Programování hry KRKAL

Jan Krček (jan.krcek@matfyz.cz)

Úvod

Tento text je určen všem, kteří chtějí proniknout do programování hry Krkal, přidávat nové prvky, uzpůsobovat si Krkala pro sebe a díky tomu všemu tvořit velmi originální levly.

Je to ovšem náročný cíl, který vyžaduje trochu programátorské zběhlosti a hodně trpělivosti při pronikání do tajů Systému Krkal, tedy systému pro tvorbu objektově orientovaných 2D her.

Tento text předpokládá znalost hry Krkal. Zahrajte si ji. Můžete si i přečíst dokumentaci ke hře. Dále je nutné seznámit se s obecným povídáním o programování v Systému Krkal – „*Jak psát skripty*“.

Zdrojový kód hry Krkal se nachází v souboru:

```
test_0001_FFFF_0001_0001.kc
```

Základní popis hry

Hra Krkal se odehrává na čtvercích 40 x 40 pixelů. Je jednopatrová, odehrává se na obdélníkovém, různě velikém plánu. Smyslem hry je ovládat postavičku, která se po levlu pohybuje. V Krkalovi je možno proměňovat postavičku na jinou a dokonce mít postaviček i více.

Ve hře mají téměř všechny objekty velikost jedné buňky, výjimky tvoří pouze triggery a podobné objekty, které vlastně nehrají. Objekty buď stojí na místě, v tom případě jsou zarovnány přesně na buňku. A nebo se můžou pohybovat po takzvaných **trajektoriích**. Pohyb vždy začíná i končí také přesně na buňce.

Cílem hry je překonat všechny nástrahy a dostat se do **exitu**. Tedy většinou. Od toho jsme přeci programátoři, abychom mohli vymyslet a naprogramovat i nějaké jiné cíle. Třeba sebrání alespoň třiceti diamantů, zničení nepřítele, přežití po určitou dobu a pod.

Běžné objekty se dělí do čtyř základních skupin podle toho, jakou část buňky zabírají. Běžné umístitelné objekty mají společného předka a to objekt **`placeable`**. Jsou to:

- **podlahy**, které kolidují jen s jinými podlahami. Předek podlah je objekt `floors`.
- **stěny**, ty kolidují s jinými stěnami, s věcmi a s LezeNadVecma. Žádného speciálního předka nepotřebují.
- **věci**. Kolidují se stěnami a s jinými věcmi. Předek: `Veci`.
- **LezeNadVecma**. Kolidují se stěnami a s jinými LezeNadVecma. Předek: `LezeNadVecma`.

Existují i jiné objekty, které se tomuto rozdělení vymykají. Dost často nekolidují s ničím nebo naopak se vším. Nemívají nastavenou grafiku (jsou neviditelné). Většinou jde o **triggery**.

A pak je v Krkalovi i velké množství objektů, které jsou určeny pouze k dědění. Tyto objekty si s sebou nesou nějakou vlastnost, která ovlivní potomky, a nejsou určeny k vytváření, či umísťování.

Typické situace

Vymysleli jsme si nový objekt s nějakými zajímavými vlastnostmi a chceme ho do hry přidat. Nejčastěji bude prostě stačit oddědit nový objekt od té správné kombinace předků a přidat objekt do takových množin, aby s ním ostatní objekty ve hře správně nakládaly.

U složitějších objektů implementujeme metody, ve kterých reagujeme buď na zprávy z kernelu (`@MapPlaced`) nebo na zprávy od ostatních objektů.

Příklad 1

Řekněme, že chceme naprogramovat hrbolatou podlahu, která bude objektům zabraňovat v pohybu. vytvoříme tedy objekt `oHrbolataPodlaha` a oddědíme ho od `floors`. A co dál?

Teď je třeba seznámit se s objektem `oMoveable`, od kterého jsou odděděny všechny pohybující se objekty, a zjistit, jakým způsobem probíhá rozhodování o budoucím pohybu. A do tohoto rozhodování je třeba zasáhnout tím, že řekneme: „Tak, a přes hrbolatou podlahu nikdo nesmí!“. Tím bychom se ale dopustili velké chyby. Problém je tu v tom slově „**nikdo**“. Můžou přeci existovat i objekty, které se po hrbolaté podlaze budou chtít pohybovat. A jestli už neexistují nyní, mohou existovat někdy v budoucnu. Je nutné na tyto věci pamatovat a situaci řešit tak, že zamezíme v pohybu **jen objektům z množiny `BlokovanoHrbolatouPodlahou`**.

Konkrétní program by pak vypadal takto:

```
voidname oBlokovanoHrP;  // množina, která se po hrbolech nepohybuje
depend oBlokovanoHrP << {oKamen, oklic, oKrabice, /* ... */ }
objectname oHrbolataP;
depend {floors, oModifyTr} << oHrbolataP;
      // jsem podlaha a jsem objekt, který modifikuje trajektorie.

object oHrbolataP {
     edit {InMap}
     int ModifyTrajektory(int ptrpos) {
          // jestliže se na de mnou zkoušel hýbat objekt z množiny oBlokovanoHrP,
          // tak pohyb zamítnu.
          if (typeof(sender) <= oBlokovanoHrP) return 0;
          return ptrpos; // trajektorii nechám beze změny.
     }
}
```

Příklad 2

někdy je situace složitější. Neprogramuji jen nový objekt, ale je třeba přidávat nové metody k již stávajícím objektům, aby s novým objektem mohli správně nakládat a komunikovat. **Důležité**: Snažte se situaci řešit čistým přidáváním nových těl metod a nových atributů. Nepoužívejte pokud možno **nečisté** operace, které dokáží měnit již existující těla metod v uzavřených verzích.

Příkladem může být nová sbíratelná věc. Chceme, aby ji maník uměl sbírat a pokládat. Aby tohle bylo možné je třeba udělat řadu modifikací (vše lze řešit čistým způsobem): U maníka je třeba naprogramovat sbírání a pokládání, dále je nutné rozšířit kopírovací funkci, která slouží k okopírování inventáře, když dojde ke změně podoby maníka. Pak je tu herní menu, kam bychom rádi přidali nový obrázek naší věci. (o menu se stará objekt `oManikControler`) A ještě je třeba postarat se o aktualizaci údaje o počtu sebraných věcí v případě, že došlo k přepnutí na jiného maníka.

Důležité objekty ve hře Krkal

Programování a přidávání nových věcí může být sice docela jednoduché, ale téměř vždy je třeba znát již naprogramované objekty a vědět, jak s nimi komunikovat. Následující text bude obsahovat popis těch důležitých objektů, které ovlivňují celou hru a od kterých je možno dědit.

Základní objekty

Od těchto objektů se dědí všechny ostatní umístitelné objekty. Objekty řeší souřadnice, umísťování a kolize.

`placeable`	Předek všech umístitelných objektů s obrázkem. Definuje souřadnice
`Veci`	Sbíratelné objekty
`LezeNadVecma`	
`floors`	
`oSeSmerem`	// Obsahuje atribut `smer`, který může být `Sever, Vychod, Zapad` a `Jih` a který určuje natočení objektu. Je editovatelný a ovlivňuje automatickou grafiku.

Triggery

Je několik druhů triggerů. Liší se podle typu oblasti, kterou zabírají. Všechny kolidují jak se stěnami, tak s podlahami. Všechny mají implementovánu metodu `SetClzGr`, kterou je možno nastavovat kolizní množiny a redirekci zpráv. (`@clzAddGr, @clzSubGr, @MsgRedirect`). Dále obsahují metodu pro nastavování souřadnic a pozice (`SetPosSz` nebo `SetPos`).

`otrigger`	Zabírá obdélník n x m buněk
`oPointTr`	Zabírá 1 bod. Má také nastaven `@eKCCcenterColBit`, takže reaguje jen na oběkty, které mají přesně stejné souřadnice jako trigger

`oLineTr`	Kombinace obdélníkového triggeru a triggeru s nastaveným `@eKCCcenterColBit`. Je zamýšlen na kontrolování obdélníků 0 x n nebo n x 0. Tedy čar vedoucích středy buněk.
`oAreaTrigger`	Zabírá obdélník n x m pixelů

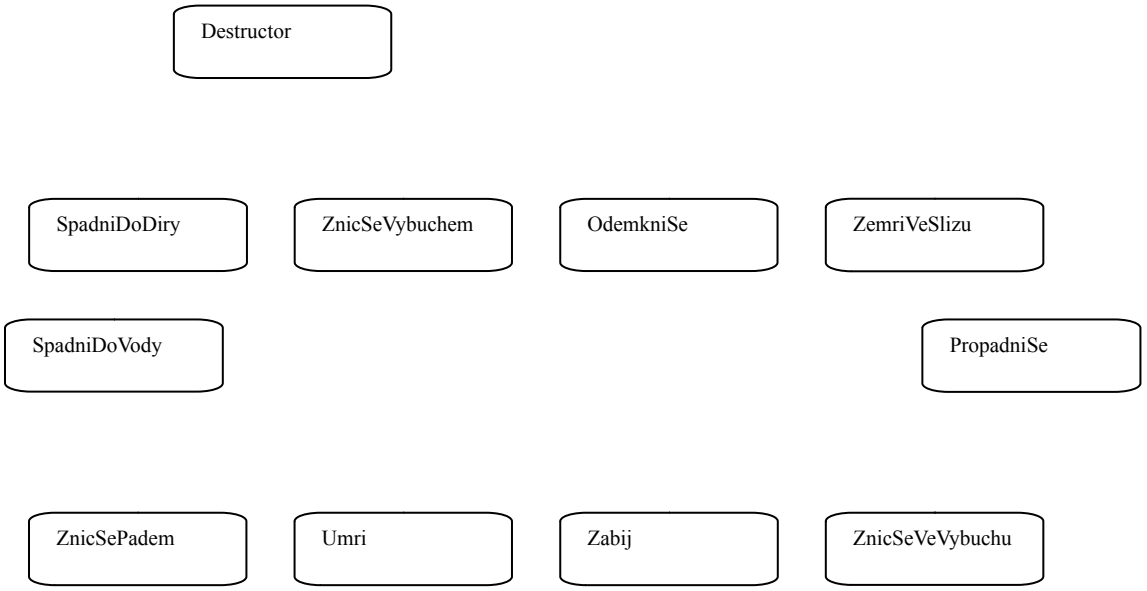
Výbuchy, díry a umírání

Výbuch

Vybuchující objekty jsou odděděny od `oVybuchuje`. Tento objekt obsahuje metodu `ZnicSeVybuchem`, která postaví kolem objektu výbuch o velikosti 3 x 3 buňky a která poté objekt zničí. Vybuchující objekty jsou také často odděděny od `ZnicitelneVeVybuchu`. To pro ně znamená, že mohou umřít ve výbuchu, což většinou vede k zavolání metody `ZnicSeVybuchem` a tedy k vybouchnutí. Vznikají takzvané **řetězové výbuchy**.

Výbuch volá na všechny objekty `ZnicitelneVeVybuchu` metodu `ZnicSeVeVybuchu`.

Schéma propojení „zabijáckých metod“



Zabijácké objekty, množiny na které působí a metoda, kterou volají:

- `oDira` a `oLava` ničí `VeciPadajiciDoDiry` zavoláním metody `SpadniDoDiry`.
- `oVoda` ničí `PadaDoVody` zavoláním metody `SpadniDoVody`.
- `oSliz` působí na objekty `oUmiraNaSlizu`. Tyto objekty si počítají velikost své otravy. Když velikost otravy přesáhne určitou mez, tak si objekty zavolají metodu `ZemriVeSlizu`.
- Výbuch volá na všechny objekty `ZnicitelneVeVybuchu` metodu `ZnicSeVeVybuchu`.
- Když `oZabijak1` chce jít na místo, kde mu překáží `oObet1`, tak zabiják bude (možná) zničen metodou `Zabij`, kdežto oběť metodou `Umri`.

- Když padající objekt `oZabijiPohybem1` spadne na `oObetPohybu1`, tak oběti je zavolána metoda `ZnicSePadem`.

Další zajímavé množiny ve hře Krkal

- `AktivujeFotobunku`
- `AktivujeMinu`
- `KlouzePoLedu`
- `NebezpeciProPriseru`. Inteligentnější příšery se vyhýbají nejen objektům, se kterými kolidují, ale i dalším nástrahám. Jako například díře a vodě.
- `oPusobiKulate`. Když se potkají dva objekty působící kulatě a jeden stojí a druhý se přes něj chce pohybovat, tak ten druhý se kolem prvního obkutálí
- `oMoveable`. Může se pohybovat
- `oStartsMove`. To by měli být všechny objekty, které někdy mohou blokovat pohyb. Tedy vše kromě podlah.
- `oTeleportujeSe`. Objekt může vstupovat do teleportů.
- `oStrkatelne`
- `oStka`. Objekty, které před sebou mohou tlačit jiné objekty (`oStrkatelne`)
- `oVybuchuje`
- `VeciStrkatelnePasem`
- `PritahovanoMagnetem`
- `ZapinaPrepinac`. Ten klasický, který reaguje na strčení.

Pohyby

Všechny pohybující se objekty jsou odděděny od objektu **`oMoveabe`**. Tento objekt obsahuje veškerou inteligenci pro rozhodování „kam se pohnout“ a pro řízení následného pohybu.

Všechny pohyby začínají a končí vždy přesně na buňce. Pokud pohyb už jednou začal, bude dokončen a nic ho neovlivní. Je tu jediná výjimka a to odebrání pohybujícího se objektu z mapy, či jeho zničení. Během pohybu si objekt blokuje svou trajektorii speciálním objektem odděděným od `oBlockedPath`. To se dělá proto, aby si objekt už dopředu zabral potřebné buňky a nehrozilo, že se někde po cestě střetne s jiným objektem ani že dva objekty, které současně chtějí jít na stejnou cílovou buňku, tam nakonec dojdou.

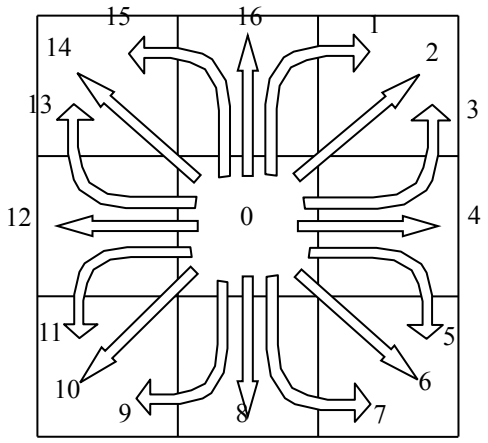
Síly

Než se objekt začne pohybovat, je třeba, aby na něj působila nějaká **síla**. Na jeden objekt může působit až několik sil a objekt `oMeveable` se podle nich pak musí rozhodnout pro výslednou trajektorii (viz dále). Sílu je možno zapnout zavoláním:

```
void AddForce(int pfdir, name pfpriority, name pftype, int pfspeed)
```

Parametr `pfdir` určuje směr, viz následující obrázek.

`pfpriority` určuje prioritu síly. Priorita je klíčová při rozhodování. `oMoveable` nejprve zkouší trajektorie podle sil s největší prioritou. Když je takových sil více, bude rozhodnuto náhodně.



`pftype` určuje typ síly. Každý objekt, který způsobuje pohyby by měl generovat jiný typ síly. Dále jsou určité typy, které mají speciální význam. (`BanningForce`).

`pfspeed` určuje rychlost. Do parametru se ale nezadává rychlost (to je chyták :-)), ale čas v milisekundách, který říká, jak dlouho se má objekt po trajektorii pohybovat. `pfspeed` je nepovinný parametr. Když se nezadá, objekt se bude pohybovat touž rychlostí jako se pohyboval posledně.

Voláním `RemoveForce` odstraníme všechny síly, které volající objekt přidal.

Žádost o vypočítání pohybu

`oMoveable` nezkouší propočítávat pohyby vždy, když stojí, ale jen tehdy požádá-li ho o to někdo zavoláním metody

```
CalcMove()
```

Tato metoda se volá i automaticky, když je pohyblivý objekt vložen do mapy, po ukončení pohybu a pokud někdo zavolal metodu `AddForce` nebo `RemoveForce`.

Proto je nutné všechny objekty, které potenciálně můžou blokovat pohyb oddědit od `oStrtsMove`. Tento objekt, v případě, že je odebrán z mapy, odblokuje okolní pohyblivé objekty tím, že jim zavolá metodu `CalcMove`. I objekt `oMoveable` nesmí zapomínat odblokovávat okolní objekty, když se kolem nich pohybuje.

`CalcMove` je jen žádost. Vlastní výpočet se ještě neprovádí, s tím se čeká až na konec kola. (To proto, aby i další objekty mohly přidávat síly a odblokovávat)

```
void CalcMove() {
     if (!WillCalculate && !Moving && @IsGame()) {
          // pozdavek, ktery je uznan, je jen ten prvni
          WillCalculate = 1;
          ::CalcMove2() end;  // vlastní výpočet
          MoveCalculating() message; // zpráva, která oznamuje, že se
          // bude počítat. Využití: Když chci těsně před výpočtem přidávat síly.
     }
}
```

Výpočet pohybu a průběh pohybu

Žádný směr se nezkouší vícekrát. Pokud se v některém směru ptám objektů po cestě, jestli nechtějí modifikovat sílu, tak se každého objektu ptám maximálně jednou za směr.

V Krkalovi jsou možné trajektorie a směry sil omezeny. Je to jen malá diskrétní množina (v současnosti 16 prvků). Když hledám výslednou sílu, nemůžu použít klasické fyzikální skládání sil, ač by to bylo mnohem jednodušší! V Krkalovi se tedy výsledná síla počítá na základě úplně jiného mechanismu.

1. Mezi silami najdi síly jejichž typ je z množiny `BanningForce`. Pokud působí dvě banning síly proti sobě, tak se vyruší a v následujícím výpočtu je už nebudu uvažovat.
2. Rekurzivně testuji skupinu sil.
    a. Vyberu ještě netestovaného kandidáta s nejvyšší prioritou. Pokud jich je více, tak se rozhodnu randomem. Pokud žádný takový není, tak return
    b. Banning test. Síla typu `BanningForce` znemožňuje pohyb proti ní. Testuji tedy navrhovanou sílu proti každé `BanningForce`. Když má síla shodný až kolmý směr, nebude změněna. Když má mírně protichůdný směr, bude pootočena. Jiné směry, především ty naprosto protichůdné budou zcela zamítnuty.
    c. Najdu objekty na buňkách po kterých vede trajektorie. Pro každý objekt:
        i. Pokud patří do množiny `TTMOConsumer`, bude mu zavolána metoda `TrayingToMoveOver` přímo. Jinak bude alespoň zavolána jako zpráva
        ii. Objekty z množiny `oModifyTr` dostanou příležitost modifikovat trajektorii. Mohou ji zamítnout, změnit libovolné údaje včetně směru, přidat několik nových trajektorií na rozhodování
        iii. Pokud některá z těchto metod na mě zavolala `CalcMove`, tak to znamená, že mám s výpočtem pohybů ještě počkat. Konec výpočtu.
        iv. Změnila-li modifikace směr síly, nebo přidala nové směry, testuj je krokem 2
    d. Kolizní test. Testuje se jak klasická kolize na všech místech trajektorie, tak navíc dodatečná uživatelská kolize `IsInMoveCollision`. Pokud je kolize a já jsem rozjetý tím směrem, tak hlásím náraz posláním zprávy `NarazilJsem`.
    e. Pokud některý z předchozích kroků selhal, vyberu dalšího kandidáta. Jdi na krok a
    f. Jinak jsem úspěšně zvolil trajektorii a můžu se po ní začít pohybovat. Jdi na krok 3
3. Zablokuji si vybranou cestu. Zapnu stav pohybu (`Moving` = 1)
4. Pošlu si zprávu `MoveStarted`
5. Pohybuji se
6. Pohyb skončil. Odeberu blokovací objekty a odblokuji jiné pohyblivé objekty
7. Počkám na konec kola
8. Zruším pohyb (`Moving = 0`). Požádám o výpočet nových pohybů `CalcMove` a pošlu si zprávu `MoveEnded`.

```
void TraingToMoveOver(int pfdir,int pfspeed,name pftype)
int ModifyTrajektory(int ptrpos, inta Pfdir,namea Pfpriority,
      inta Pfspeed, namea Pftype)
```

`TrayingToMoveOver` slouží jen k informování ostatních objektů, že se objekt pohybuje přes ně. Složí například k zabíjení pohybem, k odemykání zámků, přepínání, aktivování teleportů a pod.

`ModifyTrajectory` umožňuje ostatním objektům, přes které trajektorie vede, trajektorii jak zcela změnit, tak zachovat. Umožňuje také přidávat trajektorie nové. Funkce dostává pole s informacemi o trajektoriích. `ptrpos` ukazuje na právě testovanou trajektorii. trajektorii s tímto indexem může funkce měnit, na vyšší indexy může postupně přidávat nové návrhy na trajektorie. Na indexy nižší sahat **nesmí**.

Funkce vrací nejvyšší index trajektorie, kterou navrhla. Je možno vrátit i něco menšího, než bylo `ptrpos` a tím celou trajektorii zamítnout. (0 zamítá vždy)

oManik

`oManik` je objekt, od kterého jsou odděděny všechny herní postavičky.

Maník si každé kolo zavolá metodu `::EveryTurn`, kde reaguje na zmáčknuté klávesy. Podle nich si zapíná síly, aby se mohl pohybovat a dále používá různé sebrané předměty, tedy `Veci`.

Maník má u sebe vytvořen trigger, který reaguje na `Veci`. Programováním obsluhy tohoto triggeru se řeší sbírání.

Focus

Statický objekt `oManikControler`, si udržuje přehled o všech manících, co jsou ve hře. Pokud počet maníků ve hře klesne na 0, `oManiControler` ukončí hru. Dále tento objekt umožňuje mezi maníky přepínat **focus** klávesou TAB. Jen jeden maník je aktivní. To znamená, že herní menu ukazuje jeho inventář, obrazovka je vycentrována kolem něho a Samonaváděcí příšery jdou po něm. Pokud není nastaveno sdílené ovládání, tak i klávesy účinkují jen na aktivního maníka.

`oManikControler` se také stará o herní menu.

Inventář

Ve svých atributech si `oManik` udržuje stav inventáře. Aktivní maník musí udržovat aktuální i herní menu. Při sebrání, použití věci je třeba zavolat příslušnou funkci objektu `oManiControler`. Při změně focusu se volá funkce `ResetMenu`. (Aby aktuální maník poslal do menu aktuální údaje)

Kvůli proměňovačům (`oHajzl`), které mění podobu maníka, bylo nutné zavést funkci `CopyManik`, která slouží k okopírování inventáře.

Přepínače

Společný předek všech přepínačů je objekt `oPrepinace`.

Každý přepínač může dělat 0 až n akcí. Popis akcí je uložen ve spojovém seznamu `Akce`. Atribut `Prepina` určuje, zda přepínač bude při aktivaci provádět tu samou akci znovu a znovu nebo zda akci provede a pak ji vrátí do původního stavu. Poznámka: Pokud přepínač umísťuje objekty a při umísťování musel nějaké jiné objekty kvůli kolizi odstranit z mapy,

tak, v rámci vrácení akce do původního stavu, znovu vrátí do mapy i tyto odebrané objekty. Přepnutí zpátky bychom mohli přirovnat k operaci undo.

Přepnutí

Přepínač přepne, pokud zavoláme metodu `Prepni()`.

Typy akcí

- `PrepUmisti`. Pouze podmnožina `PrepOdeberUmisti`
- `PrepOdeber`. Pouze podmnožina `PrepOdeberUmisti`
- `PrepOdeberUmisti`. Jedny objekty umísťuje jiné odebírá. Pokud přepínač přepíná, tak si pamatuje odebrané objekty, aby je později mohl vrátit.
- `PrepOdeberUmisti2`. Touto akcí přepínač vrací `PrepOdeberUmisti` do původního stavu. Kód je až na pár detailů stejný jako u `PrepOdeberUmisti`.
- `Vymen`. Přepínač najde jedny objekty a na jejich místo umísťuje objekty jiné.
- `PrepMessage`. Přepínač posílá objektům zprávy. Když přepíná, tak dvě, jinak jednu.

oTriggerPrepinac

Tento přepínač reaguje na zprávy z triggeru. Dá se zvolit jestli na `@TriggerOn`, `@TriggerOff` nebo oboje. `oTriggerPrepinac` je určen k dědění. Sám netvoří trigger, jen se stará o jeho obsluhu. Potomci si tedy můžou vytvořit takový trigger, jaký chtějí.

`oTriggerPrepinac` se může snadno zacyklit. Třeba tak, že je nastaven na přepínání, aktivuje se objektem O a, když je aktivován, tak objekt O odebere. Proto `oTriggerPrepinac` volá `Prepni` jako zpožděnou zprávu `nextturn`. Díky tomu dojde k cyklu jen na přepínači a ne v celém Systému Krkal.