Jak psát skripty?

Jan Krček (jan.krcek@matfyz.cz)

Úvod

Tento text je určen pokročilým uživatelům, kteří si chtějí v Systému Krkal vytvářet svoje vlastní, zcela nové hry, či rozšiřovat hry již vytvořené. Bude zde popsáno **jak v systému programovat**, popíšu, jak se dají tvořit **objekty**, a doplním to řadou konkrétních příkladů. Nebudu se zabývat detailní syntaxí našeho skriptovacího jazyka (ta je popsána v Referenční příručce k jazyku od Jiřího Margaritova), ale vezmu to obecně, popíšu hlavní myšlenky a postupy. Dále zde podrobně popíšu funkci a práci **Kernelu**, ten se o veškeré skripty stará, řídí jejich činnost a je to také prostředník (interface), přes který skripty komunikují se zbytkem systému.

Předpokládá se základní znalost programování v C.

Objektově orientovaný přístup

Objekt je stejně jako v C++ spojením dat a metod. Navíc objekty „žijí“ v průběhu času, komunikují spolu pomocí přímých volání nebo pomocí zasílání zpráv. Objektem může být bomba, stěna nebo obléhací tank. K těmto objektům je přiřazena nějaká grafika a jsou umísťovány do levlů na konkrétní souřadnice. Existovat ale můžou i zcela abstraktní objekty. (např. objekt Mapa, objekt pro spojový seznam, …)

Svět Krkala je tedy „obydlen“ objekty, které si pamatují svůj stav v **atributech,** mají u sebe kusy kódu – **metody**, tam je naprogramováno, jak má objekt v určitých situacích zareagovat, co udělá.

Čas

Kernel řídí čas tohoto světa. Každých 33 ms (defaultní nastavení) je vyvolán **takt** (kolo, tah, turn), což je příležitost pro objekty, které chtějí něco dělat, aby to udělaly. Někdy může objekt dělat jen něco sám pro sebe, ale většinou objekty komunikují s okolními objekty (vyzvídají informace, ovlivňují cizí objekty, zasílají jim zprávy, dochází k tvoření nových objektů, ničení existujících ...) Když to shrnu, tak: objekty mění svůj stav.

Během taktu se tedy provádí řada **volání**, zpracovává se řada **zpráv**. Po celou dobu taktu zůstává čas kernelu nastaven na stejné hodnotě. Záplava akcí a reakcí, která se odehrává v jednom taktu, by měla skončit a to v rozumně krátkém čase (reálném). Pokud ne, tak jsou skripty chybně naprogramovány (vznikl tam nekonečný cyklus nebo výpočet je tak náročný, že trvá příliš dlouho a negativně ovlivňuje plynulost hry). Se skončením činnosti skriptů skončí i takt a kernel může vyvolat takt nový, tentokrát s časem o dalších 33 ms vyšším.

Poznámka: Svět Krkala velmi připomíná simulaci s diskrétním časem. A Systém Krkal, jako takový, se také pro programování simulací výborně hodí.

První Objekt

Nejlepší je začít příkladem. Popíšu tedy, jak vytvořit první, nejjednodušší objekt, který půjde umísťovat do **mapy,** tedy na plán levlu.

Postup:

- Otevřeme editor skriptů (F1) a vytvoříme nový skript, který pojmenujeme třeba `pokus`.
- Napíšeme řádek: `objectname oPrvniObjekt;`
- Zkompilujeme (F7)

A Objekt je vytvořen! Jenže tenhle objekt vůbec nic nedělá, ani do mapy umísťovat nejde. Co tedy s tím?

- Předně, potřebujeme mapu. Do hlavičky našeho skriptu si přidáme řádek
  `include map_0001_000F_0001_1001.kc`
- Pak je třeba upravit objekt. Jakékoli úpravy objektů, přidávání metod, atributů,... se dělají v objektové závorce. Podle toho kompilátor pozná, k jakému objektu modifikace patří.

  ```
  object oPrvniObjekt {
  }
  ```

- Je třeba k objektu přidat tag `InMap`, aby editor levlů věděl, že je objekt umístitelný do mapy a aby umožnil uživateli objekt do té mapy umísťovat.
- Asi nechceme, aby objekt byl vždy jen na souřadnicích (0,0), proto k objektu přidáme dva atributy, kde si objekt bude pamatovat svou aktuální pozici. Atributy se musí jmenovat `@ObjPosX` a `@ObjPosY` a musí být typu `int`. Atributy mají takzvaná **známá jména**, to se dá poznat podle toho, že jejich identifikátory začínají znakem `@`. Tím, že jsme použili známá jména, říkáme kernelu, že tyto atributy mají speciální význam. Konkrétně nyní říkáme, že objekt má právě v těchto atributech uloženy své souřadnice. K atributům ještě přidáme tag `Editable`, aby editor věděl, že má dovolit atributy měnit, a aby kernel věděl, že má hodnoty atributů ukládat/nahrávat s levlem. (Atributy označeny tímto tagem jsou `LevelLoad`.)

Takto by měl vypadat celý kód našeho prvního objektu:

```
#head
game ?
author    ?
version   EA7D_EF19_00B0_A149
include map_0001_000F_0001_1001.kc
#endhead

objectname oPrvniObjekt;

object oPrvniObjekt {
     edit {InMap}
     int @ObjPosX edit {Editable}, @ObjPosY edit {Editable};
}
```

- Zkompilujeme (F7)

Tvorba Levlu

Level si můžeme představit jako plán, jakousi herní mapu, kde jsou na různých místech umístěny různé objekty. Jestliže některý objekt má `LevelLoad` atribut, tak je zadán (nastaven na konkrétní hodnotu). Součástí levlu jsou i vytvořené, ale neumístěné objekty a globální (statické) objekty, dále nastavení `LevelLoad` globálních proměnných a nastavení některých dalších údajů.

Kernel level ukládá a nahrává.

Co se děje při nahrání levlu (load)?

Tyto akce se provádí vždy, nezáleží na tom, jestli se nahrává level v editoru nebo proto, že ho chceme hrát. Je jedno, jestli otvíráme nový prázdný level nebo již existující.

- Inicializuje se kernel.
- Nahraje se skript, nad kterým je level postaven. Zde se nahrávají informace o objektech, o jménech (viz dále), o globálních proměnných a o statických objektech.
- Vyhledá se potřebná grafika a zvuky a vše se nahraje
- Kernel inicializuje globální proměnné. Nastaví je na nulu, vytvoří kernelí pole; pokud je nahráván existující level, jsou nahrány hodnoty do `LevelLoad` globálních proměnných. Podrobně viz dále.
- Nyní se začnou pouštět skripty
- Nejprve se tvoří globální (statické) objekty a volají se na ně postupně konstruktory.
- Pak se tvoří ostatní objekty v levlu. Ty umístěné se po vytvoření umísťují do mapy.
- Tímto se level podařilo nahrát
- Kernel běží dál a každých 33 ms vyvolá takt.

Příklad, pokračování – Jak vytvořit level s porůznu rozmístěnými objekty oPrvniObjekt?

- V editoru levlů vytvoříme nový level nad skriptem `pokus` (Ctrl+N)
- Na panelu vpravo nahoře je jediný objekt na výběr a to náš `oPrvniObjekt`. Zatím nejde umísťovat, protože nemá přiřazenou grafiku. Něco mu tedy vybereme:
- Otevřeme nový soubor s automatismy (Ctrl+A), třeba `pokus.`
- K objektu přiřadíme grafiku. (Je tam na to button, který vypadá jako A v kroužku, jinak viz dokumentace k editoru)
- Uložíme automatismy (Ctrl+Shift+A)
- Můžeme umísťovat objekty do mapy.
- Až budeme spokojeni s „uměleckým dílem“, uložíme level (Ctrl+S)

Verze

Systém Krkal je navržen tak, aby hry v něm programované bylo možno stále vyvíjet, modifikovat a rozšiřovat o nové prvky. A to i více nezávislými uživateli najednou. Každý nově otevřený soubor se skriptem představuje jednu novou verzi. Soubor má dlouhé unikátní jméno, aby se žádný jiný soubor nejmenoval stejně. (přesněji: pravděpodobnost, že se budou dva soubory se skripty jmenovat stejně, se blíží nule)

Nově vytvořený soubor se skripty může rozvíjet některý jiný soubor se skripty (proto mluvíme o verzích a o vývoji verzí). Toto propojení se dělá pomocí direktivy `include` v hlavičce skriptu. Soubory nerozdělují kód podle tématu, jak jsme zvyklí z C++, ale podle toho, co kdy a kde vzniklo, soubory mapují historii vývoje. V nejnovější verzi jsou tedy nejnovější modifikace.

Poté co skript vytvoříme, můžeme programovat. Jazyk je navržen tak, aby bylo možné přidávat k existujícím objektům nové atributy, nová těla metod a pod., aniž by bylo nutné předělávat kód v předchozích verzích. Předchozí verze jsou většinou označeny jako **uzavřené**. To znamená, že už nikdy v budoucnu jejich zdrojový text nebude měněn a nemůže být měněn. (poznámka: v současnosti ještě není naprogramován mechanismus, který by znemožňoval změnu uzavřených verzí. Je nutné se tedy spolehnout na „pravidla slušného chování“)

A co dělat, když je nějaká uzavřená verze naprogramována nevhodně a my to pomocí čistých, přidávacích operací nemůžeme obejít? Pro tento případ jazyk umí i nečisté operace, což jsou příkazy, které píšeme do aktuální verze a které zpětně modifikují věci ve verzích předchozích. Nečistým operacím je třeba se snažit vyhnout, protože komplikují slučování verzí a tím celou rozšiřitelnost.

Vývoj skriptů a levlů zároveň

Je možné pracovat na nějaké otevřené verzi (programovat, přidávat, modifikovat, ladit a pak zas všechno smazat...) a zároveň nad touto verzí tvořit levly. Je snadno vidět, že zde vzniká řada problémů.

Příklad. Máme už vytvořen level L s objektem O. poté přidáme k objektu O nový atribut, který chceme zadávat v editoru levlů a nahrávat ho s levlem. Jestliže nyní staré levly zahodíme a začneme editovat nové, je všechno v pořádku. Ale co když budeme chtít nahrát level L? Nový atribut tam u objektu O ještě není, nebude tedy nahrán a jeho hodnota zůstane nedefinována. To může vést v lepším případě k ukončení kernelu, v horším případě k pádu celého Systému Krkal. Současný vývoj levlů i skriptů je nutný kvůli ladění, ale toto riziko je třeba mít na paměti.

Uzavírání

Až skript naprogramujeme a odladíme, můžeme ho **uzavřít**. Levly vytvořené nad uzavřeným skriptem, už zůstanou funkční a navíc budou fungovat stále stejným způsobem, protože skript již není možné měnit.

Vše o Objektech

Základní tagy

`OutMap` – Objekt je možné přidávat do levlu, vytvořená instance je pak vidět nahoře, v pruhu globálních objektů. Umísťovat do mapy možné není

`InMap` – Objekt je možno přidávat do levlu, jak do mapy, tak mimo mapu (vytvořit, ale neumístit objekt)

úplný popis viz dokumentace k jazyku.

Práce s Objekty

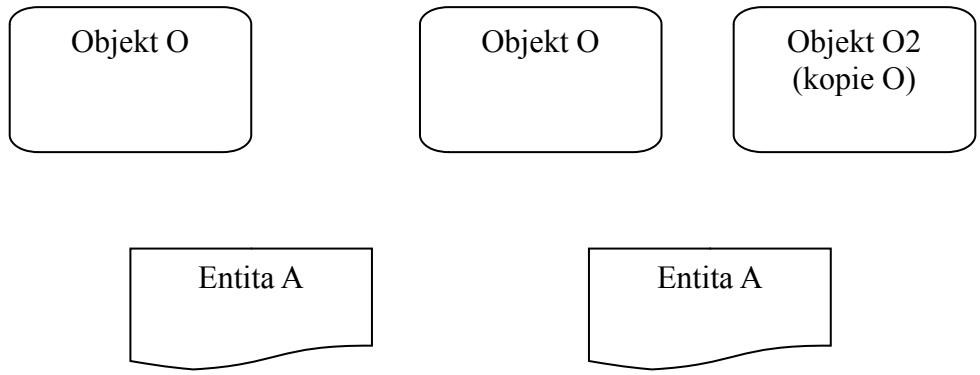
V jazyce je objekt popsaný jako typ, šablona, jde jen o informaci, která ještě „nežije“. Při startu levlu a později se podle **typů objektů** objekty vytvářejí, vznikají **instance objektů**. Ke každému typu může být vytvořeno 0 až n instancí. Vytvořený objekt již provádí svůj kód, mění hodnoty svých atributů ... prostě „žije“.

Jak může být vytvořena instance objektu?

Při vytváření nového levlu se automaticky vytvoří všechny statické objekty. Jejich atributy jsou nedefinovány (až na výjimky). Je zavolán **`constructor`**, což je speciální metoda, určená k tomu, aby si objekt do svých atributů dosadil počáteční hodnoty a zahájil svou činnost.

Při nahrávání existujícího levlu se tvoří nejdříve statické objekty a pak ostatní objekty obsažené v levlu. Do atributů, které jsou označeny jako `LevelLoad`, se nahrají z levlu hodnoty. Ostatní atributy zůstávají nedefinovány. poté se zavolá **`lconstructor`**. (`constructor` se nevolá.) Pořadí tvorby objektů při startu levlu je ovlivněno touto podmínkou: Pokud objekt A dostává v nějakém `LevelLoad` atributu pointer na objekt B, tak objekt B musí být vytvořen dříve než objekt A. Poznámka: Pokud objekty odkazující se na sebe pomocí pointerů vytvoří cyklus, level nepůjde uložit.

Je možné vytvořit **kopii** objektu, to běžně dělá například editor, když umísťuje objekty, takže **pozor na to!** Nová kopie dostane naprosto stejná data, jako původní objekt. Tedy, pokud měl původní objekt atribut typu `int` nastaven na 17, bude mít nový objekt ten samý atribut nastaven také na 17. Pokud původní objekt měl pointer na jiný objekt, bude mít nový objekt pointer na ten samý objekt. To samé platí pro ostatní pointery. Když to shrnu, tak operace kopírování vytvoří dva stejné objekty, ale pokud z objektu vede ukazatel na nějakou entitu A, tato entita již okopírována nebude a oba objekty si na ni ukazují.



Kopírování se uzavře voláním **`cconstructoru`**. V něm by měl objekt vyřešit situace s pointery.

A v neposlední řadě samy skripty mohou vytvářet nové objekty, voláním příkazu

```
new <typ>
```

nebo příkazem

```
new vartype <proměnná typu name obsahující typ objektu>.
```

Operace `new` Objekt vytvoří, atributy jsou nedefinovány, a zavolá **`constructor.`** Poté operace `new` vrátí hodnotu typu `objptr`.

Pointer na objekt - `objptr`

Jedná se o speciální typ proměnné ve které je uložen ukazatel na žijící (či už mrtvou) instanci objektu. Nejsou to klasické pointery, ale jistá identifikační čísla, o kterých platí:

- 0 je prázdný ukazatel, žádný objekt nemůže mít číslo 0. V jazyce je třeba psát `onull`.
- Čísla jsou objektům přiřazovány postupně. Každý nový objekt dostane nové číslo, které ještě nikdy předtím (myslí se „za tohoto běhu kernelu“) nebylo použito.
- Službou `@ExistsObj` lze kdykoli zjistit, zda je daný objekt živý, či mrtvý.
- K mrtvému objektu se již nedá přistupovat.

Přes pointer na objekt se dá s objektem pracovat. Je možné volat metody objektu, je možné posílat objektu zprávy, je možné říct nějaké službě kernelu, ať s objektem „něco“ udělá. Například lze objekt umístit do mapy, odebrat z mapy, číst jeho souřadnice, zjišťovat, zda je v kolizi s jiným objektem a pod. Je možné objekt zničit.

Není možné přistupovat k atributům cizího objektu.

Příkaz **`typeof(`**`<objptr>`**`)`** vrací typ objektu, jako hodnotu typu `name`.

Ničení objektů

```
delete <objptr>
```

Tento příkaz zruší objekt. A to následujícím způsobem:

- Počká se, až se ukončí všechny běžící metody daného objektu. (použije se mechanismus **callend** zprávy, viz dále)
- Pokud je objekt umístěn v mapě:
  - Objektu je zavolána metoda `@MapRemoved`
  - Objekt je odebrán z mapy
- Objektu je zavolána metoda `destructor`
- Objekt je zrušen.

Objekt je možno zničit i přímým voláním metody `Destructor`. Zde pozor, aby objekt během destrukce ještě neběžel, to by vyvolalo panický error. Je lepší používat zprávy, třeba callend. I v tomto případě je objekt napřed odebrán z mapy.

Poznámka

metody `constructor, lconstructor, cconstructor, destructor` jsou nepovinné. Pokud je objekt nepotřebuje, není nutné je programovat. Existuje ještě metoda `uconstructor`, která se volá jak na místě `constructor`u, tak `lconstructor`u.

Editor Mod a Game Mod

Jak již bylo řečeno výše, není rozdíl v tom, zda se level nahrává za účelem hraní nebo editace. Start kernelu a konstrukce objektů probíhá pokaždé stejně. Jak tedy například zabránit tomu, aby neposedná příšera se během editace nesebrala a nezačala prohledávat všechny kouty levlu a nekradla tam jablka, která tam uživatel zrovna pracně aranžuje?

Každý takovýhle „akční“ objekt musí mít ve svém kódu napsanou podmínku, která mu během editace takové chování znemožní. V editoru příšera nedělá nic, jen si správně srovná a uloží své atributy, zato ve hře se pustí do sbírání jablek.

Podmíněné chování je možné naprogramovat pomocí služeb kernelu:

```
@IsGame()
@IsEditor()
```

Typy Proměnných

základní typy

`int` – 4 byty, celá čísla, znaménkový
`char` – 1 byte, znaky nebo nezáporná čísla 0 až 255
`double` – 8 bytů, reálná čísla
`name` – 4 byty, obsahuje identifikaci KSID jména. Podrobný popis viz dále.
`objptr` – 4 byty, pointer na objekt

Další, kernelem podporované typy

`string[x]` – obsahuje textový řetězec o maximálně x znacích, kde x je maximálně 250(?). Řetězec je při předávání funkcím kopírován. Nepředává se tedy pointer jako u klasického `char*`

`inta`,`chara`,`doublea`,`namea`,`objptra` – pointery na kernelí pole. Pole se tvoří příkazem `new`, obsahuje pak 0 až n položek. Pole má proměnnou velikost a narůstá podle potřeby.

Pokročilé, kernelem nepodporované typy

struktura
pole proměnných
pointer na proměnnou

Více informací o typech a o práci s nimi viz dokumentace k jazyku.

Atributy

Atribut je popsán typem a jménem, volitelně můžou být k atributu připsány ještě tagy.

Typ atributu může být kterýkoli z typů, ale jen ty kernelem podporované typy lze jednoduchým způsobem editovat v editoru a ukládat jejich hodnoty s levlem.

Jméno je tvořeno jednoduchým identifikátorem. (složené identifikátory zde přípustné nejsou) Existuje i několik známých jmen pro atributy, které, podobně jako tagy, říkají, že atributy mají pro kernel nějaký speciální význam.

Jako atribut lze chápat i speciální položky ovlivňující editaci. Položka `scripted` a skupinové proměnné. Viz dokumentace k jazyku a k editoru.

Ukládání hodnot atributů u levlu – kernelem podporované typy

- Tag `LevelLoad` určuje, že se má hodnota atributu ukládat a nahrávat s levlem
- Tag `Editable` nastavuje `LevelLoad` a dále říká, že má být atribut v editoru editovatelný.
- kombinace `Editable, NoLevelLoad` vytvoří editovatelný atribut, který se neukládá
- Jakékoli tagy ovlivňující editování atributu automaticky zapínají tag `Editable`.

Jak ukládat atributy složených typů, např. spojáky?

Během ukládání levlu kernel postupně zavolá u každého objektu metodu `@ESaveMe`. Objekt v této metodě může ukládat hodnoty jakýchkoli základních a kernelem podporovaných typů do speciálního datového streamu. Každá instance má svůj stream. Zapisuje se sekvenčně, pomocí služeb kernelu, hodnotu za hodnotou.

V `lconstructoru` má objekt příležitost si uložené hodnoty ze streamu zase nahrát. Je třeba nahrávat hodnoty stejných typů a ve stejném pořadí jako byly ukládány.

Varování. Vyvarujte se dat, která pomocí pointerů sdílí několik objektů. Můžou pak vznikat problémy s ukládáním/nahráváním objektů. Který objekt data vlastní? Který je má uložit? Jak je naloadovat? Jinými slovy: Situace se stává nepřehlednou.

Jak editovat atributy složených typů, např. spojáky?

Editor umí editovat pouze proměnné základních, kernelem podporovaných typů. Něčemu, jako je spoják, prostě nerozumí. Skript tedy musí sám editoru říct, co, kde a jak se má editovat. K tomu slouží abstraktní editovatelná položka `scripted`. Vnitřek položky (rozmístění a funkce editačních polí, buttonů) je určován až za běhu, skriptem.

```
scripted <id> constructor <methodname>();
```

`methodname` je KSID jméno metody, kterou editor volá, když chce vnitřek položky vytvořit. Viz podrobný popis služeb kernelu a viz následující příklad. (`oSeSpojakem`)

Atributy, ovlivňující automatickou grafiku

V Systému Krkal se k objektům nepřiřazuje grafika přímo, ale přes takzvané **automatismy**. Automatismus je soubor rozhodovacích pravidel, uspořádaných ve stromu, který, nezávisle na objektu, volí výslednou grafickou podobu.

Automatismy se můžou mimo jiné rozhodovat i podle hodnot v atributech objektu. Tyto atributy musí být speciálně označené a musí k nim být přiřazeno KSID jméno.

Rozhodování pak funguje následovně: Pokud se chce automatismus podle nějaké proměnné rozhodnout, zkusí ji u objektu najít podle KSID jména. Automatismus se poté rozhodne. Automatické rozhodování funguje i tehdy, pokud příslušný atribut u objektu nebyl nalezen nebo obsahuje nekorektní hodnoty. (V tomto případě mám na mysli, že atribut je inicializován na něco jiného, než se očekává, ne, že **nebyl** inicializován a jeho obsah je tedy nedefinován!)

Atributy ovlivňující automatismy musí být typu `name`. Objekt k atributům přistupuje běžným způsobem přes identifikátor atributu (na volbě identifikátoru nezáleží). Automatismus k atributu přistupuje přes přiřazené KSID jméno.

Příklad.

```
objectname oSeSmerem; // založím si nový objekt

// nadefinuji si množinu Smery a 4 její prvky
voidname Smery, Sever, Vychod, Zapad, Jih;
depend Smery << {Sever, Vychod, Zapad, Jih}

// upřesním vlastnosti objektu oSeSmerem
object oSeSmerem {
     name smer edit {Editable, Auto=Smery, Is < Smery};
     // atribut "smer" je typu "name",
     // je editovatelný
     // je to atribut ovlivňující automatismy, přiřazené KSID jméno je "Smery",
     // Is < Smery říká, že v editoru se do atr. dají přiřazovat jen hodnoty z množiny Smery,
     // ukládá se s levlem

     constructor() { smer=Sever;}
     // atribut je třeba inicializovat. lconstructor není potřeba,
     // protože atribut se nahrává s levlem

     // metody pro čtení a zápis atributu
     name decl GetSmer() { return smer; }
     void decl SetSmer(name ::Smer) {
          smer = Smer;
          @ResetAuto(this,0,1); // po změně chci, aby se grafika přepočítala
     }
}
```

Atributy známých jmen

K objektu mohu přidat celou řadu atributů známých jmen. Jejich úplný výčet a popis viz referenční příručka. Atribut známého jména je proměnná, u které je předem dohodnut její význam, použití, funkce, ... A tento význam zná jak kernel, tak programátor. (tedy měl by znát :-))

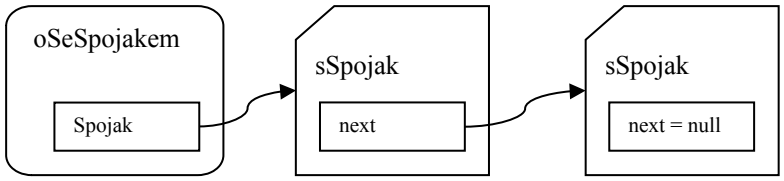
Kernel se čas od času potřebuje objektu zeptat na nějakou informaci a tato informace může být uložena právě v atributu známého jména. Kernel tyto atributy dokonce může sám měnit.

K objektům není třeba atributy známých jmen přidávat, pokud nejsou potřeba. Kernel, jestliže potřebuje přistoupit k nějakému atributu, ale nenajde ho, tak místo něj vezme jeho defaultní nastavení.

Příklad. Když nechci objekt umísťovat do mapy, nemusí mít atributy `@ObjPosX`, `@ObjPosY, @ObjPosZ.` Když přesto objekt do mapy umístím, bude umístěn na defaultní souřadnice, které jsou (0,0,0).

Příklad – Objekt, u kterého chci ukládat spojový seznam

Tento příklad je určen k procvičení si konstruktorů a jako ukázka složitějšího kódu v jazyce Krkal C. Vytvoříme objekt se spojovým seznamem intů, seznam bude možno v editoru editovat a objekty i se seznamem pak ukládat i nahrávat.




```
objectname oSeSpojakem;

object oSeSpojakem {
     edit {OutMap}  // aby byl objekt v editoru vidět.
                    // Budeme moci objekt vytvořit umístěním mimo mapu
     struct sSpojak {     // definice spojového seznamu typu zásobník
          int a;
          sSpojak *next;
     }

     sSpojak *Spojak;     // oddut začíná spoják.

     // konstruktor pro případy, když vzniká nová instance objektu
     constructor() {
          Spojak = null; // nechť je seznam prázdný
     }

     // Když dojde k okopírování objektu, je třeba okopírovat si i celý spojový seznam
     cconstructor() {
          sSpojak *p = Spojak;
          sSpojak **p2 = &Spojak;
          while (p) {
               *p2 = new sSpojak;
               (**p2).a = p->a;
               p2 = &(**p2).next;
               p = p->next;
          }
          *p2 = null;
     }

     // Pří zániku objektu odstraním i celý spojový seznam
     destructor() {
          sSpojak *p2, *p = Spojak;
          while(p) {
               p2 = p;
               p = p->next;
               delete p2;
          }
     }
```

```
// Tatu metodu volá kernel, když ukládá level. Uložíme si i spoják.
void @ESaveMe() {
     sSpojak *p = Spojak;
     while (p) {
          @SaveInt(p->a); // postupně uložíme hodnoty ze spojáku
          p = p->next;
     }
}

// v loadconstructoru spoják nahrajeme
lconstructor() {
     sSpojak **p2 = &Spojak;
     while(!@SLEof()) {  // dokud ve streamu něco je
          *p2 = new sSpojak;
          (**p2).a = @LoadInt(); // nahráváme to postupně do spojáku
          p2 = &(**p2).next;
     }
     *p2 = null;
}

// řekneme editoru, že chceme mít položku „spoják“,
// kterou si chceme sami naskriptovat
scripted spojak constructor ::Init();

// Initace položky, zde zobrazíme příslušné ovládací prvky
void ::Init() {
     sSpojak *p = Spojak;

     @EAddButton(0,1,"PridejPolozku",null,0,::PridejPolozku);
     // Služba přidá button, po jehož zmáčknutí se zavolá metoda ::PridejPolozku

     while (p) {
          // Pro každý člen spojového seznamu, přidám editační pole
          // Editoru předávám adresu na p->a, takže během editace, nesmím
          // s p->a  nikam hnout.
          @ECreateInt(&p->a,"polozka");
          @EPlaceItem();
          p = p->next;
     }
}

// Vytvořím nový prvek seznamu. A umístím ho na první místo.
void ::PridejPolozku() {
     sSpojak *p = Spojak;
     Spojak = new sSpojak;
     Spojak->a = 0;
     Spojak->next = p;
     @ECreateInt(&Spojak->a,"polozka");
     @EPlaceItem(1,0);
```

```
	}
}
```

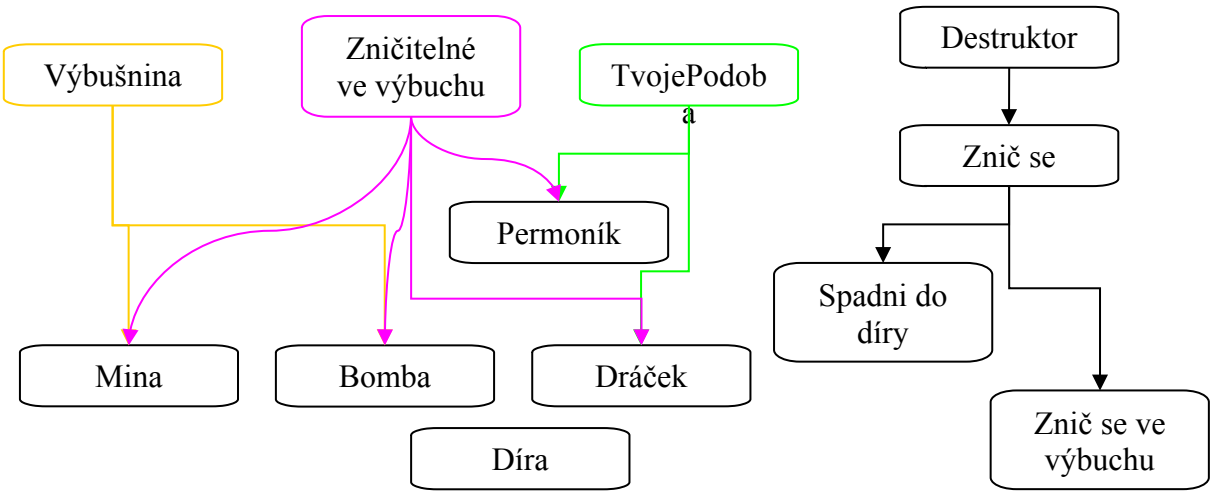
Tentýž objekt řešený pomocí kernelího pole

Situace z minulého příkladu se dá naprogramovat i mnohem jednodušeji, spojový seznam obsahoval jen jednu proměnnou typu `int`. Můžeme tedy využít kernelího pole `int`ů. Kernelí pole mají proměnnou velikost 0 až n prvků, podobně jako spojové seznamy.

```
objectname oSeSpojakem2;

object oSeSpojakem2 {
	edit {OutMap}
	inta spojak edit {Editable};  // pole je možno v editoru editovat
				// a kernel ho sám automaticky ukládá, nahrává s levlem

	// pole je třeba vytvořit
	constructor() {
		spojak = new inta; // pole po vytvoření bude prázdné
	}

	// a zničit
	destructor() {
		delete spojak;
	}

	// spojak je pointer na pole proměnné velikosti. Když dojde k okopírování
	// objektu, musím si okopírovat i pole.
	cconstructor() {
		spojak = @CopyIntA(spojak);
	}
}
```

KSID Jména

V Systému Krkal je možno deklarovat **jména** a mezi nimi vytvářet závislosti. Můžeme se na to dívat i tak, že tvoříme množiny a do nich přidáváme prvky. KSID jména jsou pak široce používána na mnoha různých místech.



Deklarace Jmen

```
voidname
objectname
methodname
paramname
```

Deklarovat jméno je možné jedním z výše uvedených klíčových slov. Jména typu void nemají přiřazen žádný zvláštní význam, způsob jejich využití je čistě na uživateli. Deklarací jména typu objekt je vytvořen nový typ objektu a od této chvíle je možno k objektu přidávat metody, atributy a pod. Další dva typy jmen jsou určeny pro pojmenování `safe` metod a jejich parametrů. (Poznámka: V kernelu existuje ještě mnoho dalších typů KSID jmen, například pro automatismy. K těm ale skripty nemají přístup.)

Někdy lze deklarovat jména i při jejich prvním použití. K tomu slouží direktiva `decl`. A takzvaná lokalizovaná jména není třeba předem deklarovat vůbec. Lokalizovaná jména začínají :: Tento znak před jménem naznačuje, že jménu chybí začátek. Začátek se automaticky doplňuje podle toho, k čemu je jméno lokalizováno. A možnosti jsou dvě: Pokud se jméno vyskytne uvnitř objektu, je lokalizováno ke jménu objektu. Pokud se jméno vyskytne jako parametr metody, je lokalizováno ke jménu metody.

Příklad.

```
objectname oObjektik;
methodname Metoda;

object oObjektik {
     void ::Metoda(int ::P1) {}
     void Metoda(int ::lokal, int decl global) {
          ::Metoda()  //volání první metody u této instance oObjektiku, parametr nepředávám
     }
}
```

Jsou zde tyto jména:

- `oObjektik` (jméno objektu)
- `oObjektik::Metoda` (jméno první metody)
- `oObjektik::Metoda::P1` (jméno parametru u první metody)
- `Metoda` (jméno druhé metody)

- `Metoda::lokal` (jméno parametru u druhé metody)
- `global` (jméno parametru, který využívá druhá metoda)

Zápis se čtyřbotkou na začátku a lokalizování je tedy jen zkratkou za dlouhá strukturovaná jména (jak jsou vidět výše). KSID jména mají globální platnost. A můžu klidně napsat například:

```
objectname oPermonik;
object oPermonik {
     void oObjektik::Metoda(float ::P1, name ::P2) {}
}
```

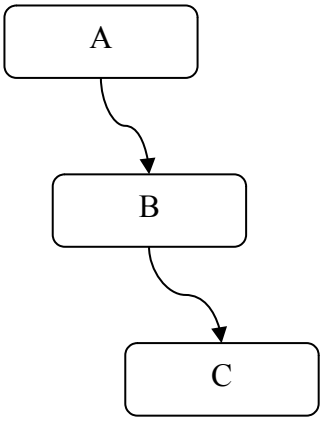
Přidal jsem do objektu `oPermonik` metodu stejného jména, jako byla u objektu `oObjektik`. Ale pozor, přidám-li tam metodu jména `::Metoda`, přidal jsem tam metodu, která se jmenuje úplně jinak, a to `oPermonik::Metoda`.

Některá jména si s sebou nesou i nějakou další informaci. U objektových jmen je to vnitřní struktura objektu, u jmen pro `safe` metody je to jejich návratový typ. Není důležité u kolika objektů metodu téhož jména vytvoříme a jaké bude mít v konkrétních případech parametry a tělo, ale návratový typ, ten musí zůstat pořád stejný. Jména parametrů jsou speciální jenom tím, že se ví, že jde o parametr. Jedno a totéž jméno můžeme použít k pojmenování parametrů u různých metod, může jít i o parametry různých typů. Ale nemělo by jít o parametry různých významů, to vnáší do programování nepřehlednost.

Závislosti mezi jmény

Mezi jmény lze definovat závislosti.

```
depend A << B; // přiřadili jsme B do množiny A
```



Pak lze říkat, že:

- jméno B je menší než jméno A, (`B < A`)
- B je prvkem množiny A, A obsahuje B
- A je obecnější případ B
- B se z A vyvinulo specializací a doplněním o dalších podrobnosti

Závislosti se chovají **tranzitivně**. Napíšeme-li ještě:

```
depend B << C;
```

Pak platí také že:

- A > C
- C je nejen prvkem B, ale také je prvkem A

Práce se jmény

KSID jména je možno ukládat do proměnných typu `name`. Proměnnou typu `name` je možno použít na většině míst, kde se píší KSID jména jako konstanty. Například můžu vytvořit objekt typu A, kde A je proměnná, a pak tento „neznámý“ objekt třeba umístit do mapy nebo mu poslat zprávu... Při používání proměnných typu `name` má jazyk řadu výjimek a proto doporučuji se napřed podívat do dokumentace k jazyku.

Operacemi <=, <, >, >= můžu zjišťovat závislosti mezi jmény. Zde pozor, jména jsou uspořádána jen částečně. Když například zjistím, že jméno A není menší nebo rovno než jméno B, **nemůžu** z toho odvozovat, že jméno A je větší než B.

Velmi užitečné je sdružovat do skupin objekty. Můžeme založit například množinu objektů zničitelných ve výbuchu. Výbuch pak, než bude ničit objekty, které zasáhl, se ještě podívá, jestli typ postiženého objektu patří, či nepatří mezi objekty zničitelné ve výbuchu. Pro tyto situace je možné kombinovat jména typu objekt a jména typu void.

Příklad.

```
objptr o;
o = ... // nechť v o je nějaký neznámý objekt

// zjistím zda typ objektu patří do množiny objektů zničitelných ve výbuchu.
if (typeof(o) <= ZnicitelneVeVybuchu) {
     o->ZnicSe() message; // v tom případě řeknu objektu, ať se zničí.
}
```

Dědění

Další možnost využití závislostí je **dědění.** Nejdříve je nutné označit, co se bude dědit (které metody, které atributy, ...). To se dělá klíčovým slovem **`inherit`**. `inherit` můžeme napsat i k celé objektové závorce. Zdědí se pak vše, co je uvnitř.

```
objectname oPlaceable;
object oPlaceable inherit {
     int @ObjPosX edit {Editable};
     int @ObjPosY edit {Editable};
}

objectname oStena;
depend oPlaceable << oStena;
```

// stěna zdědí od objektu `oPlaceable` jeho dva atributy a případné další věci, které `oPlaceable` má, či **bude mít** označeny k dědění.

Při dědění se k potomkům „kopírují“ atributy a těla metod. Jména těchto entit však **zůstávají** taková, jaká byla u toho předka, kde entita vznikla.

Příklad

```
objectname oDeda, oOtec, oPotomek;
depend oDeda << oOtec << oPotomek;
     // Otec dědí od dědy. Potomek dědí od Dědy i od Otce.
methodname Metoda;

object oDeda inherit {
     void ::Metoda() {}  // oDeda::Metoda
     void Metoda() {}    // Metoda

     int a;
     int b;
}
```

```
// oOtec zdědil: oDeda::Metoda, Metoda, oDeda::a, oDeda::b;
object oOtec inherit {
     void ::Metoda() {}  // oOtec::Metoda
     void Metoda() {     // Metoda
          a = 1;         // pristup k oOtec::a
          oDeda::a = 2;  // pristup k oDeda::a
          ::Metoda();    // volám svou metodu oOtec::Metoda
          oDeda::Metoda();// volám metodu dědy
     }

     int a;
}
```

```
// oPotomek zdědil: oDeda::Metoda, Metoda, oDeda::a, oDeda::b;
// oOtec::Metoda, Metoda, oOtec::a;
object oPotomek inherit {
     void Metoda() {     // Metoda
          Metoda();      // metodu tohoto jména mám třikrát, jednou od otce,
                         // jednou od dědy a jednu svojí (teď v ní píšu komentář!)
                         // zavolají se všechny tři těla v nedefinovaném pořadí.
          oDada::Metoda();      // volám metodu od dědy
          // ::Metoda(); // nelze, tuhle metodu nemám

          b = 1;         // přístup k proměnné oDeda::b
          // a = 2;      // nelze, kompilátor neví, ke kterému a má přistoupit
          oOtec::a = 3;  // přístup k proměnné od otce
     }
}
```

Metody

Metoda je jediná oblast v jazyce, která může obsahovat kód. Každé tělo metody patří k nějakému objektu. Existují dva druhy metod. **`Safe`** metody a **`direct`** metody.

safe metody

Mezi `safe` metody patří i konstruktory, destruktor a všechny metody známých jmen. `Safe` metody jsou pojmenovány KSID jménem, stejně tak jejich parametry jsou pojmenovány KSID jménem. Volání `safe` metod se provádí přes kernel. Lze je volat opožděně, lze takzvaně **posílat zprávy.**

`Safe` metod může být u objektu **více stejného jména**. Ještě jednou: Je stejné jméno, stejná návratová hodnota, ale mnoho různých nezávislých těl, která přijímají potenciálně různé

parametry, různých typů. Tato vlastnost `safe` metod je klíčová pro rozšiřování vyvíjených her, bez nutnosti zasahovat do už existujícího kódu.

Mechanismus safe volání

Co potřebuje kernel vědět, než provede volání?

- Cílový objekt. Zadává se pointer na instanci objektu (hodnota typu `objptr`). Není-li objekt zadán, má se zato, že se má zavolat metoda u právě běžícího objektu (**`this`**).
- KSID jméno metody. Možno zadat jako konstantu nebo přes proměnnou typu `name` (zde pozor, jazyk má na tento způsob jistá omezení, viz dokumentace k němu)
- Typ volání. Zda se jedná o přímé volání nebo nějaký druh zprávy.
- Parametry. Ty jsou volitelné, lze jich zadat 0 až n, jejich pořadí, jména a typy určuje volající.
- S každým parametrem je třeba zadat dvojici <KSID jméno parametru>, <předávaná hodnota> , kompilátor údaje ještě rozšíří o <předávaný typ>, ten je zjištěn podle typu předávané hodnoty. Typy parametrů u `safe` volání a `safe` metod můžou být pouze **kernelem podporované typy**. Žádné struktury, pointery a pod. Klíčovým slovem `ret` lze říct, že chci do parametru vracet hodnotou.

příklady

```
::Metoda(::arg1 : 5, ::arg2 : s) message;
// Posílám si sám sobě zprávu se dvěma parametry.

s->oObjektik::Metoda();
// s je pointer na nějaký objekt a já volám jeho metodu přímo.
```

Jak kernel vyhodnocuje volání?

- Kernel ověří, zda je cílový objekt živý a zda jméno metody je opravdu platné KSID jméno metody.
- Kernel se snaží u cílového objektu najít tělo nebo těla metod, jejichž KSID jména odpovídají volanému KSID jménu. Provádí se zde takzvané **zobecňování**. Když kernel u objektu nenajde přímo volanou metodu, zkouší zavolat nějakou obecnější metodu, ale takovou, která má k volané metodě nejblíže. Obecnější metoda je ta, která je podle závislostí nad KSID jmény větší než volaná metoda. Viz výše.
- Pokud kernel žádné metody nenašel, volání bude ukončeno, pokud to bylo přímé volání, bude nahlášena běhová chyba.
- Přepne se kontext. To znamená, že se zpřístupní volaný objekt
- Kernel má seznam těl, která bude volat. Začne je tedy volat v nedefinovaném pořadí:
    - přiřadí k sobě volané parametry a cílové parametry, podle jejich KSID jmen, opět se zde provádí zobecňování. Je možné předávat parametry v různém pořadí, některý parametr nezadat (tělo do nezadaných parametrů dostává defaultní hodnoty a může si zjistit, zda byl, či nebyl parametr zadán). Je možné předávat při volání i nějaké parametry navíc – tělo je prostě nedostane.
    - Do cílových parametrů jsou překopírovány volané parametry. Typ je automaticky zkonvertován
    - Zavolá se tělo metody
    - Pokud jde o přímé volání a volající si některé parametry označil jako `ret` a i volaný má odpovídající parametr označen jako `ret`, dojde k takzvanému

vracení hodnotou. Hodnota z volané funkce je kopírována zpátky volajícímu. (hlídají se tu případy, že se na jedno místo vrací víckrát nebo že se naopak nevrací vůbec, a vše se hlásí jako errory.)
    - Stejný postup se provede s případnou návratovou hodnotou funkce.
- kontext je vrácen do původního stavu.
- Nyní je příležitost pro vyvolání takzvaných **`callend`** zpráv.
- Pokud se provádělo přímé volání, vrátí se řízení do volající funkce, pokud šlo o zprávu, zůstává řízení v kernelu a ten třeba vyvolá další zprávu ve frontě ...

Zprávy

Přímé volání přeruší běh stávající metody, vykoná se kód metody volané a teprve potom se řízení vrátí zpět. Metody volané přímo mohou vracet hodnoty. (návratová hodnota funkce, vracení hodnotou (`ret`))

Volání metody jako **zpráva** probíhá jinak. Při zavolání se zpráva uloží do příslušné fronty, včetně všech předávaných parametrů. A volající funkce pokračuje v práci. Volaná metoda se provede až někdy později, až na ní přijde řada. Záleží na typu zprávy.

Při používání zpráv si nemusíme lámat hlavu s tím, jestli cílový objekt danou zprávu přijímá nebo jestli je cílový objekt vůbec živý. Kernel v těchto případech zprávu jednoduše zahazuje.

Všechny zprávy se ukládají do front, front je sice mnoho, ale pokud vyvolám za stejných okolností stejný typ zprávy, tak se zpráva uloží do stejné fronty a mohu si být jist, že dříve vyvolaná zpráva se i dříve provede.

Takt

Když kernel chce vyvolat nový takt má k dispozici hned čtyři fronty zpráv. **`message, end, nextturn`** a **`nextend.`** Kernel se nejprve podívá do další fronty **`timed`**, kde jsou časované zprávy a vyzvedne ty, kterým už uzrál čas, a přidá je na konec fronty `message`**`.`**

Pak provádí postupně všechny zprávy z fronty `message`. Když frontu `message` vyprázdní a ve frontě `end` něco je, prohlásí frontu `end` za frontu `message` a frontu `end` nastaví jako prázdnou. A vše se opakuje.

Vyprázdněním prvních dvou front skončí takt. Kernel převede zprávy z front `nextturn` a `nextend` do front `message` a `end` a tím je vše připraveno k vyvolání nového taktu.

Typy zpráv

- `message`
- `end`
- `nextturn`
- `nextend`
  zprávy se vkládají na konce příslušných front.
- `timed <čas>` - časovaná zpráva. Zpráva se vyvolá až po uplynutí příslušného počtu milisekund kernelího času. Pokud tato doba nastane někdy mezi takty, zpráva musí počkat až na další takt, vyvolá se tedy trošičku později.
- `callend <objptr>` - Tyto zprávy se vyvolávají po ukončení metody. Pokud se v rámci volání metody provádí více nezávislých těl, tak se `callend` zprávy vyvolávají až po ukončení činnosti **všech** těl. `callend` front je mnoho, s každým

vnořeným voláním se vytvoří jedna, do které se pak ukládají zprávy, které čekají na ukončení právě tohoto volání. `Callend` zpráva se vyvolá **co nejdříve** (vybírá se tedy, co možná nejbližší a nejzanořenější fronta), ale **ne** tehdy, pokud ještě **běží nějaká metoda od objektu `objptr.`** `Callend` tedy čeká, až daný objekt ukončí veškerou svoji činnost. (Zprávy ve frontách se nepočítají jako činnost, to nejsou běžící metody.)

Direct metody a direct volání

`direct` metody se volají přímo. Jména `direct` metod a jejich parametrů jsou obyčejné identifikátory, které podléhají stejným pravidlům jako jména atributů. Volání probíhá stejně, jak to známe z jazyka C. Počet parametrů, jejich pořadí a typy musí sedět. Je možno předávat i pointery, na druhou stranu není možno vracet hodnotou. `Direct` volání je vždy přímé. Není přípustné volání typu zpráva, stejně tak není možné mít více těl `direct` metod pod stejným jménem.

I při `direct` volání se napřed inicializuje kontext, aby měla metoda přístupný svůj objekt. Při ukončování metody, dojde k vrácení kontextu do původního stavu a k případnému zavolání `callend` zpráv. I `direct` metody mohou vracet funkční hodnoty.

Jaký typ metody používat?

`Direct` volání je rychlejší, proto `direct` metody by měly být různé pomocné rutiny s úzce specializovaným použitím, které se často volají během nějakého náročnějšího výpočtu.

Vše ostatní by mělo být `safe`. To umožňuje použití zpráv a další výhody, ale hlavně **rozšiřitelnost**. `Safe` by měly být všechny metody, pomocí kterých komunikují objekty mezi sebou.

Poznámka: Kernel při vyhledávání vhodných metod a parametrů u `safe` volání využívá perfektní hashovací tabulku. Tedy veškerý postup má předpočítán. Pomalé je na `safe` volání to, že se tam musí na několika místech kopírovat všechny parametry a konvertovat jejich typy a je třeba za běhu získávat a ověřovat řadu údajů (u `direct` k tomu dochází už během kompilace)

Příklad – Naprogramujeme Bombu

```
////////////////////////////////////////////////
//  Základ všech umístitelných objektů
objectname oPlaceable;

object oPlaceable inherit {
      int @ObjPosX edit {Editable}, @ObjPosY edit {Editable};
      name @APicture edit {Editable};      // To abychom mohli v editoru volit grafiku
      // pro @APicture není třeba programovat konstruktor (jedna z výjimek)
}


////////////////////////////////////////////////
// Výbuch. Zazáhne oblast 3x3 políčka (Pracujeme s mapou na čtvercích o
// velikosti 40x40 pixelů.) A vše zničitelné výbuchem v té oblasti bude
// zničeno. Ničení neprobíhá okamžitě, ale trvá určitý čas.
```

```
objectname oVybuch;
depend oPlaceable << oVybuch;

voidname ZnicitelneVeVybuchu; // Množina všeho zničitelného
methodname ZnicSeVeVybuchu, Vybouchni;
depend Destructor << Vybouchni << ZnicSeVeVybuchu;
// Pokud objekt nemá metodu ZnicSeVeVybuchu, tak se napřed zkusí zavolat metoda
// Vybouchni, pokud ani ta u objektu není, tak bude objekt zničen.

object oVybuch {
      char @CollisionCfg;      // V tomto atributu předáváme kernelu informaci o tom,
                               // jak je objekt veliký, jakou část buněk zabírá
      int @NumCellX, @NumCellY;
      name @clzAddGr;          // S kterými objekty má objekt kolidovat?
      constructor() {
            @CollisionCfg = @eKCCrect1 | @eKCCcell;
                  // Objekt bude zabírat obdélníkovou oblast a bude zabírat celou buňku, včetně podlahy
            @NumCellX = 3; @NumCellY = 3; // Velikost obdélníku nechť je 3x3 buňky
            @clzAddGr = ZnicitelneVeVybuchu;      // Nechť koliduji jen s těmito objekty
      }

      // Po umístění do mapy začnu škodit.
      void @MapPlaced() {
            Destructor() timed 231; // Sám sebe zničím za 231 ms.
            objptra objs;           // pointer na pole objektů proměnné velikosti
            int f;
            objs = @FindCollidingObjs(this);      // Fumkce vrátí objety, se kterými
                                    // koliduji, jsou to jen samé objekty zničitelné ve výbuchu.
            for (f=0; f<objs->GetCount(); f++) { // projdu pole
                  objs[f]->ZnicSeVeVybuchu() timed 165;
                  // A všechny objekty zničím za 165 ms.
            }
            delete objs;   // Pole už nepotřebuji, zničím ho.
      }
}


////////////////////////////////////////
// Základ vybuchujících objektů
// Vybouchnutí znamená, že objekt sám sebe zničí a vytvoří kolem sebe
// výbuch
objectname oVybuchuje;
depend oPlaceable << oVybuchuje;

object oVybuchuje inherit {
      void Vybouchni() {
            objptr o;
            if (@IsObjInMap(this)) {      // zda jsem ještě umístěn v mapě
                  // Zjistím souřadnice své buňky a umístím kolem sebe výbuch
                  o = new oVybuch;
                  int x,y,z;
                  @FindObjCell(this,&x,&y,&z);  // Zjišťuju souřadnice své buňky
                        // Dělám to proto, aby Výbuch byl zarovnaný přesně na buňky.
                  @WriteObjCoords(o,(x-1)*40,(y-1)*40);
                        // Objektu předávám souřadnice v pixelech
                  @PlaceObjToMap(o);
            }
            delete this;       // Zničím sám sebe.
      }
}
```

```
////////////////////////////////////
// Bomba. Po umístění chvíli tiká a pak vybouchne
objectname oBomba;
depend {oPlaceable, ZnicitelneVeVybuchu, oVybuchuje} << oBomba;
object oBomba {
      edit {InMap}
      void @MapPlaced() { // tato metoda se zavolá, když je objekt umístěn do mapy
            if (@IsGame()) Vybouchni() timed 660;
            // za 660 ms od umístění vybouchnu
      }
}


////////////////////////////////////////
// Neaktivni Bomba. Nevybuchuje sama od sebe, ale jen pokud je
// zasažena vybuchem.
objectname oNeaktivniBomba;
depend {oPlaceable, ZnicitelneVeVybuchu, oVybuchuje} << oNeaktivniBomba;
object oNeaktivniBomba {
      edit {InMap}
}


//////////////////////////////////////////////////
// Jeste priklad znicitelne a neznicitelne veci

objectname oStena; // neznicitelna
depend oPlaceable << oStena;
object oStena {
      edit {InMap}
}

objectname oKlic; // znicitelny
depend {oPlaceable, ZnicitelneVeVybuchu} << oKlic;
object oKlic {
      edit {InMap}
}
```

Práce s Mapou

Mapa je plán, do kterého umísťujeme umístitelné objekty. Umísťujeme na zvolené souřadnice, objekty se po umístění **zobrazí**. Dále objekty na mapě zaberou určité místo. Pokud je stejné místo zabráno více objekty, říkáme, že objekty **kolidují**.

Objekty můžeme umísťovat, odebírat a pohybovat s nimi po mapě. Mapy se můžeme ptát, jaké objekty kde jsou. Také můžeme umísťovat takzvané **triggery**, což jsou neviditelné objekty, které hlídají určitou oblast na mapě a, když v té oblasti něco přibude, či ubude, posílají o těchto událostech zprávy.

Objekt MAPA

Aneb první objekt nebyl první.

Systém je navržen tak, aby si samy skripty mohly spravovat objekty umístěné v mapě. (Ony to vlastně dělat musí, protože objekt mapa je povinný.) Objekt mapa musí obsahovat datovou strukturu, ve které si pamatuje umístěné objekty a musí umět odpovídat na dotazy typu „Vrať mi všechny objekty v té a té buňce“.

Skripty tak samy určují, jaká datová struktura je pro mapu vhodná, a mohou si nad mapou naprogramovat celou řadu nadstandardních vyhledávacích funkcí. Mohou prostě využít toho, že objekt mapa **ví** o všech umístěných objektech.

Objekt mapa dále říká kernelu, jaký tvar a velikost mají buňky, jak je level veliký, jestli má jen jedno patro nebo více. Bohužel v současné době jsou plně implementovány jen buňky ve tvaru čtverce, či obdélníka, jehož strany jsou zarovnány se souřadnými osami. V budoucnu počítáme i se zkosenými čtverci a s hexy. Vícepatrová grafika je už podporována lépe. Umí s ní pracovat jak Kernel, tak Grafický Engine, ale bohužel není implementována v editoru.

Požadavky na Objekt Mapa

- Objekt musí být statický, během svého konstruktoru musí inicializovat svou datovou strukturu a zavolat funkci `@RegisterMap`. (na jméně objektu mapa nezáleží)
- Objekt Mapa musí mít správně implementovány následující metody známých jmen:

```
void @MPlaceObjToMap(objptr @Object, inta @CellsArray)
```

Tuto metodu volá kernel, když přidává objekt `@Object` do mapy. V `@CellsArray`, předává kernel mapě seznam buněk, do kterých objekt zasahuje. V poli jsou předávány souřadnice buněk, pro každou buňku je tam trojice hodnot: x, y, z.

```
void @MRemoveObjFromMap(objptr @Object, inta @CellsArray)
```

Tuto metodu volá kernel, když oznamuje mapě, oby si objekt odebrala ze své datové struktury. V `@CellsArray` je opět seznam buněk, ve kterých se objekt nachází.

```
void @MMoveObjInMap(objptr @Object, inta @RemoveCellsArray,
inta @KeepCellsArray, inta @PlaceCellsArray)
```

Tuto metody volá kernel poté, co objektem `@Object` bylo posunuto na nějaké nové souřadnice. V `@RemoveCellsArray` jsou buňky, které objekt opustil; v `@KeepCellsArray` jsou buňky, kde objekt zůstává, a v `@PlaceCellsArray` jsou buňky, které objekt nově zabral. V budoucnu plánuji této funkci předávat i staré a nové souřadnice objektu. Tohle ale zatím implementováno není.

```
objptra @MGetObjects(inta @CellsArray, objptra @ObjectArray)
```

Tato funkce slouží ke zjišťování objektů na daných buňkách. Funkce by měla vrátit seznam všech objektů, které se vyskytují na buňkách z `@CellsArray`. Pokud je `@ObjectArray` null, funkce by měla vytvořit pole typu `objptra` a to vrátit. Jinak by funkce měla pole z `@ObjectArray` vyprázdnit, uložit do něj výsledek a vrátit `@ObjectArray`.

```
void @MResizeMap()
```

Tuto funkci volá kernel, pokud v editoru dojde ke změně velikosti mapy. Mapa by měla zrušit svou datovou strukturu. Vytvořit ji znova prázdnou a znovu se zaregistrovat. Další krok provádí kernel. Vezme všechny objekty, které byly umístěny v mapě. Pokud je stále možné umístit je do mapy, umístí je (volá

`@MPlaceObjToMap`. Objekt se o této změně nedozví, `@MapPlaced` zavolána není). Objekty, které se ocitly mimo mapu, kernel zničí.

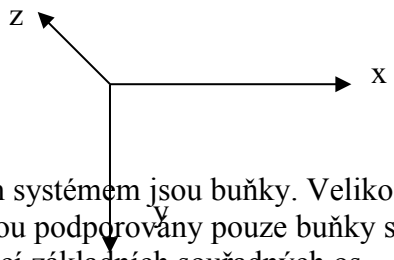
Objekt Mapa ze hry KRKAL

Příkladem implementace objektu Mapa je objekt Mapa pro hru Krkal. Tato hra má jedno patro a čtvercové buňky o velikosti 40x40 pixelů, velikost mapy je volitelná. Mapu je možné najít v souboru Games/Krkal_4F88_78B7_A01C_48AB/map_0001_000F_0001_1001.kc

Doporučuji toto mapu používat. Pokud chcete ve své hře mít mapu jinou, lze ji jednoduše naprogramovat upravením mapy pro hru Krkal.

Souřadnice

Základní systém souřadnic je trojrozměrný a je v pixelech.

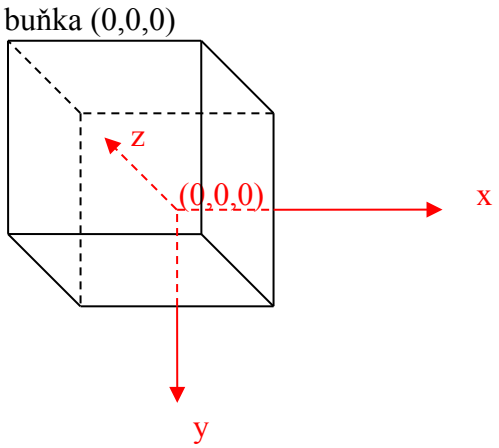


Druhým souřadným systémem jsou buňky. Velikost a tvar buněk definuje objekt Mapa. V současné době jsou podporovány pouze buňky s obdélníkovou základnou. Orientace buněk se shoduje s orientací základních souřadných os.

Buňka má svůj počátek ve středu své základny. To znamená, že pokud umístíme objekt, který zabírá právě jednu buňku, na souřadnice (40,0,0), bude zabírat na ose x pixely 20 až 59, na ose y pixely -20 až 19 a na ose z pixely 0 až 39. Za předpokladu, že máme buňky veliké 40x40x40 pixelů.

Buňka (0,0,0) je umístěna kolem pixelu (0,0,0).

Objekt z minulého příkladu přesně vyplní buňku (1,0,0). Objekty nemusíme umísťovat zarovnané na buňku. Neuděláme-li to, tak nezarovnaný objekt zabírající jednu buňku, zasáhne hned do několika buněk.



Umísťování objektů do mapy

Specifikace kolizní oblasti

Nejprve je třeba zadat, jakou oblast bude objekt zabírat a s čím bude kolidovat. Defaultní nastavení je, že objekt zabírá jednu buňku a koliduje se vším, kromě podlah.

```
char @CollisionCfg;
```

Tento atribut nastavuje typ kolize. Nejprve je třeba nakonfigurovat kolizi v horizontální směru. Máme tyto možnosti:

```
@eKCCpoint      // Objekt zabírá bod. Koliduje jen tehdy, pokud je ten bod ostře uvnitř
                // jiného objektu
@eKCConeCell    // Objekt zabírá přesně jednu buňku
@eKCCrect1      // Objekt zabírá obdélník o velikosti n x m buněk
     int @NumCellX, @NumCellY, @NumCellZ;
     // V těchto atributech je popsána velikost obdélníka. Povolené hodnoty jsou 0 až n
     // buněk. Obdélník má počátek ve středu základny své první buňky.
@eKCCcolCube    // Oblast je vymezena krychlí. Souřadnice krychle jsou v
     // pixelech, relativně vzhledem k objektu. Zadávají se dva rohy krychle.
     int @BCubeX1, @BCubeY1, @BCubeZ1;
     int @BCubeX2, @BCubeY2, @BCubeZ2;
@eKCCoutOfMap   // nebude žádná kolize, ani s mapou
```

Dále je třeba určit, jak objekt koliduje vertikálně, vzhledem k buňce. Konkrétní konstanty z obou sad se spojují operátorem or.

```
@eKCCwall       // Objekt koliduje s vnitřkem buňky
@eKCCfloor      // Objekt koliduje s podlahami buňky
@eKCCcell       // Objekt koliduje se vším. I s vnitřkem i s podlahami.
@eKCCnothing    // Objekt, nekoliduje s žádnými objekty, ale do příslušných buněk v
                // mapě umísťován je.
```

Příklad.

```
@CollisionCfg = @eKCConeCell | @eKCCwall; // Defaultní nastavení
```

Kromě oblasti je ještě možno přímo určit množiny objektů, se kterými daný objekt koliduje, či naopak nekoliduje. Tyto údaje se zadávají do atributů typu `name`: `@clzAddGr`, `@clzSubGr`. Defaultně je `@clzAddGr` nastavena na `@Everything`, a `@clzSubGr` na null.

A nakonec je tu ještě možnost si naprogramovat **kolizní funkci**. Tu kernel zavolá během testu kolize, pokud by mělo dojít ke kolizi a pokud druhý objekt patří do množiny `@clzFceGr` (defaulně nastavená na null). Kolizní funkce má tento tvar:

```
     int retor TestCollision(objptr @Object);
```

Kde `@Object` je objekt, se kterým možná koliduji. Pokud chci kolidovat, vrátím 1, pokud ne, vrátím 0.

Kolize můžu testovat funkcemi:

```
int IsObjInCollision(objptr o, int relx=0, int rely=0, int relz=0)
```

```
objptra FindCollidingObjs(objptr o, objptra ret=0, int relx=0, int rely=0,
      int relz=0)
```

Kde `o` je testovaný objekt (nemusí být umístěn v mapě). `rel?` je relativní posun objektu vůči jeho souřadnicím (v pixelech). A `ret` mohu zadat, pokud nechci, aby funkce návratové pole alokovala, ale místo toho chci použít pole `ret`. První funkce vrací 1 i v případě, že už je objekt mimo mapu. „Oblast mimo mapu koliduje se vším."

Zadání souřadnic

```
int @ObjPosX;
int @ObjPosY;
```

Před umístění do mapy zadám souřadnice v pixelech do těchto atributů známých jmen. Souřadnice mohu číst i zapisovat také pomocí těchto služeb kernelu :

```
void @ReadObjCoords(objptr o,int *x, int *y, int *z)
void @WriteObjCoords(objptr o,int x, int y, int z=0)
```

A tahle služba vrátí primární buňku objektu. (Zjistí, do které buňky patří souřadnice objektu). Funkce vrátí 1, pokud je buňka v mapě, 0 pokud je buňka mimo mapu.

```
int @FindObjCell(objptr o,int *cx, int *cy, int *cz)
```

Umístění

```
void @PlaceObjToMap(objptr o)
void @PlaceObjToMapKill(objptr o)
```

Je možné umístit objekt kamkoli do mapy. Umístění se neprovede, pokud je objekt mimo mapu. První funkce objekt umístí do mapy i v případě, že objekt koliduje. Druhá funkce nejprve pozabíjí všechny kolidující objekty a pak teprve objekt umístí.

Po umístění je objektu zavolána speciální metoda známého jména **`@MapPlaced`**. Metoda je volána jako zpráva.

Pohyby

Když je objekt umístěn v mapě, nemohu měnit jeho souřadnice, ani nastavení kolizní oblasti. (Výjimku tvoří jen nastavování kolizních množin.) Místo toho musím používat služby pro pohybování.

```
void @MoveObjTo(objptr o, int x, int y, int z=0)
void @MoveObjRel(objptr o, int rx, int ry, int rz=0)
void @InitMoveTo(objptr o, int time, int rx, int ry, int rz=0)
```

Kde `o` je pohybovaný objekt, `x`,`y`,`z` jsou absolutní souřadnice, `rx`, `ry`, `rz` jsou relativní souřadnice a `time` je čas v milisekundách po který bude plynulý pohyb trvat.

Pokud chceme s objektem hýbat plynule, je třeba použít třetí funkci. (Trhané pohybování s objektem vždy o kousek každý takt je neefektivní a nevypadá dobře.) První dvě funkce s objektem hnou okamžitě na cílové místo. Pokud zavoláme jakoukoli z funkcí v době kdy ještě běžel nějaký plynulý pohyb, tento plynulý pohyb bude zrušen.

Funkce dále informují o změnách objekt mapu a aktualizují hodnoty objektových souřadnic.

Odebrání

```
void @RemoveObjFromMap(objptr o)
```

Před odebráním je ještě objektu zavolána speciální metoda známého jména **`@MapRemoved`**. Metoda je volána jako volání.

Poznámka. Pokud umísťujeme už umístěný objekt nebo odebíráme neumístěný, nestane se nic.

Propojené objekty

Umístěné objekty mohu mezi sebou propojit. Pak, pokud pohnu jedním z objektů, ostatní objekty se pohnou také. (Ne na stejné místo, ale po stejné trajektorii. Mohu si představit, že objekty jsou spojeny neviditelnou železnou tyčí.)

```
void MvConnectObjs(objptr o1, objptr o2)
```

Funkce objekty propojí. Pokud se už objekty během propojení pohybovaly, budou se od teď pohybovat přesně tak, jak se pohybuje `o2`.

Příklad – Šmejdič

```
objectname oSmejdic;
depend oPlaceable << oSmejdic;

object oSmejdic {
     edit {InMap}

     void @MapPlaced() {
          if (@IsGame()) @MoveEnded();
     }

     void @MoveEnded() { // Tuto metodu zavolá kernel po ukončení pohybu
          int dx,dy;
          switch(@randInt(3)) {     // Náhodně si vyberu směr pohybu
               case 0: dx=40;dy=0; break;
               case 1: dx=0;dy=40; break;
               case 2: dx=0;dy=-40; break;
               case 3: dx=-40;dy=0; break;
          }
          // A jestli to místo, kam chci, není obsazené, tak se tam rozjedu
          if (!@IsObjInCollision(this,dx,dy))
               @InitMoveTo(this,330,dx,dy);
     }

}
```

Scrolling

Tím že scrollujeme herním oknem, posouváme svůj pohled na různá místa herního plánu.

Scrollovat můžeme buď přímo přes služby kernelu a nebo můžeme vytvořit a umístit do mapy speciální objekt typu `@ScrollObj`. Jak se tento objekt bude pohybovat po mapě, bude s ním

scrollovat i okno. A to tak, aby se objekt držel ve středu okna. Nejlepší je nastavit `@ScrollObj` tak, aby s ničím nekolidoval, aby byl neviditelný (nadat mu grafiku nebo nastavit do kolizní konfigurace `@eKCCinvisible` bit) a pak ho propojit s postavičkou, která se má držet ve středu okna.

```
void SetScrollCenter(int x, int y)
void WindowScroll(int relx, rely, int time)
```

Triggery

Triggery jsou speciální neviditelné objekty, které si po umístění hlídají oblast, se kterou kolidují.

Když do této oblasti přibude nový objekt, se kterým trigger koliduje, dostane trigger zprávu `@TriggerOn`. S tou jdou dva parametry: `@Object` – pointer na objekt, který trigger aktivoval a `@ObjType` – typ toho objektu. (Typ je zde proto, protože z mrtvého objektu se typ získat nedá)

Když naopak kolidující objekt opustí kolizní oblast, posílá kernel zprávu `@TriggerOff` se stejnými parametry.

Zprávy `@TriggerOn` dostává trigger i pokud byl právě umístěn do mapy a už od začátku s něčím koliduje.

Obráceně to nefunguje. Žádné zprávy už nechodí, pokud odeberu trigger z mapy a on přesto ještě s něčím kolidoval.

Umístěné triggery si spravuje sám kernel. Objekt Mapa o nich neví. Jinak se s triggery pracuje úplně stejně jako se všemi ostatními umístitelnými objekty, můžu konfigurovat jejich kolizní chování, umísťovat je do mapy, pohybovat s nimi, odebírat je z mapy, propojovat jejich pohyb s jinými objekty, ...

Aby se objekt stal triggerem, musím:

Nastavit do kolizní konfigurace `@eKCCtriggerBit`.

```
@CollisionCfg = @CollisionCfg | @eKCCtriggerBit;
```

Ještě jeden bit, nastavitelný do kolizní konfigurace, je pro triggery zajímavý. A to `@eKCCcenterColBit`. Ten říká, že triggery při vyhodnocování kolizí neporovnávají svojí kolizní oblast proti kolizní oblasti jiného objektu, ale porovnávají svojí kolizní oblast proti pouhým souřadnicím jiného objektu.

Dále můžu nastavit atribut `@MsgRedirect` na objekt A. To způsobí, že zprávy `@TriggerOn` a `@TriggerOff` se nebudou posílat triggeru, ale objektu A.

Příklad – Mina

Tento příklad je pokračováním příkladu **bomba**.

```
////////////////////////////////////////////////
// TRIGGER
objectname otrigger;
object otrigger {
      int @ObjPosX, @ObjPosY;       // souřadnice
```

```
	char @CollisionCfg;		// typ kolize
	int @NumCellX, @NumCellY;	// velikost kolizního obdélníka
					// v bunkach
	name @clzAddGr, @clzSubGr;	// s čím koliduji a čím ne
	objptr @MsgRedirect;		// kam přepošlu @triggerOn/Off

	constructor() {
		// Nechť mám tvar obdélníka, koliduji se stěnami i s podlahami
		// a nechť jsem trigger.
		@CollisionCfg = @eKCCrect1 | @eKCCcell | @eKCCtriggerBit;
		@NumCellX = 1; @NumCellY = 1;
		@clzAddGr = @Everything; @clzSubGr = @Nothing;
		@MsgRedirect = onull;
	}

	// Metoda pro nastavení souřadnic a velikosti obdélníka v buňkách
	void decl SetPosSz(int ::X, int ::Y, int ::ncX=1, int ::ncY=1) {
		@ObjPosX = X; @ObjPosY = Y;
		@NumCellX = ncX; @NumCellY = ncY;
	}

	// Metoda pro nastavení kolizních množin a redirectu
	void decl SetClzGr(name ::AddGr=@Everything, name ::SubGr=@Nothing,
	objptr ::Redirect=onull)  {
		@clzAddGr = AddGr; @clzSubGr = SubGr;
		@MsgRedirect = Redirect;
	}
}



//////////////////////////////////////////
// Mina. Po umistění bude koukat kolem sebe na oblast 3x3 buňky
// A Když tam zahlédne něco, co ji aktivuje, tak po čase vybouchne.
objectname oMina;
voidname AktivujeMinu;
depend {oPlaceable, oVybuchuje} << oMina;
depend AktivujeMinu << {oSmejdic,oVybuch};
depend ZnicitelneVeVybuchu << {oMina, oSmejdic};

object oMina {
	edit {InMap}
	objptr triger; // Zde si budu pamatovat svůj trigger
	uconstructor() { triger = onull; }

	void @MapPlaced() {
		if (@IsGame()) {
			// vytvořím a umístím trigger
			triger = new otrigger;
			// Nechť trigger zabíra 3x3 buňky a já jsem v jeho středu
			triger->SetPosSz(::X:@ObjPosX-40,::Y:@ObjPosY-40,
			::ncX:3,::ncY:3);
			// Ať trigger zajímají jen věci, které aktivují minu
			triger->SetClzGr(::AddGr:AktivujeMinu, ::Redirect:this);
			@PlaceObjToMap(triger);
			@MvConnectObjs(triger,this); // přivážu si trigger k sobě
				// Když se mnou někdo pohne, trigger půjde se mnou.
		}
	}

	// reakce na aktivování triggeru
```

```
	void @TriggerOn() {
		// Mina byla aktivovana
		@LogDebugInfo(1,0,"Mina aktivovana!");
		Vybouchni() timed 396;
		delete triger;
		triger = onull;
	}

	// Nezapomenu odstranit trigger
	void @MapRemoved() {
		delete triger;
		triger = onull;
	}


}
```

Ladění

Za běhu skriptů může vznikat celá řada chybových hlášení. Ty nejzávažnější, ze třídy **KernelPanic**, okamžitě způsobují ukončení kernelu a tím i zastavení hry, či editace. Ostatní chyby se většinou jen logují a je na uživateli, aby z toho sám vyvodil případné důsledky.

Úplný log je možno si prohlížet po ukončení levlu a kernelu buď na úvodní obrazovce (Ctrl+L) nebo v editoru (Ctrl+W). Dále je možno sledovat chybová hlášení, jak vznikají za běhu, a to v editoru (Ctrl+W). Poznámka: V editoru se dá zapnout takzvaný GameMod, který umožní současné hraní hry a editování, což je vynikající pro ladění skriptů.

Pomocí následujících služeb lze uživatelsky vyvolat některý error a také vypsat do logu debugovací hlášku.

```
@LogUserError(int group, int info1, char *info2)
@LogDebugInfo(int num, int info1, char *info2)
```

`group` říká, do které skupiny má error patřit (`@eKEGPanicError`, `@eKEGWarning`, ...), `num` si uživatel může zvolit. Další parametry by měly obsahovat popis chyby, jsou nepovinné.

Kompilované a Interpretované skripty

Kernel umí pracovat se dvěma druhy skriptů. Interpretované skripty jsou takové, jejichž kód přeložil kompilátor do svého interního jazyka (assembleru). Když pak interpretovaný skript běží, interpret čte tento assembler, rozpoznává jeho instrukce a provádí je. Interpretované skripty jsou pomalejší, ale díky kompilátoru jsou vždy k dispozici.

Kompilované skripty přeložil kompilátor do C++ a my je pak přidali ke kódu celého projektu Krkal. Skripty jsou pak rychlejší, protože jsou prováděny přímo, nejsou interpretovány. Běžný uživatel nemůže zasahovat do kódu Systému Krkal a tudíž nemůže vytvářet kompilované skripty.

Myšlenka je taková, že stabilní základ skriptované hry přeložíme do kompilovaných skriptů, aby vše bylo rychlejší. Různá drobná vylepšení od různých uživatelů zůstanou interpretována.

Transparentnost

Když skripty programujeme, nevíme zda v budoucnu poběží interpretovaně nebo kompilovaně. A v podstatě nám to může být jedno.

Propojení Interpretovaných a Kompilovaných skriptů

Interpretovaný a kompilovaný svět je oddělený. Metody z různých světů spolu mohou komunikovat jen přes `safe` volání nebo přes služby kernelu. V jednom objektu je možno mít jak interpretované tak kompilované metody, ale pokud metody přistupují ke stejným atributům nebo pokud se vzájemně volají `direct` voláním, musí být všechny ze stejného světa (metody jsou propojené). Tyto situace hlídá kompilátor. Když propojíme interpretovanou metodu s kompilovanou, dopadne to tak, že budou obě interpretované.