ROČNÍKOVÁ PRÁCE

Obor č. 18: Informatika

Vývoj logické hry v GameMaker Studiu 2

Autor: **Martin Znamenáček**
Konzultant: **Mgr. Čestmír Mniazga**
Škola: **Gymnázium ALTIS s.r.o., Dopplerova 351, 110 00 Praha**
Kraj: **Praha**
Ročník: **5. A (kvinta), 2018/2019**

Prohlášení

Prohlašuji, že jsem svou práci vypracoval samostatně a použil jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů.

Nemám závažný důvod proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Praze dne 6. 5. 2019

Martin Znamenáček

Poděkování

Tímto bych milerád poděkoval svému konzultantovi panu Mgr. Čestmíru Mniazgovi za jeho ochotu, neustálou podporu i motivaci, díky níž jsem byl schopen produktivně pracovat a v neposlední řadě též za množství nápadů a zajímavých doplňků v programu samotném.

V programování jako takovém jsem měl neustálou možnost odborné pomoci mimo jiné i od samotných administrátorů, rovněž pak od nadšených fanoušků platformy GameMaker, kteří mi pomohli vyřešit nemalé množství problémů ve hře jako takové, a proto bych ještě rád poděkoval pěti uživatelům, a to jsou jmenovitě:

HeartBeast, Shaun Spalding, FriendlyCosmonaut, Jonti Sparrow a TheouAegis.

Anotace

Ve své práci jsem se zabýval vývojem logické hry, která má za úkol hráče nejen zabavit, ale i rozvinout jeho/její myšlení, hlavně ve strategii a kombinatorice.

K vytvoření výsledného programu jsem využil herní engine GameMaker Studio 2, jakožto platformu od společnosti YoYo Games.

Cílovým výsledkem mé práce byla funkční hra, dále pak výsledky testování rozhraní GameMaker Studia 2. Jedním z hlavních cílů práce též bylo užití dosavadních znalostí o programování, rovněž pak nabití nových zkušeností a praxe v daném oboru, jež by mohly být přínosnými body pro budoucí zaměstnání.

Klíčová slova

objektově orientované programování; GameMaker Studio 2; C#; GML; logická hra

OBSAH

1 ÚVOD / NÁPAD

Již od malička jsem byl fascinován ledajakými hrami, a to už na tehdejším zařízení playstation portable či na mém prvním počítači s Windows 98, který jsem v té době dostal. Sotva jsem začal chodit do školy, napadla mě myšlenka, ze které se postupem času stal i můj sen. Chtěl jsem totiž vytvořit hru. Neboť jsem byl v té době ještě moc mladý a nezkušený na to, abych si tento sen splnil, rozhodl jsem se při příležitosti vybírání tématu této ročníkové práce splnit můj sen a též jej pokud možno co nejlépe zrealizovat.

V této práci jsem se proto zabýval programováním počítačové hry, do které jsem se snažil implementovat jednoduchý příběh, který tak slouží jako vodítko pro hráče, poněvadž jej žene kupředu a objasňuje jisté nereálné kreace. Mimo to jsem usiloval o vytvoření specifických mechanik a funkcí, které tak hráči dávají volnost a mě, jako programátorovi, poskytují nástroje k vytvoření všemožných hádanek a logických situací. Zároveň jsem u hry zvolil co nejjednodušší ovládání, avšak pro větší rovnováhu jsem u hry zvýšil obtížnost.

2 CÍLE PRÁCE

- Obeznámení čtenáře s prostředím, které bylo využito pro tvorbu nejen samotného programu, ale i grafiky či hudby
- Vysvětlení stěžejních skriptů, funkcí a mechanik, které byly použity nebo vytvořeny v průběhu programování
- Vysvětlení a představení konkrétních řešení specifických situací a problémů
- Testování výkonnostních rozdílů běžně používaných částí kódu spolu s objasněním, či domněnkami
- Vymyšlení konkrétního schématu, které by bylo návodem pro chronologii, ale i samotné tvoření her

3 CÍLE PROJEKTU

- Zlepšení mých dosavadních znalostí v programovacím jazyce C#
- Nasbírání nových zkušeností v oborech tvorby grafiky a animace
- Naprogramování hry, která by uživatele nejen zabavila, ale i přinutila myslet
- Vytvoření hry s možností dalších vylepšení a přidání nového obsahu v budoucnosti

4 PROSTŘEDÍ

Svou práci jsem započal na začátku října minulého roku (2018), kdy jsem se po několika letech po předchozích zkušenostech s programováním v jazyce C# rozhodl pro program GameMaker, tentokráte již v nové verzi, nazývané GameMaker Studio 2. Na trhu existuje mnoho oficiálních verzí této programovací platformy, a to od verze zkušební, která je zcela zdarma, avšak umožňuje velmi omezené množství funkcí, až po 12měsíční licence, které stojí 1 500 $. Já jsem zakoupil profesionální licenci pro vývojáře v hodnotě 100 $.

Druhým krokem byla snaha získat nástroj pro tvorbu grafiky mimo programovací prostředí GMS 2, čímž se stala online aplikace Piskel, kterou lze najít na doméně piskelapp: https://www.piskelapp.com/.

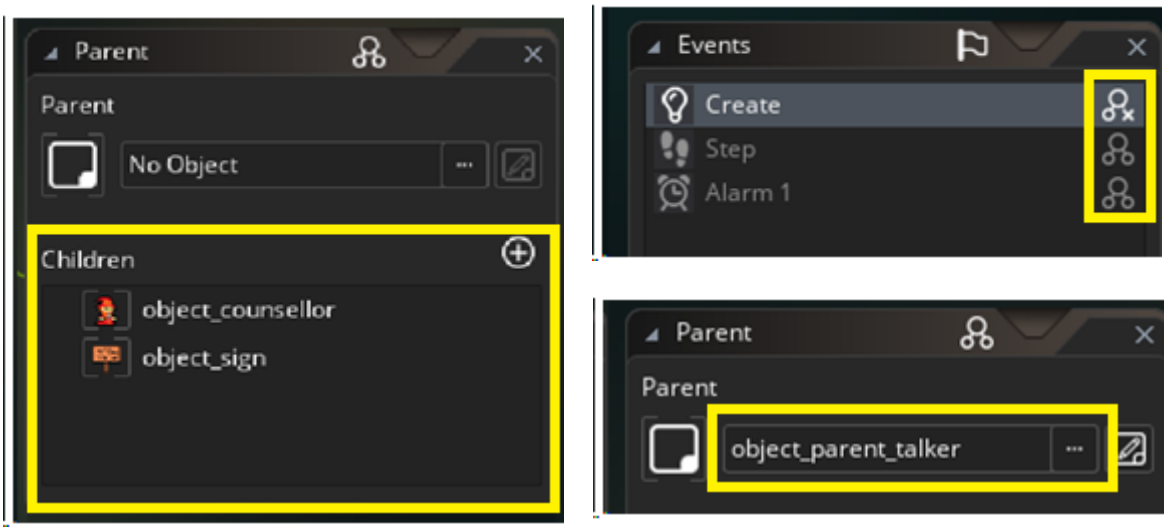
V neposlední řadě jsem čerpal inspiraci či samotnou grafiku z domény itch.io, kde lze nalézt všemožné sprity pro prakticky všechny herní žánry: https://itch.io/game-assets/free/tag-pixel-art.

5 JAZYK GML

GameMaker language či zkráceně GML je jazykem, jehož run-time system, neboli běhové prostředí běží v jazyce C++, které umožňuje implementaci softwarových knihoven, funkcí a podprogramů, dále pak ještě ladění a jednodušší optimalizaci kódu, zatímco Integrated development environment, zkráceně IDE, v češtině pak Vývojové prostředí je vytvořeno v jazyce C#, jež obsahuje editor zdrojového kódu, kompilátor, který překládá algoritmy z jazyka vyššího do jazyka nižšího, de facto z kódu napsaného programátorem vše převede na strojový kód, který naopak dokáže přečíst jedině stroj, jako třeba počítač a debugger, skrze který se program ladí, a to i za jeho běhu a kromě toho umožňuje vývojáři lepší kontrolu nad hodnotami a fungováním celého programu.

GMS 2 je zároveň specifické programovací paradigma OOP, Object-oriented programming, v češtině objektově orientované programování, které se na rozdíl od standardního procedurálního liší v umístění výkonného kódu, který je přidružen k jednotlivým prvkům, k objektům, což umožňuje hlavně ve vývoji her jednoduchost a snadnější řešení vzájemných vztahů jednotlivých entit. Mezi hlavní atributy i výhody, kterých jsem hojně využíval, patří:

- dědičnost, díky které lze vytvořit rodičovské třídy, podtřídy a potomky tříd, kde rodičovská třída zavádí hlavní funkcionalitu, která je pak potomky zcela zděděna, modifikována či potlačena, což je výborně využitelné, pokud vytváříme jednotlivé objekty, které spojuje určitá vlastnost či hodnota, a naším cílem není znovu opakování a duplikování kódu, nýbrž pozměnění určitých aspektů jednotlivých objektů, pro což jsem níže uvedl ukázku, uplatnění pro postavy, takzvaná NPC, které komunikují s hráčem, přičemž používají stejný formát, pochopitelně až na jméno a text, nechť díky funkci *event_inherited()* poupravujeme či v našem případě přepisujeme a přidáváme hodnoty pro *name* a *text*, které slouží jako nosiče textových dat



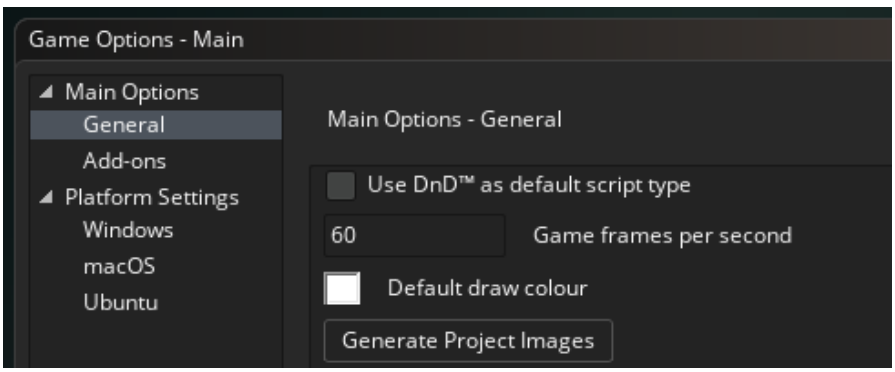

```
textbox = noone;

name = "";
text[0] = "";
text[1] = "";

yfloat = 100;
endtext = 0;
```

```
event_inherited();
name = "The Counsellor";
text[0] = "Prithee..."
text[1] = "Avaunt.. now..."
text[2] = "Wherefore has such bootless and scurvy scapegrace come to me.."
text[3] = "Wight such as ye has nought to be here for."
text[4] = "Or mayhap.."
text[5] = "Oh, yes indeed."
text[6] = "'Tis but a malapert malison, I see.";
```

- eventy, které se řadí do 14 základních kategorií, přičemž všechny mají své vlastní opodstatnění ve využití, protože se chovají zcela odlišně
  - step event umožňuje neustálé opakování segmentu kódu, který se vykonává každou sekundu v mém případě 60 krát, což určuje primární rychlost a reaktivitu samotného programu, a tak se jedná o časovou jednotku, se kterou programátor pracuje a na bázi které jsou neustále přepočítávány dané hodnoty



- create event, ve kterém jsou všem hodnotám prvotně přiřazena data, se kterými jednotlivé objekty později pracují, a proto lze kupříkladu deklarovat množství z počátku identických proměnných, které později měníme

```
35 hpmax = 25;
36 hp = hpmax;
37 hp_defined = hpmax
```

- mimo to lze ještě proměnné „recyklovat“, nebo alespoň ušetřit paměť, a to skrze funkci var, kterou lze deklarovat hodnoty i mimo, již zmíněný, create event, což je praktické hlavně při opakovaném počítání, v loopech, hlavně pak ve skriptech

```
21 var ox,oy,dir,range,object,prec,notme,dx,dy,sx,sy,distance;
22 ox = argument0;
23 oy = argument1;
24 dir = argument2;
25 range = argument3;
26 object = argument4;
27 prec = argument5;
28 notme = argument6;
29 sx = lengthdir_x(range,dir);
30 sy = lengthdir_y(range,dir);
31 dx = ox + sx;
32 dy = oy + sy;
```

- draw event, který funguje jako malíř, neboť postupuje podle kódu, a proto krok za krokem překresluje předchozí vrstvy, což může znít velmi jednoduše, avšak samotný draw event byl mým největším problémem, co se výkonu týče

6 DRAW EVENT EXPERIMENT

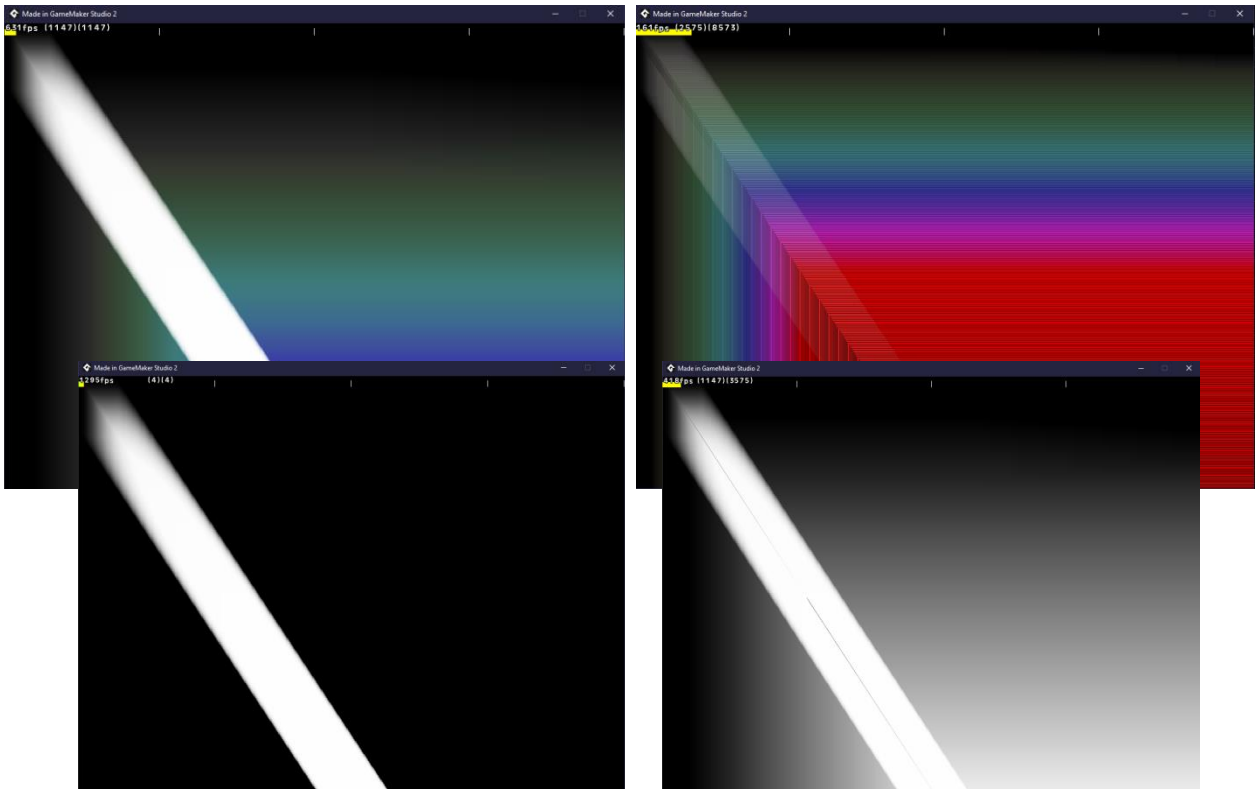
```
1  var loop = 1000;
2  var i = 0;
3  repeat(loop)
4  {
5     var col = make_color_hsv(i, i, i);
6     draw_set_color(col);
7     draw_set_alpha(i/loop);
8     draw_text(i, i, "i");
9     draw_text_color(power(i, i), i*2, "i", col, col, col, col, i);
10    draw_sprite(sprite, 0, i, i);
11    draw_rectangle(i*2, i*2, i*i, i*i, 0);
12    draw_point(i, i);
13    draw_line(i, i, i*i,i*i);
14    draw_healthbar(i, i, i*i, i*i, i, col, col, col, 0, 1, 1);
15    draw_circle(i, i, i, 0);
16    i ++;
17 }
```

Neboť jsem dlouhou dobu přesně nevěděl, jak funguje systém texture swaps a texture batches, vytvořil jsem jednoduchý looper, který 1000 krát zopakuje zvolené předdefinované funkce *draw_*. To mi umožnilo, abych je jednotlivě i skupinově podrobil testování, jehož výsledky jsem dále využíval k vyladění hry samotné a též k nalezení nejproblematičtějších funkcí, kterých by se měl každý vývojář pokud možno vyvarovat, neboť se jejich využití neobejde bez velkých dopadů na výkon hry.

Samotný test byl návodem k tomu, jak vykreslování neboli vrstevní funguje, z čehož se také daly vyvodit závěry ohledně toho, jak dané funkce pracují.
Všechny funkce, až na *draw_healthbar*, samy o sobě nezpůsobují problémy, jak si lze všimnout z prvního obrázku, kde byl vykreslen pouze 64pixelový sprite.

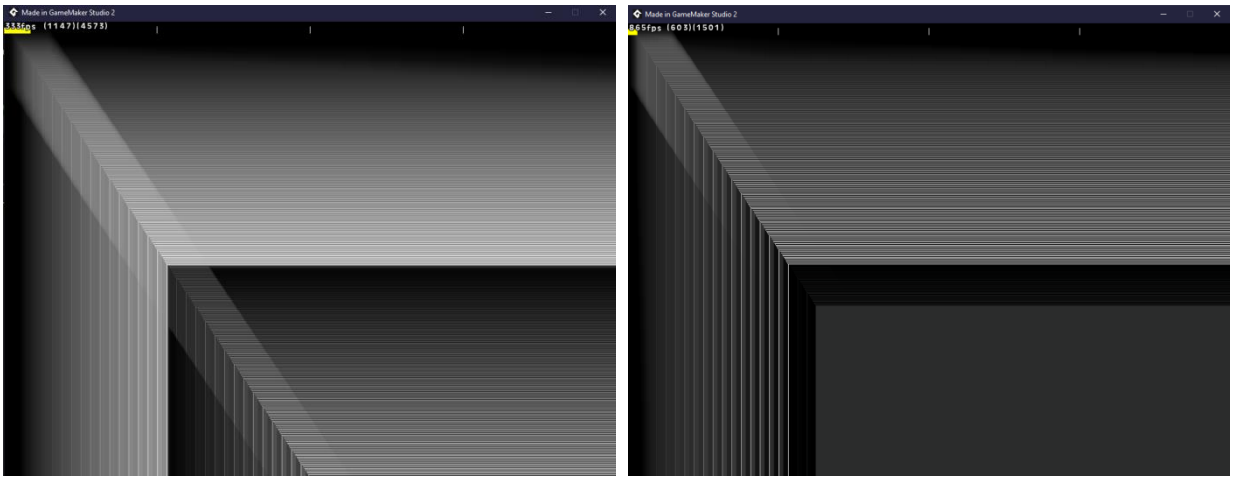
Dále pak obě textové funkce, na deklarování textu a barvy, dohromady perfektně fungují, aniž by porušily jediný vertex batch.

Avšak problémy nastaly v případě, že bylo využito více různorodých funkcí na vykreslení, což lze vidět v obrázku druhém. Protože se jednotlivé funkce překrývaly a vzájemně vyrušovaly, kvůli čemuž nemohly být všechny naráz vykresleny, došlo k ohromnému nárůstu grafických výměn, včetně porušených vertex batches.



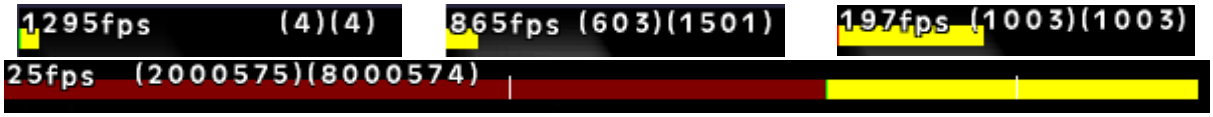
Ve třetím obrázku si lze povšimnout stejných hodnot swaps i batches, poněvadž se jedná o funkce, které se sice vzájemně neruší, avšak překrývají, a proto dochází k více než 1000 grafickým prohozením, společně se stejným množstvím přerušených texturových vrstev.

Čtvrtý obrázek je naprosto katastrofálním efektem zkombinování nesourodých funkcí a jejich špatného pořadí, kvůli čemuž dochází k enormnímu množství grafických prohozů a rozbíjení texturových vrstev. Jedná se ale o dokonalý obrázek, schéma, který poukazuje na postupné vrstvení stále se zmenšujících obdélníků, šikmé linie, kterou vytváří směsice bodů, a bílého 64pixelového čtvercového spritu, jež zdůrazňuje šikmou linii. Z obrázku je zároveň zřejmé, jak se ve velkém měřítku jednotlivé grafické prvky překrývají, kvůli čemuž dochází k extrémnímu snížení výkonu.



Pátý obrázek je znázorněním chyby, ke které často docházelo v mé hře, neboť jsem nesprávně navrstvil sprite a vestavěnou funkci pro healthbar, bar, který ukazuje poměr jistých hodnot, například zdraví. Poslední obrázek je prakticky stejný, s tím rozdílem, že jsem snížil počet opakování, hodnotu *var loop,* a tak lze vidět konečnou, svrchní vrstvu všech grafických vykreslování.

Celé testování bylo založené na dvou, již zmíněných, hodnotách, jejichž zobrazení umožňuje *show_debug_overlay(true)*, přičemž jejich vzájemnou kompatibilitou, stabilitou a funkčností byla velikost těchto dvou čísel, která by v nelepším případě, tedy pro nejvyšší výkon, měla být obě stejná a pokud možno u slabších grafických karet, třeba u mobilů, jednociferná, u počítačů se silnou grafickou kartou ideálně do 30. V levé závorce je znázorněna okamžitá hodnota texture swaps, zatímco vlevo je okamžitá hodnota vertex batches.



Pro ještě lepší znázornění dopadů na výkon je barevný obdélník, který poukazuje na celkovou náročnost programu, a to svou délkou, taktéž pak na jednotlivé prvky, které proces zpomalují, v mém případě nejvíce výkonu zabíraly grafické procesy pro vykreslení, znázorněné žlutou barvou, avšak v posledním případě bylo vykreslování natolik náročné, že program nestihl vykonat veškerý kód v daném step eventu, a proto došlo ke zbrzdění jeho obnovovací rychlosti, viz červený proužek, aby byly všechny funkce splněny, a tak byl program extrémně zpomalen, čímž ztratil reaktivitu, což by u hry znamenalo fatální selhání ovládání, a proto nehratelnost

7 DRAW EVENT FIX

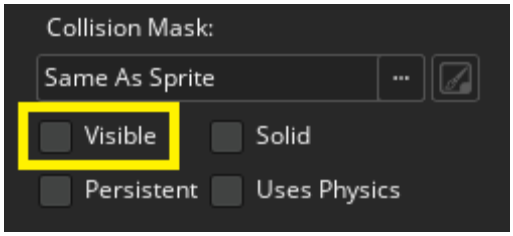
Co se vertex batches týče, jedná se o data, která jsou posílána přímo do grafického zařízení, které je zodpovědné za renderování obrazu. Obsahem dat je zpravidla pozice, barva, či kupříkladu normála, která se dostávají rovnou na video kartu, namísto toho, aby se ukládala v systému.

Veškerá grafika v GameMakeru je rozdělena do tzv. texture groups, texturových skupin, které mají své vlastní texture pages, texturové stránky, které si lze představit jako stránky se stranami s velikostí mocnin čísla 2, obvykle se tedy jedná o obdélník s velikostí 1028x512 pixelů či o čtverec, třebas s rozpětím 2048x2048 pixelů, s maximem 8192x8192 pixelů. Na tyto stránky se de facto rozdělí všechny sprity z dané texturové skupiny, včetně jednotlivých subimages, tedy sérii spritů, které vytváří animaci, za předpokladu, že velikost spritu nepřesahuje maximum texturové stránky, což by vedlo ke zmenšení spritu v poměru 1:2, neboť sprity nikdy nemohou být rozděleny na menší kousky, a proto může dojít k plýtvání plochou prázdnými místy mezi sprity, které se do texturové stránky nevešly.

Za předpokladu, že bychom měli texturovou stránku v dané texturové skupině o velikosti 1024x1024, vešlo by se do ní právě 64 128x128 spritů. Pokud bychom ale spritů měli 65, už by se do dané texturové stránky nevešly, a proto bychom při vykreslování 65. spritu zaznamenaly nárůst v texture swaps, protože by GameMaker musel změnit texturovou stránku, uložit ji do paměti, a až pak by se opět mohl vrátit na původní texturovou stránku. Zároveň by na druhé texturové stránce zbylo 1 032 192 nevyužitých pixelů, které by i tak musely být načteny.

Pořadí těchto úkonů je též velmi důležité, a proto je zapotřebí předem vytvořit skupiny, které je nutné vykreslovat neustále, to je například grafika hráče. Ideálně by se jednotlivé sprity měly dělit kupříkladu podle levlů, úrovní, aby se předešlo zbytečnému plnění texturových stránek, jejichž plocha by mohla být využita právě používanými sprity. Již zmíněné rozvržení je pro výkon zcela klíčové, neboť se v mnoha případech veškerá grafika na jednu texturovou stránku nevejde, a tak k texturovým prohozením dojít musí, avšak nutný počet těchto prohození ovlivnit lze, a to logickým sledem vykreslování. Za předpokladu, že chceme ve hře zobrazit kolečko a 2 čtverce, a tyto 2 odlišné sprity se rozdělily na rozličné texturové stránky, přičemž my chceme co nejvíce ušetřit výkon, vykreslíme nejprve kolečko, na straně A, a pak dvakrát čtverec, na straně B, címž dojdeme ke schématu A, B, B, a zaznamenali bychom pouze 2 swapy. Pokud bychom ale nejprve vykreslili jeden čtverec, pak kolečko a posléze poslední, druhý, čtverec, naše schéma by vypadalo jako B, A, B, čímž bychom namísto 2 způsobili hned 3 swapy, neboť by GameMaker musel z B stránky vykreslit čtverec, přejít ke stránce A, a závěrečně, znovu, se vrátit na stránku B.

Pokud máme texturových skupin a stránek stále moc, můžeme výkon i tak o něco zlepšit, a to tím, že problematické objekty udělám neviditelnými, a kdekoliv v místnosti umístíme objekt, který tyto problematické objekty díky kódu v eventu draw vykreslí jako první, aby GameMaker opakovaně v odlišných intervalech dále nemusel hledat jednotlivé sprity v texturových stránkách.




```
40 with(object0)
41 {
42     draw_self();
43 }
```

```
45 with (object1)
46 {
47     draw_self();
48 }
```

Nutno zmínit, že se tento problém nedá vyřešit skrze netexturové objekty, které bychom pomocí vrstvy tileset „přetřeli“, se tileset, stejně jako background, pozadí, řadí do texturových skupin.

V neposlední řadě se dá mnoha chybám v texture swaps a vertex batches předejít nepoužíváním jistých problematických *draw_* funkcí.

Z testování programu jsem zjistil, že použití funkce *draw_set_color* a *draw_set_alpha()* nemá žádný efekt na výkon.

Malé výkonnostní rozdíly může v přiměřeném měřítku způsobovat funkce *draw_text()* a *draw_text_color()*, přičemž jsem byl překvapen, že na výkonnost má vliv i samotná délka textu.

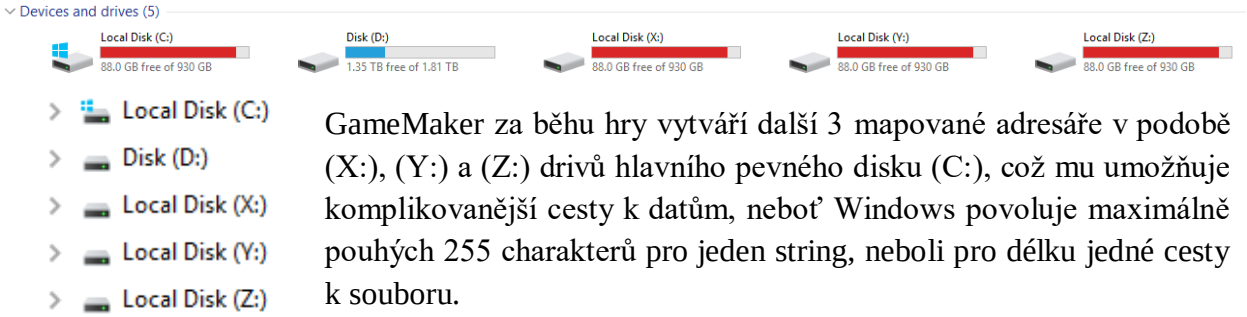
Velké problémy může způsobovat vykreslování obzvláště velkých spritů či, jak již bylo zmíněno, mnoho spritů z odlišných texturových stránek, a to funkcí *draw_sprite()*.

Negativním faktorem, kterého by se měl každý programátor v GameMakeru vyvarovat jsou funkce, které vykreslují tvary jako úsečky, obdélníky, kruhy či body - *draw_line()*, *draw_rectangle()*, *draw_circle()*, *draw_point()*.

Zcela nejhůře vyšla funkce na vykreslování baru, která je využívána v téměř všech hrách, avšak z draw eventu je, leč praktická, ale výkonnostně úplně nejhorší *draw_healthbar()*, poněvadž poruší vertex batch hned několikrát, což může být při rozsáhlém použití přímo katastrofické.

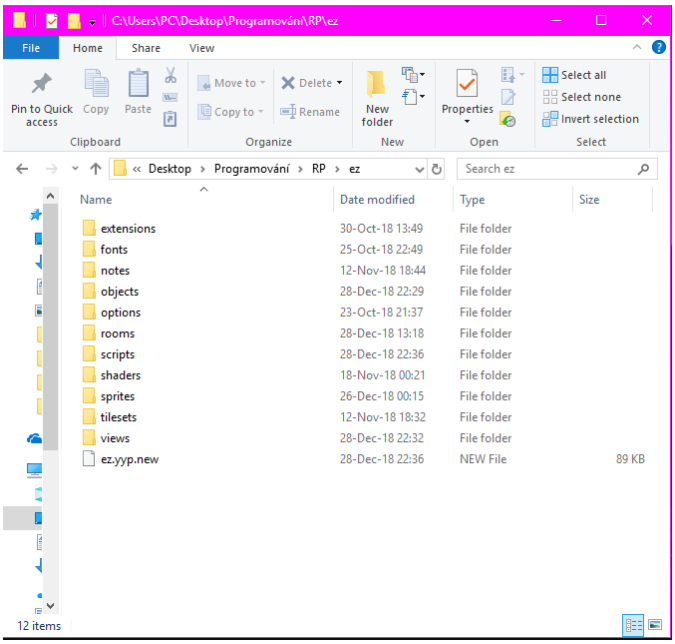
8 SMAZÁNÍ

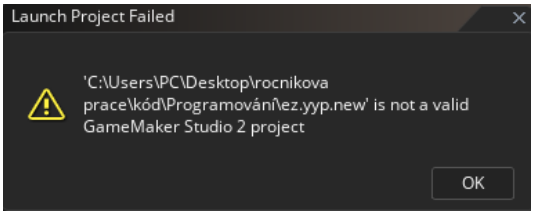
Při programování jsem mimo jiné narazil na chybu, se kterou si ani samotná GameMaker podpora nebyla poradit.



GameMaker za běhu hry vytváří další 3 mapované adresáře v podobě (X:), (Y:) a (Z:) drivů hlavního pevného disku (C:), což mu umožňuje komplikovanější cesty k datům, neboť Windows povoluje maximálně pouhých 255 charakterů pro jeden string, neboli pro délku jedné cesty k souboru.

Namísto klasického GameMaker Studio 2 project souboru (.yyp) se pravděpodobně právě při deaktivování 3 zmíněných úložišť X, Y, Z stala chyba, která měla za následek přepsání předchozího souboru na .new soubor, jakožto na typ souboru, který v sobě uchovává informace ohledně aplikací a programů. Podle samotných administrátorů YoYo Games podpory se jednalo o chybu automatického vytvoření zálohy, backup souboru, která se měla po vypnutí programu přeměnit zpět na projekt soubor (.yyp), k čemuž ale z neznámých důvodů nedošlo.





Byl jsem též obeznámen s informací, že se s podobnou chybou nikdo z komunity ani vývojářů doposud nesetkal, a proto pro ni ještě nebyla vytvořena oprava či možnost se této vadě vyvarovat.

Jediným řešením, které jsem osobně našel, bylo znovuvytvoření všech souborů, které jsem od posledního ukládání vytvořil. Naštěstí jsem veškerý kód nemusel znovu programovat, neboť se vše dalo s trochou cviku rozluštit zapomocí zápisníku, ve kterém bylo možné majoritu souborů otevřít.

9 SKRIPTY

9..1 Bar

Do hry jsem implementoval mnoho nepřátel, a proto bylo nutné zajistit vykreslení jejich životů a dalších hodnot, což bylo rovněž zapotřebí i u samotného hráče. Po již zmíněném testování jsem zcela vyřadil využití implementované funkce *draw_healthbar()*, protože by její použití při každém vykreslení, de facto tedy u každého objektu s touto funkcí, způsobilo 2 texturové prohození (swaps) a 5 vertex batches. Pro příklad bych v případě pouze 4 nepřátel nashromáždil 10 swaps a 20 batches, společně s hráčem pak dalších 6 swaps a 15 batches, a

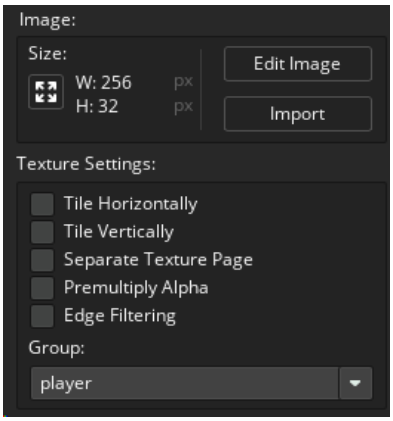
proto by mi v součtu tato funkce nakupila 16 swaps a 35 batches, díky čemuž by už hra jako taková byla neovladatelná.

Pro tento problém jsem vytvořil vlastí skript, *drawbar()*, který se jeví zcela totožný jako ostatní implementované funkce. Skrze něj jsem schopen vykreslit nekonečně mnoho barů, ukazatelů, které vypadají zcela stejně, jako ty, které jsou vykreslené skrze *draw_healthbar()*, avšak co se výkonu týče, použijí pouze 1 texturovou stránku, a tudíž 1 swap a 1 batch, v případě pohyblivých objektů, tedy na příklad u samotného hráče dojde ještě k porušení 1 vertex batch, neboť není využíváno standardního *draw* eventu, avšak jeho podtřídy, *draw gui*, která nebere v potaz rotaci, ani scale (zvětšení/zmenšení) dané instance, a proto umožňuje plynulý chod na bázi pohyblivé x a y souřadnice kamery, viewportu.

Nahoře si lze povšimnout porovnání obou výsledných produktů. Implementované funkce, *draw_healthbar()*, tvoří horní bar, zatímco mnou vytvořený skript vytváří dolní bar.

Povedlo se mi tedy nejen docílit prakticky nulového deficitu ve výkonu, ale zároveň jsem vytvořil dokonalou repliku vestavěné funkce.

Zprvu jsem si vytvořil 2 sprity, které jsem zařadil do speciální texturové stránky *player*.



Tyto sprity jsem posléze využil k vykreslení barů. Jednalo se o obyčejné obdélníky, které jsem udělal dostatečně vysoké i široké, abych se nesetkal s chybou, která muže nastat pouze za předpokladu že námi zvolená šíře přesahuje sprite, a tak se pravá či dolní chybějící část nedokáže vykreslit.

Samotný skript pak vypadal takto:

```
/// @param koeficient
/// @param thickness
/// @param length
/// @param yposition
/// @param uppersprite
/// @param backgroundsprite
/// @param background
/// @param player
/// @param playergapx
/// @param playergapy

var varK = argument0;
var thickness = argument1;
var length = argument2;
var sidegap = floor((sprite_width - argument2)/2);
var ypos = y + argument3;
var uppersprite = argument4;
var backsprite = argument5;
var drawback = argument6;
var player = argument7;
var playergapx = argument8;
var playergapy = argument9;

if player
{

    var vx = camera_get_view_x(view_camera[0]);
    var vy = camera_get_view_y(view_camera[0]);
    if drawback draw_sprite_part(backsprite, image_index, 0, 0, length, thickness,
    vx + playergapx, vy + playergapy);
    draw_sprite_part(uppersprite, image_index, 0, 0, length*varK, thickness,
    vx + playergapx, vy + playergapy);
}
else
{
    if drawback draw_sprite_part(backsprite, image_index, 0, 0, length, thickness,
    x - sprite_width/2 + sidegap, ypos);
    draw_sprite_part(uppersprite, image_index, 0, 0, length*varK, thickness,
    x - sprite_width/2 + sidegap, ypos);
}
```

a je založen na vykreslení určitého segmentu spritu, za pomoci funkce *draw_sprite_part*. Obsahuje mnoho parametrů, poněvadž jsem se jej snažil naprogramovat co nejdůmyslněji, tak abych pokud možno obsáhl všechny budoucí požadavky a skript tak později nemusel předělávat.

Nultým parametrem je koeficient, tedy poměr mezi okamžitou hodnotou a maximální teoretickou hodnotou, na základě čehož je určována procentuální část vykresleného baru.

První parametr určuje výšku vykrojení spritu, a tudíž tloušťku výsledného baru, spolu s ním pak druhý parametr, délka, určuje šířku vykrojení spritu, a tak výslednou délku baru. Hodnota délky je mimo jiné využita pro zarovnání baru na střed vůči spritu objektu. V proměnné *sidegap* totiž dojde k odečtení vybrané délky od šířky spritu objektu, jež je posléze rozpůlena a zbavena desetinného čísla pomocí funkce *floor()*. V samotném kódu je pak důležitá pro souřadnici x, tedy pro odsazení, které zajišťuje, že je počátek baru vzdálen polovinu délky od středu spritu objektu.

Třetí argument, proměnná *ypos*, je rozdílem mezi okamžitou y souřadnicí objektu a vykreslovaného baru a není využívána u hráče.

Do čtvrtého a pátého parametru je vložen název spritu s příslušnou barvou plochy, jež byl předem vytvořen.

Šestý argument, *drawback*, je switchem, přepínačem, mezi zapnutým/vypnutým vykreslováním pozadí.

V neposlední řadě jsem do skriptu implementoval 3 další argumenty, které jsou používány ryze hráčem, a to na vykreslení HUDu, heads-up displaye, neboli hráčova rozhraní, které obsahuje nejdůležitější informace o stavu. Sedmý argument je opět switchem, který umožnuje zapnutí funkce vykreslování pro hráče. Poslední 2 argumenty pak byly využity k nastavení mezery mezi x a y souřadnicí baru vůči levému hornímu rohu obrazovky.

Samotné využití v mých parent objektech nepřátel pak vypadalo například takto:

```
60 drawbar(hp/hpmax, 16, sprite_width, sprite_height + 8, sprite_health, sprite_healthback, 1, 0, 0, 0);
```

a u hráče:

```
61 drawbar(hp/hpmax, 25, round(hpmax/1.15), 0, sprite_health, sprite_healthback, 1, 1, 32, 16);
```

9..2 Pravděpodobnost

Do hry jsem taktéž potřeboval implementovat mnoho náhodných prvků, jako třeba rozličný počet zkušeností, životů či například kritické útoky. Abych mohl všechny zmíněné komponenty efektivně předpovídat a ovlivnit tak jejich množství, vytvořil jsem pravděpodobnostní skript. Efektivitu daného skriptu jsem otestoval generováním dvou hodnot, a to jedniček pro pravdu a nul pro nepravdu. Z milionů hodnot za vteřinu jsem vytvořil aritmetický průměr, na bázi kterého jsem poupravil znaménka, respektive zanedbal rovnost, tedy namísto <= jsem využil <.

```
1 /// @param percent
2
3 var chance = random(100);
4 if chance < argument0
5 {
6     return true;
7 }
```

Proměnná *chance* náhodně vygeneruje reálné číslo od 0 do 99. Ta je posléze porovnána s parametrem, který jsme nastavili, a pokud je podmínka pravdivá, skript je taktéž pravdivý a naopak.

Pokud bych chtěl například stanovit 25% pravděpodobnost, že hráč získá 1 život, mohl bych to velmi jednoduše zapsat takto:

```
102 if probability(25) hp ++;
```

Například by tedy pro náhodně vygenerovaná čísla 15,37; 67,2378; 43,211 a 98,1 byl skript pravdivý právě v jednom ze 4 čísel, a tak by splňoval 25% pravděpodobnost.

Důležitou součástí funkčnosti tohoto skriptu, bez které by program vždy vygeneroval stejná čísla je funkce *randomise()*, díky níž se inicializační seed programu, unsigned 32bit integer, číslo od −2 147 483 648 do 2 147 483 647, před každým zapnutím vygeneruje odlišný.

10 AI

V rámci vytváření určité obtížnosti jsem potřeboval navrhnout funkční a nezcela předvídatelný systém, pomyslnou umělou inteligenci, která by reagovala na hráčem vykonané akce a předvídala potenciální chování.

Celý systém je založen na podmínkách, které například zjišťují, zda má hráč málo životů, a tak by se vyplatilo zaútočit, či zda se například nenachází ve výhodné pozici, a tak by se spíše vyplatilo pohnout směrem k němu, a tím hráče zahnat do rohu, kde je pak zablokován a odkázán pouze na útočení, které v takové situaci není efektivní.

Jednotlivé podmínky se u každého typu nepřítele liší, a to podle jejich individuálních vlastností.

Samotné splnění podmínky nutně neznamená její vykonání, ale markantní nárůst pravděpodobnosti, která je později zohledněna ve skriptu *probability ()*. Tímto způsobem se v určitých situacích může nepřítel zachovat zcela nepředpokládatelně a zmařit hráčovy plány.

Nejdůležitější součástí tohoto systému bylo definovat, který nepřítel bude v daný moment reagovat, aby se zamezilo splňování již neplatných podmínek či v případě 2 a více nepřátel duplikování akcí.

```
18 i = 0;
19 defid = 1;
20 global.ides = 0;
21 global.ns = instance_number(object_enemyslime);
22 global.IDEs = 0;
23 global.defineid = 0;
```

```
105 if defid
106 {
107     defid = false;
108     global.ides += id;
109 }
110
111 global.IDEs = global.ides/global.ns - (global.ns + 1)/2;
112
113 if i > global.ns
114 {
115     i = 0;
116 }
117 i ++;
118
119 var ID = global.IDEs + i;
```

Každá instance objektu má své vlastní *id*, několikamístné číslo, které jej zastupuje. Například pro 3 stejné nepřátele by to mohla být čísla 100004, 100001, 100002. Poslední cifra (pokud je počet < 9) určuje pořadí vytvoření dané instance objektu, přičemž její maximální hodnota je celkový počet všech instancí daného objektu. To se dá efektivně využít pro výpočet čísla 100000, které je tak zcela originální pro daný objekt. K tomuto číslu se pak vždy přičte 1, a v případě, že tato instance objektu existuje, je splněna základní podmínka pro vykonání jakékoliv akce. Každý objekt má svůj totožný systém s odlišným indexem (v tomto případě s) a nezávisle na ostatních typech nepřátel nikdy nebude vykonávat jakoukoliv akci ve stejný moment společně s jiným typem nepřítele.

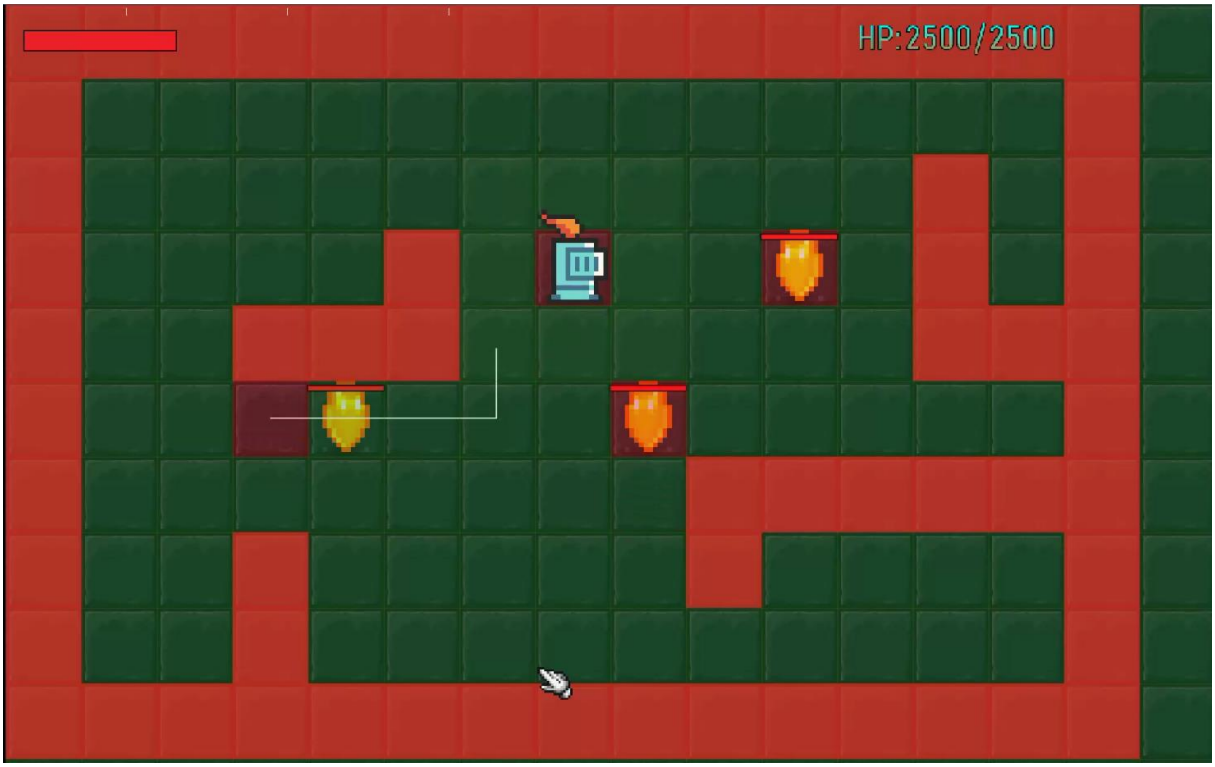
Nejdůležitější částí bylo vyřešení pohybu, které funguje na základě tzv. gridu.

	0	1	2	3	4	5	6	7	8	9
0	4	3	6	4	4	8	2	6	5	1
1	5	3	7	0	4	6	8	1	4	8
2	0	4	5	1	4	0	1	1	7	5
3	4	8	2	9	1	2	4	3	8	6
4	2	2	0	2	2	6	9	3	2	0
5	5	1	1	8	5	7	3	3	0	5
6	4	7	7	0	2	7	4	6	4	0
7	3	3	3	8	9	4	8	7	4	7
8	9	3	6	1	1	0	0	6	4	3
9	2	1	6	2	4	7	2	1	8	0

Jedná se o 2dimensionální array systém, což je de facto tabulka s přesně danou šířkou a výškou buňky, stejně tak i s definovaným počtem *x* (nahoře) a *y* (vlevo) souřadnic.

Využil jsem jej též na flexibilní systém inventáře, který uchovává obdržené věci, například meč, u kterého se na jednotlivých souřadnicích uchovávají odlišné druhy informací. Pro ilustraci [0, 0] = typ, [0, 1] = název, [0, 2] = útok, [0, 3] = váha.

V případě řešení pohybu je grid daleko jednodušší. V mém případě jsou červeně vyznačena pole, kam nelze vkročit, zatímco zeleně jsou pole, kam vkročit lze.



Obrázek tak zachycuje situaci, kdy byla definována instance objektu k pohybu, díky čemuž mohla být z gridu smazána původní buňka, na které se nepřítel nacházel. Byla vypočtena nejkratší cesta (zvýrazněna bílou čarou) a díky tomu se mohl nepřítel pohnout o jedno pole vpravo.

```
var ID = global.IDEs + i;

global.grid = mp_grid_create(0, 0, room_width div 64, room_height div 64, 64, 64);
mp_grid_clear_all(global.grid);
mp_grid_add_instances(global.grid, object_obstruction, true);
mp_grid_add_instances(global.grid, object_destructible, true);
mp_grid_add_instances(global.grid, object_player, true);
path = path_add();

if instance_number(object_flooraround) != 0 && !place_meeting(x, y, object_playerproximity)
{
    var goal = instance_nearest(x, y, object_flooraround);
}
else
{
    var goal = false;
}
```

```
19 if action && id == ID && goal != false
20 {
21     mp_grid_clear_cell(global.grid, x div 64, y div 64);
22     if mp_grid_path(global.grid, path, x, y, goal.x, goal.y, false)
23     {
24         path_start(path, 64, path_action_stop, false);
25         action = false;
26     }
27 }
28 else
29 {
30     path_end();
31 }
```

11 EXPERIMENTY

Přestože v současné době již existuje mnoho návodů a pouček pro začínající programátory v GameMakeru, narazil jsem na mnoho zajímavých a prakticky neprobíraných problémů, které ovlivňují do jisté míry celkový výkon ve hře. Všechny své otázky jsem se pak v rámci této práce snažil objasnit, a to ničím jiným, než experimentováním a testováním.

PODMÍNKY IF, ELSE IF A ELSE

Podmínky jsou v každém programovacím jazyce zcela klíčové, a GameMaker není výjimkou. Snad v každém objektu lze nalézt alespoň jednu. Jedná se tedy o stěžejní součást programovacího kódu. Podmínkami jsou klíčová slova *if, else if*, a *else*. Přestože lze stejných cílů vždy dosáhnout za použití pouze jedné z uvedených podmínek, je potřeba vždy zvolit tu, která bude v dané situaci nejvýkonnější. Protože jsem k tomuto problému nenašel na internetu data, vytvořil jsem si jednoduchý looper, na kterém jsem všechny podmínky podrobil testu.

Hodnoty, ze kterých jsem vyvozoval úsudky, jsem zaznamenával pomocí vestavěného debuggeru, který mi umožňoval všechny hodnoty sledovat, a též pozorovat rychlost vykonávání kódu jako takového. Vytvořil jsem si 3 objekty, každý pro jednu podmínku.

```
1  var loop = 1000000;
2  var number = random(loop);
3
4  repeat loop
5  {
6      if number >= loop/2
7      {
8          ifs ++;
9      }
10
11     if number < loop/2
12     {
13         ifs ++;
14     }
15 }
```

```
4  repeat loop
5  {
6     if number >= loop/2
7     {
8        elseifs ++;
9     }
10    else if number < loop/2
11    {
12       elseifs ++;
13    }
14 }
```

```
4  repeat loop
5  {
6     if number >= loop/2
7     {
8        elses ++;
9     }
10    else
11    {
12       elses ++;
13    }
14 }
```

Všechny tři objektu fungují jako repeatery, které kód vykonávají 1 000 000 krát za 16,6 ms. Přičemž první objekt je sestaven ze dvou *if* podmínek, zatímco u druhého objektu byla využita podmínka kromě podmínky *if* ještě *else if* , která byla u třetího objektu nahrazena podmínkou *else*.

Name	Call Count	Time (Ms)	Step %
+ object_if (Step)	1000	311824	39.699
+ object_elseif (Step)	1000	261710	33.319
+ object_else (Step)	1000	211832	26.969

Z přiložených hodnot je zcela patrné, že nejméně času, a tedy výkonu zabírá podmínka *else*, neboť namísto rozhodování a porovnávání hodnot GameMaker automaticky vyhodnotí sekundární výstup, nehledě na hodnoty jako takové.

Podmínka *else if* je patrně o něco méně výkonná nežli pouhé *else*, avšak je velmi efektivně využitelná v případě mnoha možných výsledků, u kterých by byla pouhá *else* podmínka zcela irelevantní.

Logicky vyšla podmínka *if* zcela nejhůře, neb GameMaker musí každou *if* podmínku kontrolovat zvlášť, zatímco již zmíněné *else* podmínky pomyslně přeskakují nepravdivé podmínky, a jakmile jednu z nich shledají pravdivou, zcela ignorují podmínky následující.

NESTING PODMÍNEK

Takzvaný nesting v programování značí mnoho podmínek, které jsou namísto sjednocení do jedné rozděleny do mnoha bloků. Pro lidské oko jsou daleko přehlednější, avšak mnohým připadají výkonnostně náročnější, přičemž pravdou je pravý opak.

```
if arg1 && !arg2 && !arg3 && !arg4 && arg5
{
   result ++;
}
```

```
if arg1
{
   if !arg2
   {
      if !arg3
      {
         if !arg4
         {
            if arg5
            {
               result ++;
            }
         }
      }
   }
}
```

Jediná nested podmínka mi po mnohých testech vždy vyšla daleko výkonnější, standardně 5 až 9krát rychlejší nežli podmínka rozepsaná. Sice se jedná pouze o rozdíly v desítkách milisekund, avšak v případě mého programu, kde jich je circa 200, by taková změna mohla znamenat mnoho zbytečně promeškaných sekund běhu.

Name	Call Count	Time (Ms)	Step %
object_single (Step)	11107	63	6.264
object_nested (Step)	11107	7	0.696

PODMÍNKA V BLOKU A ŘÁDKU

Dalším mým zájmem, navázav na předešlý pokus, bylo porovnat podmínku zapsanou v bloku se zkrácenou verzí v 1 řádku, neboť tuto verzi používá majorita programátorů.

```
if arg result ++;
```

```
if arg
{
   result ++;
}
```

Stejně jako u předchozího testu jsem byl překvapen, když jsem zjistil, že je podmínka rozepsaná do bloku tentorkráte 8 až 9krát výkonnější.

Name	Call Count	Time (Ms)	Step %
object_oneline (Step)	42233	246	5.368
object_block (Step)	42233	29	0.633

ZÁPIS LOGICKÝCH OPERACÍ

Logické operace se v GameMakeru dají zapsat dvěma způsoby. Oba zápisy by tudíž měly být zcela identické, avšak po otestování jsem zjistil, že tomu tak není.

Konjunkci lze značit pomocí symbolu && či slova *and*, podobně jako lze disjunkci zapsat || nebo *or*.

Disjunkce fungovala tak jak jsem předpokládal, zcela totožně v obou případech zápisu. Zatímco konjunkce zapsána jako && způsobovala oproti slovu *and* patrný deficit, a to až 10násobný rozdíl.

Name	Call Count	Time (Ms)	Step %
object_and (Step)	5650	29	5.357
object_andSIGN (Step)	5650	3	0.554
object_or (Step)	5650	2	0.369
object_orSIGN (Step)	5650	2	0.369

Pokud jde o ekvivalenci, ta je zapisována == a její negace pak *!=*. V tomto případě bylo mou otázkou, zda je rychlejší (pokud to tak lze zapsat) ekvivalence non výroku, či znegovaná ekvivalence výroku.

De facto rozdíl mezi *x != true* a *x == false*. Již znegovaný výrok je i dle testu rychlejší neboť GameMaker nemusí vytvářet negaci výroku sám, a až pak jej porovnávat.

Name	Call Count	Time (Ms)	Step %
object_nonequal (Step)	6186	34	6.07
object_equal (Step)	6186	2	0.357

KVANTITA PROMĚNNÝCH

Ve své hře jsem u mnoha objektů zcela bezmyšlenkovitě vytvářel množství proměnných, přestože jsem se na ně odkázal třebas jenom jednou, a to na začátku kódu, a dále jsem je nepoužíval. Zajímalo mě, zda samotné vytvoření a následné nevyužívání proměnných má na výkon programu vliv.

Vytvořil jsem si 2 objekty, s totožným step eventem, ve kterém jsem k jedné proměnné přičítal každý step 1. V create eventu jsem měl u prvního objektu proměnnou pouze 1, zatímco u druhého jsem jich měl právě 100.

Name	Call Count	Time (Ms)	Step %
object_valuesmany (Step)	2514	11	5.001
object_valuesless (Step)	2514	0	0

U prvního objektu jsem zaznamenal téměř nulové poklesy ve výkonnosti, zatímco, jak jsem předpokládal, objekt se zbytečnými, ač nevyužívanými hodnotami program mírně ovlivňoval.

Po dalších dlouhodobějších měřeních jsem zaznamenal, že mých zbytečných 99 proměnných způsobovalo 4 až 5násobný deficit ve výkonnosti.

Name	Call Count	Time (Ms)	Step %
object_valuesmany (Step)	42815	50	6.605
object_valuesless (Step)	42815	11	1.453

HODNOTY PROMĚNNÝCH

Mým dalším zájmem byla myšlenka operování s velmi malými hodnotami, neb mě napadlo, že by se kupříkladu neustálé přičítání 1 000 dalo zmenšit na 0,1, a v případě potřeby práce s číslem jako takovým by se hodnota pouze 10 000krát nazpět pronásobila.

Tento problém jsem opět proměřoval repeatrem, ve kterém jsem u prvního objektu přičítal 1 ke všem 6 proměnným, avšak u druhého objektu jsem namísto 1 pracoval s miliardami.

```
8  repeat 10000
9  {
10     valuea += 5000000000;
11     valueb += 5000000000;
12     valuec += 5000000000;
13     valued += 5000000000;
14     valuee += 5000000000;
15     valuef += 5000000000;
16 }
```

Kupodivu velikost čísel měla vliv na výkon, avšak tento deficit byl natolik minimální, že by se již zmíněná myšlenka práce s relativně malými hodnotami pravděpodobně nevyplatila.

Name	Call Count	Time (Ms)	Step %
object_valuelow (Step)	10114	49898	48.891
object_valuehigh (Step)	10114	51347	50.311

KVANTITA VAR PROMĚNNÝCH

Při svém programování jsem měl pocit, že využívání proměnných *var*, namísto jejich deklarování v create eventu má pozitivní vliv na výkonnost, neboť se proměnné *var* po jejich využití pomyslně recyklují.

Experiment probíhal zcela totožně, jako u předešlých dvou testování, s rozdílem nahrazení proměnných za *var*.

Name	Call Count	Time (Ms)	Step %
object_valuesmanyvar (Step)	23747	102	4.98
object_valueslessvar (Step)	23747	11	0.537

Přestože výsledky na první pohled vypadají lepší, nežli u předešlých porovnávaných proměnných, je nutno si povšimnou, že sice proměnné *var* nezahlcují natolik pamět, a proto je poměr výkonu, a tedy i jeho efektivita zdárně příznivější, avšak kvůli samotné recyklaci je zase využíváno mnohonásobně více času na vykonání stepu.

HODNOTY VAR PROMĚNNÝCH

V opět zcela totožném experimentu, tedy za přeměnění proměnných na *var* jsem byl výsledky zcela překvapen, neboť jsem očekával zcela opačné hodnoty.

Name	Call Count	Time (Ms)	Step %
object_valuelowvar (Step)	5653	28830	50.366
object_valuehighvar (Step)	5653	27936	48.804

Name	Call Count	Time (Ms)	Step %
object_valuelowvar (Step)	22137	113501	50.395
object_valuehighvar (Step)	22137	109880	48.787

Ze zcela nevysvětlitelného důvodu mi po mnoha testováních vycházely stále ty stejné výsledky, a to, že proměnné *var*, s výrazně většími hodnotami vyly o něco méně výkonnostně náročné, nežli ty s čísly o několik řádu menšími.

TRUE NEBO 1

Již při úplném počátku mého programování jsem si nebyl jist, zda užívat klíčová slova *true, false, self, other, all, noone* nebo číslice *1, 0, -1, -2, -3,* respektive *-4.*

Jak i test potvrdil, *true* je slovo, a tak je s ním pracováno jako s jiným datovým typem, který tak buďto zabírá více paměti, nebo je převeden na číslici *1*, což zase nepatrně zpomalí proces, neb se jedná o zbytečný krok navíc.

Name	Call Count	Time (Ms)	Step %
object_true (Step)	1	0.006	5.986
object_1 (Step)	1	0.001	0.544

A proto je schůdnější volbou kratší a přehlednější číslice *1*.

HODNOTA NEBO ODKAZ

Pro rychlou možnost modifikace kódu je využívání proměnných namísto číselných hodnot efektivní, avšak na výkon to má paradoxně horší vliv.

Ve svém posledním experimentu jsem opět využil jednoduchého repeateru, přičemž u prvního objektu byly využity číselné hodnoty, ale u druhého objektu jsem se odkazoval na předem vytvořené globální hodnoty.

```
repeat 10000
{
    val += 10000;
}
```

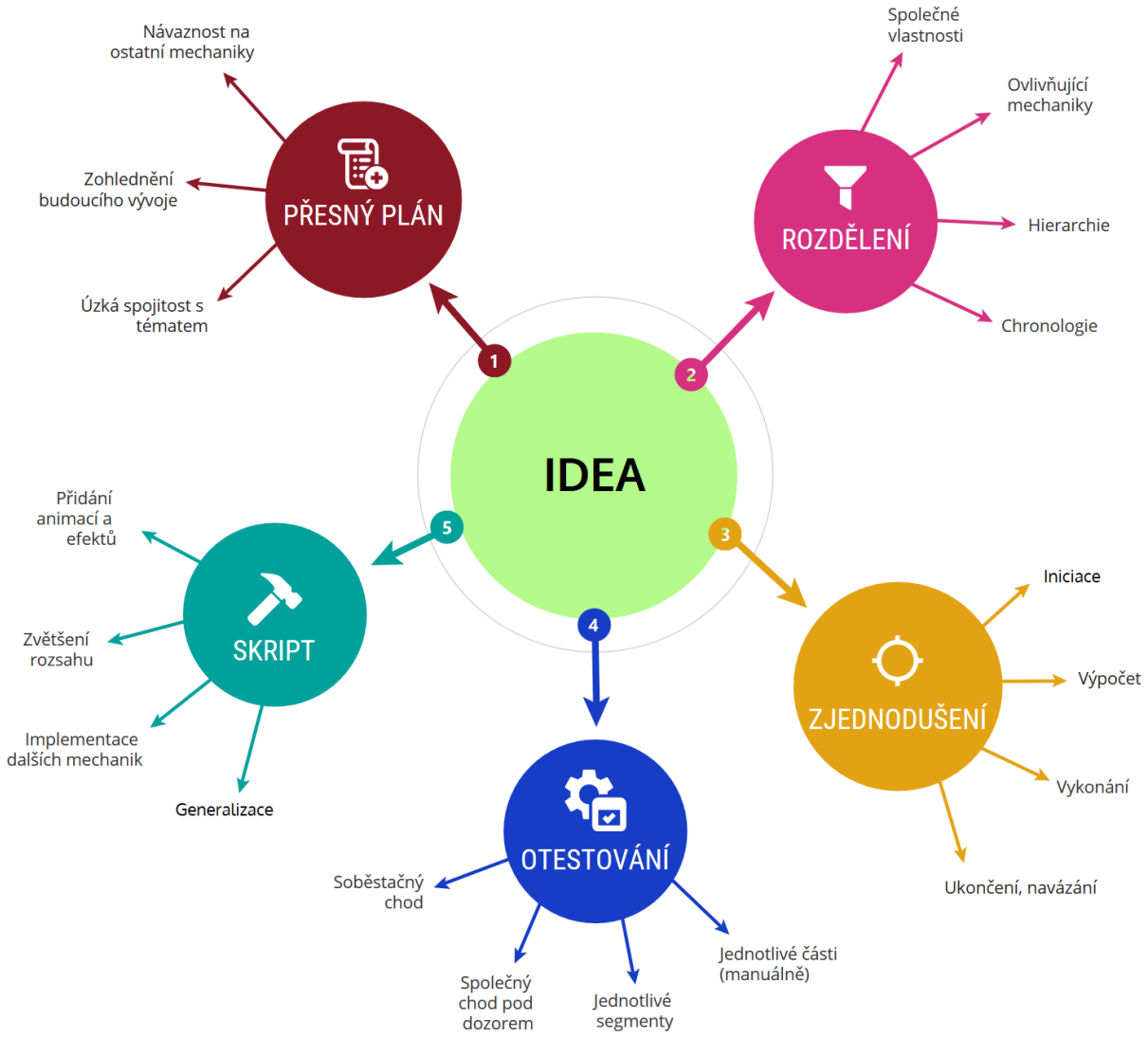
```
repeat global.LOOP
{
    VAL += global.VALUE;
}
```

Přesné číselné vyjádření bylo logicky o něco málo rychlejší, neboť GameMakeru ulehčujeme cestu k těmto datům tím, že je rovnou naformátujeme a uvedeme již v kódu samotném. Přestože využívání hodnot způsobuje mírný deficit na výkonu programu, opětovně se jedná o zcela nevyužitelnou výhodu, neboť by používání pouhých hodnot znemožnilo flexibilitu a celkovou přehlednost kódu.

Name	Call Count	Time (Ms)	Step %
objectset (Step)	19371	23558	46.79
objecthyper (Step)	19371	25142	49.936

12 MYŠLENKOVÁ MAPA

Dalším důležitým aspektem mého programování bylo vytvoření pomyslného plánu, myšlenkové mapy s pravidly, jehož dodržováním jsem byl schopen dosáhnout větší efektivity a prakticky nulové chybovosti všech nových idejí, v podobě mechanik hry.



Základem jakéhokoliv programování by prvotně měl být přesný plán. Skrze tento rozvrh je potřeba zajistit, aby všechny zamýšlené mechaniky úzce souvisely s celkovým tématem hry, aby se předešlo zbytečnému přidávání irelevantního obsahu. Součástí plánu by zároveň měla být představa o budoucích úmyslech společně s propojením dosavadních fungujících herních mechanik.

Po zhotovení plánu je zapotřebí rozdělit nápad dle několika kategorií. A to podle chronologie, tedy návaznosti jednotlivých akcí či eventů a hierarchie, tedy nadřazenosti, která bude hrát roli v posloupnosti podmínek. Posléze je potřeba separovat dosavadně fungující mechaniky, a to tak, aby nezávisle na ostatních pracovali. Dále je pak potřeba sjednotit společné vlastnosti, například totožné konstanty či výpočty, pro zjednodušení podmínek.

Až po rozsortování na jednotlivé segmenty přichází psaní nového kódu, který se dá optimálně rozdělit do 4 částí. Primární je iniciace, jež určí kdy či za jakých podmínek dojde k výpočtu. Výpočet připraví všechny potřebné hodnoty, které jsou zohledněny v podmínkách a umožní nejdůležitější část, vykonání. Vykonání v sobě skrývá stěžejní kód, který určuje chování daného objektu, a přirozeně pokračuje v poslední část, ukončení. Ukončení uzavře celý segment a dovolí pokračování v ostatních segmentech.

Otestování, a tak nalezení chyb, je nejen důležité pro opravení nedostatků v kódu samotném, ale zároveň se jedná o výborného učitele, který mě vždy dovedl na správnou cestu. Testování jsem prováděl v mnoha fázích. Nejprve jsem manuálně otestoval jednotlivé části, dále zvlášť všechny segmenty. Pak jsem manuálně kontroloval vzájemný chod všech segmentů. Finálním testem byl chod všech segmentů bez jakéhokoliv zásahu do podmínek, pod kterými bloky kódy fungují.

Závěrečným krokem, který markantně zlepší přizpůsobivost kódu je převedení do skriptu, a to skrze zobecňování. Hlavní výhodou je možnost implementace malých skriptů do větších, což umožní jednoduchou implementaci dalších mechanik a ovlivňujících aspektů, přičemž se zachová přehlednost a flexibilita. Úplně poslední pak přichází samotný design, tedy grafické efekty a animace, který je ideálním zakončením, neboť by v průběhu programování zbytečně komplikoval podmínky a odvracel pozornost od funkčnosti jako takové.

13 ZÁVĚR

Od začátku roku jsem se díky této práci v programování posunul daleko více, než jsem čekal. Zodpověděl a ověřil jsem své otázky, ba dokonce jsem i lépe pochopil fungování, tedy postupování GameMakeru jednotlivými částmi kódu. Zároveň jsem se setkal s mnohými anomáliemi, které se pokusím v budoucnu prokonzultovat s podporou YoYo Games. Mimo jiné jsem získal obecný přehled o ostatních programovacích jazycích.

Vývoj hry zdaleka není u konce, a tak bych na něm milerád pracoval i příští rok, poněvadž bych se mohl pustit do daleko komplikovanějších problémů, k jejichž řešením jsem na začátku programování neměl dostatečné znalosti.

14 POUŽITÁ LITERATURA / ODKAZY

https://cs.wikipedia.org/wiki/B%C4%9Bhov%C3%A9_prost%C5%99ed%C3%AD

https://en.wikipedia.org/wiki/GameMaker_Studio

https://cs.wikipedia.org/wiki/Debugger

https://cs.wikipedia.org/wiki/V%C3%BDvojov%C3%A9_prost%C5%99ed%C3%AD

https://cs.wikipedia.org/wiki/P%C5%99eklada%C4%8D

https://cs.wikipedia.org/wiki/Objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD

https://cs.wikipedia.org/wiki/Zapouzd%C5%99en%C3%AD_(objektov%C4%9B_orientovan%C3%A9_programov%C3%A1n%C3%AD)

https://docs.yoyogames.com/source/dadiospice/002_reference/data%20structures/ds%20grids/index.html

15 PŘÍLOHA 1: HRA

16 PŘÍLOHA 2: GRAFICKÝ A VÝKONNOSTNÍ EXPERIMENT