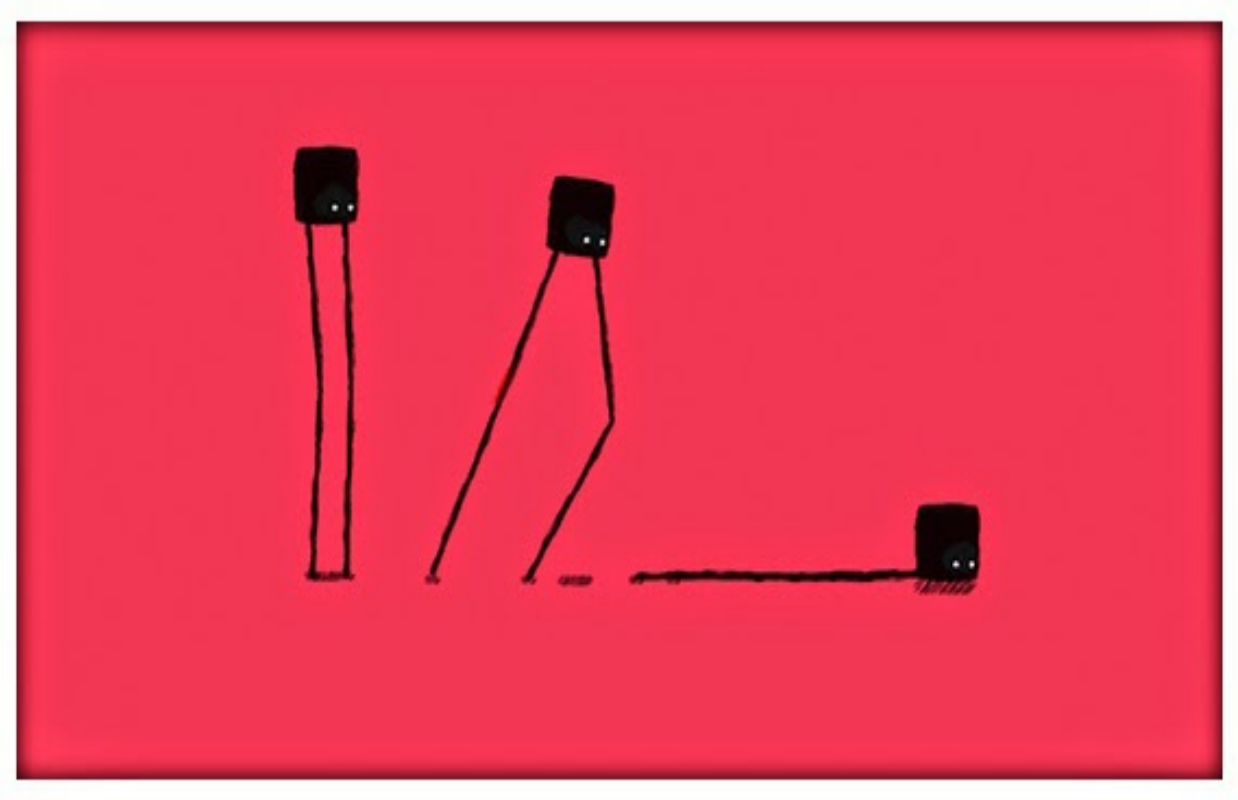
Compte Rendu

Chloé Leric et *Victor Duvivier*.

Notre projet consiste en l'application d'un ou de plusieurs algorithmes génétiques sur un mini-jeu. Nous nous sommes inspirés du mini-jeu Daddy Long Legs



Sujet

Le but de notre algorithme est de générer un ensemble de `moustiks` et de les faire jouer dans un environnement physique 2D. Un algorithme génétique est ensuite appliqué pour améliorer les performances de déplacement des `moustik`. Le critère utilisé pour départager les individus est la distance (positive) maximale parcourue par le `moustik`.

Déroulement du projet

Nous avons travaillé sur ce projet en 2 grandes étapes qui se sont superposées au milieu du projet. La première partie du projet a été de modéliser l'environnement physique et les déplacements du personnage, et la seconde d'implémenter un algorithme génétique qui permettrait de faire avancer le plus loin possible notre personnage. Durant environ le premier

mois, nous nous sommes concentrés uniquement sur le fait d'implémenter les mouvements du moustik, contrôlés par l'usager. Notre approche a été tout d'abord d'imiter un environnement physique, sans pour autant faire entrer en jeu des concepts d'accélération ou d'inertie, simplement en faisant "tomber" le moustik à chaque fois que sa jambe effectuait une rotation par l'utilisateur. Mais rapidement, nous nous sommes rendu compte que cette méthode laissait place à trop d'incertitude et de subjectivité quant au déplacement du moustik : le moindre changement dans une des variables d'entrée du fonctionnement du

Description du code

Description classes

Les différentes classes existant dans notre code sont:

- `Forme` cette classe correspond à un rectangle physique. Elle permet d'automatiser la création de boîtes physiques et facilite l'écriture des fonctions d'affichage de ces formes sous `OpenGL`.
    - Cette classe contient un attribut `b2Body* body` issu de la librairie moteur physique `Box2D` qui correspond à un point 2D de l'espace.
    - Le constructeur rattache à ce corps une `Box` qui vas permettre de gérer les collisions et de donner un "*volume*" à la boîte. Cette boîte est de demie-largeur `width` et de demie-hauteur `height`.
    - L'attribut `type` définit si l'objet créé est fixe ou si il est dynamique (le sol est fixe, le `moustik` est dynamique).
    - `diag` et `angleIn` correspondent à des données géométriques invariantes, mais utilisées lors du processus d'affichage. Pour éviter de faire des calculs trigonométriques coûteux en processus inutilement, on préfère stocker ces valeurs directement en attribut.

- `Genome` cette classe contient l'information de notre moustique
    - Une séquence d'entiers `vector<int>` correspondant aux frames où le `moustik` change d'appui.
    - Ainsi qu'un réel `fitness` qui correspond à la distance maximale atteinte par le `moustik` avec cette séquence *(le constructeur fixe sa valeur à -1)*.
    - `tauxMutation` correspond à la probabilité qu'a un `moustik` de muter lorsqu'il rentre dans la phase de mutation.

- `Moustik` cette classe correspond au concept de notre animal bipède.

    - Il possède trois attributs `Forme`, correspondant à sa tête et à ses deux jambes.

- Il possède deux attributs de type `b2RevoluteJoint` permettant de définir les points de liaison entre la tête et chacune des jambes, les rotules peuvent aussi être activées en tant que moteur en définissant une vitesse de rotation désirée ainsi qu'un couple maximal.

    - L'attribut `com` correspond à la commande utilisateur. Il vaut 0 initialement *(aucune jambe commandée)* puis il oscille entre 1 et 2 pendant le jeu *(jambe gauche et jambe droite activée respectivement)* . Un appui sur s applique la transformation suivante : $com=1+com\% 2$

    - Le booléen `dead` correspond à si le `moustik` est tombé ou non.

    - Le réel `angleMax` correspond à l'angle maximal toléré pour les rotules.

    - L'attribut `controlType` vaut "human" ou "IA" et est utilisé pour l'écriture dans des fichiers texte des génomes respectifs.

    - Un attribut `génome` toutes les données génétiques nécessaires.

- Le booléen `seqWritten` permet de n'écrire qu'une fois la séquence de jeu du moustique.

- `MoustikIA` correspond à un `moustik` géré par l'ordinateur.

    - un entier `int` "ID" permet de le différencier de ses camarades générés par dizaines.

- `Population` correspond à une génération de moustiks.

- Cette classe contient donc une liste `vector<MoustikIA*>` .

    - Ainsi que l'entier `generation` correspondant à la génération de cette population.

Diagramme de classe

`+` public
`#` protected

```
classDiagram
Population--MoustikIA
MoustikIA--Genome
Moustik--|>MoustikIA
Moustik--Coord
Moustik--Forme
```

```
class Coord {
  + float x
  + float y
  + Coord()
  + Coord(float, float)
}

class Forme {
  # b2Body* body
  # float width
  # float height
  # int type
  # float diag
  # float angleIn
  + Forme(b2World*, Coord, float, float, int)
  + Forme(b2World*, Coord, int)
  + ~Forme()
  + float getX()
  + float getY()
  + Coord getPos()
  + b2Body* getBody()
  + float getAngle()
  + float getAngleIn()
  + float getDiag()
  + Coord getHL()
  + Coord getHR()
  + Coord getTL()
  + Coord getTR()
  + GLvoid drawOpenGL()
  + GLvoid drawOpenGL(float, float, float)
}

class Moustik {
  # Forme* ptrHead
  # Forme* ptrLegL
  # Forme* ptrLegR
  # b2RevoluteJoint* rotuleL
  # b2RevoluteJoint* rotuleR
  # int com
  # bool dead
  # float angleMax
  # string controlType
  # Genome* genome
  # bool seqWritten
  + Moustik(b2World*, Coord)
  + ~Moustik()
  + void commande(b2World*, int)
  + Coord getPos()
  + bool isDead()
  + int getAge()
  + void upAge()
  + Genome* getGenome()
  + void setGenome(Genome*)
  + virtual bool undertaker(int)
  + void updateFitness()
```

```
  + void reset(b2World*)
  + float getAbs()
  + string getType()
  + GLvoid drawOpenGL()
  + void writeSeqDown(int, string, bool)
}

class MoustikIA {
  # int id
  + MoustikIA(b2World*, Coord, Genome, int)
  + MoustikIA(b2World*, Coord, vector[int], int)
  + void setID(string)
  + string getID()
  + void activation(bool)
  + void play(b2World*, int)
  + virtual bool undertaker(int)
}

class Genome {
  # int fitness
  # vector[int] seq
  # float tauxMutation
  + Genome(vector[int])
  + Genome(int)
  + Genome()
  + ~Genome()
  + vector[int] getRelativeSeq()
  + vector[int] getAbsoluteSeq()
  + float getFitness()
  + void setFitness(float)
  + void addAbsoluteDate(int)
  + Genome* crossSplit(Genome*)
  + Genome* crossAvg(Genome*)
  + Genome* mutation()
  + bool betterThan(Genome*)
}

class Population {
  # vector[MoustikIA*] moustiks
  # int generation
  + Population()
  + Population(vector<MoustikIA*>, int)
  + Population(Population*)
  + ~Population()
  + vector<MoustikIA*> getMoustiks()
  + void addMoustik(MoustikIA*)
  + int getGeneration()
  + void setGeneration(int)
  + Population bests(int)
  + Population reproduction(Population)
  + Population mutateGroup(Population)
  + Population getChildren(int)
  + void playLive(int)
  + void playOff()
  + void writeGenomes()
  + vector<Genome*> readGenomes(string)
```

```
}
```

Description méthode algorithme génétique

`getChildren`

La classe `Population` possède une méthode `getChildren` qui permet d'obtenir la génération suivante en renvoyant une nouvelle population.

```
Population* Population::getChildren(int n){
  //renvoie la population fille contenant le même nombre d'individus. Elle a été formée à p
  if (n>moustiks.size()){
    //pour ne pas faire planter l'algorithme
    n=moustiks.size();
  }
  //on trie la population active selon leur fitness et on récupère les n meilleurs individu
  Population* bestPop = bests(n);
  //on produit n enfants à partir des n meilleurs parents
  Population* newGeneration = reproduction(bestPop);
  while (newGeneration->getMoustiks().size()<moustiks.size()){
    //on complète la population fille avec les meilleurs parents pris aléatoirement.
    newGeneration->addMoustik(bestPop->getMoustiks()[rand()%n]);
  }
  //chaque date de chaque génome a une probabilité d'évolution.
  newGeneration = mutateGroup(newGeneration);
  return newGeneration;
}
```

`bestPop`

La méthode `bestPop` issue de la classe `Population` est simplement un algorithme de tri suivi d'un remplissage d'un vecteur. Peu spécifique, nous ne détaillerons pas son fonctionnement ici.

`reproduction`

On fait ensuite appel à la méthode `reproduction`. Celle-ci produit une population aussi nombreuse que le nombre de parents qu'on lui fait entrer. On détermine aléatoirement 2 parents que l'on fait se reproduire selon la méthode `crossAvg` et l'on répète cette opération jusqu'à avoir créé suffisamment d'enfants.

```
Population* Population::reproduction(Population pop){
```

```
    //size = nombre de parents
    int size=pop.getMoustiks().size();
    //on vas générer autant d'enfants que de parents
    vector<MoustikIA*> children = pop.getMoustiks();
    //pour initialisation des nouveaux moustiks, on a besoin d'un pointeur vers une classe is
    b2World* ptrWorldIAs = moustiks[0]->getWorld();
    for (int i=0;i<size;i++){
      //on cherche un papa et une maman pour l'enfant au hasard.
      int dad=rand()%size;
      int mom=rand()%size;
      while (dad==mom){
        //papa et maman doivent être différents.
        mom=rand()%size;
      }
      //on croise papa et maman pour donner un enfant
      Genome* genomei = pop.getMoustiks()[dad]->getGenome()->crossAvg(pop.getMoustiks()[mom]-
      //on instancie le fils dans l'environnement
      children[i] = new MoustikIA(ptrWorldIAs, Coord(0.0,3.0), genomei, to_string(i));
      //on le désactive pour ne pas que les collisions ne soient pas calculées
      children[i]->activation(false);
    }
    return new Population(children, pop.getGeneration()+1);
  }
```

`crossAvg`

La méthode appelée à la ligne 17 issue de la classe `Génome` permet de croiser deux parents. Cette méthode produit un enfant génome à partir de deux parents en faisant la moyenne de chaque paire de dates. On a nommé les séquences `litSeq` et `bigSeq` dans l'éventualité où une séquence serait plus grande que l'autre.

Si la séquence du père est plus grande que la séquence de la mère par exemple, alors on moyenne les (nombres-de-dates-dans-la-séquence-de-la-mère) premières dates du père et de la mère et on ajoute ensuite à la séquence fille les dernières dates du père.

```
Genome* Genome::crossAvg(Genome* genome){
  vector<int> seqout;
  vector<int> bigSeq;
  vector<int> litSeq;
  if (seq.size()>=genome->getRelativeSeq().size()){
    bigSeq = seq;
    litSeq = genome->getRelativeSeq();
  } else {
    bigSeq = genome->getRelativeSeq();
    litSeq = seq;
  }
  for (int i=0; i<litSeq.size(); i++){
    //moyenne pondérée des dates par le ratio des fitness
    seqout.push_back((bigSeq[i]+litSeq[i])/2);
```

```
    }
    for (int j=litSeq.size(); j<bigSeq.size();j++){
      //on complète sans moyenner avec la séquence qui est plus grande
      seqout.push_back(seq[j]);
    }
    return new Genome(seqout, 0);
  }
```

`mutateGroup`

Cette méthode permet de modifier les `Genome` de tous les individus aléatoirement. Elle appelle la méthode `mutation` de la classe `Genome` pour tous les `moustik`. La séquence de jeu d'un individu ne doit être constitué que de dates strictement positives car elles correspondent à des intervalles de temps entre lesquels le `moustik` n'a pas changé de jambe. Il ne peut pas non plus changer deux fois de jambe à la même frame.

```
Genome* Genome::mutation(){
  vector<int> mutseq=seq;
  //on parcourt toutes les dates de la séquence.
  for (int i=0; i<seq.size(); i++){
    int prob = rand()%100/100;
    if (prob<tauxMutation){
      //alors mute et se modifie aléatoirement de +/- 5 frames
      int mudate=abs(seq[i]+rand()%11-5); //on veut une valeur positive...
      if (mudate==0){
          mudate=1; //... positive stricte!
      }
      mutseq[i]=mudate;
    }
  }
  return new Genome(mutseq, -1);
}
```

Conclusion

Difficultés rencontrées

Bibliographie

pdf
: NatureOfCode

youtube

: DaddyLongLeg game