Projet IEVA : Pingouins

Mathis Baubriaud et Juliette Grosset

13 janvier 2021

Table des matières

1 Question 1

```
// Question 1
var nbrHerbes = 10;
for (var i = 0; i < nbrHerbes; i++) {
  x = pickRandomInt(20);
  y = 0;
  z = pickRandomInt(20);
  var herbe = new Herbe("herbe" + String(i), {couleur:0xaaff55},this);
  herbe.setPosition(x, y, z);
  this.addActeur(herbe);
}
```

FIGURE 1 – Instanciation de touffes d'herbes

On crée une boucle for qui instancie une nouvelle touffe d'herbe à une position aléatoire. On utilise une fonction placée dans un fichier séparé pour choisir un nombre aléatoire :

```
pickRandomInt = function(intRange){
    return Math.floor(Math.random() * intRange);
}
```

FIGURE 2 – Fonction de choix d'un nombre aléatoire

2 Question 2

```
// La classe décrivant les pingouins
// ================================

function Pingouin(nom, data, sim){
  Acteur.call(this, nom, data, sim);

  var repertoire = data.path + "/" ;
  var fObj       = data.obj + ".obj" ;
  var fMtl       = data.mtl + ".mtl" ;
  this.nimbusType = "sphere";
  this.vitesse = 0.05;
  this.FOV_angle = 100;
  this.visionRange = 70;
  this.lastproduction=0;
  this.isFleeing = false;
  this.plantToEat = 'None';
  this.cible = new THREE.Vector3(pickRandomInt(10), 0, pickRandomInt(10));

  var obj = chargerObj("tux1",repertoire,fObj,fMtl) ;
  this.setObjet3d(obj) ;
  this.setNimbus(nom, {rayon: 2, hauteur: 3, opacity: 0.3});
}
```

FIGURE 3 – Fonction de choix d'un nombre aléatoire

De manière similaire à la classe Acteur1, on créer une classe qui décrit le fonctionnement d'un pingouin. Ce dernier possède une vitesse en float et une cible. La cible est placée aléatoirement dans le monde.

```
Pingouin.prototype = Object.create(Acteur.prototype);
Pingouin.prototype.constructor = Pingouin;
Pingouin.prototype.actualiser = function(){
  //moving randomly
  this.objet3d.lookAt(this.cible);
  this.objet3d.translateOnAxis(new THREE.Vector3(0,0,1), this.vitesse);
  if (this.objet3d.position.distanceTo(this.cible) < 0.1){
    this.cible = new THREE.Vector3(pickRandomInt(30), 0, pickRandomInt(30));
  }
```

FIGURE 4 – Fonction actualiser du Pingouin

On définit la fonction “actualiser” du Pingouin. Avec la méthode “lookAt” de la classe Object3D on oriente le Pingouin vers sa cible et avec la méthode “translateOnAxis” on le déplace jusqu’à celle-ci. Lorsqu’il atteint sa cible, une nouvelle est définie avec une position aléatoire dans le champ.

3 Question 3

Pour permettre la détection autour de l’acteur, on lui attribue un nimbus. Celui-ci se présente sous forme d’un cylindre autour de l’acteur :

```
// Affectation d'un nimbus à l'acteur sous une forme géométrique de détection autour de lui
Acteur.prototype.setNimbus = function(nom,  params){
  if(this.objet3d){
    this.nimbus = creerCylindre(nom, params);
    this.objet3d.add(this.nimbus);
  }
}
```

FIGURE 5 – Fonction setNimbus

Pour créer le cylindre, on utilise la méthode CylinderGeometry de THREE.js :

```
function creerCylindre(nom, params){
  params = params || {};
  var rayon = params.rayon || 1;
  var couleur = params.couleur || 0xf47878;
  var hauteur = params.hauteur || 1;

  const geometry = new THREE.CylinderBufferGeometry( rayon, rayon, hauteur, 20 );
  const material = new THREE.MeshBasicMaterial( {color: couleur, transparent: true, opacity: 0.3} );
  const mesh = new THREE.Mesh( geometry, material );

  return mesh;
}
```

FIGURE 6 – Fonction creerCylindre du module utils.js

On instancie le nimbus avec un rayon et une hauteur proportionnel à la taille de l’acteur :

```
this.setNimbus(nom, {rayon: 1, hauteur: 2, couleur: 0xf4ec2e, opacity: 0.3});
```

FIGURE 7 – Instanciation d’un nimbus pour une touffe d’herbe

Deux fonctions sont définies pour faire fonctionner le nimbus :

— Une fonction “nimbusDetection” qui parcourt les nimbus de tous les acteurs présents dans la simulation et qui calcule si deux nimbus sont en contact.

```
Acteur.prototype.nimbusDetection = function(){
  for (let otherActeur of this.sim.acteurs){
    if (otherActeur != undefined && otherActeur.nimbus != null && (otherActeur.objet3d.position != this.objet3d.position)){
        var intersection = cylinderIntersectsCylinder(
          this.objet3d.position.x,
          (this.nimbus.geometry.parameters.height - this.objet3d.position.y) / 2,
          this.objet3d.position.z,
          this.nimbus.geometry.parameters.height,
          this.nimbus.geometry.parameters.radiusTop,
          otherActeur.objet3d.position.x,
          (otherActeur.nimbus.geometry.parameters.height - otherActeur.objet3d.position.y) / 2,
          otherActeur.objet3d.position.z,
          otherActeur.nimbus.geometry.parameters.height,
          otherActeur.nimbus.geometry.parameters.radiusTop
      )
      //console.log(intersection);
      if (intersection !== false){
        this.nimbusBehavior(otherActeur);
      }
    }
  }
}
```

FIGURE 8 – Fonction nimbusDetection d'un Acteur

Pour calculer si deux nimbus sont en contact on utilise une fonction "cylinderIntersectsCylinder" qui renvoi false ou la distance de chevauchement entre deux cylindres qui s'intersectent. Nous avons décidé de faire le calcul d'intersection par nous même en utilisant les paramètres des cylindres. Au préalable, nous avons testé différentes techniques (notamment avec boundingBox et boundingSphere qui sont des composantes de la classe geometry) mais aucune ne remplissait toutes les conditions demandées.

```
cylinderIntersectsCylinder = function( ax, ay, az, ah, ar, bx, by, bz, bh, br ){
  var ah2 = ah / 2;
  var bh2 = bh / 2;

  var overlapY = Math.min( ay + ah2, by + bh2 ) - Math.max( ay - ah2, by - bh2 );
  if ( overlapY < 0 ) return false;

  var x = ax - bx;
  var z = az - bz;

  var distance = Math.hypot( x, z );
  if ( distance > ar + br ) return false;

  var overlapXZ = ar + br - distance;

  var minOverlap, y;

  if ( overlapY < overlapXZ ) {
    minOverlap = overlapY;

    y = Math.sign( ay - by );
    x = 0;
    z = 0;
  } else {
    minOverlap = overlapXZ;

    x /= distance;
    z /= distance;
    y = 0;
  }

  return {
    minOverlap: minOverlap,
    mtvX: x,
    mtvY: y,
    mtvZ: z,
  };
}
```

FIGURE 9 – Fonction cylinderIntersectsCylinder

— La seconde fonction associée au nimbus est “nimbusBehavior” qui va être spécifiée pour chaque acteur en fonction du comportement voulu. Voici la fonction pour le comportement des pingouins :

```
Pingouin.prototype.nimbusBehavior = function(otherActeur){
  if(otherActeur.nom.startsWith("herbe") && otherActeur.isEaten == false || this.plantToEat == otherActeur.nom){
    // state machine so that when one pingouin is going to eat the grass,
    // others wont go for it
    this.plantToEat = otherActeur.nom;
    otherActeur.isEaten = true;
    console.log(this.nom, ' eat ', otherActeur.nom);
    this.cible = otherActeur.objet3d.position;
    if (this.objet3d.position.distanceTo(this.cible) < 0.1){
      //destroy the grass
      this.sim.destroy(otherActeur);
    }
  }
  else if(otherActeur.nom.startsWith("pingouin")){
    //pingouin is going close to the other pingouin
    if (this.objet3d.position.distanceTo(otherActeur.objet3d.position) > 5){
      //move towards the other pingouin
      this.cible = otherActeur.objet3d.position.clone();
    }
  }
}
```

FIGURE 10 – Fonction nimbusBehaviour

Si le nimbus détecté est celui d’une herbe, alors le pingouin se dirige vers celle-ci puis la mange. On définit grâce à un système de machine à état que si un pingouin se dirige vers une herbe pour la manger alors les autres pingouins ne sont plus intéressés par cette herbe. Si c’est celui d’un autre pingouin, alors on se dirige vers celui-ci. Pour détruire l’herbe, on utilise cette fonction :

```
Sim.prototype.destroy = function(act){
  this.acteurs.splice(this.acteurs.indexOf(act), 1);
  act.setVisible(false);
  this.scene.remove(act.objet3d);
}
```

FIGURE 11 – Fonction destroy

Il ne nous reste plus qu’à ajouter le comportement pour fuir l’utilisateur. Pour ce faire, on va utiliser la position de la caméra. On définit également que la fuite de l’utilisateur prévaut sur tous les autres comportements.

```
Pingouin.prototype.actualiser = function(){
  //moving randomly
  this.objet3d.lookAt(this.cible);
  this.objet3d.translateOnAxis(new THREE.Vector3(0,0,1), this.vitesse);
  if (this.objet3d.position.distanceTo(this.cible) < 0.1){
    this.cible = new THREE.Vector3(pickRandomInt(30), 0, pickRandomInt(30));
    // state machine to give the priority to fleeing the user above all other behaviors
    this.isFleeing = false;
  }
  //fleeing the user
  if (this.objet3d.position.distanceTo(this.sim.camera.position) < 5)
  {
    this.cible = this.sim.camera.position.clone();
    //selecting a point by linear interpolation between the two vector3
    this.cible.lerp(this.objet3d.position, 2);
    this.cible.setY(0);
    this.isFleeing = true;
  }
  //associating an action for actor in nimbus
  if (this.isFleeing == false){
    this.nimbusDetection();
  }
}
```

FIGURE 12 – Fonction actualiser du pingouin

À la fin de la question 3, voici l'état de notre projet avec les différents comportements qui fonctionnent :

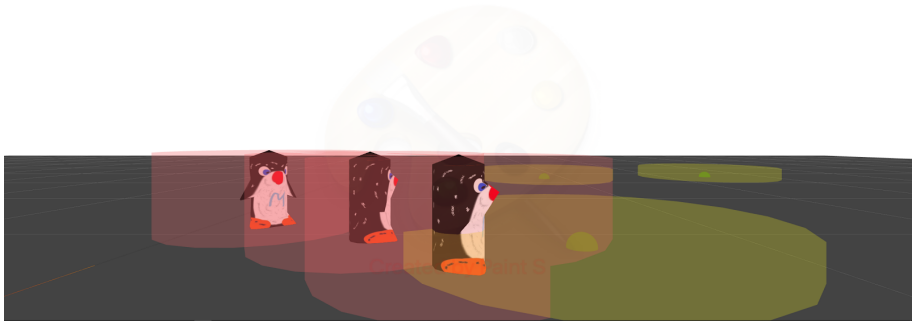

FIGURE 13 – Visualisation à la fin de la question 3

4 Question 4

Nous avions déjà une fonction pour créer des sphères ou des cylindres, de la même façon, on implémente une fonction pour créer des cônes :

```
function creerCone(nom, params){
  params = params || {};
  var rayon = params.rayon || 1;
  var couleur = params.couleur || 0xf47878;
  var hauteur = params.hauteur || 1;

  const geometry = new THREE.ConeBufferGeometry( rayon, hauteur, 20 );
  const material = new THREE.MeshBasicMaterial( {color: couleur, transparent: true, opacity: 0.3} );
  const mesh = new THREE.Mesh( geometry, material );

  return mesh;
}
```

FIGURE 14 – Fonction creerCone

On ajoute un attribut aux acteurs "nimbusType" qui peut être redéfini, sa valeur d'origine sera "cylindre", mais on peut au choix lui passer "sphère" ou "cône" (en modifiant les paramètres associés si nécessaire). On modifie la fonction qui créer le nimbus en prenant en compte le type de la géométrie.

```
// Affectation d'un nimbus à l'acteur sous une forme géométrique de détection autour de lui
Acteur.prototype.setNimbus = function(nom,  params){
  if(this.objet3d){
    if (this.nimbusType == "cylindre"){
      this.nimbus = creerCylindre(nom, params);
    }
    else if(this.nimbusType == "sphere"){
      this.nimbus = creerSphere(nom, params);
    }
    else if (this.nimbusType == "cone"){
      this.nimbus = creerCone(nom, params);
    }
    this.objet3d.add(this.nimbus);
  }
}
```

FIGURE 15 – Fonction setNimbus avec attribut possible pour la géométrie

Désormais, il nous faut gérer la collision entre toutes ces formes, ces méthodes sont définies dans le fichier "utils.js" et voici un exemple pour une interaction entre une sphère et un cylindre :

```
function sphereIntersectsCylinder( sx, sy, sz, sr, cx, cy, cz, ch, cr ) {

  var ch2 = ch / 2;
  var overlapY = Math.min( sy + sr, cy + ch2 ) - Math.max( sy - sr, cy - ch2 );
  if ( overlapY < 0 ) return false;
  var newRadius; var h1, h2;

  if ( overlapY < sr ) {
    h1 = sr - overlapY;
    newRadius = Math.sqrt( sr * sr - h1 * h1 );}
  else { newRadius = sr; }

  var x = sx - cx; var z = sz - cz;
  var distance = Math.hypot( x, z );
  if ( distance > newRadius + cr ) return false;
  var overlapXZ = newRadius + cr - distance;
  var minOverlap, y;

  if ( overlapY < overlapXZ ) {
    minOverlap = overlapY;
    y = Math.sign( sy - cy );
    x = 0; z = 0;}
  else if ( overlapY < sr ) {
    var newerRadius = newRadius - overlapXZ;

    h2 = Math.sqrt( sr * sr - newerRadius * newerRadius );
    minOverlap = h2 - h1;
    y = Math.sign( sy - cy );
    x = 0; z = 0;}

  else {
    minOverlap = overlapXZ;
    x /= distance;
    z /= distance;
    y = 0;}

  return {
    minOverlap: minOverlap,
    mtvX: x,
    mtvY: y,
    mtvZ: z,
  };
}
```

FIGURE 16 – Fonction sphereIntersectsCylinder

Ensuite, en fonction du type des nimbus présents dans la simulation, on appelle la fonction de collision correspondante :

```
else if (this.nimbusType == "sphere" && otherActeur.nimbusType == "cylindre"){
  var intersection = sphereIntersectsCylinder(
    this.objet3d.position.x,
    (this.nimbus.geometry.parameters.radius - this.objet3d.position.y) / 2,
    this.objet3d.position.z,
    this.nimbus.geometry.parameters.radius,
    otherActeur.objet3d.position.x,
    (otherActeur.nimbus.geometry.parameters.height - otherActeur.objet3d.position.y) / 2,
    otherActeur.objet3d.position.z,
    otherActeur.nimbus.geometry.parameters.height,
    otherActeur.nimbus.geometry.parameters.radiusTop
  )
}
```

FIGURE 17 – Fonction sphereIntersectsCylinder

Voici à quoi ressemble l'environnement avec des cônes pour les herbes et des sphères pour les pingouins :

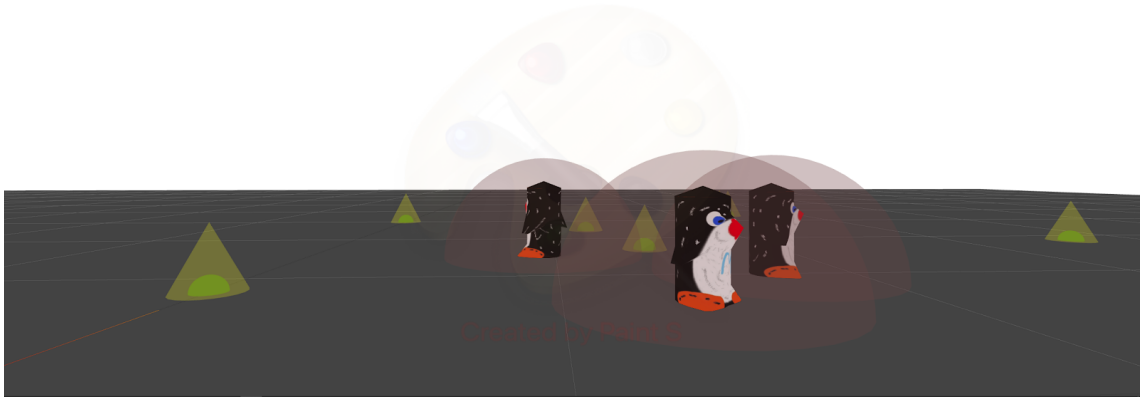

FIGURE 18 – Visualisation à la fin de la question 4

Dans nos fonctions pour détecter si deux nimbus s'inter sectionnent, dans le cas de collisions, la valeur du chevauchement est renvoyée. C'est cette valeur qui nous permet de définir l'intensité d'un acteur présent dans le nimbus d'un autre, plus elle est grande, plus il est proche.

5 Question 5

Pour ajouter un champ de vision à nos pingouins, nous définissons deux nouveaux attributs à celui-ci qui sont la distance à laquelle il peut voir "visionRange" et son angle de vue "FOV_angle". Si l'on prend comme exemple ce schéma :

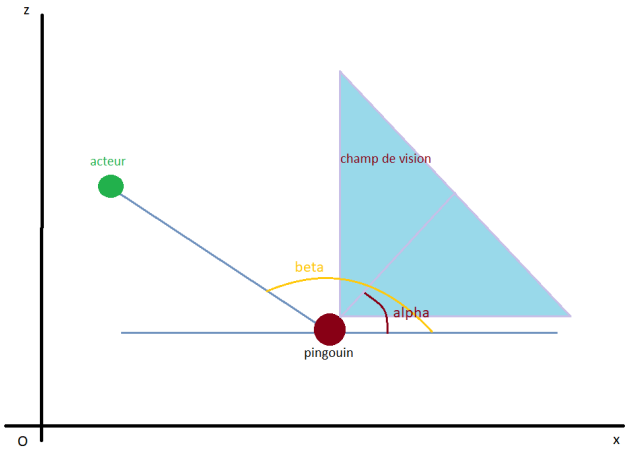

FIGURE 19 – Champ de Vision

Alors calculer si un acteur est présent dans le champ de vision d'un pingouin revient à calculer : — — champvision / 2. Nous implémentons ce calcul dans une méthode "champVision" (voir la figure ci-dessous). Ensuite on peut imaginer que si un pingouin est détecté dans le champ de vision alors le pingouin ira vers lui en priorité.

```
Pingouin.prototype.champVision = function(otherPos){
  var pos = this.objet3d.position;
  // b is the angle also called field of view
  var b = Math.atan2(otherPos.z-pos.z, otherPos.x-pos.x) * 180 / Math.PI;
  var beta = (b+360)%360;
  var alpha = (this.FOV_angle+360)%360;
  var delta = Math.abs(beta-alpha);
  return (delta <= this.visionRange/2);
}
```

FIGURE 20 – Fonction champVision d'un Pingouin

6 Question 6

On crée une nouvelle classe d'acteur "Pheromone" qui à la forme d'une sphère bleue et qui dispose d'un nimbus ainsi que d'un attribut pointant sur le pingouin qui l'a lâché.

```
function Pheromone(nom,data,sim,pingouin){
  Acteur.call(this,nom,data,sim) ;

  var rayon   = data.rayon || 0.1 ;
  var couleur = data.couleur || 0x0000ff ;
  this.horloge = 0;
  this.pingouin = pingouin;
  this.nimbusType = "sphere";
  this.isEaten = false;
  var sphere = creerSphere(nom,{rayon:rayon, couleur:couleur, opacity: 0.9}) ;
  this.setObjet3d(sphere);
  this.setNimbus(nom, {rayon: 0.1, couleur: couleur, opacity: 0.00001});
}
Pheromone.prototype = Object.create(Acteur.prototype) ;
Pheromone.prototype.constructor = Pheromone ;
```

FIGURE 21 – Classe d'acteur Pheromone

On actualise l'opacité des phéromones tout les x temps et on les détruit passé une opacité de 0 :

```
Pheromone.prototype.actualiser = function(){
    if(this.sim.horloge-this.horloge>0.3){
        this.horloge=this.sim.horloge;
        var opacity=this.objet3d.material.opacity;
        opacity-=0.1;
        this.objet3d.material.opacity=opacity;
        if(opacity<=0){
             this.sim.destroy(this);
        }
    }
}
```

FIGURE 22 – Fonction actualiser de la classe acteur Pheromone

On définit une méthode “déposerPheromone” dans la classe Pingouin pour instancier des phéromones dans le sillon des pingouins.

```
Pingouin.prototype.deposerPheromone = function(){
    if(this.sim.horloge-this.lastproduction>0.3){
        this.lastproduction=this.sim.horloge;
    var sph = new Pheromone(this.nom,{},this.sim) ;
    var pos = this.objet3d.position;
    x = 0.5 + Math.random() + pos.x;
    y = Math.random() + pos.y;
    z = pos.z;
    var pheromone = new Pheromone("pheromone", {couleur:0x0000ff},this.sim, this);
    pheromone.setPosition(x, y, z);
    this.sim.addActeur(pheromone);
    }
}
```

FIGURE 23 – Methode deposerPheromone de la classe Pingouin

Il nous reste plus qu’à ajouter le comportement des pingouins quand ils rencontrent un phéromone. Celui-ci va alors aller vers le pingouin responsable.

```
else if(otherActeur.nom.startsWith("pheromone")){
  //check if it is not our own pheromone
  if (otherActeur.pingouin != this){
    //move towards the pingouin who dropped the pheromone
    this.cible = otherActeur.pingouin.objet3d.position.clone();
  }
}
```

FIGURE 24 – Ajout du comportement pour un pingouin en vue d’un pheromone

Désormais, la simulation ressemble à la figure ci-dessous. Le souci étant que les pingouins peuvent sentir leurs phéromones mutuellement, les poussant à aller l’un vers l’autre et à rester bloqués. Pour régler ce problème, j’ai opté pour l’utilisation d’un timer qui agit dans ce cas. Ainsi le pingouin qui suit les phéromones d’un autre va changer de direction si ce dernier se met à suivre les siennes.

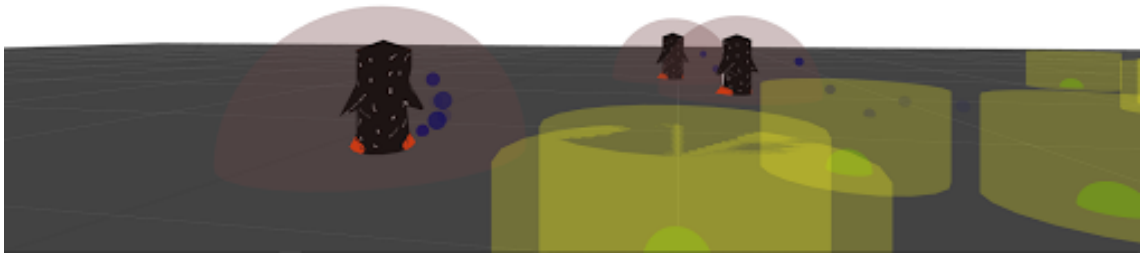

FIGURE 25 – Visualisation à la fin de la question 6

Table des figures