CADOUR Erwann **M2 SIIA**

Rapport de Projet sur les Mondes Virtuels

Janvier 2021

Travail réalisé:

Question 1: modifiez la classe Appli du fichier projet.js de façon à créer un terrain sur lequel des touffes d'herbes sont disposées de façon aléatoire.

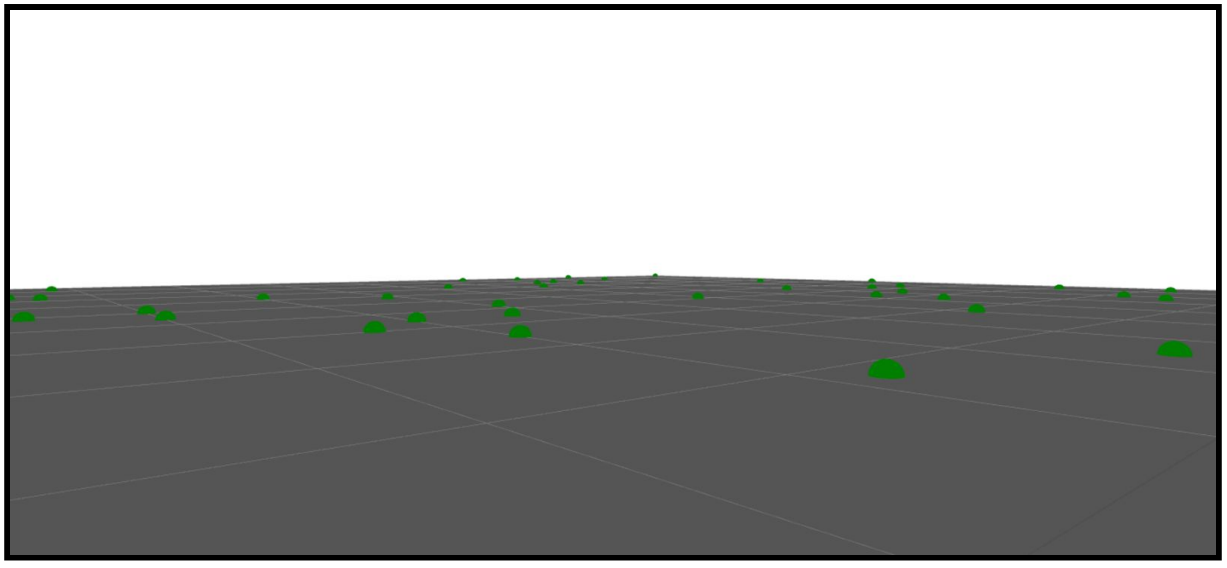
Réponse et déroulement:

Afin de créer des touffes d'herbes disposées de manière aléatoire sur le terrain, il a fallu ajouter une boucle for dans la fonction creerScene() de la classe Appli.

```
// Placement aléatoire d'herbe sur le terrain
for (let i = 0; i < 66; i++) {
    var herbe_num = "herbe" + i;
    var herbe = new Herbe(herbe_num,{},this);
    herbe.setPosition(getRandomInt(50),0,getRandomInt(50));
    this.addActeur(herbe);
}
```

Dans cette boucle, le nombre de touffes d'herbes souhaité est renseigné. Ici, c'est 66.
A chaque tour de boucle, une touffe d'herbe va être créée et ajoutée à la simulation.
Dans un premier temps, on définit un nom pour la touffe d'herbe en cours de création, il s'agit ici du mot clé 'herbe' et du tour de boucle actuel.
Ensuite on crée un acteur herbe avec le nom créé, on lui affecte une position aléatoire dans les limites du terrain.
Enfin, on ajoute ce nouvel acteur de type herbe à la simulation.

Voici le résultat:



Pour pouvoir placer de manière aléatoire ces touffes d'herbes, il a également fallu créer la fonction *'getRandomInt'*. Cette fonction permet d'obtenir une valeur entière aléatoire. On y renseigne la valeur maximale possible et la valeur renvoyée sera située entre cette valeur et son opposé.
Voici le contenu de la fonction: *return Math.floor((Math.random() * (max*2)) - max);*

Question 2: mettez en œuvre une nouvelle classe d'acteurs correspondant à des pingouins qui se déplacent en changeant de façon aléatoire de direction tout en restant dans leur « champ ».

Réponse et déroulement:

Pour mettre en œuvre cette nouvelle classe d'acteurs, il a fallu définir un constructeur pour cette nouvelle classe, puis une fonction actualiser.

Composant:

Étant donné les consignes pour les questions suivantes du sujet, il a été question de créer une classe **composant** comme vu en cours avant de préparer le comportement de ce nouveau type d'acteur.
Il a ensuite fallu ajouter un attribut 'composants' à la classe **acteur**, cet attribut correspondant à un tableau de composants.
Après avoir créé cette nouvelle classe composant, une nouvelle fonction a été ajoutée à la classe **acteur**, il s'agit de la fonction *'addComposant'* qui permet d'ajouter un composant au tableau de composants de l'acteur.
Pour finaliser la mise en place de ce nouveau système de composants pour les acteurs, il a été nécessaire d'adapter la fonction *'actualiser'* de la classe sim pour qu'ils puissent fonctionner. Pour cela, le parcours de l'ensemble des composants de chaque acteur y a été ajouté pour faire appel à la fonction *'actualiser'* de chaque composant.

Déplacement aléatoire:

J'ai ensuite créé une classe composant du nom de CompAlea, à sa construction un composant de type CompAlea se voit fournir en paramètre, l'acteur auquel il est lié.
Ensuite, la fonction actualiser du composant permet de définir le comportement aléatoire que l'on souhaite. Pour cela, on tire une valeur aléatoire. Si celle-ci est inférieure à notre limite, on tire deux nouvelles valeurs aléatoires que l'on affecte aux paramètres x et z de notre vecteur de force. Enfin, cette force est normalisée, multipliée puis appliquée à notre acteur.

Utilisation du composant:

Pour utiliser ce composant et donc que l'acteur se comporte comme on le souhaite, il faut l'ajouter à la liste des composants de notre nouvel acteur.
Voici la ligne de code à ajouter au constructeur de l'acteur:

```
this.addComposant(new CompAlea(this,{}));
```

Question 3: les acteurs sont dotés d'un « nimbus ». Modifiez le comportement des acteurs pingouins de façon à ce qu'ils se déplacent de façon aléatoire mais que dès qu'ils sont dans le nimbus d'un autre acteur leur comportement se modifie :

Réponse et déroulement:

Ajout de la notion de nimbus:

Pour mettre en œuvre les solutions des différentes parties de la question, il est nécessaire d'ajouter la notion de nimbus.
Chaque acteur de la simulation peut posséder un nimbus, on ajoute donc un attribut correspondant à la classe acteur.
Un nimbus est lié à un acteur et possède un rayon qui permet de définir la portée de celui-ci.
Voici le constructeur d'un nimbus:

```
// --- Nimbus
function Nimbus(actor,rayon){
    this.entite = actor;
    this.rayon = rayon || 7;
}
```

Ensuite, j'ai créé une fonction isInNimbus. Cette fonction permet de vérifier si un acteur est positionné dans la zone du nimbus d'un autre acteur. Si c'est le cas, cette fonction renvoie True et False sinon. Voici la fonction:

```
Nimbus.prototype.isInNimbus = function(posCible){
    var CibleDist = this.entite.objet3d.position.distanceTo(posCible);
    if(CibleDist < this.rayon){
        return true;
    } else {
        return false;
    }
}
```

Il est maintenant possible d'ajouter un nimbus à un acteur et de s'en servir pour guider des comportements.

Attirance des touffes d'herbes:

Le premier comportement à ajouter à notre acteur consiste en son déplacement vers une touffe d'herbe si l'acteur de type pingouin se situe dans le nimbus de la touffe d'herbe.

Pour ce faire, un nimbus a été ajouté à chaque touffe d'herbe, ce nimbus possède un rayon de 5 unités. Pour mettre en place ce comportement, un composant est créé comme ce fût le cas pour le déplacement aléatoire, ce composant se nomme AllerVersHerbe et il fonctionne comme suit:

- La liste des acteurs de la simulation est parcourue
- Dès qu'on détecte un acteur de type herbe, on regarde si l'acteur faisant appel au comportement se situe dans le nimbus de l'herbe en question

- Si c'est le cas, un vecteur trajectoire par défaut est créé puis:
    - La distance entre notre acteur et l'herbe est calculée
    - Le vecteur trajectoire prend comme valeur le résultat de la soustraction de la position de la touffe d'herbe et de celle de notre acteur
    - Le vecteur trajectoire est normalisé, on y ajoute une importance liée à la distance entre les deux acteurs
    - On applique le vecteur trajectoire comme force sur notre acteur

Ainsi, lorsqu'un pingouin possédant le comportement AllerVersHerbe entre dans le nimbus d'une touffe d'herbe, il y est attiré.

Consommation des touffes d'herbes:
Après implémentation d'un comportement d'attirance de l'herbe pour le pingouin, il faut désormais lui permettre de la consommer. Pour cela, j'ai une nouvelle fois créé un composant pour mettre en place le comportement souhaité.
Ce composant fonctionne comme suit:

- La liste des acteurs de la simulation est parcourue
- Si on détecte un acteur de type herbe et que notre acteur se situe à moins d'1 unité de celui-ci
- La touffe est mangée : pour cela on la déplace à des coordonnées accessibles par les pingouins

La solution du déplacement de la touffe d'herbe une fois mangée est très loin d'être optimale mais je n'ai pas trouvé d'autre moyen.

Fuite à proximité de l'utilisateur:
Un autre comportement demandé consistait en la fuite d'un pingouin si celui-ci se trouvait trop proche de l'utilisateur. Une nouvelle fois, j'ai créé un composant pour mettre en place ce comportement. Il s'agit du composant FuirCamera.
Ce composant fonctionne comme suit:

- Un vecteur trajectoire par défaut est créé
- La position de la caméra est récupérée et la distance entre notre acteur et celle-ci calculée
- Si la distance calculée est inférieure à 3 unités
    - On affecte au vecteur trajectoire la valeur résultant de la soustraction de la position de l'acteur et de la caméra. Ici la position de l'acteur est la première opérande comme l'on veut que celui-ci s'éloigne de la caméra.
    - Le vecteur trajectoire est normalisé, on y ajoute une importance liée à la distance entre les deux acteurs
    - On applique le vecteur trajectoire comme force sur notre acteur

Ainsi, lorsque le pingouin se retrouve trop près de l'utilisateur (la caméra), il cherche à s'en éloigner.

Attirance entre pingouins:

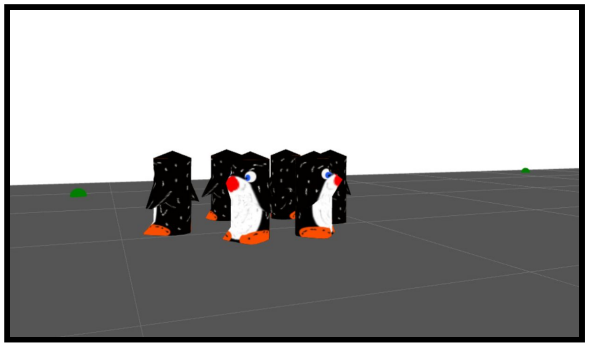
Le dernier comportement à implémenter pour la question était l'attirance des pingouins. Un pingouin se rapproche d'un autre pingouin s'il entre dans son nimbus. Si ceux-ci se retrouvent à distance trop petite l'un de l'autre, ils s'éloignent. Ce comportement est finalement la combinaison de la recherche des touffes d'herbes et de la fuite de l'utilisateur.
Le composant correspondant est appelé AllerVersPing.
Il fonctionne comme suit:

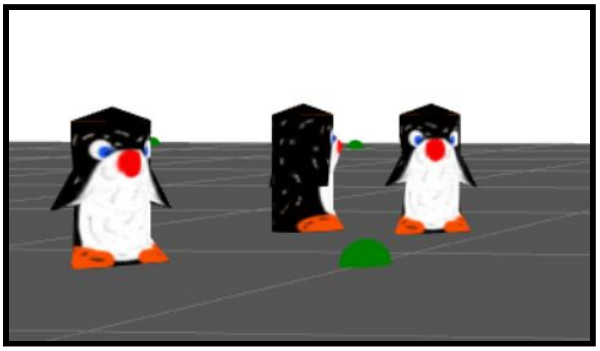
- La liste des acteurs de la simulation est parcourue
- Dès qu'on détecte un acteur de type pingouin, on regarde si l'acteur faisant appel au comportement se situe dans le nimbus du pingouin en question
- Si c'est le cas, un vecteur trajectoire par défaut est créé puis:
    - La distance entre notre acteur et l'autre pingouin est calculée
    - Si les deux acteurs ne sont pas trop proches
        - Le vecteur trajectoire prend comme valeur le résultat de la soustraction de la position du pingouin cible et de celle de notre acteur
    - Si les deux acteurs sont trop proches
        - Le vecteur trajectoire prend comme valeur le résultat de la soustraction de celle de notre acteur et de celle du pingouin cible
    - Le vecteur trajectoire est normalisé, on y ajoute une importance liée à la distance entre les deux acteurs
    - On applique le vecteur trajectoire comme force sur notre acteur

Il est à noter qu'avant de lié une importance au vecteur, on effectue des tests sur la distance entre les acteurs et selon le résultat, il est possible qu'on modifie cette valeur pour une valeur fixe. Ceci est dû à un problème lors du calcul suivant:

```
trajectoire.multiplyScalar(this.attirance/(Math.pow(distToPing,2)));
```

Voici des groupes de pingouins s'étant formés:





Question 4: Cette question a été répondu en partie lors de la 3

Réponse et déroulement:
Une importance est donnée aux différentes cibles des acteurs dans les différents composants de comportement.
Pour le composant AllerVersHerbe, il s'agit de :

```
trajectoire.multiplyScalar(1/(Math.pow(distToHerbe,2)));
```

Pour le composant AllerVersPing, il s'agit de :

```
trajectoire.multiplyScalar(this.attirance/(Math.pow(distToPing,2)));
```

Informations supplémentaires:

Ajout de composants supplémentaires:

CompFrott : ce composant permet d'ajouter la notion de frottement au déplacement des acteurs, comme vu en cours.

```
// Composant de frottement
function CompFrott(entite, opts){
    Composant.call(this,entite) ;
    this.force = new THREE.Vector3(0.0,0.0,0.0) ;
    this.k = opts.k || 0.1 ;
}

CompFrott.prototype = Object.create(Composant.prototype) ;
CompFrott.prototype.constructor = CompFrott ;
CompFrott.prototype.actualiser = function(dt){
    this.force.copy(this.entite.vitesse) ;
    this.force.multiplyScalar(-this.k) ;
    this.entite.applyForce(this.force) ;
}
```

RegardPing : ce composant permet de faire en sorte que l'acteur regarde dans la direction vers laquelle il se déplace. Cela permet d'éviter d'observer des acteurs se déplaçant dans toutes les directions sans jamais s'orienter différemment.

```
// Regard de l'acteur
function RegardPing(entite, opts){
    Composant.call(this,entite);
    this.regard = new THREE.Vector3(0.0,0.0,0.0);
}

RegardPing.prototype = Object.create(Composant.prototype);
RegardPing.prototype.constructor = RegardPing;
RegardPing.prototype.actualiser = function(dt){
    this.regard.copy(this.entite.getPosition());
    this.regard.addScaledVector(this.entite.vitesse,1.0);
    this.entite.objet3d.lookAt(this.regard);
}
```

Affectation des composants à un acteur:

```
// ActeurAlea
function ActeurAlea(nom,data,sim){

    Acteur.call(this,nom,data,sim, new Nimbus(this,30)) ;

    var repertoire = data.path + "/" ;
    var fObj       = data.obj + ".obj" ;
    var fMtl       = data.mtl + ".mtl" ;

    var obj = chargerObj("tux1",repertoire,fObj,fMtl) ;
    this.setObjet3d(obj) ;

    this.addComposant(new RegardPing(this,{}));
    this.addComposant(new CompAlea(this,{}));
    this.addComposant(new CompFrott(this,{}));
    this.addComposant(new AllerVersPing(this,{}));
    this.addComposant(new FuirCamera(this,{}));
    this.addComposant(new AllerVersHerbe(this,{}));
    this.addComposant(new ConsoHerbe(this,{}));

}
```