée de la façon suivante (\vec{V} est la vitesse actuelle) :

$$\vec{F}_{\text{seek}} = \vec{V}_{\text{desired}} - \vec{V}$$

Pour calculer la vitesse désirée, on calcule le vecteur normé entre la position du visiteur virtuel et la position qu'il souhaite atteindre et on multiplie par l'attribut *maxSpeed* :

$$\vec{V}_{\text{desired}} = \frac{\overrightarrow{P_{\text{point}}P_{\text{visiteur}}}}{\|\overrightarrow{P_{\text{point}}P_{\text{visiteur}}}\|} \times V_{\text{max}}$$

Si la distance *dist* entre le visiteur virtuel et le point est inférieure à l'attribut *slowDistance*, on multiplie alors la vitesse désirée par $\frac{\text{dist}}{\text{slowDist}}$ pour que le visiteur ralentisse à l'approche du point :

$$\vec{V}_{\text{desired}} = \frac{\overrightarrow{P_{\text{point}}P_{\text{visiteur}}}}{\|\overrightarrow{P_{\text{point}}P_{\text{visiteur}}}\|} \times V_{\text{max}} \times \frac{\text{dist}}{\text{slowDist}}$$

A partir de la force, on peut calculer la future position du visiteur virtuel grâce à cette formule :

$$\vec{P} += \vec{V} + \left(\left(\frac{\vec{F}}{\text{masse}} \right) \times dt \right) \times dt$$

On ne se sert pas de cette dernière formule pour déplacer l'agent car le *GameObject* possède un composant *RigidBody* (décrit dans la partie 1) qui permet de faire le calcul automatiquement à partir de la force en utilisant la méthode *AddForce*. Par contre, en calculant la future position de l'agent avec cette formule, on peut utiliser la méthode *LookAt* qui modifie la rotation du *GameObject* pour qu'il regarde dans la direction de la position passé en argument. Ainsi on s'assure que notre visiteur regarde vers l'endroit où il se déplace.

Une fois que la distance est inférieure à *distArrive*, l'agent va alors se rendre au Point suivant dans l'attribut *path* s'il n'est pas à la fin de la liste. S'il est arrivé au dernier point, l'agent attend que l'utilisateur sélectionne un nouveau tableau.

b) Algorithme Breadth First Search

Le Breadth First Search est un algorithme qui permet de parcourir un graphe mais qui peut également permettre de faire de la recherche de chemin. Pour l'implémenter, on instancie deux listes de points, « waypoints » qui contiendra les points par lesquelles il faut passer pour se rendre à destination et « alreadyView » qui contient les points déjà examinés ou en attente d'être examinés par l'algorithme. On instancie également une structure de données de type « File » (ou Queue) dans laquelle on va mettre les points en attente de traitement par l'algorithme. La raison pour laquelle on utilise une File est que cette structure de données repose sur le principe du First In, First Out (FIFO) qui est également le principe de base de l'algorithme Breadth First Search.

Pseudo code :

```
Mettre le point de départ dans la file ;
Tant que le point que l'on souhaite atteindre n'est pas celui qui est examiné :
    Retirer le premier point de la file pour l'examiner ;
    Pour chaque point connecté au point qui est examiné :
        Si il n'a pas déjà été examiné :
            Ajouter dans la file ;
            Définir son parent comme étant le point qui est examiné ;
```

Pour retrouver les points intermédiaire entre le point de départ et le point d'arrivé on utilise l'attribut « parent ».

Avec cet algorithme, on peut donc trouver le chemin pour se rendre d'un point à un autre mais il n'y a pas de prise en compte de la distance entre les points.

c) Algorithme A*

L'algorithme A* sert à faire de la recherche de chemin dans un graphe et utilise des évaluations heuristiques de la distance pour trouver le chemin le plus court. Il existe différentes manières pour faire un calcul heuristique de la distance, dans notre cas nous avons décidé d'utiliser la distance euclidienne. Pour réaliser l'algorithme, nous avons ajouté deux nouveaux attributs à la classe Point :

- « scoreG » qui représente la distance à parcourir depuis le point de départ jusqu'à ce point.
- « score » qui est la somme de « scoreG » plus l'heuristique de la distance entre ce point et le point que l'on souhaite atteindre.

Comme pour le Breadth First Search, on instancie deux listes de points, « waypoints » et « alreadyView ». On crée également une troisième liste « possibleWaypoints » qui contient les points qu'il faut examiner.

Pseudo code :

```
Mettre le point de départ dans la liste des points à examiner ;  
Calculer scoreG et score pour le point de départ ;  
Tant que le point que l'on souhaite atteindre n'est pas celui qui est examiné :  
    Retirer le point de la liste qui a le plus petit score pour l'examiner ;  
    Pour chaque point connecté au point qui est examiné :  
        S'il n'a pas déjà été examiné :  
            Calculer scoreG et score ;  
            Ajouter dans la liste ;  
            Définir son parent comme étant le point qui est examiné ;
```

Puis on utilise l'attribut parent pour retrouver les points intermédiaires entre le point d'arrivée et de départ. Comme on examine le point ayant la plus petite valeur de score en priorité, on s'assure que le chemin trouvé est le plus court.

d) Algorithme A* sans lien

Pour simplifier la création des points qui composent le graphe, notamment pour les connections entre les différents points, nous avons implémenté une variante de l'algorithme précédent. Pour déterminer si deux points sont reliés entre eux, nous utilisons l'API « Physics » de Unity3D et plus précisément la méthode « Raycast » de la façon suivante :

```
Physics.Raycast (Vector3 origine, Vector3 direction, float distance);
```

Cette méthode va créer une droite à partir des arguments et renvoyer un booléen pour indiquer si la droite passe à travers des obstacles. Ainsi nous pouvons vérifier s'il est possible de se rendre d'un point à un autre sans être bloqué.

5) Faire se déplacer un groupe de visiteurs virtuels vers UN tableau :

Pour permettre à un ensemble de visiteur de se déplacer en groupe, nous avons créé une nouvelle classe « SteeringBehaviour » qui va permettre aux visiteurs virtuels de se déplacer en restant proche et sans se percuter. Pour cela nous nous sommes inspiré du programme de Boids développé par Craig W. Reynolds en 1986, simulant le comportement d'une nuée d'oiseaux en vol. Nous avons donc calculé trois nouvelles forces à appliquer aux agents :

- Une force de séparation pour éviter les collisions.
- Une force de cohésion pour que les agents restent proches.
- Une force d'alignement pour qu'ils se déplacent dans la même direction.

Nous allons également faire le calcul de la force *seek* et de la direction dans cette nouvelle classe pour que tout ce qui concerne les forces appliquées aux agents soit géré dans une même classe. Pour réaliser ces calculs, chaque visiteur virtuel a besoin d'une liste qui contient tous les visiteurs virtuels qui sont proche de lui et qui ont donc une influence sur son comportement. Nous ajoutons donc à la classe « AgentBehavior », un attribut *visiteurs* de type *List<GameObject>*. Pour définir quels sont les visiteurs virtuels à mettre dans cette liste, nous utilisons un *Trigger* (présenté dans la partie 1) de forme circulaire, ainsi tous les visiteurs se trouvant dans le *Trigger* sont considérés comme étant proche et sont ajouté à la liste. La liste est mise à jour grâce aux méthode « OnTriggerEnter() » et « OnTriggerExit() ».

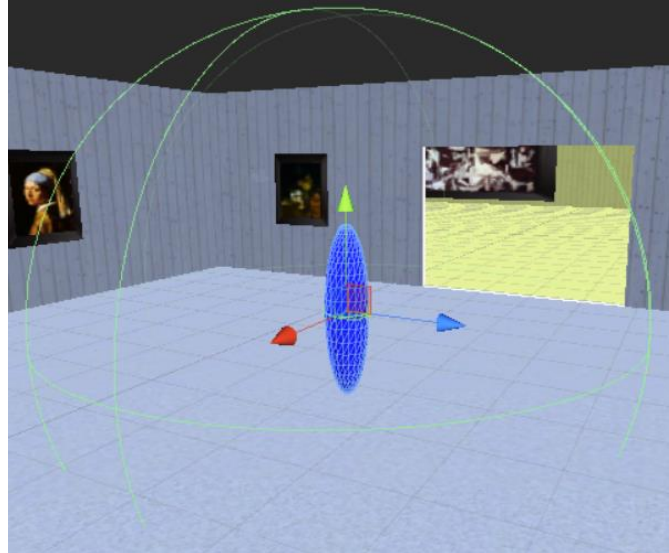


Fig. 13 Un visiteur virtuel (bleu) et son trigger (cercle vert)

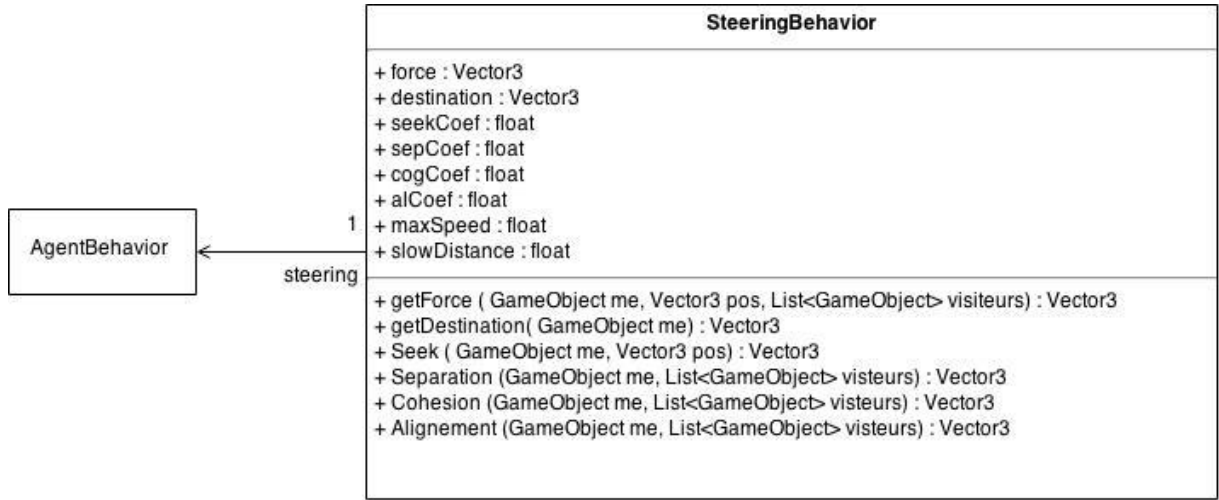


Fig. 14 Diagramme UML de la classe *SteeringBehavior*

Pour calculer la force de *séparation*, nous calculons une force qui éloigne le visiteur virtuel de tous les autres qui sont autour de lui. L'amplitude de cette force dépend de la distance entre les visiteurs virtuels. Plus un visiteur est proche, plus la force pour s'éloigner de lui sera forte :

$$\vec{F}_{\text{separation}} = \sum_{i=0}^{i=N} \frac{1}{\text{dist}_i} \times \frac{\overrightarrow{P_1 P_{\text{visiteur}}}}{\|\overrightarrow{P_1 P_{\text{visiteur}}}\|}$$

Pour calculer la force de *cohésion*, nous calculons le centre de gravité entre tous les visiteurs proches et nous calculons une force pour diriger le visiteur vers ce point comme pour la force *seek* :

$$\vec{P} = \frac{1}{N} \times \sum_{i=0}^{i=N} \vec{P}_i$$

$$\vec{F}_{\text{cohésion}} = \left(\frac{\overrightarrow{P_{\text{visiteur}} P}}{\|\overrightarrow{P_{\text{visiteur}} P}\|} \times V_{\text{max}} \right) - \vec{V}$$

Pour calculer la force d'*alignement*, nous utilisons la vitesse moyenne des visiteurs autour :

$$\vec{F}_{\text{Alignement}} = \left(\frac{1}{N} \times \sum_{i=0}^{i=N} \vec{V}_i \right) - \vec{V}$$

L'attribut *destination* représente la position où le visiteur va se rendre et est utilisé pour appeler la méthode *LookAt*.

Nous avons également ajouté un coefficient pour chaque force qui peut être modifié en temps réel durant la simulation via le panneau des options afin de voir leur impact sur le comportement des visiteurs virtuels.

6) Faire se déplacer un groupe de visiteurs virtuels vers DES tableaux :

On souhaite à présent que chaque visiteur puisse choisir eux même le tableau qu'ils vont aller voir, ce qui signifie qu'il ne faut plus appliquer les forces de cohésion et d'alignement entre tous les visiteurs proches mais seulement sur ceux qui ont la même destination. Pour cela on crée une nouvelle liste en ajoutant dans la classe « AgentBehavior » un attribut *visiteursWithSameDestination* qui contiendra tous les visiteurs proches qui vont dans la même direction que lui. On considère que deux visiteurs vont dans la même direction si le tableau qu'ils vont voir est le même ou si le prochain point du graphe qu'ils vont atteindre est le même. On modifie également le prototype de la méthode *getForce* de « SteeringBehavior » pour pouvoir passer l'attribut *visiteursWithSameDestination* en appelant la méthode.

La sélection d'un tableau par un visiteur virtuel se fait grâce à un algorithme de sélection par roulette (Roulette Wheel Selection). Pour cela nous avons rajouté à chaque tableau un attribut « fitness » qui représente sa probabilité d'être sélectionné. Ainsi, si tous les tableaux ont la même valeur de fitness, ils ont tous la même probabilité d'être sélectionné mais si on souhaite que certains tableaux aient plus de chance d'être choisis, il suffit d'augmenter leur fitness.

Pseudo code de l'algorithme de sélection :

```
sumFitness = 0 ;
currentFitness = 0 ;
Pour tous les tableaux :
    sumFitness += tableau.fitness ;
nb = nombre aléatoire entre 0 et sumFitness ;
Pour tous les tableaux :
    currentFitness += tableau.fitness ;
    Si nb < currentFitness :
        Retourner tableau ;
```

7) Créer une visite (1)

Pour créer une visite nous voulons que les visiteurs se déplacent de tableaux en tableaux en s'arrêtant pour les observer. Pour cela, nous avons donc implémenté une Machine à Etats Finis (Finite State Machine). La classe « StateMachine » permet de passer d'un état à un autre grâce à la méthode *ChangeState*. La méthode *Execute* de StateMachine est appelée toutes les frames via la méthode Update du visiteur virtuel et appelle la méthode *Execute* de l'état *currentState*. Un état est une classe qui hérite de la classe abstraite *States* qui force l'état à avoir au moins trois méthodes :

- *Enter* qui est appelée lorsque le visiteur virtuel rentre dans cet état.
- *Execute* qui est appelée toute les frames.
- *Exit* qui est appelée lorsque le visiteur virtuel sort de cet état.

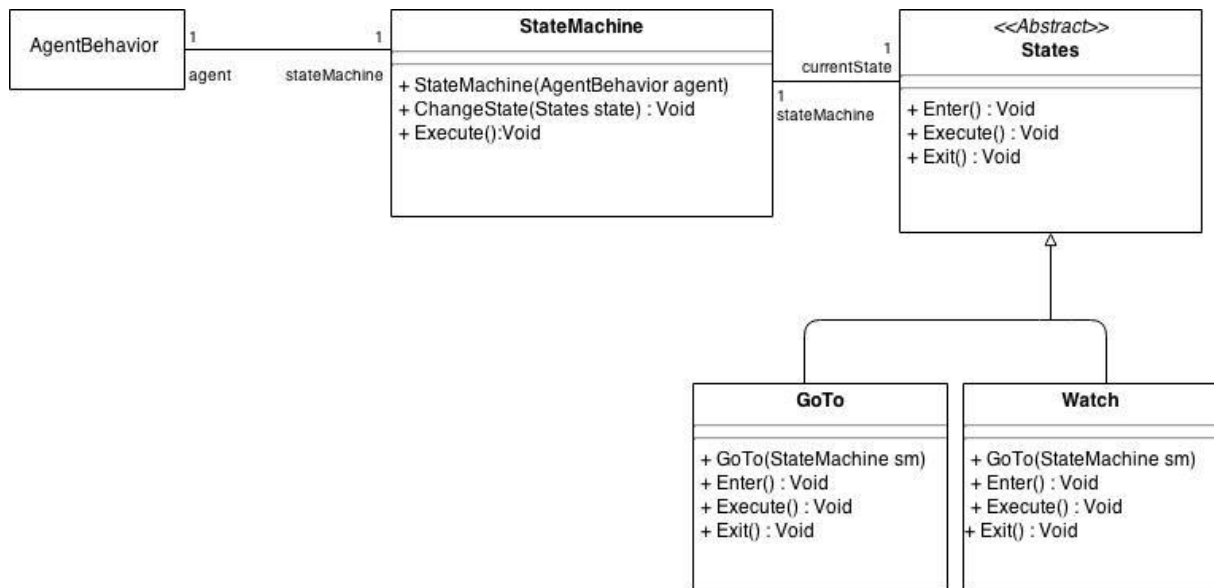


Fig. 15 Diagramme UML de la machine à états

Nous avons créé deux états : « GoTo » et « Watch ». L'état GoTo permet au visiteur virtuel de se rendre à un tableau en gérant l'avancement dans la liste *path* qui était auparavant réalisé dans la méthode Update de « AgentBehavior » et il sort de cet état lorsqu'il est arrivé à destination (fin de la liste *path*). Il va ensuite entrer dans l'état Watch où il va rester devant le tableau jusqu'à ce que la condition de changement d'état soit remplie, il retournera alors dans l'état GoTo et ainsi de suite. Trois conditions pour sortir de l'état Watch ont été implémentées et peuvent être choisies durant la simulation via le panneau des options :

- *Timer* : le visiteur virtuel va vers un autre tableau après un certain temps.
- *Visiteur à côté* : le visiteur virtuel va vers un autre tableau lorsque l'utilisateur est à côté de lui.
- *Nombre d'agents limite* : le visiteur virtuel va vers un autre tableau lorsqu'un certain nombre de visiteurs virtuels sont à côté de lui en train d'observer le même tableau.

8) Créer une visite (2)

A présent on souhaite que les tableaux vus précédemment par un visiteur virtuel aient une influence sur le choix des tableaux qu'il va aller observer par la suite. Comme on a pu le voir précédemment, la probabilité d'un tableau d'être choisi dépend de leur attribut de fitness. Le problème est le suivant, comme l'attribut est lié au tableau, sa probabilité est la même pour tous les visiteurs. Cependant dans notre cas nous souhaitons que la probabilité d'un tableau d'être choisi puisse être différente d'un visiteur à l'autre afin de pouvoir prendre en compte leurs actions précédentes. Nous avons donc décidé de définir la valeur de fitness des tableaux dans le visiteur virtuel en utilisant un Dictionnaire qui prend pour Clé : un tableau et pour Valeur : sa valeur de fitness. Ainsi nous pouvons modifier les différentes valeurs de fitness individuellement pour chaque visiteur.

Le comportement que nous souhaitons implémenter pour cette partie est le suivant :

Après qu'un agent ai vu un tableau, la probabilité qu'il aille voir un tableau ressemblant à celui-ci augmente.

Afin de déterminer si deux tableaux se ressemblent, il faut d'abord les classer. Pour cela nous créons une énumération appelée Tags et qui peut prendre les valeurs suivante :

- | | | |
|------------|-------------|---------------|
| • Femme | • Musique | • Futurisme |
| • Combat | • Nature | • Surréalisme |
| • Animaux | • Urbanisme | • Romantisme |
| • Portrait | • Baroque | |
| • Science | • Cubisme | |

Nous ajoutons à chaque tableau une liste de Tags qui sert à le décrire et nous considérons que deux tableaux se ressemblent s'ils ont un ou plusieurs Tags en commun. Nous ajoutons dans la méthode *Exit* de l'état GoTo, un appel de la méthode *UpdateFitness* de l'agent. La méthode *UpdateFitness* compare les valeurs dans la liste de Tags du tableau que le visiteur vient d'atteindre avec celles présentent dans les liste de Tags des autres tableaux et si il y a correspondance, la fitness du tableau est augmentée. Ainsi, si un tableau a deux tags en commun avec le tableau que l'agent vient d'atteindre, la fitness de ce tableau est augmentée à deux reprises.

9) Créer une visite (3)

On souhaite à présent que la probabilité d'un tableau d'être choisi dépende de l'utilisateur et de ses goûts. Contrairement à la question précédente, la probabilité d'un tableau ne change pas d'un visiteur à un autre alors on reconsidère la fitness d'un tableau comme étant un attribut de la classe « Painting ». Pour permettre à l'utilisateur de définir son profil, nous avons ajouté dans l'interface de la simulation la possibilité de sélectionner ses préférences à partir de la liste de Tags vu précédemment. Pour chaque Tags choisi par l'utilisateur, nous augmentons la fitness des tableaux qui possèdent ce Tags. Ainsi les tableaux vers lesquels vont se diriger les visiteurs virtuels dépendent des préférences de l'utilisateur.

En utilisant le même principe, on donne à l'utilisateur la possibilité de donner son avis sur un tableau. Lorsque l'utilisateur est proche d'un tableau et clique dessus avec son curseur, une fenêtre s'ouvre pour lui demander s'il aime ce tableau. S'il répond « oui » alors la fitness des tableaux sera modifiée en fonction des tags du tableau sur lequel il a cliqué comme expliqué précédemment.

Conclusion

Via l'utilisation de plusieurs algorithmes et systèmes nous avons pu répondre à la problématique du sujet. Nous avons cherché à montrer les différentes étapes de notre travail et c'est pourquoi nous avons choisi d'utiliser plusieurs scènes illustrant notre raisonnement pour chaque question. De la même façon, nous avons testé plusieurs algorithmes et il nous a semblé judicieux de présenter les différents résultats obtenus c'est la raison pour laquelle nous avons choisi d'afficher les différentes options dans le panneau des options.

Ce projet nous aura permis de mieux appréhender, comprendre et concevoir des univers virtuels. Il aura aussi été un bon entraînement pour préparer notre stage.