DUM č. 6 v sadě

35. Inf-11 Objektové programování v Greenfoot

Autor: Lukáš Rýdlo

Datum: 27.06.2014

Ročník: studenti semináře

Anotace DUMu: Detekce nepřítele, měření úhlu a vzdálenosti mezi objekty, útěk před nepřítelem.

evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Projekt: Simulace života zajíce v Greenfoot
Detekce nepřítele a útěk

Úvod

Poslední podstatná chybějící funkcionalita je detekce nepřítele a potravy. Běžný živočich zpravidla reaguje jinak v případě ohrožení, jinak v případě zpozorování potravy a jinak v klidové situaci. Implementujeme zajíci schopnost na určitou vzdálenost a v určitém poli zpozorovat lišku (případně i potravu) a zareagovat útěkem (nebo zaměřením se směrem k potravě). Podobně necháme lišku cíleně mířit ke zpozorovanému zajíci.

Naprogramování takového chování už vyžaduje rozsáhlejší úvahy a v případě simulace rozsáhlé znalosti o chování simulovaného subjektu. Nabízí se jednoduché řešení spočívající v nastavení postupných podmínek: „Spatříš-li nepřítele, utíkej. Jinak spatříš-li potravu, jdi k potravě. Jinak se chovej náhodně.“ Otázkou zůstává, jak se zachovat, pokud má zajíc už jen málo energie a v blízkosti je potrava. Bylo by logické uzpůsobit trasu útěku přes potravu, aby během útěku nevyhladověl. Podobně by se liška měla rozhodnout, zda poběží k nejbližšímu zajíci, nebo zvolí útok na vzdálenější skupinku, kde je ale větší pravděpodobnost ulovení alespoň jednoho kusu. Také může být zajímavé zkusit implementovat rozhodování na základě předpokladu, zda oběť útočníka zpozorovala nebo ne (je otočená zády). Jistě bude výhodnější strategií zamířit k vzdálenějšímu zajíci, který lišku nezpozoroval, než k blízkému, který už utíká. To ale vyžaduje komplikovanější interakci mezi objekty…

Úkoly s řešeními

1. Zajistěte, aby se ve středu obrázku zajíce objevilo zelené nebo červené kolečko v závislosti na tom, zda v jeho okolí s poloměrem 100 pixelů se nachází nebo nenachází liška. Vytvořte k tomu privátní metodu foxDetection s návratovou boolean hodnotou.

   Vytvoříme novou privátní metodu foxDetection(), která bude vracet booleovskou hodnotu podle toho, zda bude liška v dohledu. Později do této metody přidáme i „zorné pole“, zatím stačí jakákoliv liška v okolí. Vhodnou metodou získáme seznam lišek v okolí a pak zjistíme, kolik jich je. Podle toho upravíme obrázek. Řešení je s vynechávkou (nezapomeňte také na dva nezbytné importy):

```
private boolean foxDetection() {
   List foxes = this.□(100, Fox.class);
   GreenfootImage i = □;
   if (foxes.□()>0) {
       i.setColor(Color.□);
       i.fillOval(i.□()/2-2,i.□()/2-2,4,4);
       return true;
   }
   i.setColor(Color.□);
   i.fillOval(i.□()/2-2,i.□()/2-2,4,4);
   return false;
}
```

2. Předchozí příklad rozšiřte tak, aby kolečko bylo červené pouze tehdy, pokud bude liška vůči zajíci umístěná v zorném poli ±45 °. Pokud bude mimo toto zorné pole, ale uvnitř radiusu, nechť je kolečko modré. Pro výpočet úhlu mezi liškou a zajícem si pomozte vytvořením privátní metody `int countActorAngle(Actor a)` a zaměřte se na výpočet odchylky vektorů (či přímek), protože úhel bude nutné spočítat „ručně“. Tato metoda bude počítat úhel ve směru hodinových ručiček s vrcholem v aktuální instanci zajíce, počátečním ramenem mířícím vodorovně vpravo (v Greenfootu úhel 0 natočení hráče) a umístěním hráče „a“.

   Při výpočtu odchylky narazíme na několik drobných problémů k řešení. Předně odchylka vektorů (či přímek) je vždycky z intervalu od 0 do 180, ale my potřebujeme počítat úhel 0 až 359 ve směru hodinových ručiček. To můžeme kompenzovat přičítáním nebo odčítáním v závislosti na směru vektoru resp. umístění hráče „a“ vůči aktuální instanci. Uvědomme si, že ve vzorci vzniká problém umocněním složek vektoru, tedy zkladněním někdy záporné hodnoty. Lze tedy operovat s kladností vektoru. Vzorec předpočítáme a zjednodušíme díky tomu, že použijeme jednotkový vektor „vodorovně vpravo“ s příjemnými hodnotami (1;0). Zbývá si ještě uvědomit, že funkce pro výpočet arkus kosinu vrací úhel v obloukové míře, ale my jej chceme ve stupních. Funkce pro matematiku (odmocnina a arkus kosinus) se nachází ve statické třídě Math. Hledejte v dokumentaci. Pro dokončení úkolu musíme ještě identifikovat mezi liškami ve zvoleném radiusu tu, která je zároveň v zorném poli. Procházíme lišky v seznamu a počítáme úhel (nejen lišky, ale i natočení sebe). Při prvním úspěchu končíme…

```
private int getActorAngle(Actor a) {
   int vx = □.getX()-□.getX();
   int vy = □.getY()-□.getY();
   int result = (int) (Math.□(□/Math.sqrt(□*□+□*□))*□/□);
   if (vy□0) result = □ - □;

   return result;
}

private boolean foxDetection() {
   List<Fox> foxes = this.□(100, Fox.class);
   GreenfootImage i = □();
   if (foxes.□()>0) {
      for(Fox fox : □) {
         int foxAngle = Math.abs(this.□(fox)-this.□());
         if (foxAngle<45 □ □-foxAngle<45) {
            i.setColor(Color.□);
            i.fillOval(i.□()/2-2,i.□()/2-2,4,4);
            return true;
         }
       }
       i.setColor(Color.□);
       i.fillOval(i.□()/2-2,i.□()/2-2,4,4);
       return false;
   }
   i.setColor(Color.□);
   i.fillOval(i.□()/2-2,i.□()/2-2,4,4);
   return false;
}
```

3. Upravte chování zajíce, aby se v případě detekce lišky otočil opačným směrem oproti první rozpoznané lišce, zvýšil svou rychlost (místo pohybu 4 až 14 polí za jeden act, ať se hýbe 10 až 20 polí) a snížil kličkování (výchylka do stran ať není 45 stupňů, ale jen 15). Obrácení se řešte novým boolean atributem „stayOpposite“ metody foxDetection.

   Metoda vyžaduje jen drobnou úpravu (jen podstatné dva řádky kódu):

```
private boolean foxDetection(boolean stayOpposite) {
if (stayOpposite) this.setRotation((this.getActorAngle(□)+□)□360);
```

   A dále doplnění metody act:

```
if (this.foxDetection(□)) {
    this.turn(this.randomGenerator.nextInt(□)*15-□);
    this.move(this.randomGenerator.nextInt(□)+□);
} else {
    this.turn(this.randomGenerator.nextInt(□)*15-□);
    this.move(this.randomGenerator.nextInt(□)+□);
}
```

4. Jelikož zajíc po otočení v následujícím actu už nevidí lišku v zorném poli, přestal by hned utíkat. Vyřešte tedy problém tak, že si zajíc zapamatuje, že má utíkat, po dobu 8 actů.

   Zavedeme nový privátní atribut `int runAwayTimer=0`, který bude odpočítávat v případě detekce lišky od pěti k nule s každým actem. Snižování hodnoty atributu s každým actem vložíme do metody oneActOlder, kterou již používáme kvůli měření fáze dospívání z dětských variant pohlaví na dospělé. Při detekci lišky se nastaví na 8. Zároveň se změní v actu podmínka útěku z detekce lišky na situaci, kdy je hodnota runAwayTimer vyšší než nula.

5. Implementujte podobnou detekci u lišky – ať v případě, že v okolí s poloměrem 160 pixelů je alespoň jeden zajíc, vybere zajíce, který je nejblíž a maximální rychlostí (10) a bez uhýbání míří přímo k němu.

   Nejprve z třídy Rabbit zkopírujeme metodu getActorAngle, jelikož se bude hodit i zde. Doplníme privátní metodu distance(Actor a), která bude měřit vzdálenost jiného objektu (snadno přes Pythagorovu větu). Nakonec Vytvoříme metodu rabbitDetection, která bude tentokrát mít boolean parametr „rotate“, aby se liška případně natočila směrem k nejbližšímu zajíci. Výběr nejbližšího zajíce se provádí standardním sekvenčním algoritmem výběru nejmenšího prvku. Na začátku se zvolí první prvek a nastaví jeho vzdálenost (nebo číslo vyšší než maximum). Pak se postupně prochází všechny prvky a nalezne-li se prvek s menší vzdáleností, výběr se opraví. Po projití celého seznamu máme správně zvoleno. Ukázka kódu opět obsahuje vynechávky:

```
private int distance(Actor a) {
   return (□) Math.sqrt((a.□()-this.□())*(a.□()-this.□())+
                        (a.□()-this.□())*(a.□()-this.□()));
}

private boolean rabbitDetection(boolean rotate) {
   List<□> rabbits = this.□(160, Rabbit.class);
   if (rabbits.□()>0) {
      Rabbit chosenRabbit = rabbits.□(0);
      int distance = □;
      for(Rabbit □ : rabbits) {
         if (this.distance(□)□distance) chosenRabbit=rabbit;
      }
      if(rotate) this.setRotation(this.getActorAngle(□));
      return true;
   }
   return false;
}
```

   Nezapomeneme také na úpravu actu (a jeden chybějící import). V actu na základě návratu true z rabbitDetection zvýšíme rychlost na 10 a vynecháme možnou změnu směru.

6. Rozšiřte předchozí úkol, aby liška viděla zajíce jen pod úhlem ±60 °.

   Stačí provést pár úprav metody rabbitDetection. Předně je ale nutné si uvědomit, že i v případě, že jsme nějaké zajíce v okolí lišky našli, nemusí být v zorném poli a proto budeme ze všech zajíců vybírat nejprve ty, které v něm jsou, a teprve poté mezi nimi hledat toho nejbližšího. Je samozřejmě možné, že jakmile se k němu liška pootočí, změní se její zorné pole a uvidí jiného

zajíce ještě blíže, ale to je v pořádku. Změna má proto nejdůležitější aspekt v tom, že na začátku je vybraný zajíc „null“ (a jeho vzdálenost je větší než radius, ve kterém hledáme), což symbolizuje, že jsme žádného nenašli. Vzorové řešení s vynechánými:

```
private boolean rabbitDetection(boolean rotate) {
   List<□> rabbits = this.□(160, Rabbit.class);
   Rabbit chosenRabbit = null;
   if (rabbits.□()>0) {
      int distance = □;
      for(Rabbit □ : rabbits) {
         int rabbitAngle = Math.abs(this.□(rabbit)-this.□());
         if ((rabbitAngle<60 □ □-rabbitAngle<60) □
             this.distance(□)□distance) {
             chosenRabbit=rabbit;
         }
      }
      if(rotate □ chosenRabbit!=□) {
         this.setRotation(this.getActorAngle(□));
      }
   }
   return chosenRabbit!=□;
}
```

7. BONUS: Zvažte a případně naprogramujte možná rozšíření v chování lišky nebo zajíce:
   a. Je-li liška do určité míry sytá, nebude se pohybovat a nebude žrát ani zajíce, kteří by byli v jejím dosahu.
   b. Liška při sežrání zajíce zcela zastaví a nepohybuje se.
   c. Liška může detekovat, zda je k ní zajíc otočený tak, že ji nevidí, a upřednostnit ho při útoku, i když je vzdálený více, než zajíc, který lišku zpozoroval. Jak ošetřit fakt, že zajíc, který lišku zpozoroval a utíká před ní, je k ní při útěku zády a proto čistě algoritmicky ji nevidí a tedy o ní zdánlivě neví?
   d. Zajíc cíleně vyhledává potravu, je-li vyhladovělý (mí nízký koeficient energie). Vyhladovělý zajíc se také nemnoží, dokud nepřijme potravu (a nemá energii nad určitou minimální hranicí).
   e. Co upřednostní vyhladovělý zajíc v blízkosti lišky? Útěk nebo potravu? Zkuste upravit algoritmus útěku tak, aby v případě potravy v blízkosti zajíc utíkal přes pole s potravou.
   f. Naučte zajíce utíkat před více liškami naráz, tedy zvolit vektor útěku jako výslednici skládání vektorů útěku od všech lišek v dosahu.

Tyto rozšiřující bonusové příklady jsou jen návrhem pro nadané a zapálené studenty k individuální domácí práci a proto není uvedeno žádné řešení ani návrhy, jak úlohy řešit.

Úkoly

1. Zajistěte, aby se ve středu obrázku zajíce objevilo zelené nebo červené kolečko v závislosti na tom, zda v jeho okolí s poloměrem 100 pixelů se nachází nebo nenachází liška. Vytvořte k tomu privátní metodu foxDetection s návratovou boolean hodnotou.
2. Předchozí příklad rozšiřte tak, aby kolečko bylo červené pouze tehdy, pokud bude liška vůči zajíci umístěná v zorném poli ±45 °. Pokud bude mimo toto zorné pole, ale uvnitř radiusu, nechť je kolečko modré. Pro výpočet úhlu mezi liškou a zajícem si pomozte vytvořením privátní metody `int countActorAngle(Actor a)` a zaměřte se na výpočet odchylky vektorů (či přímek), protože úhel bude nutné spočítat „ručně“. Tato metoda bude počítat úhel ve směru hodinových ručiček s vrcholem v aktuální instanci zajíce, počátečním ramenem mířícím vodorovně vpravo (v Greenfootu úhel 0 natočení hráče) a umístěním hráče „a“.
3. Upravte chování zajíce, aby se v případě detekce lišky otočil opačným směrem oproti první rozpoznané lišce, zvýšil svou rychlost (místo pohybu 4 až 14 polí za jeden act, ať se hýbe 10 až 20 polí) a snížil kličkování (výchylka do stran ať není 45 stupňů, ale jen 15). Obrácení se řešte novým boolean atributem „stayOpposite“ metody foxDetection.
4. Jelikož zajíc po otočení v následujícím actu už nevidí lišku v zorném poli, přestal by hned utíkat. Vyřešte tedy problém tak, že si zajíc zapamatuje, že má utíkat, po dobu 8 actů.
5. Implementujte podobnou detekci u lišky – ať v případě, že v okolí s poloměrem 160 pixelů je alespoň jeden zajíc, vybere zajíce, který je nejblíž a maximální rychlostí (10) a bez uhýbání míří přímo k němu.
6. Rozšiřte předchozí úkol, aby liška viděla zajíce jen pod úhlem ±60 °.
7. BONUS: Zvažte a případně naprogramujte možná rozšíření v chování lišky nebo zajíce:
    a. Je-li liška do určité míry sytá, nebude se pohybovat a nebude žrát ani zajíce, kteří by byli v jejím dosahu.
    b. Liška při sežrání zajíce zcela zastaví a nepohybuje se.
    c. Liška může detekovat, zda je k ní zajíc otočený tak, že ji nevidí, a upřednostnit ho při útoku, i když je vzdálený více, než zajíc, který lišku zpozoroval. Jak ošetřit fakt, že zajíc, který lišku zpozoroval a utíká před ní, je k ní při útěku zády a proto čistě algoritmicky ji nevidí a tedy o ní zdánlivě neví?
    d. Zajíc cíleně vyhledává potravu, je-li vyhladovělý (mí nízký koeficient energie). Vyhladovělý zajíc se také nemnoží, dokud nepřijme potravu (a nemá energii nad určitou minimální hranicí).
    e. Co upřednostní vyhladovělý zajíc v blízkosti lišky? Útěk nebo potravu? Zkuste upravit algoritmus útěku tak, aby v případě potravy v blízkosti zajíc utíkal přes pole s potravou.
    f. Naučte zajíce utíkat před více liškami naráz, tedy zvolit vektor útěku jako výslednici skládání vektorů útěku od všech lišek v dosahu.