DUM č. 10 v sadě

35. Inf-11 Objektové programování v Greenfoot

Autor: Lukáš Rýdlo

Datum: 30.06.2014

Ročník: studenti semináře

Anotace DUMu: Implementace obousměrného dynamicky alokovaného řetězeného seznamu s řazením. Vytvoření možnosti vkládat dynamicky větší množství hráčů, vykreslení všech hráčů...

esf evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
2007-13 OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Projekt: Simulace života zajíce v C/Allegro
Uspořádaný, řetězený seznam hráčů

Úvod

V minulé lekci jsme do hracího plánu vložili jednoho dynamicky alokovaného hráče, na kterého jsme se odkazovali jeho ukazatelem. Je zřejmé, že nemůžeme mít ukazatel pro každého hráče zvlášť, ale je zapotřebí je uchovávat dynamicky.

Obzvláště důležité bude zavedení správných datových struktur k uložení informací tak, aby v paměti zabraly přiměřené množství místa a přitom umožnily rychlou manipulaci a hledání dat. Tento konflikt mezi množstvím potřebné paměti a rychlostí s jakou lze s daty pracovat je nutné vyvážit a správně zvolit, co je v konkrétním případě výhodnější. Zároveň samozřejmě hraje roli i náročnost implementace. My budeme za běhu vytvářet velké množství nových „hráčů“, protože zajíci se hojně množí, zároveň budeme spoustu hráčů odstraňovat (lišky spoustu zajíců sežerou). Budeme také potřebovat často prohledávat seznam lišek i zajíců, abychom poznali, kdy se vzájemně překrývají dospělý zajíc a zaječka (a tedy zplodili nového zajíce) nebo kdy se překrývá liška ze zajícem, kterého sežere.

Z uvedených důvodů je nesmyslné používat statická pole, která bychom jednak pomalu procházeli, jednak by se v nich těžko udržoval pořádek při neustálém přidávání a odebírání prvků a v neposlední řadě nedokážeme určit jejich potřebnou velikost.

Dynamická pole jsou také nevhodná, jelikož budeme často odebírat a tudíž bychom museli pole neustále přerovnávat a realokovat. Proto použijeme dynamicky alokovaný lineární spojový seznam, nejlépe v obousměrné variantě, abychom při znalosti ukazatele na hráče mohli přejít na následujícího i předchozího. Jako vhodný studijní materiál lze použít překvapivě dobrou ukázku na Wikipedii: http://cs.wikipedia.org/wiki/Line%C3%A1rn%C3%AD_seznam.

Další variantou řešení by mohla být stromová struktura, která oproti lineárnímu seznamu přináší velkou výhodu v logaritmické rychlosti vyhledávání dat. To znamená, že pokud má stromová struktura vždy dvě větve a je vyvážená (snažíme se zachovat na všech větvích stejnou „hloubku“), pak rychlost, s jakou budeme hledat nějaký prvek od vrcholu až k nalezení (nebo rozhodnutí o nepřítomnosti) je daná logaritmem (o základu dva) z počtu prvků, protože to je „hloubka“ stromu. Zatímco u spojového seznamu musíme procházet lineárně. Nicméně tato vlastnost je vykoupena pomalejším vkládáním a odebíráním (musíme vyvažovat – tj. přerovnávat – strukturu).

Pro zjednodušení implementujeme seznam přímo nad strukturou sActor, do které proto doplníme dva ukazatele: na předka a následovníka. Abychom urychlili detekci interakcí, hodilo by se nám mít seznam uspořádaný tak, aby blízké objekty byly v seznamu blízko sebe. V ploše bude problém takové uspořádání najít, ale můžeme použít třeba seřazení podle vzdálenosti na ose x a v rámci stejných x-hodnot podle vzdálenosti na ose y. Prohledávání by šlo ještě více urychlit, pokud bychom kromě ukazatelů na předchůdce (resp. následníka) s menší (resp. větší) nebo stejnou hodnotou udržovali ukazatel i na ty s hodnotou x ostře menší nebo větší.

Další možností je seřadit hráče podle jejich vzdálenosti vůči konkrétnímu bodu (např. počátku). Opět to bude nespolehlivé řešení, protože body o stejné vzdálenosti leží na kružnici a mohou být od sebe navzájem poměrně daleko vzdálené, ale je nízká pravděpodobnost, že všichni hráči budou ležet na jedné kružnici, což by způsobilo pomalé vyhledávání.

Nakreslete si obrázek a promyslete, jak se budou kontrolovat kolize, v kterém případě bude detekce pomalá a kdy rychlá. Jaké jsou možnosti optimalizace. Jak časově náročné bude vkládání a odebírání prvku nebo vyhledání všech prvků v okolí o zadaném poloměru.

Při řešení úkolů se všichni studenti rozdělí na 3 skupiny: první skupina bude implementovat seznam bez jakéhokoliv seřazení, druhá skupina implementuje řazení podle velikosti x s vnořeným řazení podle velikosti y. Poslední skupina implementuje řazení podle vzdálenosti od počátku. Nejšikovnější studenti ze druhé skupiny se mohou pokusit doimplementovat ještě zmíněnou optimalizaci ukazateli na následníky s ostře menší a větší hodnotou x.

Úkoly s řešeními

1. Přidejte do struktury sActor ukazatel na následníka a předchůdce (next a prev). Vytvořte globální proměnnou firstActor, která bude sloužit k uchování odkazu na prvního hráče.

   Řešení nevyžaduje návod.
2. Upravte funkce createActor, moveActor a removeActor tak, aby využívaly ukazatele next a previous a zachovávaly zvolené řazení. Ukazatel na prvního hráče v pořadí ať je vždy v globální proměnné firstActor. Při řešení se rozdělte na tři skupiny implementující funkce bez řazení, s řazením podle souřadnic x a y a s řazením vzdáleností od počátku.

   Tento úkol je velmi náročný, protože pracujeme s komplikovaným dynamickým datovým typem a ještě navíc jej máme v seřazené podobě, což vyžaduje provádět správně opravy pořadí během vkládání a změny souřadnic hráče. Běžný student by měl zvládnout při troše snahy variantu bez řazení (1. skupina) a je proto vhodné touto variantou začít. Až funguje (nebo nadaní studenti) přejdeme k řešení s některým druhem řazení (2. a 3. skupina). Je velmi užitečné důkladně si rozmyslet jaká je podmínka řazení (u 2. skupiny podmínkou opravdu <u>není</u>, že x i y musí být zároveň menší nebo rovno – podmínka je komplikovanější, jelikož hodnota y se bere v úvahu pouze tehdy, pokud je x už rovna) a přerovnání řešit systematicky nejprve bez ohledu na to, že předchůdce nebo následník uvažovaného hráče může být NULL (tj. nejprve uvažujeme jen situaci, kdy je hráč hluboko uvnitř řady a ne na okrajích). Až toto funguje, teprve ošetříme možné problémy, které nastávají, pokud byl hráč na okrajích řady. Při řešení velmi pomůže nakreslit si obrázek, jak se mění ukazatele na předchůdce a následovníky, když se prvek z řady vyjímá a nebo do řady přidává. Užitečný je i obrázek na kterém si demonstrujeme pohyb prvku v řadě, když se během posunu jeho hodnota zmenšila nebo zvětšila pod či nad hodnotu sousedů. Nezapomeňte, že někteří hráči se pohybují výrazně rychleji než jiní, proto je možné, že se přesunou i o několik hráčů! V řešení jsou vynechané podstatné části kódu, aby bylo opravdu nutné se nad algoritmem přidávání, odebírání a řazení zamyslet. Nicméně pro snažší pochopení je kód doplněn o komentáře (pochopitelně anglicky). Vzhledem ke komplikovanosti kódu je možné, že funkční bude i jinak postavené řešení. Ukázka obsahuje řešení pouze pro 2. skupinu (řazení nejprve podle x, pak y), bez řazení se pouze vynechají bloky, které hráče přerovnávají, s řazením do kružnic (podle vzdálenosti od středu) se mění pouze podmínka řazení. Je užitečné přejít na řešení 4. úkolu a během vykreslování hráčů si nechat zobrazovat jejich pořadové číslo funkcí textout_ex nebo textprintf_ex. Nejlépe tak učinit až po 7. úkolu a vyzkoušet, že se přeskládávání chová korektně při různých „extrémních“ přesunech hráčů…

```
struct sActor* createActor(□ type, □ x, □ y, □ speed, □ direction) {
   □ actor = (struct sActor *) malloc(sizeof(□));
   □ it;      //actor iterator (pointer for traversing actors)
   if (!□ || !(□(type)) || x□SCREEN_□ || y□SCREEN_□) {
     return NULL;
   }
   actor□type      = type;
   actor□x         = x;
   actor□y         = y;
   actor□speed     = speed;
   actor□direction = direction;
```

```
   if (firstActor == □) {  //this actor is the first one
      actor->next = □;
      actor->prev = □;
      firstActor  = □;
      return actor;
   }

   //search biggest smaller actor than already created
   for(it=firstActor; (it->x□actor->x □
                       (it->□==actor->□ □ it->□ □ actor->□))□
                      it->next□=NULL; it = it->□);
   if(it->next□=NULL □
      □(it->x□actor->x □ (it->□==actor->□ □ it->□ □ actor->□))) {
      it = it->□;
   }

   actor->□ = it;
   if (it == □) {  //biggest smaller doesn't exist
      actor->next      = □;
      firstActor->prev = □;
      firstActor       = □;
   } else {
      actor->next       = □;
      it->next          = □;
      if(actor->□!=NULL) {
         actor->□->□ = actor;
      }
   }

   return actor;
}

void moveActor(□ actor) {
   struct sActor * it;
   int dx = (□) □(actor->□*□((actor->□)/□.0*□.□));
   int dy = (□) □(actor->□*□((actor->□)/□.0*□.□));
   if (actor->x+dx□0 □ actor->x+dx□SCREEN_□ □
       actor->y+dy□0 □ actor->y+dy□SCREEN_□) {
       actor->x□=dx;
       actor->y□=dy;

       if (actor->□!=□) {   //some bigger actor exists
          //search smallest bigger actor (if actor is now bigger)
          for(it=actor->□; it!= □ && (it->x□actor->x □
                                (it->□==actor->□ □ it->□ □ actor->□))□
                               it->□!=NULL; it = it->□);
          //get last smaller or equal
          if(it->next□=NULL □
```

```
      □(it->x□actor->x □ (it->□==actor->□ □ it->□ □ actor->□))){
       it = it->□;
    }
    if (□ != □) {  //do nothing, if there will be no change
       //deconnect actor
       if(actor->prev □= NULL) {
          □ = actor->next;
       } else {
          actor->□->□ = actor->next;
       }
       if (actor->next □= NULL) {
          actor->□->□ = actor->prev;
       }

       //reconnect actor
       actor->□ = it;
       actor->□ = it->next;
       it->□    = actor;
       if (actor->next□=NULL) {
          actor->□->□ = actor;
       }
    }
 }
 if (actor->□!=□) {  //some smaller actor exists
    //search biggest smaller or equal actor
    //(if actor is now smaller)
    for(it=actor->□; it!=□ && (it->x□actor->x □
                          (it->□==actor->□ □ it->□ □ actor->□))□
                         it->□!=NULL; it = it->□);
    //get last bigger
    if(it->prev□=NULL □
      □(it->x□actor->x □ (it->□==actor->□ □ it->□ □ actor->□))){
       it = it->□;
    }
    if (□ != □) {  do nothing, if there will be no change
       //deconnect actor
       actor->□->□ = actor->next;
       if (actor->next □=NULL) {
          actor->□->□ = actor->prev;
       }

       //reconnect actor
       actor->□ = it->prev;
       actor->□ = it;
       it->□    = actor;
       if (actor->prev□=NULL) {
          actor->□->□ = actor;
       } else {
          □ = actor;
```

```
            }
          }
        }
    }
}

void removeActor(□ actor) {
   if (!□) return;
   if (actor->□ == NULL) {
      firstActor = actor->□;
      if (actor->next != □) {
         actor->next->prev = □;
      }
   } else {
      actor->prev->next = actor->□;
      if (actor->next!=□) {
         actor->next->prev = □;
      }
   }

   free(actor);
}
```

3. Vytvořte funkci int getActorCount(), která bude vracet počet hráčů. Zvažte možné implementace s ohledem na co nejvyšší rychlost výpočtu.

   Funkci je samozřejmě možné implementovat tak, že bude procházet celý seznam od prvního hráče (firstActor) a počítat, kolik jich je. Takové řešení má výhodu v jistotě správnosti (konzistenci) údaje. Je ale příliš pomalé, kvůli sekvenčnímu průchodu polem. Bude rozumnější zavést globální proměnnou a v jednotlivých funkcích (createActor a removeActor) upravovat její hodnotu. Tato funkce pak vrací pouze hodnotu globální proměnné.

4. Napište funkci `void drawAllActors(□ buffer, □ images, □ info)`, která bude vykreslovat všechny hráče do bitmapy předané parametrem buffer. Parametr images je jako u drawActor pole ukazatelů na bitmapy hráčů. Seznam hráčů se přebere z globální proměnné firstActor. Parametr info je malé číslo (0 nebo 1) udávající jako „boolean“, zda se mají nebo nemají přes obrázek hráče vypisovat informace o jeho pořadí (kolikátý byl vykreslen), souřadnice x a y, rychlost a směr. Výpis ponechejte v této funkci, neupravujte kvůli němu drawActor.

   Po náročné implementaci funkcí pro vytvoření, přesun a mazání hráčů je tato funkce oddychovým úkolem. Musí pouze procházet postupně všechny hráče v seznamu od prvního a volat na nich funkci dwarActor. Přitom počítá kolikátý hráč je a případně vypisuje text, k čemuž je výhodné použít nikoliv textout_ex, která neumožňuje vkládat formátovaný text, ale podobnou funkci z knihovny Allegro, která se podobá klasické printf a umožňuje formátovat výstup. Při procházení hráčů používáme ukazatel na aktuálně vybraného hráče, takovému ukazateli se říká iterátor – můžete si na internetu přečíst o tzv. návrhovém vzoru iterátor k lepšímu porozumění.

```
void drawAllActors(□ buffer, □ images, □ info) {
   □ it = firstActor; //actor iterator
```

```
    int i = 0;
    for (;it□=□;it = it->□) {
       i++;
       drawActor(□, buffer, images);
       if (info) {
          □(buffer,font, it->x, it->y, make□(255,255,255), -1,
            "%i [%i,%i]", i, it->x, it->y);
          □(buffer,font, it->x, it->y+12, make□(255,255,255), -1,
            "S%i D%i", it->speed, it->direction);
       }
    }
}
```

5. Napište funkci `void moveAllActors()`, která bude provádět aktualizaci posunutí všech hráčů. Seznam hráčů získá z globální proměnné firstActor.

   Funkce moveAllActors je ještě jednodušší než předchozí drawAllActors a proto zde není její řešení. Student, který ji nedokáže sám zapsat nepochopil nic z předchozích úkolů.
6. Napište funkci deallocActors, která dealokuje postupně všechny hráče.

   Při dealokaci musíme dávat pozor, jak postupujeme, abychom si nedealokovali i ukazatel na následovníky. Je snad zřejmé, že nemá smysl volat v cyklu funkci removeActor, protože by to bylo zbytečně zdlouhavé. Jestliže rušíme všechny hráče, není důvod je postupně „přepojovat“ v seznamu. Řešení je možné udělat tak, že si vždy zapamatujeme adresu na následníka a pak teprve prvek smažeme. Nebo možná ještě lépe (viz řešení) stačí začít od druhého prvku a vždy dealokovat jeho předchůdce. Jen je zapotřebí ošetřit situaci s jedním a dvěma prvky. Nezapomeňte správně nastavit firstActor po dealokaci!

```
void deallocActors() {
   □ it;
   actorCount=0;
   if (firstActor==□) return;
   if (firstActor->□==NULL) {
      free(□);
      firstActor = □;
      return;
   }
   for (it=firstActor->□;□!=NULL □ □!=NULL;it = it->□) {
      free(it->□);
   }
   free(□);
   firstActor = □;
}
```

7. Ve funkci main opravte ukazatel na hráče z „actor“ na „lastActor“. Zrušte vytváření hráče při spuštění programu. Zaveďte novou klávesovou zkratku „n“ pro přidání nového hráče, který bude vždy typu zajíc, bude se vkládat na souřadnice středu plánu s nulovým natočením a rychlostí 10. ostatní zkratky nechejte pracovat s ukazatelem lastActor.

   Řešení je v celku jednoduché a proto bude vzor až u následujícího příkladu, který jej rozvíjí.

8. Do mainu přidejte klávesové zkratky „/“ a „*“, které budou měnit hodnotu lastActor na předka či následovníka (v závislosti na aktuálním pořadí) aktuálního hráče. Zamezte nastavení NULL (tj. přetečení za posledního nebo před prvního z hráčů). Můžete také zavést zkratku „k“ pro odstranění aktuálního hráče z herního plánu. Nezapomeňte pak nastavit lastActor na nějakou rozumnou hodnotu (když původní hráč je dealokovaný).

   Tato úprava (stejně jako u předešlého úkolu) je spíše rutinní. Aktuální main by mohl vypadat takto:

```
int main() {
    init();

    □ bgBuffer;
    □ actorImages[3];
    □ buffer = □(SCREEN_□,SCREEN_□);
    if (!□) return 1;

    □ lastActor = □;

    if (!□(&bgBuffer)) return 2;
    if (!□(actorImages)) return 3;

    while (!key[KEY_ESC] □ !key[□]) {
       □(bgBuffer, buffer, □, □, 0, 0, SCREEN_□, SCREEN_□);
       drawAllActors(□, □, 1);
       □(buffer, screen, □, □, 0, 0, SCREEN_□, SCREEN_□);
       int k = readkey()□0x□;
       switch (k) {
          case 'f':
               lastActor->□=AT_FOX;
               break;
          case 'r':
               lastActor->□=AT_RABBIT;
               break;
          case 'a':
               lastActor->□=AT_FRABBIT;
               break;
          case 'c':
               lastActor->□=AT_CARROT;
               break;
          case '+':
               lastActor->speed□;
               break;
          case '-':
               lastActor->speed□;
               break;
          case '/':
               if(lastActor->□!=□) lastActor = lastActor->□;
               break;
```

```
            case '*':
                 if(lastActor->□!=□) lastActor = lastActor->□;
                 break;
            case 'm':
                 moveActor(lastActor);
                 break;
            case 'k':
                 {removeActor(lastActor);lastActor=□;};
                 break;
            case 'o':
                 {lastActor->direction□=10;lastActor->direction□=360;}
                 break;
            case 'n':
                 lastActor =
                    createActor(AT_RABBIT,SCREEN_□/□,SCREEN_□/□,10,0);
                 break;
        }
    }

    deinit();
    destroy_bitmap(□);
    deallocActors();
    return 0;
}
END_OF_MAIN()
```

Úkoly

1. Přidejte do struktury sActor ukazatel na následníka a předchůdce (next a prev). Vytvořte globální proměnnou firstActor, která bude sloužit k uchování odkazu na prvního hráče.
2. Upravte funkce createActor, moveActor a removeActor tak, aby využívaly ukazatele next a previous a zachovávaly zvolené řazení. Ukazatel na prvního hráče v pořadí ať je vždy v globální proměnné firstActor. Při řešení se rozdělte na tři skupiny implementující funkce bez řazení, s řazením podle souřadnic x a y a s řazením vzdáleností od počátku.
3. Vytvořte funkci int getActorCount(), která bude vracet počet hráčů. Zvažte možné implementace s ohledem na co nejvyšší rychlost výpočtu.
4. Napište funkci `void drawAllActors(□ buffer, □ images, □ info)`, která bude vykreslovat všechny hráče do bitmapy předané parametrem buffer. Parametr images je jako u drawActor pole ukazatelů na bitmapy hráčů. Seznam hráčů se přebere z globální proměnné firstActor. Parametr info je malé číslo (0 nebo 1) udávající jako „boolean“, zda se mají nebo nemají přes obrázek hráče vypisovat informace o jeho pořadí (kolikátý byl vykreslen), souřadnice x a y, rychlost a směr. Výpis ponechejte v této funkci, neupravujte kvůli němu drawActor.
5. Napište funkci `void moveAllActors()`, která bude provádět aktualizaci posunutí všech hráčů. Seznam hráčů získá z globální proměnné firstActor.
6. Napište funkci deallocActors, která dealokuje postupně všechny hráče.
7. Ve funkci main opravte ukazatel na hráče z „actor“ na „lastActor“. Zrušte vytváření hráče při spuštění programu. Zaveďte novou klávesovou zkratku „n“ pro přidání nového hráče, který bude vždy typu zajíc, bude se vkládat na souřadnice středu plánu s nulovým natočením a rychlostí 10. ostatní zkratky nechejte pracovat s ukazatelem lastActor.
8. Do mainu přidejte klávesové zkratky „/“ a „*“, které budou měnit hodnotu lastActor na předka či následovníka (v závislosti na aktuálním pořadí) aktuálního hráče. Zamezte nastavení NULL (tj. přetečení za posledního nebo před prvního z hráčů). Můžete také zavést zkratku „k“ pro odstranění aktuálního hráče z herního plánu. Nezapomeňte pak nastavit lastActor na nějakou rozumnou hodnotu (když původní hráč je dealokovaný).