Rapport du projet CRV

Cindy Even (c9even@enib.fr)

Guillaume Biannic (g9bianni@enib.fr)

Décembre 2014

Sommaire

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 musé 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 répondre à ces questions.

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 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 :

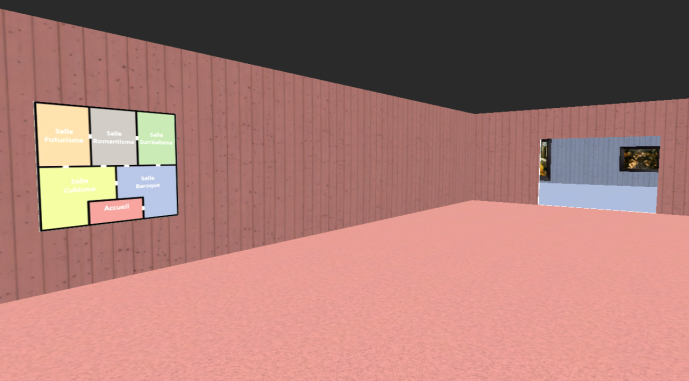
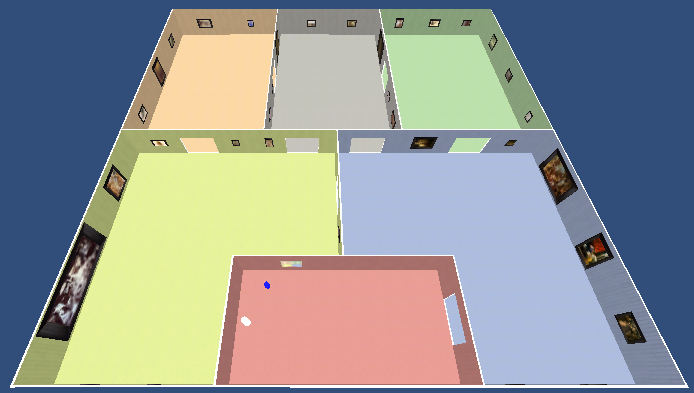
- GameObjet (GO) : ce sont les entités les plus importantes dans Unity3D. Tout objet dans l’application est un GO, cependant, ils ont besoin de composants (qui héritent de la classe Component) pour se différencier les uns des autres. On peut voir un GO comme une boîte vide à laquelle il faut ajouter 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 pour 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) et le second permet de voir le musé en entier. Pour le FPV il suffit d’attacher la caméra au visiteur et de la placer au niveau des yeux.

- 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 pourrons alors utiliser les méthodes Start(), Update() et autres pour implémenter les fonctionnalités voulues.

- 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 simplement d'autres Colliders de passer à travers. Quand un Collider entre son espace, ceci va appeler la fonction OnTriggerEnter().

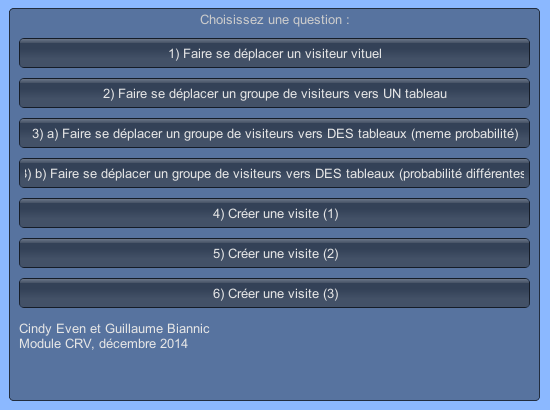
- Tag : Un tag est un mot qui permet d’accéder à un ou plusieurs GO. Nous pouvons les utiliser dans les Scripts pour trouver un GO qui contient le Tag souhaitée. Ce résultat est obtenu en utilisant la fonction GameObject.FindWithTag().

Ces explications devraient être suffisantes pour comprendre la suite 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>.

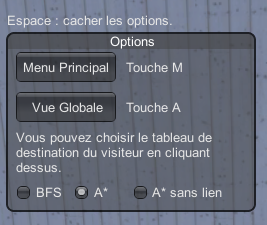
1. Présentation de l’interface

Pour lancer l’application, double cliquez sur le fichier exécutable. 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. A partir de ce menu vous pouvez accéder à 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 (voir image ci-dessous). Pour voir notre travail sur la première question : Faire se déplacer un visiteur virtuel, il suffit de cliquer sur le bouton associé.



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 en cliquant sur la touche Espace (voir figure ci-dessous).



A partir de ce tableau vous pouvez :

\* Revenir au menu principal

\* Changer la vue de la simulation

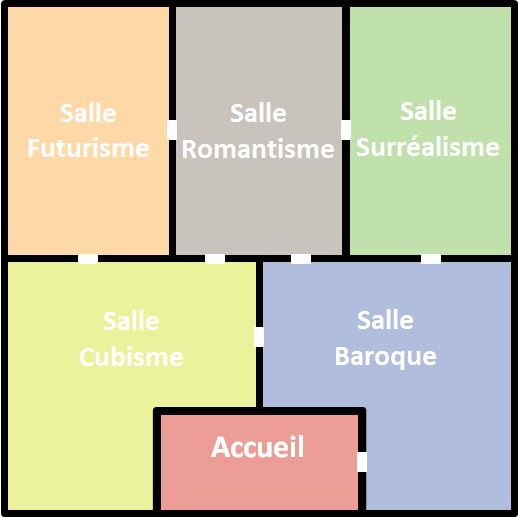
\* Modifier d’autres paramètres qui sont spécifiques à la question courante.

Notez que 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.

1. Création du musée :
   1. 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 :

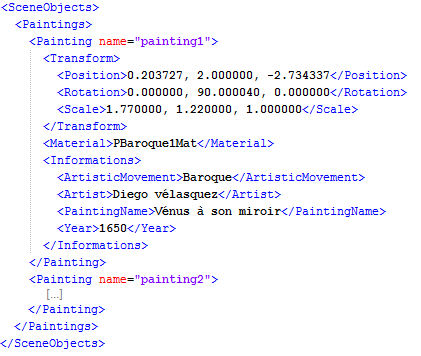


La salle Accueil est la salle où commence la simulation. C’est à partir de cette salle que tous les agents virtuels et le visiteur commencent la visite du musée. Les cinq autres salles contiennent dix tableaux chacune, répartis de façon uniforme. Chaque salle représente l’un de ces mouvements artistiques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

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

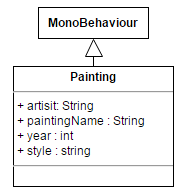


Le modèle 3D du tableau possède deux matériaux (Materials dans Unity3D) : un pour le cadre et un autre pour la peinture. De ce fait, il nous suffit de remplacer le Material de la peinture et de modifier les dimensions du tableau pour créer de nouveaux tableaux dans le musée. Afin d’économiser du temps nous avons choisi d’automatiser ce processus de création et 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 :



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 représentation UML ci-dessous), avec la position, rotation, taille et Material définit 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 l’erreur dans le fichier XML.



* 1. 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

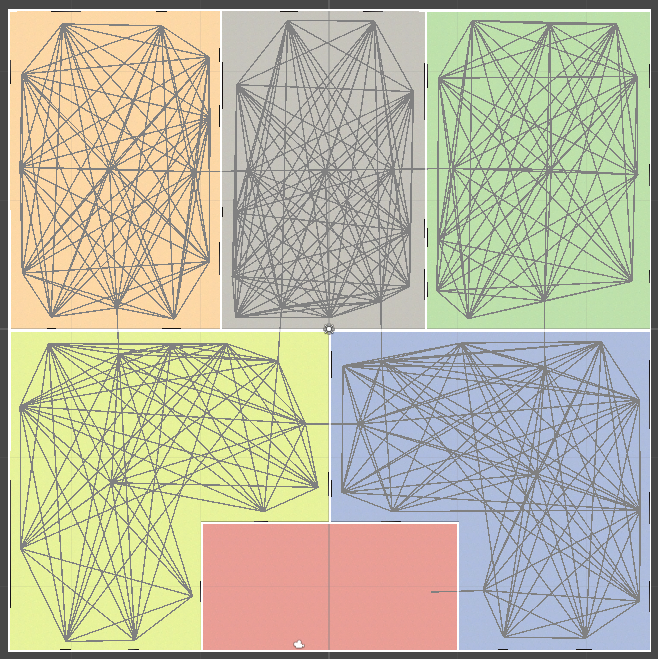
→ ou D pour se déplacer latéralement sur la droite.

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

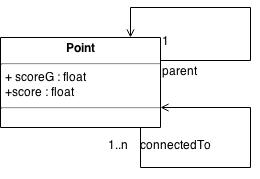
Ces comportements sont implémentés dans les classes VisitorMouvement et MouseLook.

Lorsque l’utilisateur est suffisamment proche d’un tableau et que le curseur de la sourie 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).

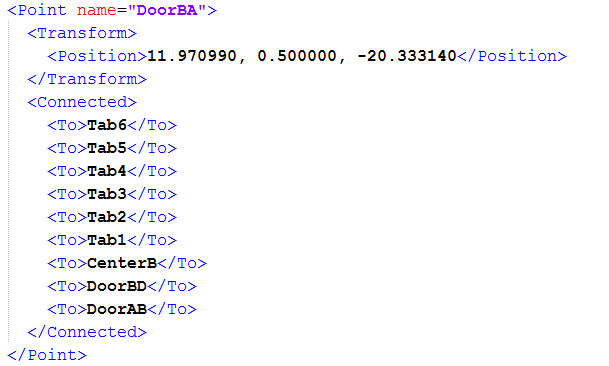
* 1. Graphe de navigation (Ajouter le schéma UML de la classe POINT)



Pour créer les points qui composent le graphe de navigation, On crée une classe « Point » qui contient un GameObject « Vide », c’est-à-dire qu’il est invisible dans la scène, pour nous permettre de les positionnés et de définir les liens entres les points (Voir schéma UML) les attributs *parent, score, scoreG* sont utilisé pour les algorithmes de recherche de chemin détaillé plus loin dans le rapport.



Afin de simplifier la création et la modification du graphe de navigation, nous avons repris de le principe utilisé pour les tableaux et implémentés 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 :

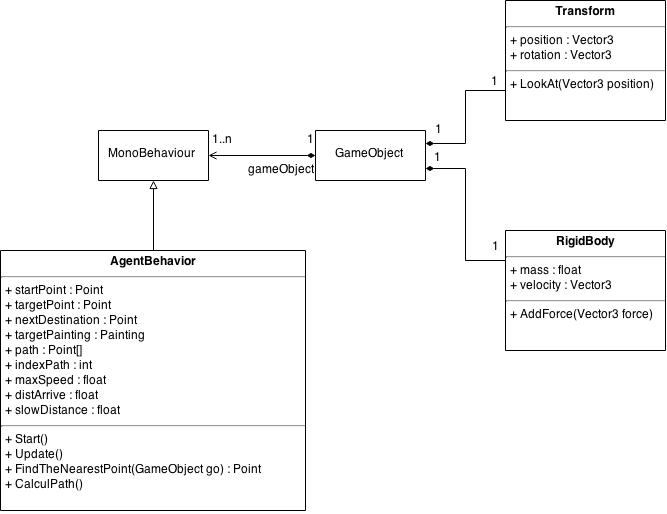


1. Faire se déplacer un visiteur virtuel :

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’arriver, va retourner la 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. Pour commencer nous allons voir comment est défini le visiteur virtuel.

* 1. Visiteur virtuel

Le comportement d’un visiteur est définit par la classe « AgentBehavior » (Voir schéma UML). 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, on va définir le point de départ et d’arrivé de l’agent avec la méthode FindNearestPoint. Cette méthode permet de trouvé le point du graphe accessible le plus proche du GameObject passé en paramètre. Pour avoir le point de départ on appelle donc la fonction avec le GameObject du visiteur en paramètre et pour avoir le point d’arrivé, on utilise le GameObject du tableau targetPainting. Une fois que nous avons ces deux points, on appelle la méthode CalculPath qui calcul 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.



Le déplacement de l’agent est réalisé dans la méthode Update, qui calcul la force seek pour permettre à l’agent de se rendre au point du graphe désiré.

Calcul de la force où est la vélocité :

Pour calculer la vélocité désirée, on calcule le vecteur normé entre la position de l’agent et la position que l’agent souhaite atteindre et on multiplie par l’attribut maxSpeed :

Si la distance dist entre le visiteur et le point est inférieur à l’attribut slowDistance, on multiplie alors la vélocité désirée par pour que le visiteur ralentisse à l’approche du point :

A partir de la force, on peut calculer la future position de l’agent grâce à cette formule :

On ne se sert pas de cette dernière formule pour déplacer l’agent car le GameObject possède un composant RigidBody d’Unity3D 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 à 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 si il n’est pas à la fin de la liste. Si il est arrivé au dernier point, l’agent attend que l’utilisateur sélectionne un nouveau tableau.

* 1. 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 « Queue » (ou File) dans laquelle on va mettre les points en attente de traitement par l’algorithme. La raison pour laquelle on utilise une Queue 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 ; |

Après, on retrouve les points intermédiaire entre le point de départ et le point d’arrivé grâce à l’attribut « parent ».

Avec cet algorithme, on peut dont 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.

* 1. A\*

L’algorithme A\* sert à faire de la recherche de chemin dans un graphe et utilise des évaluations heuristique de la distance pour trouver le chemin le plus court. Il existe différente manière pour faire un calcul heuristique de la distance, dans notre cas nous avons décidés 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 :  Si 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’arrivé 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.

* 1. 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 qui, pour déterminer si deux points étaient reliés entre eux, utilisait l’API « Physics » de Unity3D et plus précisément la méthode « Raycast » qui s’utilise de la façon suivante :

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

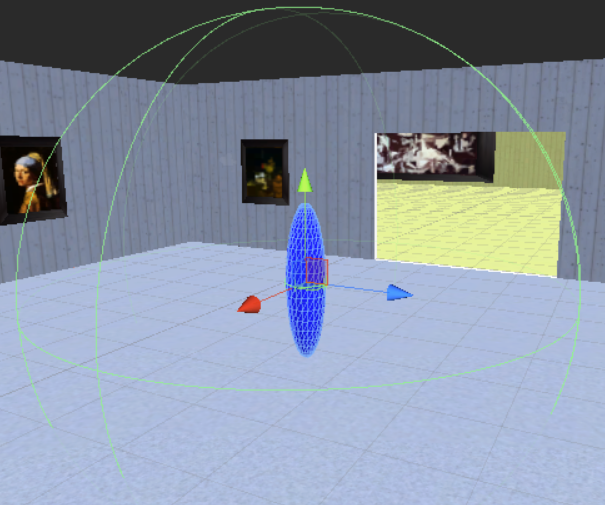
Cette méthode va créer une droite à partir des arguments et renvoyer un booléen pour indiquer si la droite passe à travers des obstacles. Ainsi on peut vérifier si il est possible de se rendre d’un point à un autre sans être bloqué.

1. Faire se déplacer un groupe de visiteurs virtuels vers UN tableau :

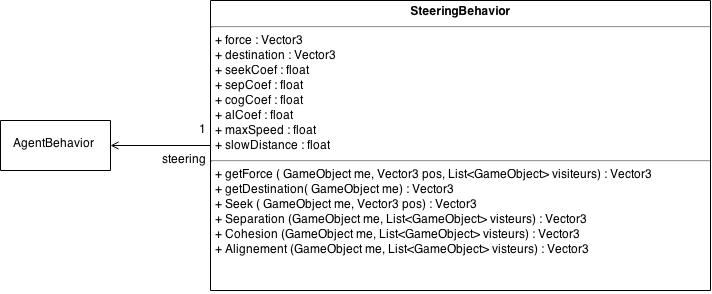
Pour permettre à un ensemble de visiteur de se déplacer en groupe, nous avons créé une nouvelle classe « SteeringBehaviour » qui va permettre aux agents de se déplacer en restant proche sans se percuter. Pour cela nous allons nous inspirer du programme de Boids développé par Craig W. Reynolds en 1986, simulant le comportement d’une nué d’oiseaux en vol et calculer 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, nous allons avoir besoin d’une liste pour chaque agent qui contient tous les agents qui sont près de lui et qui ont donc une influence sur son comportement. Nous allons ajouter un Trigger autour de chaque agent grâce au composant Collider de Unity3D et ainsi pouvoir connaitre tous les agents qui sont proche d’un autre agent grâce aux méthode « OnTriggerEnter() » et « OnTriggerExit() ». Nous ajoutons à la classe « AgentBehavior », un attribut *visiteurs* de type *List<GameObject>* dans laquelle on va mettre les visiteurs proches.



*Un agent (bleu) et son trigger (vert)*



Pour calculer la force de *séparation*, on calcule une force qui va éloigner l’agent de tous les autres agents qui sont autour de lui et l’amplitude de cette force dépend de la distance entre les agents. Plus un agent est proche, plus la force pour s’éloigner de lui sera forte :

Pour calculer la force de *cohésion*, on calcule le centre de gravité entre tous les agents proches puis on calcul une force pour diriger l’agent vers ce point comme pour la force *seek :*

Pour calculer la force d’*alignement*, on utilise la vélocité moyenne des agents autours :

L’attribut *destination* représente la position où l’agent 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 menu afin de voir leur impact sur le comportement de l’agent.

1. 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’il va 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 à 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 veuillent aller voir est le même ou si le prochain point du graphe qu’ils veuillent atteindre est le même. On modifie également le prototype de la méthode *getForce* de « SteeringBehavior » pour pouvoir passer l’attribut v*isiteursWithSameDestination* en appelant la méthode.

La sélection d’un tableau par un visiteur se fait grâce à un algorithme de sélection par roulette (Roulette Wheel Selection). Pour cela on a 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 sélectionnés mais si on souhaite que certains tableaux aient plus de chances d’être choisi, il suffit d’augmenter leurs fitness.

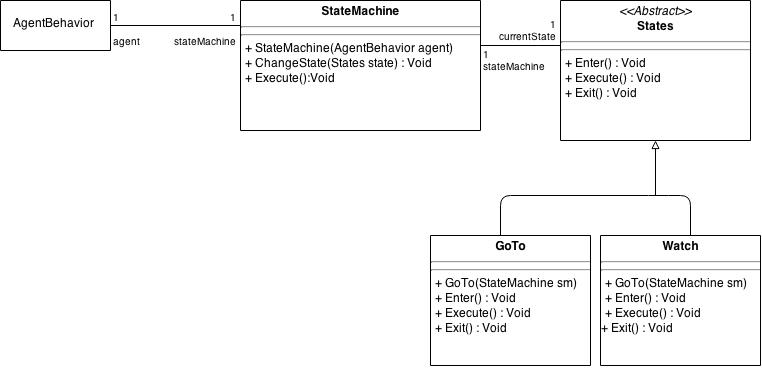
Pseudo code de l’algorithme de sélection :

|  |
| --- |
| *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 ;* |

1. Créer une visite (1)

Pour créer une visite nous voulons que nos visiteurs aillent de tableaux en tableaux et s’arrêtent entre chaque pour les observer, nous avons 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* et la méthode *Execute* appel la méthode *Execute* de l’état *currentState,* la méthode *Execute*  de la machine à état est appelée toute les frames via la méthode *Update* de l’agent. 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 l’agent rentre dans cet état.
* *Execute* qui est appelée toute les frames comme vu avant.
* *Exit* qui est appelée lorsque l’agent sort de cet état.



Nous avons créé deux états : « GoTo » et « Watch ». L’état GoTo permet à l’agent 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. Trois conditions pour sortir de l’état Watch ont été implémentées et peuvent être choisi durant la simulation via l’interface :

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

1. Créer une visite (2)

A présent on souhaite que les tableaux qu’un agent a vus avant aient une influence sur le choix des tableaux qu’il va aller observer par la suite en modifiant les probabilités. Comme on a pu le voir précédemment, la probabilité d’un tableau d’être choisi dépend de leur attribut de fitness seulement, comme l’attribut est lié au tableau, sa probabilité est la même pour chaque agent. Cependant dans notre cas on souhaite que la probabilité d’un tableau d’être choisi puisse être différente d’un agent à l’autre afin de pouvoir prendre en compte les actions précédentes de chaque agent. On a donc décidé de définir la valeur de fitness des tableaux dans l’agent en utilisant un Dictionnaire qui prend pour clé un tableau et donne en retour la valeur de fitness du tableau passé en clé. Ainsi on peut modifier les différentes valeurs de fitness individuellement pour chaque agent.

Le comportement que l’on souhaite implémenter pour cette partie est le suivant :

*Après qu’un agent a vu un tableau, la probabilité qu’il aille voir un tableau ressemblant à celui qu’il vient de voir augmente.*

Afin de déterminer si deux tableaux se ressemblent, il faut d’abord catégoriser chaque tableau. Pour cela on crée une énumération qu’on appelle Tags et qui peut prendre les valeurs suivante :

* Femme
* Combat
* Animaux
* Portrait
* Science
* Musique
* Nature
* Urbanisme
* Baroque
* Cubisme
* Futurisme
* Surréalisme
* Romantisme

On ajoute à chaque tableau une liste de Tags qui sert à le décrire tableau et on considère que deux tableaux se ressemble si ils ont un ou plusieurs Tags en commun. On ajoute dans la méthode *Exit* de l’état GoTo, qui est appelé lorsque l’agent sort de cet état, c’est-à-dire quand il est arrivé au tableau qu’il souhaite observer, un appel de la méthode *UpdateFitness* de l’agent. La méthode *UpdateFitness* va alors comparer les valeurs dans la liste de Tags du tableau que l’agent vient d’atteindre avec celles présentent dans les liste de Tags des autres tableaux et si il y a correspondance, la fitness du tableau va être augmenté. Ainsi, si un tableau a deux tags en commun avec le tableau que l’agent vient d’atteindre, la fitness de ce tableau va être augmenté à deux reprises.

1. 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, on a ajouté dans l’interface de la simulation la possibilité de sélectionner à partir de la liste de Tags vu précédemment ses préférences. Pour chaque Tags choisi par l’utilisateur, on augmente la fitness de chaque tableau qui possède ce Tags dans sa liste. 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 si il aime ce tableau. Si il répond « oui » alors la fitness des tableaux sera modifiée en fonction des tags du tableau sur lequel il a cliqué comme vu précédemment.

Conclusion

Bibliographie