Rapport projet IEVA

Pingu

Elouan EVEN & Dorian LEVEQUE

Table des matières

Description

Le projet était de créer un environnement virtuel contenant des agents autonomes.
Chaque agent aurait la capacité de percevoir au préalable son environnement pour pouvoir agir en conséquence. Par exemple, suivre ou fuire un objectif à la vue d'un autre acteur.

Pour réaliser l'environnement virtuel, nous avons utilisé la librairie Three.js. C'est une bibliothèque JavaScript qui permet de créer des scènes 3D dans un navigateur web à l'aide de webGL. Elle peut être utilisée avec une balise canvas de HTML5.

Organisation du code

Dans l'archive vous trouverez l'organisation suivante:

├ assets: répertoire des images et modèles 3d
├ css: fichier de définitions pour la mise en page
├ docs: répertoire des documents utilisé dans le projet
├ lib: répertoire des bibliothèques et package utilisé
└ src
┆ ├ actors: répertoire contenant l'ensemble des acteurs du monde virtuel
┆ ┆ ├ components: répertoire contenant divers module pour les acteurs
┆ ┆ └ triggers: répertoire contenant l'ensemble des capteurs disponible pour les acteurs
┆ ├ App.js: notre application (s'appuie sur la base du fichier simulation)
┆ ├ Prims.js: fichier contenant des fonctions pour créer des formes 3D basique et charger des fichiers 3D
┆ ├ Sim.js: classe de base d'une simulation avec Three.js
┆ └ Utils.js: fichier contenant diverses fonctions utilisé dans le projet.
├ index.html: intégration du code dans une page Web
├ index.js: script principal
├ package.json
└ readme.md

L'ensemble du code est découpé de la manière suivante. Chaque fichier contient un ensemble de fonctions et classes exportés qui sont ensuite réutilisés à divers endroits de notre application.

Le fichier à exécuter pour lancer l'application est index.html (en prenant soin au préalable de désactiver la règle de sécurité de type fileuri). Sinon, si vous avez NodeJs d'installé sur votre machine, vous pouvez juste installer les dépendances nécessaire décritent dans le package.json (à l'aide de la commande: "npm install"), puis exécuter la commande "npm start" pour lancer un serveur sur votre machine. Cette seconde technique ne requiert pas à devoir désactiver des règles de sécurité dans le navigateur.

L’application

Nous avons souhaité réaliser une simulation d’une colonie de pingouins en Antarctique. Cette simulation est constituée d’un terrain, de 200 herbes, de 200 rochers, d’une dizaine de pingouins adulte qui embarquent leur propre comportement (machine à état que nous verrons plus tard) ainsi que d’une dizaine de bébés pingouins qui agissent suivant les concepts introduits par les boids de Reynolds.
Sachez que ce réglage est paramétrable via le fichier d’entrée du projet (index.js).

Notre application est une redéfinition de la classe Simulation. Cette classe est abstraite et permet de créer une simulation de l’environnement à l’aide de Three.js.
Notre classe Application redéfinit la méthode pour créer la scène (cf createScene dans App.js). On récupère en paramètre de cette fonction, l’ensemble des options décrit dans le fichier index.js, pour pouvoir créer et placer de manière aléatoire, le nombre désiré de type d’acteur sur la scène.

Lors de la création des acteurs, nous avons fait le choix d’ajouter le paramètrage aléatoire de certaines de leurs options. Ainsi les rochers n’auront pas tous la même taille et les pingouins auront une masse différente.

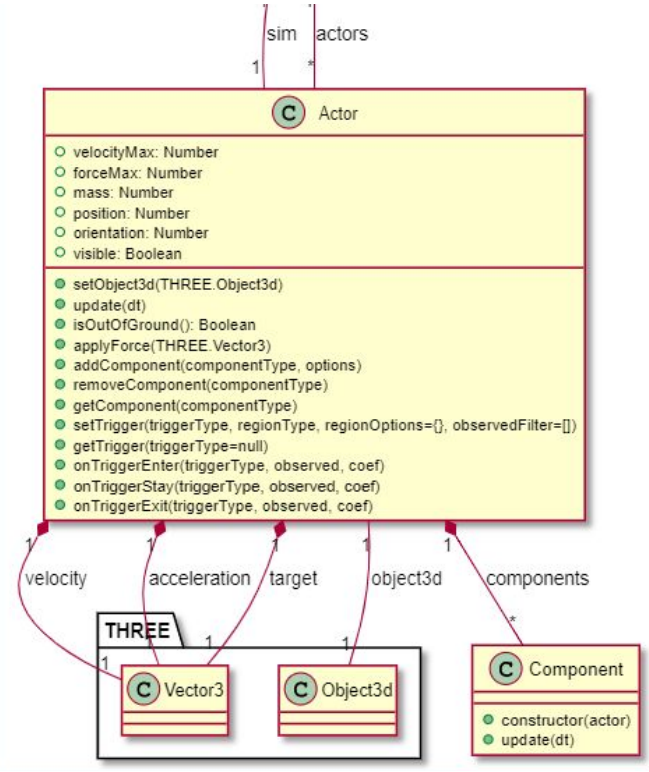
La classe simulation quant à elle, exécute dans un premier temps l'évaluation de l'ensemble des capteurs associé à chaque acteur (que nous verrons plus tard). Puis dans un second temps, met à jour chaque acteur pour leur déplacement. Cette architecture permet à ce que tous les acteurs puissent analyser le même état de l’environnement avant d’effectuer leurs décisions/actions.

Les acteurs

Dans notre simulation, nous avons créé de nombreux acteurs. Chaque acteur est un agent pouvant se déplacer ou non dans notre simulation et interagir avec l’environnement.

Nous avons créé les acteurs Grass, Human, Penguin, BabyPenguin, Pheromone et Rock. Tous héritent des propriétés définies dans la classe abstraite Actor.

Un acteur est composé d’une masse, de composants et des vecteurs target, position, vitesse et accélération. Pour faire déplacer un acteur, il suffit de lui appliquer une force d'accélération dans la direction voulu et les vecteurs vitesse et position seront calculés a chaque pas de simulation par la méthode update de la classe Actor. Des paramètres supplémentaires comme velocityMax et forceMax permettent de limiter l’application de force beaucoup trop importante.



Les composants

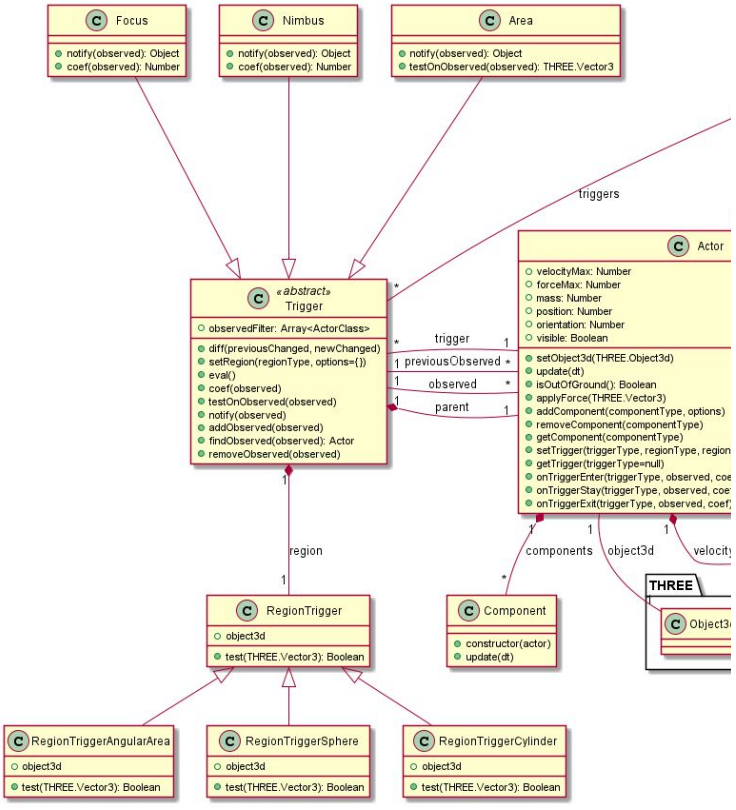
Afin d'éviter des redondances de code, on a choisi d'utiliser le principe des composants pour décrire les différents comportements de nos acteurs. Ainsi à chaque pas de simulation, chaque acteur exécute tous les composants qui le composent. Ces composants peuvent lui appliquer des forces pour le mouvoir dans une direction (exemple avec les composants Seek et Flee dans Steering.js), lui faire varier sa vitesse max pour le faire ralentir à l'arrivée de sa destination (composants Arrive et AbruptDeparture dans SpeedVariation.js), ou bien lui rajouter de nouvelles fonction comme la faim (composant Hungry), le calcul de la sensibilisation par rapport à un acteur observé (Awareness.js), l'attribution d'une destination aléatoire (RandomTarger.js), l'évitement d'obstacles (ObstacleAvoidance.js), ou encore qu'il puisse lâcher de manière aléatoire des phéromones (comme c'est implémenté avec les pingouins, composant ReleasePheromone.js).
Les composants permettent d'obtenir une architecture très modulaire et indépendante. Ces composants peuvent être ajoutés ou retirés à n'importe quel moment.

Les capteurs virtuels

Pour que l'acteur puisse décider des choix à effectuer, il a besoin d'analyser au préalable son environnement. Pour ce faire, nous nous sommes inspirés du pattern observateur/observé vue en cours qui utilise une zone de détection et notifie les observateurs lorsqu'un observé entre, est ou sort de la zone. Nous l'avons adapté à notre cas pour qu'il soit le plus générique possible.

Choix de l'architecture

Ici chaque agent pourra créer une ou plusieurs zones de détection autour de lui. Ces zones sont composées d'une région géométrique, d'un parent (l'agent qui a créé la zone) et d'une liste d'acteurs observées (autre que le parent). Nous avons ajouté trois géométrie de zone: une région angulaire, une région sphérique et une région cylindrique et avons implémenté trois types de zones de détection (Nimbus, Focus et Area) qui héritent chacune de la classe abstraite Trigger et redéfinissent la manière d'évaluation des observés dans la zone et de notification.



Pourquoi la notification est-elle spécifique ?

Comme expliqué plus haut, il existe une différence entre l'architecture vue en cours et la nôtre. La classe Trigger n'a pas de liste observateur. C'est dû au fait que pour le focus et l'Area, représentant respectivement la vue et un espace d'interaction, l'acteur notifié est lui-même, le parent de cette zone. Les autres acteurs n'ont pas à être notifié lorsqu'un acteur observé entre ou sort de sa zone. De plus pour le cas du Nimbus, il s'agit d'une zone de détection ou seul l'observé qui a pénétré dans le Nimbus d'un autre acteur doit en être informé, l'observateur aurait été le même que l'observé. Par conséquent, comme il y a une redondance non nécessaire, nous avons décidé de ne pas la conserver.
Cependant, cela nécessite pour chaque type de détection à devoir spécifiquement déclarer le trigger, l'agent à notifier (via l'attribut observer) et l'agent observé (via l'attribut observed). Ces informations seront récupérées par la méthode *eval* de Trigger qui se chargera de la notification des acteurs.
La notification s'effectue en appelant les méthodes *onTriggerEnter*, *onTriggerStay* et *onTriggerExit* des acteurs à notifier. Ces acteurs devront redéfinir ces méthodes de leur côté afin de pouvoir recevoir ces informations.

```
notify(observed) {
  return {
    'trigger': this,
    'observer': observed,
    'observed': this.parent
  }
}
```

Spécification des notifications d'un Nimbus

```
notify(observed) {
  return {
    'trigger': this,
    'observer': this.parent,
    'observed': observed
  }
}
```

Spécification des notifications de Focus et Area

Pourquoi la méthode d'évaluation des acteurs observés est-elle spécifique ?

Un capteur est composé d'une région qui définit la zone de détection. Lors de l'évaluation (méthode eval de la classe Trigger appelé depuis la simulation), on cherche a savoir si l'observé est contenu dans la zone. Pour cela, on va généralement regarder si la position de l'acteur observé est contenu dans cet espace géométrique. Cette technique fonctionne bien pour le Nimbus et le Focus mais malheureusement pas pour l'Area. En effet pour l'Aire on doit détecter si une partie de l'Aire de l'observé est contenu dans l'Aire de l'observateur. Le point n'est donc plus le centre de l'observé mais l'extrémité de la zone de l'observé la plus proche de l'observateur. On redéfinit alors la méthode *testOnObserved* de la manière suivante:

```
testOnObserved(observed) {
  const trigger = observed.getTrigger(Area);
  if (trigger) {
    const radius = trigger.region.radius;
    const dir = new THREE.Vector3();
    dir.subVectors(this.region.object3d.parent.position, observed.position);
    dir.divideScalar(dir.length()).multiplyScalar(radius);
    return observed.position.clone().add(dir)
  }
  else return observed.position;
}
```

Affichage de la zone de détection

Afin d'évaluer correctement nos implémentations et vérifier que nos capteurs fonctionnent correctement suivant la géométrie choisie, nous avons rendu visible ces régions par un mesh de ces figures en fil de fer de la couleur du parent.
Cette fonction est activable en appuyant sur les touches suivantes :

Touche	Rôle
0	Active / désactive l'affichage des capteurs de l'ensemble des acteurs
1	Active / désactive l'affichage des capteurs de l'humain
2	Active / désactive l'affichage des capteurs des Pingouins
3	Active / désactive l'affichage des capteurs des bébés Pingouins
4	Active / désactive l'affichage des capteurs de l'herbe
5	Active / désactive l'affichage des capteurs des phéromones

Cela permet de mieux se rendre compte de l'espace occupé par ces zones.

Pour faciliter la déclaration des triggers, nous avons décidé de ne pas avoir besoin de spécifier directement la liste des acteurs observés. En effet, dans la fonction setTrigger de la classe Acteur, on spécifie juste le type de Trigger, sa région et des options liés à la zone. La liste des acteurs observés sera défini lors de l'appel de la méthode *eval* du Trigger. Pour des raisons de performance et pour la raison que nous n'ajoutons pas en cours de route des acteurs qui nécessiteraient à devoir remettre à jour l'ensemble des listes observed de chaque capteur, nous avons limité la mise à jour de cette liste lorsque celle-ci est vide (c'est à dire au premier pas de simulation).

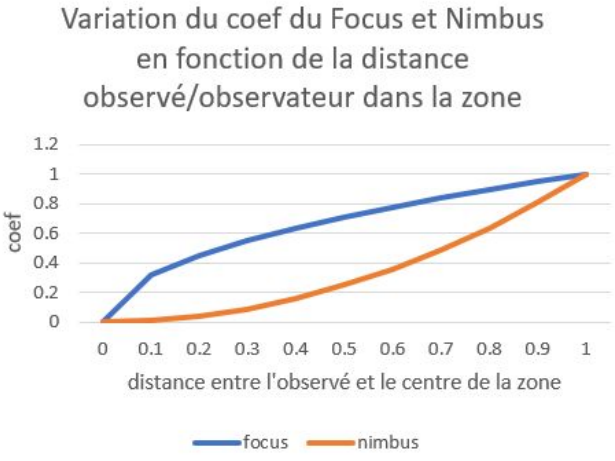
Pour des raisons de performance, un filtre peut être appliqué sur les capteurs en renseignant le quatrième paramètre de la fonction *setTrigger* afin de sensibiliser le capteur à des types d'acteurs que l'on souhaite observés. Par défaut un capteur observe l'ensemble des acteurs de la simulation. Cela permet de réduire les évaluations sur des acteurs qui n'intéressent pas les observateurs et par conséquent d'améliorer les performances.

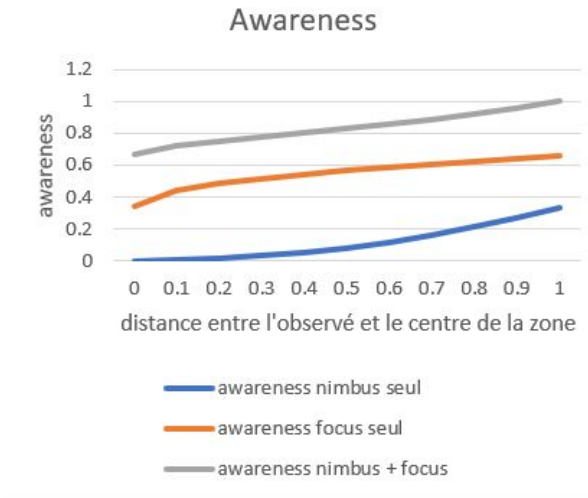
Awareness

Une fois que l'analyse de l'environnement est réalisée, chaque acteur peut prendre une décision et agir en conséquence. Ici les seuls agents qui doivent prendre une décision sont les Penguins. Étant donné que les pingouins ont un Nimbus et Focus, on peut créer un niveau de conscience pour chaque acteur observé afin de les aider dans leur décision à venir.
Pour ce faire, on définit que l'awareness est une valeur comprise entre 0 et 1. Plus cette valeur sera élevée, plus le pingouin aura conscience de cet objet observé.

A cet instant, nous avons décidé de modifier la variation du coef de chacune des zones de la manière suivantes :

- Area: $1 - (distanceBetween(parent,\ observed) / zoneRadius)$ (coef de base dans Trigger)
- Focus: $\sqrt{1 - (distanceBetween(parent,\ observed) / zoneRadius)}$
- Nimbus: $(1 - (distanceBetween(parent,\ observed) / zoneRadius))^{2}$





En effectuant ce changement, l'observateur est capable de percevoir plus rapidement un acteur lorsqu'il se trouve dans son focus (même très loin) et ressent un acteur que lorsque celui-ci est proche de lui.
A gauche, nous avons affiché une représentation graphique des variations de coefficient possible pour chaque type de capteur en fonction de la distance en pourcentage entre l'observé et l'observateur. Plus l'observé est proche de l'observateur (centre de la zone de détection), plus le coefficient du capteur sera égale à 1.
Ensuite pour obtenir une valeur de l'awareness comprise entre 0 et 1 à partir des valeurs combinées de nos coefficients Nimbus et Focus précédents, nous avons d'abord défini dans un tableur les différents paliers que celle-ci devait obtenir.

Palier	début	fin
nimbus + focus	0.67	1
focus seul	0.34	0.66
nimbus seul	0	0.33
aucun	0	0

Puis nous avons calculé l'awareness de B pour A de la façon comme suit:

- Nimbus seul: $Aw_A(B) = (0.33 - 0) * coefNimbus + 0$
- Focus seul: $Aw_A(B) = (0.66 - 0.34) * coefFocus + 0.34$
- Focus + Nimbus = $Aw_A(B) = (1 - 0.67) * Moyenne(coefFocus,\ coefNimbus) + 0.67$

Pour appliquer la fonction d'awareness à un acteur, il faut au préalable lui ajouter le composant Awareness et les capteurs focus et nimbus. Ensuite il ne faut pas oublier de mettre à jour les listes *InFocus* et *InNimbus* du composant en fonction des infos reçus des callback *onTriggerEnter (pour ajouter un acteur à l'une des listes)*, *onTriggerStay (pour mettre à jour le coef de l'acteur)* et *onTriggerExit (pour retirer l'acteur à l'une des listes)*. A chaque tour de boucle, il se chargera de calculer la nouvelle valeur d'awareness pour chaque acteur détecté. Les méthodes *getAll* et *getNearest* du composant permettent respectivement de récupérer l'ensemble des valeurs d'awareness de chaque acteur (dans l'ordre décroissant) et le type d'acteur le plus proche.

Pingouin et Machine à état

Les pingouins sont ici des acteurs plus complexes que nos acteurs Grass et Rock. En effet, ils sont capables de percevoir d'autres acteurs, d'en évaluer une "présence" de chacun afin de prendre une décision et d'effectuer une action. Afin de définir plus facilement les actions, nous utilisons une machine à état. Comme nous avons défini des composants pour déplacer, contrôler la vitesse des acteurs, chaque état activera à son entrée les composants dont il aura besoin et les désactivera à la sortie de l'état.

Un pingouin effectuera les actions suivantes:

Etat	Composants	Description
Wander	RandomTarget Arrive Seek	Il s'agit de son état par défaut. Il choisit au hasard une destination aléatoire (attribuée avec RandomTarget) et s'y rend (va en direction de la target avec Seek et ralenti avec Arrive à l'approche de sa destination). Il changera régulièrement de destination.
Search Food	Arrive Seek RandomTarget	Il cherchera un acteur Grass à manger lorsque son niveau de faim sera en dessous de 40%. Dès qu'il aura conscience d'un acteur Grass à plus de 66% (cad dans son focus), alors il se dirigera vers lui (Seek) et s'arrêtera (Arrive) lorsque son aire aura touché l'aire de l'acteur Grass.On utilise ici RandomTarget pour qu'il déplace de manière random tant qu'il n'a pas trouvé de nourriture.
GoTo Penguin	Arrive Seek	Se rapproche (Seek) vers un pingouin jusqu'à ce que leurs deux aires se touchent, puis ralentit (Arrive) et s'arrête.
Follow Pheromone	Arrive Seek	Suit (Seek) la phéromone la plus récente et lorsque son aire touche celle d'un autre pingouin, il ralentit (Arrive) jusqu'à s'immobiliser.
Stop	Arrive Seek	Il s'arrête sur place pendant une durée comprise entre 1 et 3 secondes.

Flee	Flee et Abrupt Departure	Fuit l'humain (Flee et AbruptDeparture). Dès que le pingouin en fuite n'est plus dans le nimbus de l'humain, il retourne dans son l'état Wander. Si des pingouins sont à proximité de celui-ci (que leur aire respective ce touche), il les préviendra et changeront d'état pour fuir également..

Le pingouin passe d'un état à un autre suivant le schéma ci-dessous. Pour tous les états (sauf Flee), le pingouin regarde s'il ressent plus de 50% la présence d'un humain. Si c'est le cas, il change d'état pour fuire (Flee). L'ensemble des transitions est décrit sur le schéma ci-dessous. Ces transitions ont la même couleur que le futur état afin de s'y retrouver. Par exemple, le pingouin cherchera à manger que lorsque son niveau de faim est inférieur à 40% et qu'il se trouve en train de se promener, d'aller vers un pingouin ou de suivre des phéromones.

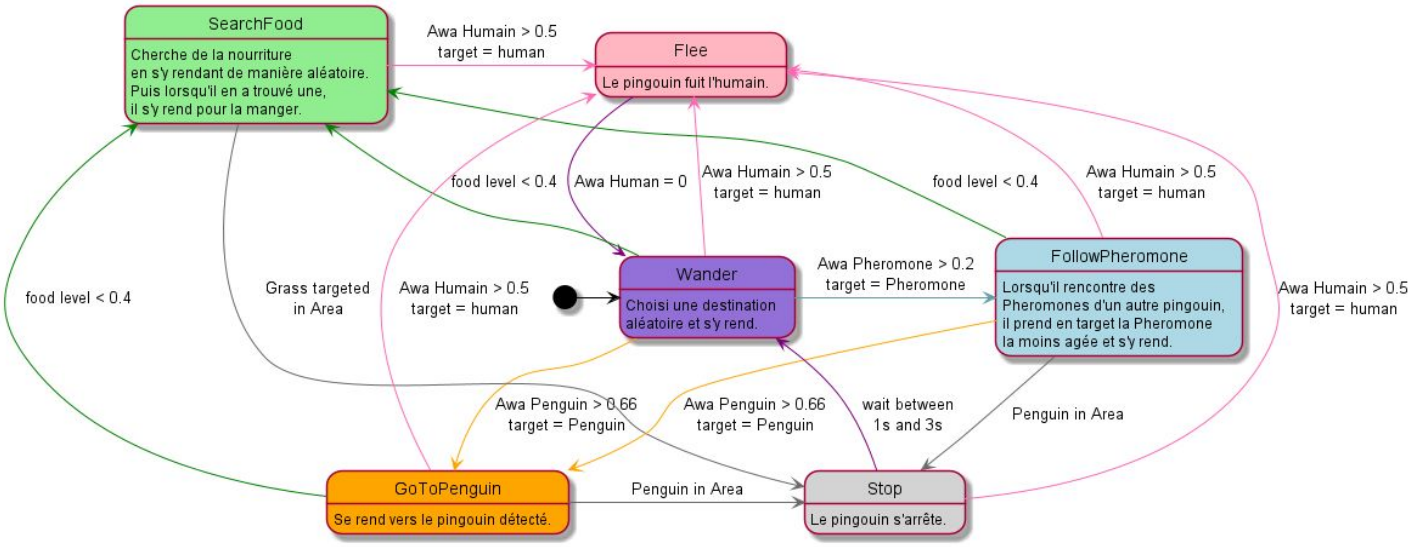


diagramme d'état de la FSM des pingouins

Les bébés Pingouin

Nous avons ajouté des bébés pingouins qui utilisent les concepts introduits par les boids de Reynolds. Pour chaque règle, nous avons créé un composant (Cohesion, Separation et Alignment), décrit dans le fichier Reynolds.js. Pour leur calcul, il est nécessaire de renseigner une liste d'acteurs. Pour les bébés pingouins cette liste provient de leur capteur focus (provenant des fonctions callback onTriggerEnter, etc).
Les règles de séparation et d'alignement sont calculées à partir des positions des pingouins adultes et bébé. Seule la règle de cohésion est calculée à partir des positions des pingouins adultes seuls. Nous avons ajouté cette différence puisque nous avions le souci que les bébés partaient dans une direction à l'infini jusqu'à en sortir du terrain.
Ce changement permet tout de même d'apercevoir les comportements souhaités avec les boids de Reynolds. Les bébés suivront les adultes, éviteront de se rentrer les uns dans les autres et regarderont tous dans la même direction. Et lorsqu'un bébé ne perçoit plus un pingouin adulte, il retourne vers le centre du terrain.

Evitement d’obstacle

Des rochers sont placés de manière aléatoire sur le terrain. Un composant nommé ObstacleAvoidance permet lorsqu’il est appliqué à un acteur de pouvoir éviter les obstacles qu’il perçoit. Il suffira de mettre à jour la liste obstacle du composant avec celle provenant d’un capteur de focus pour que notre composant puisse calculer un vecteur de force afin de corriger la trajectoire initiale et lui faire éviter l’obstacle.
Ce composant est appliqué à tous les pingouins.

Conclusion

L’ensemble du projet est fonctionnel, on peut observer nos bébés pingouins interagir ensemble ainsi qu’avec les pingouins adultes et observer le comportement des boids de Reynolds. Ils suivront les adultes en évitant les rochers sur leur trajectoire. Si le rocher n’est pas trop gros, on peut observer que le groupe est capable de se séparer le temps de passer l’obstacle, puis de se reformer. Les adultes ont également leur propre comportement et fuient l’humain lorsqu’il est dans leur focus.

Globalement, nous sommes très contents du résultat puisque nous avons réussi à implémenter l’ensemble des fonctionnalités demandées (à part la question 8). De plus ce projet est très intéressant, il permet de manipuler des notions que nous ne voyons pas dans les autres matières du master en plus de la découverte de la librairie.

Annexes

Pour résumer, voici l'architecture actuelle de l'application.