

PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO  
KATEDRA INFORMATIKY

# BAKALÁŘSKÁ PRÁCE

Hra Worms



2011

Pavel Procházka

## Anotace

*Z her, které ovlivnily svět, patří Worms k těm nejvýznamnějším. I díky tomu se stali námětem pro moji práci. Zejména je to implementace z roku 1995, která se objevila pro platformu DOS a využívala výkon tehdejších počítačů na maximum. Dnes jsou počítače mnohem rychlejší, a přesto je nutné se zabývat efektivitou programu. Z tohoto důvodu jsem se rozhodl implementovat Worms v jazyce C spolu s knihovnamy z rodiny SDL a knihovnou MGEAN, která vzniká jako podpůrná vrstva, a která umožňuje napsání libovolné dvojrozměrné hry. Výsledná hra podporuje GNU/Linux a experimentálně další UNIXy a Windows. Moje implementace obsahuje základní principy v podobě hry týmů, používání zbraně a v destrukci prostředí. Hlavní výhodou mého řešení oproti originálu je svobodná licence, snadná rozšiřitelnost a také výrazně pokročilejší kvalita obrazu.*

Poděkování patří zejména mému vedoucímu práce panu Mgr. Petru Krajčovi, Ph.D., který téma mé bakalářské práce vymyslel a byl mi dobrým rádcem při tvorbě programu i textu. Dále děkuji rodině a všem, kteří mě při tvorbě bakalářské práce podporovali.

# Obsah

<b>1. Úvod</b>	<b>7</b>
1.1. Původ hry Worms . . . . .	8
1.2. Prostředí . . . . .	8
1.3. Pravidla Worms z roku 1995 . . . . .	8
1.4. Zbraně . . . . .	8
1.5. Uživatelské rozhraní . . . . .	9
1.6. Zásady pro vypracování . . . . .	9
<b>2. Materiály a metody</b>	<b>9</b>
2.1. Architektura projektu . . . . .	10
2.2. Principy vykreslování 2D rastrové grafiky . . . . .	10
2.3. Knihovna SDL . . . . .	12
2.3.1. Důležité struktury SDL . . . . .	12
2.3.2. Důležité funkce SDL . . . . .	13
2.4. MGEAN - Micro Game Engine for Absolute Newbies . . . . .	13
2.4.1. Architektura MGEAN . . . . .	13
2.4.2. Hlavičkové soubory . . . . .	15
2.4.3. Inicializace a deinicializace . . . . .	15
2.4.4. Seznamy v knihovně MGEAN . . . . .	18
2.4.5. Asociativní seznamy . . . . .	19
2.4.6. Vykreslování . . . . .	19
2.4.7. Načítání obrázků a zvuků pomocí MGEAN . . . . .	21
2.4.8. Animace objektů . . . . .	21
2.4.9. Fyzikální zákony v knihovně MGEAN . . . . .	23
2.4.10. Řízení objektů - základní zprávy . . . . .	24
2.4.11. Odchytávání vstupu z klávesnice a myši . . . . .	26
2.4.12. Uživatelské parametry a callbacky spritu . . . . .	26
2.4.13. Práce s konfiguračními soubory . . . . .	28
2.5. Implementace hry . . . . .	29
2.5.1. První kroky s projektem . . . . .	29
2.5.2. Návrh projektu . . . . .	30
2.5.3. Inicializační a deinicializační proces . . . . .	30
2.5.4. Načítání prostředků a jejich správa . . . . .	30
2.5.5. Fáze hry . . . . .	31
2.5.6. Sestavení herního světa . . . . .	31
2.5.7. Nastavení parametrů fyziky . . . . .	32
2.5.8. Rozvržení entit . . . . .	32
2.5.9. Oživení entit . . . . .	33
2.5.10. Interakce entit se světem . . . . .	33
2.5.11. Ovládání pomocí klávesnice . . . . .	34
2.5.12. Ovládání pomocí myši . . . . .	35

2.5.13. Umělá inteligence . . . . .	36
2.5.14. Rozvržení zbraní . . . . .	36
2.5.15. Společný algoritmus pro výpočet střelby . . . . .	37
2.5.16. Heuristika bazuky . . . . .	37
2.5.17. Herní pravidla . . . . .	38
2.5.18. Herní ukazatel . . . . .	39
2.5.19. Základní systém GUI a menu . . . . .	39
2.5.20. Práce se soubory . . . . .	39
<b>3. Dosažené výsledky</b>	<b>40</b>
3.1. Stručná uživatelská příručka . . . . .	40
3.1.1. Kompilace ze zdrojových kódů . . . . .	41
3.1.2. Instalace balíků . . . . .	41
3.1.3. Odinstalace balíku . . . . .	41
3.1.4. Spuštění hry . . . . .	41
3.1.5. Ovládání menu . . . . .	42
3.1.6. Dialog výběru nové hry . . . . .	43
3.1.7. Dialog editace týmů . . . . .	43
3.1.8. Dialog o autorovi . . . . .	43
3.1.9. Dialog vedoucí k ukončení hry . . . . .	43
3.1.10. Ovládání hry . . . . .	44
3.1.11. Indikátory hry . . . . .	44
3.1.12. Stabilita Bazooworms . . . . .	45
3.2. Shrnutí výsledků . . . . .	45
3.3. Licence . . . . .	46
<b>4. Diskuze</b>	<b>46</b>
4.1. Srovnání s jinými projekty . . . . .	46
4.2. Možnosti dalších rozšíření . . . . .	46
4.3. Budoucnost projektu . . . . .	46
<b>Závěr</b>	<b>48</b>
<b>Conclusions</b>	<b>49</b>
<b>Reference</b>	<b>50</b>
<b>A. Obsah přiloženého DVD</b>	<b>51</b>

## Seznam obrázků

1.	Animace pohybu červa. . . . .	12
2.	Animace větrníku - vetrnik.png. . . . .	22
3.	Menu hry v počátečním stavu. . . . .	42
4.	Obrázek z rozehrané hry. . . . .	45

# 1. Úvod

Worms je jedním z neoriginálnějších počínů v herní historii. Princip hry spočívá ve hře týmů složených z několika červů, cílem každého týmu je zůstat posledním týmem v herním světě. K eliminaci nepřátel používá každý tým sadu účinných zbraní. Jednotlivé týmy se po odehraném kole střídají. Herní svět je představován ručně kresleným, z boku viděným terénem, pozadím a vodou. Původní hra byla implementována v 2D grafice a byla určena pro řadu platform, mezi nimiž byl například MS-DOS, Amiga, Apple Mac, Playstation a další. Do dnešních dnů bylo vývojářským týmem Team17 vyprodukováno několik pokračování této úspěšné hry. Některé díly byly dokonce implementovány v 3D.

Hra Worms probíhá sice v tazích, avšak vykreslování světa probíhá čistě v reálném čase. To znamená, že cokoli může v daný okamžik reagovat s prostředím na základě fyzikálních jevů. Vykreslované objekty proto musí být vykresleny v lidskému oku nepostřehnutelném čase. Tento přístup klade jisté nároky na optimalizace kódu a vyžaduje dostatečně výkonný procesor, případně grafický adaptér.

Implementační prostředky jsem volil s ohledem na přenositelnost, rychlost a snadnou implementaci. Jako implementační jazyk byl zvolen jazyk C, hlavní knihovnou pro grafické vykreslování je SDL spolu s několika dalšími multimediálními knihovnami z rodiny SDL. Cílovou platformou jsou zejména distribuce založené na platformě GNU/Linux, vývojovou platformou byla distribuce Ubuntu.

Moje implementace je rozdělena na dva nezávislé celky. Prvním je knihovna MGEAN, která odstiňuje uživatele od nízkoúrovňového programování 2D grafiky. Při programování této knihovny jsem se zaměřil na snadnou použitelnost a minimalistický kód. Druhý, větší celek, tvoří samotná hra, která široce využívá knihovnu MGEAN, díky ní je implementace poměrně přímočará a kód by měl být snadno čitelný i pro člověka bez hluboké znalosti programování rastrové grafiky.

Za hlavní přínos mé práce považuji nejen svobodný kód a grafiku, ale také snadnou rozšiřitelnost a modularitu výsledného díla. Svobodná licence kódu a grafiky umožňuje neomezené kopírování a upravování kýmkoliv, a tudíž zajišťuje snadný růst projektu i v budoucnosti. Po dokončení vlastních prací poskytnu tuto hru komunitě.

### 1.1. Původ hry Worms

Grafika původních Worms byla ve stylu kresleného filmu, animace byly opatřeny zábavnými zvuky, podle zdroje[6]. Vykreslování umožňovalo plynulé posouvání obrazovky a nabízelo spoustu dalších grafických efektů.

### 1.2. Prostředí

Grafika byla v originále založená na rastrovém vykreslování. Hlavními zdroji pro sestavení světa jsou tudíž bitmapy. Vykreslování probíhá dvojrozměrně a vidění světa je omezeno na pohled z boku. Prostředí je sestaveno z pozadí, země, moře a dalších dodatečných objektů jako jsou mraky nebo poletující částice. Terén může podléhat destrukci způsobené výbuchy nábojů některých zbraní. Červové se mohou pohybovat po terénu „píďalkovitými“ pohyby doprava nebo doleva, případně skokem do obou směrů.

### 1.3. Pravidla Worms z roku 1995

Samotná hra je postavena na přítomnosti různých týmů červů (jejich počet byl omezen na čtyři), jejichž primárním cílem je zneškodnit všechny ostatní týmy. Výhra nastává, pokud existuje právě jeden tým, který má alespoň jednoho žijícího červa. Remíza nastává v případě, když není žádný tým v herním světě, který disponuje žijícím červem.

Týmy se střídají stejným způsobem jako u karetních her. Střídání nastává buď po výstřelu, nebo po uplynutí časového limitu, případně po smrti aktuálně hrajícího červa. Červ, který je právě na tahu, bude příště hrát, až odehrají všichni jeho spoluhráči z týmu.

Výstřel ze zbraně lze ovlivňovat mírou natažení, čím je vyšší, tím rychleji náboj vyletí z ústí zbraně. Náboj při střetu s terénem vybuchne, případně způsobí jiné poškození.

Každý tým může být ovládán člověkem u počítače, nebo pomocí vestavěné umělé inteligence. Člověk ovládá svého aktivního červa pomocí klávesnice. Klávesami šipka vpravo a vlevo posouvá červa do odpovídající strany. Šipkami nahoru a dolů se ovlivňuje zaměření zbraně. Klávesa enter způsobí výskok červa ve směru, kterým je právě natočen. Přidržením klávesy mezerník se zvyšuje natažení zbraně, pokud dosáhne maxima, zbraň sama vystřelí. Uvolněním dříve zmáčknutého mezerníku dojde k výstřelu s aktuálním natažením. Posuv kamery je zajištěn myší.

### 1.4. Zbraně

K destrukci nepřátel v originální verzi bylo možné využít docela velkého arzenálu zbraní.



Bazooka je výchozí zbraní. Bazooka po výstřelu, na základě směru, síly natažení a intenzity větru vystřelí náboj, který letí po trajektorii známé jako vrh šikmý. Po nárazu na zeminu exploduje a způsobí poškození červům v okolí, které je spočítáno podle přímé uměry v závislosti na vzdálenosti od epicentra výbuchu. Výbuch bazooky také zničí část prostředí, což může být dobrá strategie na zničení nepřítele.

V původních Worms bylo zařazeno více zbraní, avšak jejich rozbor přenechám na osobní zkušenosti čtenáře.

## 1.5. Uživatelské rozhraní

Ve hře Worms z roku 1995 bylo na menu pohlíženo velmi střídavě. Obsahuje volbu pro novou hru, která vyvolá jednoduchou tabulku s možností výběru týmů. Další volbou je editace týmů, o autorovi a ukončení hry. Jednotlivé volby i tabulky jsou tvořeny pouze textem a obrázkem s nápisem *Worms* v pozadí. Menu se ovládá myší.

## 1.6. Zásady pro vypracování

Pro úplnost uvádím oficiální zadání práce, kterým jsem se při tvorbě této bakalářské práce řídil.

*Cílem práce je implementovat tahovou hru na motivy původní hry Worms z roku 1995. Každý hráč bude moci ovládat několik postavíček, přičemž jednotlivé postavíčky budou moci po sobě střílet pomocí šikmého vrhu. Aplikace bude umožňovat hru více hráčů včetně možnosti hry s počítačem.*

## 2. Materiály a metody

Pro účely psaní programu jsem použil jazyk C, který svojí flexibilitou a rychlostí splňuje veškeré předpoklady pro vytvoření počítačové hry. Při používání jazyka C jsem čerpal ze zdroje [2]. Vykreslování nízkoúrovňové grafiky zajišťuje knihovna SDL, SDL\_image používám pro načítání obrázků, SDL\_ttf na vykreslování písem a SDL\_gfx je přítomna z důvodu transformací obrázků. SDL\_mixer je přítomna pouze z důvodu možného rozšíření o zvukové efekty v budoucnosti. Ke správě kódu jsem založil v adresáři se hrou git repozitář. Vývoj Worms probíhal na operačním systému Ubuntu 10.10, za podpory běžných nástrojů z rodiny GNU. Jedná se zejména o Make, gcc, gdb, Gimp a mnohé další. Jako nástroj pro sledování paměti byl použit zejména nástroj Valgrind. Jako textový editor výborně posloužil GNU Emacs s modulem CEDET. Pro kreslení grafiky jsem využil výhradně služeb programu Gimp a Inkscape.

## 2.1. Architektura projektu

Projekt je rozdělen do dvou celků, které také odděleně vznikají. Podstatnou část prací na tomto projektu představuje knihovna MGEAN, která poskytuje poměrně obecný základ pro psaní dvojrozměrných her. MGEAN zajišťuje optimalizované vykreslování grafiky v reálném čase, obsahuje také podporu pro klávesnici a myš. Samotná hra se pro účely projektu nazývá Bazooworms a využívá MGEAN jako vrstvu, která odděluje programátora od nízkoúrovňových nesnází s knihovnou SDL.

## 2.2. Principy vykreslování 2D rastrové grafiky

Rastrová grafika se stala základem nejedné úspěšné hry, zejména v minulosti byl tento způsob vykreslování velmi oblíben. Podle knihy [1], se obrázek většinou hardwarově reprezentuje jako jednorozměrné pole, to zaručuje velmi efektivní uložení a poměrně snadné pixelové operace. Jednorozměrné pole sice plně neodpovídá intuici, většina programátorů by očekávala pole dvojrozměrné. Adresování pixelu v jednorozměrném poli je však jen o málo složitější než u dvojrozměrného, podmínkou je pouze znalost výšky a šířky obrázku. Algoritmus pro získání pixelu z jednorozměrného pole může v pseudo-kódu vypadat následovně.

```
function get_pixel( 1D_pole , vyska , sirka , x , y )
{
    return 1D_pole[ (y*sirka) + x ];
}
```

Obrázek v paměti se obvykleji označuje jako povrch, případně anglicky *surface*. S každým povrchem lze provádět operaci kopírování do jiného povrchu - *blitting*, která způsobí zkopírování určitého obdélníkového výřezu zdrojového obrázku do výstupního. Speciálním případem povrchu je povrch vykreslovací, který je přímo napojen na grafický výstup. To znamená, že změny na tomto povrchu jsou viditelné v reálném čase na monitoru nebo displeji.

V případě použití pouze jednoho vykreslovacího povrchu však může být pozorovatelné trhání a prokládání obrazu, které se vyskytuje, pokud vykreslování nesynchronizujeme s obnovovací frekvencí monitoru. Synchronizace pak ale stojí drahocenný čas. Tento neduh se neprojevoval na všech zařízeních, ale typicky jím trpěla platforma PC. Řešení je jednoduché, spočívá ve zdvojení vykreslovacího povrchu. Jeden slouží jako buffer a z druhého se vykresluje na monitor. Po dokončení vykreslování se pouze zamění ukazatele těchto bufferů. Tato technika dostala název *double buffering*. Operace záměny bufferů se obvykle označuje jako *flipping*. DirectX<sup>1</sup> dokonce podporuje více než dva obrazové buffery - to se označuje jako *multibuffering*. Při operacích s povrchy je třeba stále brát zřetel

---

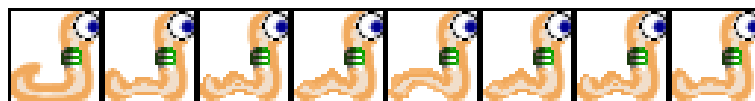
<sup>1</sup>Sada knihoven od firmy Microsoft pro práci s grafikou a s multimédií.

na optimální výkon, proto je dobré si rozmyslet, zda by měl být povrch umístěn ve video paměti nebo v systémové paměti. Oba přístupy mají své výhody a nevýhody. Pokud je povrch umístěn v hardwarové video paměti jedná se o tzv. *hardware surface*, který má tu vlastnost, že kopírování do jiného hardwarového povrchu trvá obvykle velice krátkou dobu díky masivním optimalizacím v grafických čipech. Zásadní omezení hardwarových povrchů však spočívá ve velmi pomalém přesunu dat mezi systémovou a grafickou pamětí. Toto omezení se tedy týká hlavně čtení a zapisování pixelů - rasterizace. Povrch, který je umístěn v systémové paměti se označuje jako *software surface*, jeho nevýhodou je pomalejší kopírování do vykreslovacího *bufferu*, ovšem pixelové operace jsou zásadně rychlejší než v případě hardwarového povrchu.

Další optimalizace, která se někdy využívá, je používání palet. Paleta je obvykle pole obsahující 256 barev. Obrázek je potom jen pole indexů do palety, tím se poměrně výrazně zredukuje velikost obrázku. Tato funkcionalita bývá podporována i hardwarově, tudíž mohou být operace nad takovým povrchem velmi rychlé. S paletami je spojen jeden zajímavý efekt. Paleta umožňuje snadno zaměnit barvy za jiné, čímž se efektivně změní barevné rozložení obrázku. To lze s výhodou využít v případě, kdy máme například více postavíček stejných tvarů, ale potřebujeme každé postavíčce přiřadit oblečení v jiné barvě. Nevýhodou palet je nutná redukce množství odlišných barev v obrázku, proto se využívají spíše u obrázků s malým rozpětím barev a ve starších hrách.

Podstatnou vlastností obrázků bývá také průhlednost, která je nutná při skládání scény. V praxi se používá buď modernější přístup přes alfa kanál, anebo tradiční způsob přes barvové klíče. Princip alfa kanálu spočívá v přidání aditivní informace ke každému pixelu, tato informace popisuje, jak moc bude daný pixel průhledný, což má opět zvýšené nároky na přenosy dat a výpočetní rychlost. Barvové klíče naproti tomu mnohem více spoří systémové prostředky. Jejich princip spočívá v označení jedné barvy jako průhledné, pokud se na ni narazí při kopírování, je jednoduše vynechána.

Další obvyklou součástí her bývají animace, které přinášejí vizuální efekty a pocit z pohybujícího se herního světa. Animace se obvykle skládá z posloupnosti obrázků, které se vyměňují v určitém časovém intervalu, čím je tento interval menší, tím je dojem z plynulosti animace lepší. Člověku pro dokonalé vnímání plynulého obrazu stačí zhruba 45 milisekund na snímek. V praxi se používá z důvodu vysokých nároků na kreslení grafiky spíše větší časový rozestup mezi jednotlivými snímky. Z pohledu programátora jsou jednotlivé snímky animace obdélníky, které symbolizují své umístění v povrchu. Často se tento proces automatizuje tím, že grafik poskládá snímky animace do smluveného obrazce, například do obdélníka, přičemž všechny snímky mají stejnou velikost. Tento princip je viditelný na obrázku 1., kde jsou jednotlivé snímky poskládány vedle sebe a tvoří tak obdélník.



Obrázek 1. Animace pohybu červa.

## 2.3. Knihovna SDL

Knihovna SDL - Simple Directmedia Layer vznikla jako programová vrstva pro snadné portování her z MS Windows na unixové systémy. V současné chvíli se jedná o komunitní dílo, šířené pod licencí GNU LGPL 2.1 nebo vyšší a licencí komerční. Tato knihovna je portovaná nejen na Linux, Windows, MacOSX, ale také na některé netradiční nebo dále nevyvíjené platformy. Jako příklad mohu uvést AmigaOS, Atari, OS/2 nebo mobilní systém SymbianOS. SDL je dále možno použít s velkým počtem jazyků. SDL je sice nativně napsáno v C, ale podporuje i C++, ObjectiveC, Go, Python, Haskell, Java, Lisp a mnohé další jazyky.

Z pohledu programátora SDL obsahuje funkce pro grafické vykreslování na úrovni povrchů, přičemž podporuje všechny používané hardwarové optimalizace. Dále nabízí také funkcionalitu pro práci se zvukem a se vstupními zařízeními. V SDL je také zahrnuta podpora OpenGL. SDL se ovšem často používá s mnoha podpůrnými knihovnami, které ještě zjednodušují některé operace a úkony. Typicky je to knihovna `SDL_image`, která podporuje načítání mnoha různých formátů obrázků, dále `SDL_ttf` umožňující práci s true type fonty, `SDL_mixer` nabízí abstrakci nad komplexním zvukovým subsystémem knihovny SDL, knihovna `SDL_gfx` je určena zejména pro transformaci povrchů a pro kreslení primitiv vektorové grafiky.

Jelikož MGEAN z velké části zastiňuje SDL, ve zkratce uvedu jen některé důležité struktury a funkce, které se běžně používají i s knihovnou MGEAN, a které je dobré znát při snaze pochopit kód mé práce. Struktury z knihovny SDL jsou použity z důvodu snadné manipulace s nimi, navíc jsou dobře zdokumentované. Podrobná dokumentace je umístěna na oficiálních stránkách projektu SDL <http://www.libsdl.org/>.

### 2.3.1. Důležité struktury SDL

V kapitole o rastrové grafice byl zaveden pojem povrch, knihovna SDL ho reprezentuje pomocí datové struktury **SDL\_Surface**, podle internetového zdroje [5]. Ta obsahuje nejen informaci o jednorozměrném poli, ale také o dalších pokročilejších parametrech, jako je paleta nebo barvový klíč. Další nezbytnou strukturou pro vykreslování je obdélník, ten je reprezentován typem **SDL\_Rect**, který obsahuje informace o levém horním bodu a o šířce a výšce. Ve hrách se také často využívají události ze vstupních zařízení. To je v SDL reprezentováno rozsáhlou strukturou **SDL\_Event**. Posledním význačnějším typem je **SDL\_Color**, což je přenositelná struktura nesoucí informaci o jednotlivých barevných kanálech

včetně alfy.

### 2.3.2. Důležité funkce SDL

K povrchům se pojí několik základních operací, které mají v SDL své ekvivalentní implementace. Funkce **SDL\_CreateRGBSurface** slouží k vytvoření nového povrchu zadaného šířkou, výškou a formátem. Tato funkce také umožňuje vybrat mezi hardwarovým a softwarovým povrchem. Funkce **SDL\_FillColor** provede vyplnění části povrchu jednou barvou předanou argumentem, to se nejčastěji používá pro „přemazání“ povrchu barvou nebo k vykreslení obdélníka. Volání **SDL\_SetColorKey** nastaví danému povrchu barvový klíč. Funkce **SDL\_LockSurface** slouží k uzamčení povrchu. To je nutné udělat vždy, když chceme provést vykreslení nějakého objektu do povrchu pomocí přímého přístupu k pixelům. Po dokončení vykreslování je nutné zavolat funkci **SDL\_UnlockSurface**, která povrch odemče. Jako funkce pro *blitting* je v SDL přítomna funkce **SDL\_BlitSurface**, která provede zkopírování části jednoho povrchu do druhého.

## 2.4. MGEAN - Micro Game Engine for Absolute Newbies

MGEAN je knihovna, kterou jsem vyvinul nejen pro účely Worms, ale jako obecnou knihovnu umožňující napsání běžné 2D hry strategiemi počínaje a adventurami konče. Podstatnou myšlenkou, která provází celou knihovnu je jednoduché API s minimálním počtem exportovaných funkcí, které však mají zajišťovat co možná nejsilnější funkcionalitu. Funkce, které jsou určeny pro uživatele, jsou opatřeny prefixem MG\_. Struktury definované knihovnou MGEAN jsou psány velkými písmeny.

### 2.4.1. Architektura MGEAN

Hlavním datovým typem, na kterém stojí celá implementace je seznam, nad kterým je vystavěn ještě seznam asociativní. Seznam jsem zvolil pro lineární složitost při přidávání a odebrání prvku, což jsou velmi časté operace vyžadované hrami. Operace přidání a odebrání prvku do nekoncových partií seznamu jsou dokonce rychlejší než v případě pole. Seznam je použit také pro implementaci výkonnostně nejnáročnější datové struktury - vykreslovacího zásobníku, který se každý snímek musí projít. Ani zde však použití seznamu nezpůsobuje zpomalení. Asociativní seznam je použit pro případy, kdy je potřeba pamatovat si data ve formátu klíč : hodnota. Pro tento problém není mé řešení nejrychlejší možné, avšak pro účely her dostatečné s ohledem na to, že v praxi (Bazooworms) je počet klíčů maximálně v řádu desítek. Asociativní seznam se dále používá pro účely uchovávání parametrů a zpráv spiritů, což jsou základní grafické prvky každé hry.

Prvním prvkem, který odstiňuje programátora od knihovny SDL je inicializační procedura, která zabírá zhruba padesát až sto řádků programového kódu v případě čistého použití SDL. Proto jsem ji abstrahoval do jednoho volání - `MG_init_video`, které zajistí nastavení knihovny MGEAN a SDL do výchozích hodnot. S inicializací souvisí také proces deinicializace i tato funkcionality je obsažena v mé knihovně a má za následek uvolnění všech dosud použitých prostředků, které jsou řádně zaregistrovány. V průběhu inicializace se také vytvoří objekt `MACHINE`, který reprezentuje stav knihovny a hostujícího počítače.

Daleko významnějším způsobem odstínění je systém načítání prostředků, zejména grafiky. Vytvořil jsem zde systém líného načítání, který alokuje grafiku a zvuky, až jsou vyžádány uživatelem. Registrace prostředků v knihovně probíhá v podobě záznamu do asociativního seznamu. Klíč představuje cestu k souboru a hodnotou je pointer na načtený prostředek. Při každém dalším pokusu o získání zaregistrovaného prostředku se již nic nenačítá, ale je vrácen prostředek z asociativního seznamu. V případě výkonnostního propadu je možné grafiku načíst dopředu pomocí tzv. *preloadingu*, který vynutí načtení prostředků dříve než jsou potřeba. Grafika je podorována v závislosti na knihovně `SDL_image` a operačním systémem, typicky jsou to ale rastry ve formátu `.png`, `.jpeg`, `.bmp`. Princip líného načítání se používá také pro načítání hudby.

Dalším významným datovým typem je animace. Ta se rovněž při vytvoření pojmenuje a je možné ji získat podobně, jako je tomu v případě grafiky. Vnitřně je opět uložena a zaregistrována v asociativním seznamu. Jednotlivé součásti animace - snímky jsou představovány obdélníky. Každá animace také nese informaci o čase, který představuje prodlevu mezi jednotlivými snímky. Po uplynutí tohoto časového rámce dojde k výběru následujícího snímku. Zvolí se nejbližší snímek vpravo od něj. V případě, že jsme na konci animace, dojde k opakování - vybere se znovu první snímek.

MGEAN se také stará o základní fyzikální výpočty. Implementace obsahuje deklaraci základních typů potřebných k výpočtům. Základním typem je `VECTOR`, představující vektor v dvojrozměrném prostoru. Dále je to typ `PHYSICS`, který nese fyzikální parametry objektu. Obsahuje atributy jako je rychlost objektu, což je vektor, neboť rychlost je vektorová veličina. Dále je to síla, taktéž reprezentována vektorem. Posledním atributem je hmotnost, ta musí být nenulová, pokud chceme, aby objekt podléhal fyzikálním zákonům. Každý herní snímek je přepočítána pozice objektu v závislosti na čase, působící síle a rychlosti objektu.

Nejvýznamnější a nejsložitější je implementace spritu. Každý objekt `SPRITE` obsahuje reference na použitý zdrojový obrázek, animaci, destinační obdélník, přídatné argumenty, odchytávané zprávy a další. Každý sprite musí být umístěn na vykreslovacím zásobníku (`drawing_stack`). Který je každé vykreslení projit. Respektive volání funkce `MG_draw_stack` jej projde přes mapování `MG_destructive_map`. Mapovací funkce obsahuje tuto posloupnost operací nad spritem:

- Vyvolá událost `on_iter`.
- Přepočítá fyziku.
- Vyhledá kolize mezi mapovaným objektem a ostatními objekty na vykreslovacím zásobníku, pokud došlo ke kolizi pošle zprávu `on_collide`.
- Vybere nový snímek animace, pokud uplynul časový interval pro tuto činnost, a pokud je animace nastavena.

Vstupním bodem do knihovny MGEAN je funkce `MG_draw_stack`. Tato funkce sestavuje herní snímky a zajišťuje zasílání zpráv spritům. Jejím úkolem je nejprve aktualizovat buffer klávesnice a myši, aby byly informace o zmáčknutých klávesách aktuální vůči právě vykreslované scéně. Dále je vykreslen zásobník se sprity, pomocí principu dříve popsáném. Následně se musí odstranit ze zásobníku sprity, na něž byla zavolána funkce `MG_remove_sprite`. Odstranění nelze udělat přímo ve funkci `MG_remove_sprite`, neboť by se stal `drawing_stack` nekonzistentním. Poté proběhne vykreslení scény za pomoci optimalizačních technik v podobě tzv. *dirty rects* optimalizace. To jest, projdou se pouze obdélníky, které je nutno aktualizovat a tyto oblasti se překreslí. Následně dojde k prohození předního a zadního bufferu (tzv. *flipping*). Nakonec se ještě stabilizuje snímková frekvence předaná parametrem, aby se předešlo nepříjemnému trhání obrazu. Stabilizace počtu snímků se docílí spánkem procesu po dobu, která mu zbyla po vykreslení snímku.

#### 2.4.2. Hlavičkové soubory

Pro práci s knihovnou stačí uvést hlavičkový soubor `MG_lib.h`. V konkrétním případě:

```
#include "MG_lib.h"
```

#### 2.4.3. Inicializace a deinicializace

S inicializačním procesem MGEANu souvisí datový typ `MACHINE`. Struktura `MACHINE` představuje vnitřní stav knihovny a je definována takto:

```
typedef struct MACHINE
{
    SDL_Surface *screen;
    SDL_Surface *game_world;
    Uint8 *keyboard_state;
    MOUSE mouse;
    Uint8 use_audio;
    NODE *events;
```

```

    SDL_Rect view_rect;
} MACHINE;

```

Proměnná `screen` je ukazatel na vykreslovací povrch SDL, `game_world` je povrch celého herního světa. Položka `keyboard_state` ukazuje na buffer klávesnice, `mouse` je objekt myši. Parametr `use_audio` je příznak, zda jsou povolené zvuky. Proměnná `events` je seznam událostí, které se vyvolaly během posledního snímku. Obdélník `view_rect` označuje polohu kamery v herním světě.

Každá knihovna má obvykle nějaký vstupní bod, kterým inicializuje svůj stav do výchozích hodnot. Takovou funkci v knihovně MGEAN je:

```
int MG_init_video (int width, int height, Uint32 video_flags)
```

Funkce inicializuje knihovnu a vykreslování. Parametry `width` a `height` určují rozměry obrazovky nebo okna. V případě, že jsou oba 0, je okno nastaveno na nativní velikost, kterou podporuje monitor a grafická karta. Parametr `video_flags` se dosazuje do volání SDL funkce `SDL_SetVideoMode`, určuje míru optimalizace zobrazování videa, může být 0 pro implicitní nastavení. Tato funkce vrátí 0 při úspěchu, při neúspěchu ukončí program se statusem -1. SDL je také možno inicializovat ručně před voláním `MG_init_video`, v takovém případě tato funkce zinicializuje pouze zbylé potřebné věci. Pro ilustraci chování uvedu jednoduchý příklad.

```
MG_init_video( 800 , 600 , SDL_SWSURFACE );
```

Toto volání inicializuje knihovnu a vytvoří okno s rozměry 800 na 600 pixelů. Nebude použita žádná hardwarová akcelerace. Dalším typickým příznakem je `SDL_FULLSCREEN` pro přepnutí do celoobrazovkového režimu.

Párová operace k inicializaci knihovny je ukončení knihovny. V MGEAN je ukončovací funkce:

```
void MG_quit ()
```

Funkce `MG_quit` uvolní veškeré prostředky zabrané knihovnou a korektně ukončí SDL, musí být vždy přítomna.

Ve výchozím stavu po inicializaci je herní svět velký jako obrazovka, což však v mnoha případech nestačí a je zapotřebí svět zvětšit. Obrazovka pak slouží jenom jako určitý výsek z herního světa. Funkce pro zvětšení herního světa je deklarována takto:

```
SDL_Surface * MG_set_world( int w , int h )
```

Přenasťaví velikost současného herního světa na nové rozměry `w`, `h`. Ve výchozím stavu je velikost herního světa shodná s velikostí obrazovky. Tuto funkci je možné volat až po inicializaci knihovny.

Pokud je herní svět zvětšen, požadujeme obvykle pohyb obrazovky vůči hernímu světu, tuto funkcionalitu zajišťuje funkce:



```
void MG_set_view_rect( SDL_Rect r )
```

Funkce `MG_set_view_rect` nastaví polohu a rozměry kamery. Kamera představuje viditelný výřez z herního světa. Kamera nesmí přesáhnout rozměry světa, ani být částí mimo herní svět. Jinak je chování nedefinované a obvykle vede k pádu aplikace pro neoprávněný přístup do paměti.

V mnoha případech potřebuje programátor hry znát některé parametry knihovny, jako je velikost vykreslovacího světa nebo zmáčknuté klávesy. Funkce, která získá stav knihovny je deklarována následovně:

```
MACHINE *MG_get_machine ( )
```

Výstup této funkce nesmí být uvolňován, protože s touto pamětí pracuje knihovna až do ukončení pomocí `MG_quit()`.

Herní svět je obvykle vykreslován pomocí snímků, které se mění v dostatečně malém časovém intervalu. Pro vykreslení jednoho snímku hry slouží následující funkce, která navíc stabilizuje snímkovou frekvenci.

```
void MG_draw_stack (int delay)
```

Funkce, která provede jednu iteraci nad všemi sprity. Typické použití je v nekonečné vykreslovací smyčce. Parametr `delay` zaručí, že vykonávání funkce bude trvat alespoň `delay` milisekund, v případě dostatečně rychlého počítače právě `delay` milisekund. Toto chování je výhodné pro stabilizaci počtu snímků za vteřinu. Například pro hodnotu 50 to bude 20 snímků za vteřinu. Tato funkce také může vyvolat některé zprávy. Pokud bychom chtěli vykreslit 1000 snímků po 20 milisekundách, docílíme to tímto kódem:

```
int i = 1000;
while( i-- )
{
    MG_draw_stack( 20 );
}
```

Na konec kapitoly o inicializaci MGEANu uvedu kompletní příklad základní inicializace hry.

```
#include "MG_lib.h"

int main( int argc , char * argv[] )
{
    MACHINE * m = NULL;
    MG_init_video( 800 , 600 , 0 );
    m = MG_get_machine();

    /* dokud se nezmáčkne escape */
}
```

```

while( !MG_key_down_p( SDLK_ESCAPE ) )
{
    MG_draw_stack( 20 ); /* vykresluj */
}

MG_quit(); /* ukončení */
return 0;
}

```

Tento kód inicializuje knihovnu, zjistí stav knihovny pomocí `MG_get_machine` - uvedeno pouze pro demonstraci použití. Pokud uživatel nezmáčkl klávesu escape vykresluje, jinak ukončí knihovnu. Funkce `MG_key_down_p` bude vysvětlena později. Jako výstup bude vidět pouze okno s černým obsahem, které je možno ukončit klávesou escape.

#### 2.4.4. Seznamy v knihovně MGEAN

Knihovna MGEAN vnitřně pracuje nejčastěji s datovou strukturou známou jako spojový seznam. Při implementaci seznamu jsem využíval zdroj [3]. Spojový seznam jsem zvolil pro snadné operace přidání, odebrání a vyhledání prvku. Tyto operace jsou velmi často u her používány. Pro každý spojový seznam v MGEAN platí, že `NULL` je seznam a každý pointer `NODE *` je seznam. V praxi definuji uzel seznamu následujícím způsobem.

```

typedef struct NODE
{
    void *value; /* ukazatel na hodnotu, kterou nese tento uzel */
    struct NODE *node; /* ukazatel na následníka */
} NODE;

```

Následující funkce je možné použít k práci se seznamy.

**void MG\_add (NODE \*\* root, void \*value)** – přidá na konec existujícího seznamu `root` hodnotu `value`. V případě, že chceme založit nový seznam, musí být `*root` `NULL`. Funkce vždy uspívá, pokud uspěje `malloc`, a pokud je `root` platná adresa.

**void MG\_destructive\_map (NODE \* list, void (\*func) (void \*value))** – funkce podobná funkci `mapcar` z Lispu, ale má destruktivní charakter a nevrací žádnou hodnotu. Funkce `func` musí být v předepsaném tvaru, pak za parametr `value` dostává hodnotu daného uzlu seznamu.

**void \*MG\_remove\_node (NODE \*\* l, void \*value)** – odstraní ze seznamu `l` hodnotu `value`. Vráti `NULL` pokud nebylo nic smazáno, jinak vrací hodnotu `value`. Pokud je prvek v seznamu duplicitně, je vymazán pouze jednou. Pokud se ze seznamu odebral poslední prvek, bude jeho hodnota `NULL`.

**NODE \*MG\_add\_nth (NODE \*\* l, void \*val, int nth)** – přidá hodnotu `val` na pozici `nth` v seznamu, pokud taková pozice neexistuje, je přidán na konec. Vrací ukazatel na nově vytvořený uzel.

**int MG\_list\_length (NODE \* l)** – vrací délku seznamu, pro případ, kdy se `l` rovná `NULL` je vrácena 0.

**void MG\_remove\_list (NODE \* list, void (\*func) (void \*rm))** – uvolní celý seznam, pokud není funkce `func` `NULL`, je aplikována na hodnotu všech uzlů.

**void \*MG\_list\_nth (NODE \* l, int n)** – vrátí hodnotu `n`-tého uzlu v seznamu `l`.

#### 2.4.5. Asociativní seznamy

Asociativní seznamy jsou implementačně vystavěny nad běžným spojovým seznamem. Každý asociativní seznam by měl obsahovat na sudých pozicích (včetně 0) klíče, které jsou typu `char *`. Na lichých pozicích se mohou vyskytovat libovolné hodnoty, respektive pointery na ně. Asociativní seznamy, by neměly být modifikovány funkcemi pro běžné seznamy.

Funkce operující s asociativními seznamy jsou následující.

**void MG\_add\_pair (NODE \*\* list, const char \*key, void \*value)** – přidá na konec seznamu `list` kopii řetězce `key` a za něj umístí uzel s hodnotou `value`. Vytvoří tedy asociaci klíč : hodnota.

**void \*MG\_get\_nth (NODE \* list, const char \*symbolic\_id)** – vyhledá klíč `symbolic_id` v seznamu `list` a vrátí hodnotu asociace.

**void MG\_destroy\_pair (NODE \* l, void (\*func) (void \*value))** – uvolní asociativní seznam, na klíče je zavolána funkce `free`, na hodnoty pak funkce `func`, pokud je však jiná než `NULL`.

#### 2.4.6. Vykreslování

Knihovna MGEAN definuje základní datové typy široce používané pro vývoj her. Pro práci s událostmi, grafikou a zvuky používá typy z SDL. Základní entitou v knihovně MGEAN je `sprite` představující obecný zobrazitelný objekt. `Sprite` umožňuje detekovat kolize s jinými objekty a mohou na něj být aplikovány účinky fyzikálních zákonů. Každý `sprite` také obsahuje asociativní seznam, který dovoluje ukládat uživatelské proměnné a instalovat uživatelské callbacky. Interně je definován takto:

```

typedef struct SPRITE
{
    ANIMATION *anim;
    SDL_Rect src;
    SDL_Rect dest;
    SDL_Surface *img;
    NODE *args;
    NODE * attached_sprites;
    PHYSICS physics;
    int time;
    AUDIO audio_state;
    int type;
} SPRITE;

```

Na první pohled je patrné, že obsahuje ukazatel na objekt animace `anim`. Dále obsahuje obdélník `src` představující výřez v povrchu `img`. Obdélník `dest` je oblast, do které se sprite vykreslí. Ukazatel na povrch `img` ukazuje na aktuální zdrojový povrch, který může být `NULL` - pak se nevykresluje nic. Seznam `args` je asociativní seznam, který obsahuje uživatelské callbacky a parametry, při vytvoření je prázdný. Do dalšího seznamu `attached_sprites` jsou ukládány navázané sprity ve smyslu pohybu. To znamená, že při pohybu spritu se pohnou i sprity uložené v `attached_sprites`. `PHYSICS` je struktura popisující fyzikální parametry. Proměnná `time` je použita pro interní výpočty animace. Poslední významnou proměnnou je `type`, což je proměnná pro uživatelské nastavení typu spritu. Proměnná `type` je jediná, která může být měněna přímo, se zbylými operují předepsané funkce.

K vytvoření spritu a jeho zařazení na vykreslovací zásobník se používá funkce `MG_make_sprite`, která je definovaná následovně.

```
SPRITE * MG_make_sprite (SDL_Surface * s, SDL_Rect * src)
```

Tato funkce očekává dva parametry. Parametr `s` je ukazatel na libovolný platný SDL povrch, může nabývat hodnoty `NULL`. Parametr `src` je ukazatel na zdrojový obdélník v povrchu `s`, hodnota `NULL` je interpretována tak, že se použije jako zdrojový obdélník celý povrch. Funkce `MG_make_sprite` vždy skončí úspěchem na posixových systémech, na systémech rodiny Windows může vrátit `NULL` při nedostatku paměti. Návratovou hodnotou je ukazatel na `SPRITE`, ten je současně s vytvořením zařazen na vykreslovací zásobník. To znamená, že pokud je `s` jiné než `NULL` je také zobrazen. Příklad použití:

```
SPRITE * muj_sprite = MG_make_sprite( IMG_Load( "foo.png" ),
                                     NULL );
```

Načte obrázek `foo.png` z aktuálního adresáře a použije jej celý pro sprite, od této chvíle bude vykreslován a aktualizován vždy po zavolání `MG_draw_stack`.

#### 2.4.7. Načítání obrázků a zvuků pomocí MGEAN

MGEAN poskytuje mechanismus pro cachování sdílených prostředků, typicky se u her jedná o animace, zvuky a grafiku. Výhodou tohoto přístupu je sjednocení většiny prostředků do jednoho místa, to výrazně ulehčuje práci s načítáním, uvolňováním a sdílením prostředků. Tvar funkcí určených pro naplnění cache s obrázky je:

```
int MG_preload_img( const char * path )
SDL_Surface * MG_get_img (const char *path);
```

Preload značí, že pouze přednačítáme obrázek do cache. `MG_get_img` se nejprve podívá do cache, zda už je obrázek načten, pokud ano vrátí jej, jinak ještě provede preload. To může způsobit malý výpadek v počtu vykreslovaných snímků, zvláště u velkých obrázků. Důvodem je nutné načtení z disku. Proto není příliš doporučováno za běhu používat velké necachované zdroje.

#### 2.4.8. Animace objektů

Animace v knihovně MGEAN je chápána jako posloupnost obdélníků, které určují pozice jednotlivých snímků v povrchu a je popsána strukturou `ANIMATION`.

```
typedef struct ANIMATION
{
    SDL_Rect *anim_array;
    SDL_Rect rect;
    Uint32 count;
    Uint32 delay;
    Mix_Chunk *music;
} ANIMATION;
```

**anim\_array** – pole animačních obdélníků, což jsou oblasti ve zdrojovém obrázku tvořící jednotlivé snímky animace.

**rect** – velikost jednoho obdélníku v animaci, kde není použito `anim_array`.

**count** – počet obdélníků v animaci.

**delay** – doba, po kterou bude každý snímek zobrazen.

Každou animaci je nejprve potřeba vytvořit, k tomu slouží funkce:

```
ANIMATION *MG_make_animation (const char *s, int count,
                               Uint32 delay, SDL_Rect * r)
```

Tato funkce vytvoří animaci pojmenovanou `s` (je opět cachována). Proměnná `count` je počet animačních obdélníků, `r` je ukazatel na obdélník představující rozměry a offset prvního animačního obdélníku. Při animování se posouvá obdélník doprava o svoji šířku, pokud animace přesáhne `count` je resetována na první snímek. Proměnná `delay` určuje dobu zobrazení snímku animace.

Pokud potřebujeme získat objekt animace z cache, poslouží následující funkce.

```
ANIMATION *MG_get_animation (const char *string)
```

Vrací animaci pojmenovanou `string`, NULL pokud taková animace není zařazena v cache.

Nejdůležitější funkcí, která přiřadí spritu animaci je tato:

```
void MG_set_animation (SPRITE * s, const char *string)
```

Přiřadí spritu `s` animaci pojmenovanou `string`.

Pro ilustraci použití animací v knihovně MGEAN uvedu jednoduchý příklad, který animuje rotaci větrníku. Pro představu je přiložen obrázek `vetrnik.png`.



Obrázek 2. Animace větrníku - `vetrnik.png`.

```
SDL_Rect anim_rect = {0,0,32,32};
ANIMATION * a = MG_make_animation( "rotation" , 16 ,
                                   40 , &anim_rect );
SPRITE * vetrnik = MG_make_sprite( MG_get_img("vetrnik.png") ,
                                   NULL );
MG_set_animation( vetrnik , "rotation" );
```

Tento kód vytvoří animaci pojmenovanou `"rotation"`, jeden obrázek v animaci bude mít rozměry 32 na 32 pixelů a bude posunut na pozici 0,0 ve zdrojovém obrázku. Poté se vytvoří sprite `vetrnik`, jeho obrázkem bude `vetrnik.png`. Nakonec se nastaví animace `rotation`. Obrázek `vetrnik.png` by měl mít rozměry 512 na 32 pixelů nebo více v obou směrech. Pokud bude tento obrázek vypadat jako větrník z obrázku 2. a bude rozdělen do šestnácti obdélníků vedle sebe, v nichž bude větrník pootočen vždy o 22.5 stupně, dosáhneme efektu rotace podle své osy. Tuto funkcionalitu lze zajistit i jinak než pomocí animací - použitím transformací.

### 2.4.9. Fyzikální zákony v knihovně MGEAN

V knihovně MGEAN je zabudovaná jednoduchá Newtonova fyzika. Objektům typu SPRITE umožňuje nastavovat hodnoty jako síla, rychlost a hmotnost, kde všechny položky až na hmotnost jsou jednotky vektorové. Tomu odpovídá i realizace ve struktuře pro fyziku. Účinky fyzikálních zákonů jsou vypočítávány ve dvou krocích. Prvním je vždy aktualizace zrychlení na základě působící síly podle všeobecně známého vzorce.

$$a = \frac{F}{m}$$

Kde  $a$  je zrychlení  $F$  je síla a  $m$  je hmotnost. Poté dojde k výpočtu aktuální rychlosti na základě vypočteného zrychlení, k tomu je použit zřejmý vzorec.

$$v = v_0 + at$$

Kde  $v_0$  je počáteční rychlost,  $a$  je zrychlení a  $t$  je čas. Samotný přesun objektu je ovšem počítán z lineárního vzorce pro rovnoměrný pohyb.

$$s = vt$$

Což by bylo obecně špatně, protože vztah pro výpočet dráhy zrychleného pohybu vypadá jinak:

$$s = v_0t + \frac{1}{2}at^2$$

Ovšem ve hrách se obvykle používá poměrně malé kvantum času pro vykreslení jednoho snímku (v mé implementaci Worms je to 0,03 sekundy), tudíž jsem složku

$$\frac{1}{2}at^2$$

zanedbal.

Struktura VECTOR představuje dvojrozměrný vektor.

```
typedef struct VECTOR
{
    float x;
    float y;
} VECTOR;
```

PHYSICS je struktura popisující fyzikální stav objektu. Proměnná  $m$  je hmotnost objektu. Vektor  $f$  popisuje sílu působící na objekt, vektor  $v$  popisuje aktuální rychlost.

```
typedef struct PHYSICS
{
    float m;
    VECTOR f;
    VECTOR v;
} PHYSICS;
```

Výpočet fyziky nejvíce závisí na uplynulém čase mezi dvěma snímky, k získání této hodnoty slouží funkce `MG_get_system_dt`. Lze ji použít také například pro implementaci časovače.

```
int MG_get_system_dt ()
```

Následující funkce jednoduše modifikují proměnnou `physics` ve spritu. Pokud má sprite hmotnost nulovou, nebude na něj aplikována žádná fyzikální transformace.

```
void MG_set_power (SPRITE * s, VECTOR * v)
void MG_set_speed (SPRITE * s, VECTOR * v)
void MG_set_weight (SPRITE * s, float m)
```

Hry také obvykle vyžadují nějakou fyziku prostředí, typicky to může být gravitace nebo vítr. K nastavení těchto hodnot se používá funkce `MG_set_environment_physics` definovaná následovně.

```
void MG_set_environment_physics (VECTOR env_physics)
```

Parametr `env_physics` představuje zrychlení udílené každému objektu v daném směru.

#### 2.4.10. Řízení objektů - základní zprávy

Zprávy v knihovně MGEAN patří k jedné z nejpodstatnějších funkcionalit. V této kapitole představím základní zprávy, které posílá knihovna MGEAN spritům, a jak na ně reagovat. Při každém zavolání `MG_draw_stack` může za určitých podmínek každý sprite obdržet od knihovny tyto zprávy:

**on\_iter** – tuto zprávu obdrží každý sprite při každém zavolání `MG_draw_stack`, na tuto zprávu musí být nainstalován callback ve tvaru `void (*func)(SPRITE * self)`.

**on\_collide** – je zaslána každému objektu `SPRITE`, který koliduje na úrovni obdélníků s jiným spritem. Callback je funkce ve tvaru `void (*func)(SPRITE * self , SPRITE * collision_with)`.



**on\_destroy** – objekt, který byl vyřazen z vykreslovacího zásobníku obdrží tuto zprávu, typicky používanou pro uživatelské uvolnění prostředků. Callback, nainstalovaný na tuto zprávu, nesmí vytvářet jiný sprite. Destrukční funkce musí být opět ve tvaru `void (*func)(SPRITE * self)`.

Pro nainstalování callbacku na zprávu slouží funkce `MG_install_callback`

```
void MG_install_callback (SPRITE * s, const char *key,  
                          void (*function))
```

Chování této funkce nejlépe zdokumentuje následující příklad, který vytvoří neviditelný sprite kocka, která bude na standardní výstup „mnoukat“.

```
#include "MG_lib.h"  
  
void mnoukni( SPRITE * self )  
{  
    printf("mnouk\n");  
}  
  
int main()  
{  
    MG_init_video(0,0,0);  
    SPRITE * kocka = MG_make_sprite( NULL , NULL );  
    /* instalace callbacku na zprávu on_iter  
       - zavolá se při každé iteraci      */  
    MG_install_callback( kocka , "on_iter" , mnoukni );  
    int i = 10;  
  
    while( i-- ){  
        MG_draw_stack(1000);  
    }  
    MG_quit();  
    return 0;  
}
```

Tento kód vyvolá 10 výstupů do konzole s nápisem `mnouk`. To je způsobeno tím, že jsme iterovali desetkrát a při každé iteraci byla vyvolána zpráva `on_iter` pro objekt `kocka`, která má na zprávu `on_iter` nainstalovanou funkci, která vypíše jednoduchý text na standardní výstup. Rozestup mezi dvěma výstupy bude vždy 1000 milisekund, tedy jedna sekunda, běh celého programu bude trvat 10 sekund.

#### 2.4.11. Odchytávání vstupu z klávesnice a myši

MGEAN abstrahuje přímý přístup ke klávesnici a myši. To znamená, že poskytuje aktuálně zmáčkнутé klávesy na klávesnici nebo myši, ale nesleduje změny na těchto vstupních zařízeních. K odchytávání vstupu lze také použít události ze SDL, ty jsou v MGEANu dostupné ve struktuře MACHINE pod proměnnou `events`.

Funkce pro zjištění okamžitého stavu klávesy je zavedena takto:

```
int MG_key_down_p (int key)
```

Testuje, zda je klávesa `key` zmáčknutá. Jako vstup bere index klávesy. Tyto indexy definuje SDL, jsou to definice jako `SDLK_ESCAPE` nebo `SDLK_ENTER`. Vrací nenulovou hodnotu pokud je klávesa zmáčknutá, jinak vrací 0.

Funkce vracející souřadnice myši pro x-ovou a y-ovou souřadnici vypadají následovně:

```
int MG_get_mouse_x ();  
int MG_get_mouse_y ();
```

Vrácené souřadnice jsou v mezích 0 a šířky, respektive výšky obrazovky.

Funkce, která okamžitě zjistí všechna zmáčknutá tlačítka myši, je zavedena následovně:

```
Uint8 MG_get_mouse_state ()
```

Výstup je shodný s výstupem funkce `SDL_GetMouseState`.

#### 2.4.12. Uživatelské parametry a callbacky spritu

V knihovně MGEAN je možné instalovat callbacky na předdefinované zprávy, jak bylo vysvětleno. Každý sprite, ale může obsahovat i uživatelské zprávy, které mohou rozšiřovat jeho funkcionalitu. Sprite dále může obsahovat uživatelské klíče, což jsou speciální proměnné přiřazené entitě a slouží jako slot v „objektu“ `SPRITE`. Systém zasílání zpráv je ve hrách velmi užitečný, jako modelový příklad si můžeme představit ničení několika různých objektů v herní scéně. Pomocí knihovny MGEAN lze všem ničeným objektům zaslat uživatelskou zprávu. Taková zpráva může být například `on_damage`, přičemž vůbec nezáleží, jakým způsobem je tato zpráva zpracována. V mnoha případech nemusí být zpracována vůbec, pokud není nainstalován příslušný callback. Tento přístup je velmi vstřícný k rozšíření, neboť stačí doimplementovat callbacky pro objekty podobné kategorie.

Dalším velmi užitečným rozšířením každého spritu mohou být uživatelské klíče. Ty představují za běhu definované proměnné přiřazené ke spritu. Praxe ukázala, že sprity často vyžadují uchování nějakého stavu napříč herními snímky.

Uživatelské klíče lze použít třeba pro vytvoření časovače, ale i pro mnohem komplexnější úlohy. Například lze pomocí tohoto principu vybudovat objektový systém GUI. Klíče pak představují sloty objektu.

Výhoda tohoto přístupu oproti klasickému objektově orientovanému programování, spočívá v instalaci metod a členských proměnných za běhu programu. Tento přístup má nejblíže k tzv. prototypovému objektově orientovanému programování. Získávání nedefinovaných slotů a zasílání neimplementovaných zpráv jsou bezpečné operace, při pokusu o získání nenaplněného slotu je vráceno NULL, v případě poslání neimplementované zprávy se nestane nic.

Zprávy i uživatelské proměnné jsou uloženy v proměnné struktury `SPRITE` nazvané `args`. Jedná se o asociativní seznam, avšak i zde jsou definovány speciální funkce operující s touto funkcionalitou. Jmenovitě jsou to:

```
void MG_install_callback (SPRITE * s, const char *key,
                          void (*function))
```

Funkce `MG_install_callback` nainstaluje spritu `s` klíč `key` s callbackem `function`. Od této chvíle se při vyvolání zprávy `key` vyvolá funkce `function` s parametrem `s`.

```
void * MG_send_msg( SPRITE * s , const char * msg )
```

Posílá zprávu `msg` objektu `s`, jako parametr je předán `s`. Pokud sprite `s` neobsahuje obsluhu zprávy anebo je funkce NULL, nestane se nic.

```
void MG_set_key( SPRITE * sprite , const char * key , void * val )
```

Nastaví klíč `key` objektu `sprite`, jeho hodnota bude `val`. Pokud již klíč `key` existuje, přepíše se jeho hodnota na `val`. Tato funkce se může použít k zneplatnění klíčů i zpráv, dosazením za parametr `val` hodnotu NULL.

```
void * MG_get_key( SPRITE * s , const char * key )
```

Je funkce vracející hodnotu příslušející v objektu `s` ke klíči `key`. Pokud takový klíč neexistuje, vrátí NULL.

Pro demonstraci uživatelských zpráv a klíčů rozšířím příklad z podkapitoly o základním řízení objektů.

```
#include "MG_lib.h"
```

```
void mnoukni( SPRITE * self )
{
    printf("mnouk: %d\n" ,
           (int)MG_get_key( self , "iterace_ita" ));
}
```

```

}

void iterace_kocky( SPRITE * self )
{
    /* pošle zprávu kočce "on_mnouk" */
    MG_send_msg( kocka , "on_mnouk" );
    /* sniž hodnotu uživatelské proměnné "iterace_ita" o 1 */
    int iterace = (int)MG_get_key( self , "iterace_ita" ) - 1;
    /* přenastav klíč "iterace_ita" */
    MG_set_key( self , "iterace_ita" , (void*)iterace );
}

int main()
{
    MG_init_video(0,0,0);
    SPRITE * kocka = MG_make_sprite( NULL , NULL );
    MG_install_callback( kocka , "on_iter" , iterace_kocky );
    int i = 10;
    /* nastaví klíč kočky "iterace_ita" na i */
    MG_set_key( kocka , "iterace_ita" , (void*)i );
    MG_install_callback( kocka , "on_mnouk" , mnoukni );

    while( i-- ){
        MG_draw_stack(1000);
    }
    MG_quit();
    return 0;
}

```

Pokud je tento program spuštěn, způsobí deset výpisů na standardní výstup. Každý výpis bude obsahovat mnouknutí a číslo od 10 do 1.

#### 2.4.13. Práce s konfiguračními soubory

V MGEANu je obsažen jednoduchý parser textově zapsaných seznamů. MGEAN umožňuje také zpětné převedení na text. Seznam v textové podobě může být i asociativní. Jako příklad uvádím konfigurační soubor, který nese informace o obrazovce, její velikosti a barevné hloubce. Další položkou je velikost herního světa, název hry a jména hráčů.

```

(:screen (:resx 1024 :resy 768 :bpp 32)
:game_world (:resx 2048 :resy 2048)
:game_name  foo_game
:players    (foo bar baz))

```

Výhodou tohoto zápisu je stručnost a minimalističnost, avšak konfigurační soubor v tomto tvaru dokáže zachytit celou řadu situací.

## 2.5. Implementace hry

V následujících kapitolách se budu zabývat principy a postupy, kterými jsem hru programoval za použití knihovny MGEAN a SDL. Hra se skládá ze dvou celků, prvním je menu. To se stará o nastavení hry a týmů před samotnou hrou, zajišťuje také ukončení programu. Druhou, větší část, představuje implementace hry a její logiky.

### 2.5.1. První kroky s projektem

MGEAN je knihovna postavená na multiplatformní knihovně SDL, proto by mělo být její použití snadné na jakémkoli systému včetně Windows. Nativním prostředím pro práci s knihovnou je však GNU/Linux, který obsahuje standardní nástroje pro vývoj aplikací v jazyce C. Dále musí být nainstalovány vývojové balíky pro SDL knihovny. Pro správný průběh kompilace hry a knihovny jsou zapotřebí vývojové i binární knihovny SDL, SDL\_image, SDL\_mixer, SDL\_gfx, SDL\_ttf. Pro zprovoznění kompilačního procesu na distribuci Ubuntu 10.10 stačí zadat tento konzolový příkaz:

```
$ sudo apt-get install libsdl-image1.2-dev libsdl-mixer1.2-dev  
libsdl-ttf2.0-dev libsdl1.2-dev libsdl-gfx1.2-dev
```

Na MS Windows by kompilace měla fungovat přes prostředí mingw. Na platformě GNU/Linux lze však také provádět cross kompilaci na Windows, důsledkem toho je možnost produkovat spustitelné soubory .exe z pohodlí linuxového prostředí. K tomu je však zapotřebí sada nástrojů mingw32 pro linux. V Ubuntu lze opatřit tento balík pomocí příkazu:

```
$ sudo apt-get install mingw32
```

K sestavení hry se používá nástroj Make. Soubor Makefile pro libovolnou hru využívající MGEAN pak může vypadat stejně jako v případě Bazooworms. Makefile lze najít v adresáři src. Kompilační proces dále vyžaduje, aby byly přítomny adresáře bin a resources ve stejném adresáři jako je Makefile. Do adresáře bin se provede sestavení hry včetně zkopírování datových souborů z adresáře resources. Pokud jsou přítomny všechny vývojové knihovny, je možné s úspěchem projekt sestavit napsáním příkazu:

```
$ make
```

Výsledný kompilát bude obsažen v adresáři `./bin/posix/` i s obsahem adresáře `./resources/`, kam jsem umístil prostředky nutné k běhu programu. Pokud máme správně nakonfigurován i balík mingw32, je možné provést křížovou kompilaci na platformu MS Windows pomocí příkazů:

```
$ make clean
$ make win32
```

Veškeré podpůrné knihovny nutné pro sestavení i běh exe souboru jsou umístěny uvnitř knihovny MGEAN a po sestavení se zkopírují do `./bin/win32/`. U dalších unixových platforem by kompilace měla být obdobná jako na platformě GNU/Linux.

### 2.5.2. Návrh projektu

Na začátku každé nově vznikající hry je nutné určit herní principy a pravidla. To mi bylo v případě Bazooworms ulehčeno díky předloze v podobě Worms z roku 1995. Psaní programu jsem rozdělil na dvě etapy. První bylo vytvoření herního světa se základními pravidly a herními mechanizmy. Po dokončení významných prací na herním světě jsem se soustředil na menu. V rámci tvorby těchto dvou partií programu jsem narazil na několik problémů, které v následujících kapitolách popíšu.

### 2.5.3. Inicializační a deinicializační proces

Inicializace představuje první kroky programu, který se právě spustil. V knihovně MGEAN představuje celý inicializační proces volání `MG_init_video`. Tato funkce se řídí principem „buď vše důležité, nebo nic“. To v praxi znamená, že pokud se povede zavést video a ne zvuk, je vše v pořádku, ovšem pokud se nepodaří zinicializovat například knihovnu SDL, povede to na ukončení celého programu. Bazooworms dále při inicializaci provádí některé další zaváděcí procedury, jako je zjištění nejlepšího možného rozlišení uživatelského monitoru, načtení herních prostředků nebo zjištění parametrů z konzole.

Úkolem deinicializace je korektní ukončení programu, tudíž by se aplikace vždy měla postarat o správné uvolnění zabraných prostředků. Toto však v případě Bazooworms téměř automatizovaně řeší knihovna MGEAN. Výjimkou jsou uživatelsky alokované prostředky, ty jsou v Bazooworms uvolněny při vyvolání zpráv `on_destroy`.

### 2.5.4. Načítání prostředků a jejich správa

K načítání prostředků, čímž se obvykle u her myslí obrázky nebo hudba, existují rámcově dva přístupy. Jeden přístup vsází na politiku, kdy se nejdříve načte

vše, aby posléze už nebylo potřeba načítat prakticky nic. Druhý přístup je dynamičtější a hodí se spíše pro rozsáhlé herní světy, kdy se načítají prostředky, jakmile jsou potřeba. Bazooworms vsadil na politiku načtení všeho hned na začátku, což zvyšuje jeho odolnost vůči zásahům uživatele do herních dat za běhu, jako je vymazání nebo modifikace obrázků.

### 2.5.5. Fáze hry

V tomto odstavci bude řeč o základním rozdělení Bazooworms na menu a herní prostředí, což nazývám fázemi - anglicky *stages*. Mezi každou fází je nutné udělat některé úkony, jejichž důsledkem bude vyčištění předchozí fáze a inicializování jiné. Z pohledu knihovny MGEAN ve většině případů postačí volat `MG_drawing_stack_free`, což vyčistí vykreslovací zásobník a provede některé vnitřní i uživatelské dealokace.

Po deinicializaci fáze můžeme přistoupit k inicializaci jiné. Každá fáze obsahuje nějaký vstupní bod, což je funkce, která naplní vykreslovací zásobník novými sprity. Například v případě přepnutí do menu jsou vykresleny základní tlačítka a pozadí. Mezi fázemi hry lze v Bazooworms také předávat argumenty. Toho se využívá pro informování menu o vítězi, anebo naopak pro předání nastavení hry. Pro předávání argumentů mezi fázemi v Bazooworms slouží funkce `void WRMS_stage_set_args( void * arg )` a pro vstup do jiné fáze je to `int WRMS_enter_stage( WRMS_STAGES stage )`. Úkolem funkce pro vstup do jiné fáze je tato posloupnost operací.

- Vyčistit vykreslování.
- Získat parametry.
- Nastavit požadovanou fázi hry na aktuální.
- Zavolat vstupní bod fáze s předanými parametry.
- Vynulovat parametr.

### 2.5.6. Sestavení herního světa

Bazooworms je z boku viděná hra, která obsahuje několik pevných elementů tvořících herní svět. Tyto objekty jsou poskládány ve vykreslovacím zásobníku tak, aby se korektně překrývaly. Prvním dobře patrným objektem v Bazooworms je pozadí, které je představováno obrázkem, konkrétně jde o `background.png`. Pozadí je zařazeno na dno vykreslovacího zásobníku, tudíž je nejnižší z pohledu uživatele. Pozadí je dále překryto herní mapou, což je obrázek, po němž se červové pohybují, v případě Bazooworms je to obrázek `tropical.png`. Ten je vyplněn z velké části barvovým klíčem, tyto plochy jsou prostupné, zbylé plochy naopak neprostupné a tvoří dojem povrchu zeminy. Dalšími prvky v herním světě je

moře a mrak. Moře způsobuje červovi úmrtí při pádu do něj. Mrak indikuje směr a sílu větru podle rychlosti, kterou pluje po obloze.

### 2.5.7. Nastavení parametrů fyziky

V Bazooworms se vyskytuje gravitace a boční vítr stejně jako v originální hře. Tuto funkcionalitu však zajišťuje knihovna MGEAN skrze volání `MG_set_environment_physics`, které každému objektu s nenulovou hmotností udělí konstantní zrychlení ve směru vektoru, který předáme parametrem. Hodnota pro gravitaci je v Bazooworms nastavena v souboru `config.h` následovně.

```
#define ENV_PHYSICS vector( 0.0f , 0.00025f )
```

Hodnoty větru jsou generovány náhodně v pevně daných mezích.

### 2.5.8. Rozvržení entit

Herní entitou se myslí objekt, který podléhá pravidlům hry a principu fungování světa. V případě plošinové hry si lze představit pod entitou nejen postavičku ovládanou hráčem, ale třeba i nepřátele nebo různé další elementy ovlivňující hru. Při návrhu entity musíme uvážit, co ji v konkrétním případě popisuje. V případě MGEAN je entita reprezentována instancí `SPRITE`. Entitu je vhodné popsat další strukturou, jejíž instanci potom nastavíme do nějakého klíče spritu. V Bazooworms popisuje červa tato struktura:

```
typedef struct WRMS_worm{  
    PHYS_STATES phys_state; /* fyzikální stav - vzduch, země */  
    WORM_STATES worm_state; /* animační stav - pohybuje se, stojí */  
    int direction; /* směr natočení červa */  
    int space_toggle; /* signalizuje natahování zbraně */  
    WRMS_worm_player player_spec; /* struktura popisující hráče */  
}WRMS_worm;
```

Pro vytvoření entity by měla existovat funkce, která entitu daného typu nejen vytvoří, ale i nastaví do výchozích hodnot. V případě červa se v Bazooworms používá funkce `WRMS_worm_make` a vrací ukazatel na `SPRITE`, který popisuje červa, protože má v klíči `worm_desc` nainstalovanou instanci struktury `WRMS_worm`. Ve funkci vytvářející entitu dále nastavuji parametr `type` spritu na nějakou konstantu, která umožňuje rychlé rozpoznání typu entity. To je výhodné například při rozpoznání kolize s objektem daného typu. Při zakládání objektu nastavuji všechny hodnoty na implicitní často za použití funkce `memset`.



### 2.5.9. Oživení entit

Oživením entit mám na mysli konstrukci callbacku pro reakci na zprávu `on_iter`. Musíme mít na paměti, že tento callback se vyvolá při každé iteraci hry. Tudíž se zde vždy pracuje se vstupem z klávesnice nebo myši a aktualizují se stavy entit. Pro ilustraci si můžeme představit situaci, kdy je postavička v klidu a má na základě vstupu změnit svoji animaci v běh a začít se posouvat podle definice běhu. Jistým problémem je výběr správné animace a obrázku, to jsem v Bazooworms vyřešil jednotným tvarem jmen souborů, které vystihují směr natočení a zvolenou akci pro animaci. Konkrétní názvy souborů s grafikou jsou pak ve tvaru:

`JménoEntity_Animace_Směr.Přípona`

Tyto řetězce pak lze vybudovat na základě stavu červa.

Callback nainstalovaný na zprávu `on_iter` by také měl komunikovat s herními pravidly. V případě Bazooworms se tato funkce dotazuje pravidel, zda je tato entita na tahu. Pokud je na tahu, dojde k aplikaci vstupu z klávesnice nebo od umělé inteligence. V Bazooworms dále tato funkce aktualizuje uplynulý čas hry červa.

### 2.5.10. Interakce entit se světem

Obvykle nejsložitější technickou součástí každé 2D hry bývají kolize, a to jejich rozpoznání a následná reakce na ně. Kolize je jev, při kterém jsou dva objekty ve stavu, kdy je jejich geometrický průnik nenulový. Jde o to, jak je tento geometrický průnik zaveden. V případě knihovny MGEAN je přítomná funkcionality, která rozpozná průniky obdélníků spritů. Pokud je tento průnik nenulový, knihovna zašle zprávu `on_collide` oběma zúčastněným objektům. To v mnoha případech stačí pro jednoduché rozpoznání kolize, například u plošinových her bude tento princip fungovat velmi dobře. Ovšem pokud bude terén složitější útvar než obdélník, budeme muset zapřemýšlet nad složitějšími algoritmy. Bazooworms je hra, ve které je terén tvořen obrázkem, a tak je nutné přistoupit k alternativnímu algoritmu. Mezi vývojáři her je dobře znám algoritmus „pixel perfect collision detection“. Což je jednoduchý algoritmus který spočítá průnik dvou obdélníků, a následně iteruje nad pixely patřícími do tohoto průniku obou obrázků. Zde zjišťuje, zda jsou na dané pozici dva neprůhledné pixely. Pokud nalezne dva neprůhledné pixely na stejné souřadnici, skončí úspěchem, jinak neúspěchem. V pseudo-kódu by zápis jednoduché pixel perfect kolize bez výpočtu průniku podle zdroje [4] mohl vypadat následovně.

```
function pixel_perfect_collision( obrazek1 , obrazek2 , prunik )
{
    for( y = prunik.y ; y < prunik.h ; y++ )
```

```

{
    for( x = prunik.x ; x < prunik.w ; x++ )
    {
        if((get_pixel(obrazek1 , x , y)!=TRANSPARENT_COLOR)&&
            (get_pixel(obrazek2 , x , y)!=TRANSPARENT_COLOR))
            return true;
    }
}
return false;
}

```

Z tohoto kódu je patrné, že rozhodnutí o kolizi je snadné. Složitější je samotná reakce na zjištěnou kolizi. Problém spočívá v nalezení pozice entity, kdy je „usazena“ na povrchu terénu. Toto jsem v Bazooworms vyřešil nalezením pixelu terénu, který koliduje s červem a je současně nejbližší středu červa. Červa poté posunu tak, aby se tento pixel nacházel na obvodu kružnice, jejíž průměr je stejný jako šířka červa. V důsledku zaokrouhlovacích chyb může na strmém svahu docházet k drobnému sesouvání ze svahu, což je v tomto případě žádoucí efekt.

U entit, jako jsou projektily zbraní, není zapotřebí problém reakcí řešit, protože při nárazu změni svůj stav ve výbuch, případě zmizí úplně. Tudíž postačí pouze rozpoznání srážky. V Bazooworms navíc bazukové projektily ničí terén, což provádím pomocí vykreslení kruhu do povrchu terénu. Barva kruhu je barvový klíč, je tedy neviditelný. Ke kreslení používám nízkoúrovňová volání - `SDL_LockSurface` a `SDL_UnlockSurface`, v kombinaci se zápisem do pole pixelů terénu.

#### 2.5.11. Ovládání pomocí klávesnice

V kapitolách o knihovně MGEAN jsem naznačil, že existují dva přístupy odchytávání vstupu z klávesnice. První je založen na událostech a druhý na přímém přístupu k bufferu klávesnice. V herní fázi hry Bazooworms je ke zjišťování stavu klávesnice použit přímý přístup. Důsledkem tohoto je, že objekt může okamžitě zjistit, zda je daná klávesa stisknuta nebo nikoliv. Důvod pro volbu přímého přístupu je skutečnost, že stav libovolné klávesy může být zjištěn v každé iteraci hry. V knihovně MGEAN se pro zjištění zmáčknuté klávesy používá funkce `MG_key_down_p`, která vrací stav klávesy. Na ten se reaguje obvykle v podmínkách typu `if`. Jako krátký demonstrační příklad reakce na klávesy uvedu upravený příklad z Bazooworms.

```

if( MG_key_down_p( SDLK_RIGHT ) ) {
    out.direction = RIGHT;
    out.action = MOVING;
}

```

```

}
else if( MG_key_down_p( SDLK_LEFT ) ) {
    out.direction = LEFT;
    out.action = MOVING;
}
else if( MG_key_down_p( SDLK_RETURN ) )
    out.action = JUMPING;
else
    out.action = IDLING;

```

První podmínka testuje zmáčknutí šipky vpravo. Pokud je zmáčknuta, dojde k aktualizaci objektu na pohyb vpravo. Obdobně je to v případě šipky vlevo. Pokud je zmáčknuta klávesa enter, program nastaví aktivitu na skok, přičemž směr zachovává. Pokud není zmáčknuta žádná z těchto kláves, aktivita je nastavena na nečinnost.

### 2.5.12. Ovládání pomocí myši

Práce s myší je principiálně velmi podobná práci s klávesnicí. Stejně jako u klávesnice jsou přítomny dva způsoby přístupu. Uvnitř hry jsem využil přímý přístup a pro menu jsem zvolil přístup skrze SDL události. Ty mají tu vlastnost, že se neopakují, a že událost hra dostane i tehdy, pokud uživatel kliknul a pustil myš někdy během vykreslení předchozího snímku. Připomenu, že přímý přístup k myši v knihovně MGEAN je možný přes volání `MG_get_mouse_x`, `MG_get_mouse_y` a `MG_get_mouse_state`. Práce s událostmi je poněkud obtížnější. Události jsou umístěny v objektu `MACHINE` v atributu `events`, který lze obdržet pomocí volání `MG_get_machine`. Atribut `events` je spojový seznam, který obsahuje události, které se vyskytly od posledního volání `MG_draw_stack`. Události jsou pak instance `SDL_Event` definované v SDL. Vyřízení událostí pak provádím projitím seznamu a zasláním příslušných zpráv:

```

LIST * l = MG_get_machine()->events;
while( l )
{
    SDL_Event ev = *((SDL_Event*)l->value);
    /* filtr kliknutí */
    if( (ev.type == SDL_MOUSEBUTTONDOWN)
        && pixel_in_rect_p( self->dest , mouse ) )
    {
        MG_send_msg( self , "on_click" );
    }
    else if( ev.type == SDL_KEYDOWN )
        MG_send_msg( self , "on_keydown" );
}

```

```

    l = l->node;
}

```

Tento výsek kódu projde všechny události, a pokud některá splňuje podmínku, že jde o kliknutí, je příslušnému spritu zaslána zpráva `on_click`. V případě stisknutí libovolné klávesy se obdobně zašle zpráva `on_keydown`.

### 2.5.13. Umělá inteligence

Umělá inteligence může být dalším způsobem ovládání entit, její implementace je však poměrně komplexní problém. Moje implementace je založena na principu nalezení velmi přesné dráhy letu projektilu. V praxi to vypadá tak, že simulují všechny možné úhly střely a všechny možnosti natažení intenzity zbraně. Simulace, která dopadne s nejvyšším skóre, je aplikována na výsledný výstřel. Pokud však skóre není dostatečně velké, pokusí se postavička o pohyb. Tento pohyb by měl postupně vylepšit pozici červa natolik, že bude možné dobře zacílit nepřítele. V Bazooworms se červ přednostně pohybuje směrem k nejbližšímu hráči z nepřátelského týmu. Může se však stát, že pohyb nepomůže, v takovém případě se pokusí červ odstranit překážku. Pokud je to ale nevýhodné, počká na konec časového limitu. Po dokončení každého výpočtu umělé inteligence, dojde k nastavení entity stejným způsobem, jakým se to děje v případě lidského hráče. To zajistí férové chování hráče s umělou inteligencí.

### 2.5.14. Rozvržení zbraní

Každá zbraň musí implementovat některé základní funkce, které jsou všem zbraním společné. V některých jazycích by se tyto funkce mohli označit jako virtuální. Jsou to:

**on\_draw** – funkce volaná při překreslení zbraně.

**sighting** – volá se při zaměřování.

**fire** – funkce vyvolaná při vystřelení.

**simulate\_fire** – heuristická funkce sloužící umělé inteligenci k nalezení nejlepšího zaměření.

**compute\_damage** – spočítá poškození udělené zbraní.

V případě potřeby rozšířit Bazooworms o další zbraň stačí pouze tyto funkce doimplementovat.

### 2.5.15. Společný algoritmus pro výpočet střelby

Společný algoritmus pro nalezení nejlepšího zaměření zbraně, je relativně snadný. Jeho úlohou je projít všechny možné vektory zaměření a všechna možná natažení zbraně a vyzkoušet pro každou dvojici jak dopadne heuristická simulace - funkce `simulate_fire`. Ta je implementována každou zbraní obecně nezávisle a měla by být poměrně rychlá. Jakmile známe simulaci, vypočteme zranění červů v okolí vypočítaného dopadu, které zbraň udělila. Jako nejlepší volbu bereme tu, která skončila s nejvyšším zraněním. Zranění na straně aktuálně hrajícího týmu jsou započítávána jako záporné hodnoty. Vzhledem k tomu, že funkce pro simulaci je obvykle heuristická, může se stát, že náboj nedopadne přesně podle očekávání. To se může stát zejména kvůli nepřesnému výpočtu kolize. Moje aktuální implementace zaměřovací funkce je k nalezení v souboru `ai.c`. Zapsáno pseudo-kódem může tento algoritmus vypadat následovně.

```
function get_best_fire( worm , gun )
{
    best_damage = -INFINTY;
    foreach( gun.angles as angle )
    {
        foreach( gun.intensity as intensity )
        {
            impact = gun.simulate_fire( angle , intensity );
            damage = gun.compute_damage( impact );
            best_damage =
                ( best_damage < damage ? damage : best_damage );
        }
    }
    return best_damage;
}
```

### 2.5.16. Heuristika bazuky

Simulační heuristika bazuky je založena na výpočtu bodu dopadu při vrhu šikmém. Využívá se přitom výpočtů fyziky z knihovny MGEAN. Princip je založen na simulaci letu náboje. Střela se testuje ve smyčce dokud nedojde ke střetu s terénem nebo k opuštění herního světa. Náboj je z důvodu úspory výpočetních prostředků představován jediným bodem. Kolizi v tomto případě rozpoznám velmi snadno - přečtením konkrétního pixelu. Pokud se náboj srazil se zemí nebo vyletěl z herního světa, je tento bod vrácen. Použití jediného bodu však může způsobit nepřesný výpočet bodu dopadu, a tudíž neefektivní výstřel červa.

Funkci pro výpočet zranění bazuka implementuje pomocí vztahu přímé úměry. To znamená, čím blíže náboj dopadne, tím větší zranění červ obdrží. Bazuka udě-

luje při vzdálenosti 0 pixelů od středu červa zranění o 100 bodech, při vzdálenosti větší jak 80 pixelů je zranění nulové.

### 2.5.17. Herní pravidla

V této kapitole bude řeč o implementaci herních pravidel Bazooworms. Herní pravidla jsou soubor struktur a funkcí, které se provádějí každý snímek a kontrolují stav hry z hlediska férovosti. Například sem spadá kontrola konce hry nebo výměna hráčů na tahu. S pravidly je spjato několik funkcí. Funkce pro inicializaci pravidel je `WRMS_rules_init`, která zařídí nastavení pravidel do výchozího stavu. Funkce `WRMS_team_make` je funkce, která vytváří nový tým, na nějž je možné navazovat červy. Struktura pravidel, která popisuje stav hry je deklarována takto:

```
typedef struct WRMS_game{
    LIST * initial_worms; /* červové při inicializaci */
    LIST * worms; /* seznam žijících červů */
    LIST * teams; /* fronta týmů - tým na vrcholu hraje */
    Uint32 time_used; /* čas použitý hrajícím červem */
    int waiting_for_detonation_p; /* zda se čeká na výbuch */
    int teams_count; /* iniciální počet týmů */
}WRMS_game;
```

Zajímavou proměnnou je `teams`, která se v tomto případě chová jako úložiště fronty. Tímto principem je řešena záměna týmů. Hrající tým je na prvním místě ve frontě, jakmile dojde k prohození týmu, ocitne se na posledním místě. Při změně týmů se provede operace odebrání a následně přidání odebraného týmu. Struktura popisující tým vypadá následovně.

```
typedef struct WRMS_team{
    LIST * teammates; /* fronta spoluhráčů */
    int initial_teammates_count; /* iniciální počet spoluhráčů */
    SDL_Color c; /* barva týmu */
    char * team_name; /* jméno týmu */
    WRMS_player_types type; /* lidský nebo AI hráč */
}WRMS_team;
```

Uvnitř týmů, se nachází další fronta spoluhráčů, kteří se střídají stejným způsobem, jako je tomu u týmů. Aktuální je opět na prvním místě fronty.

Rozpoznání konce hry realizují tak, že zjistím počet týmů, které mají nenulový počet hráčů. Pokud má nejvýše jeden tým nenulový počet červů, je hra u konce. Vítězný tým je ten, který má jako jediný nenulový počet spoluhráčů. Pokud žádný tým nemá žijícího červa, hra skončila bez vítěze - remízou.

### 2.5.18. Herní ukazatel

HUD – *the heads-up display* představuje herní ukazatel, který informuje v reálném čase uživatele o současném stavu hry. V Bazooworms je představován informační oblastí při horním kraji obrazovky a implementuje některé základní přehledové prvky. Jedním z nich je ukazatel zbývajících času, jehož implementace se dotazuje herních pravidel na zbývajících čas. Pokud se čas změnil, je překresleno číslo pomocí knihovny MGEAN a funkcí z knihovny SDL\_ttf. Dalším prvkem je indikátor směru a síly větru, který vykresluje podle intenzity určitý počet vzduchových pytlů, které jsou navíc natočeny ve směru větru. V levé části HUDu se ještě nachází síla natažení zbraně, její barva a intenzita závisí na síle natažení. Barva se lineárně mění ze zelené v červenou s rostoucím natažením. Červená složka se vypočítá pomocí vztahu  $r = k * 255$  a zelená  $g = (1 - k) * 255$ , kde  $k$  je číslo z intervalu  $[0, 1]$  a vyjadřuje procento natažení zbraně. V pravém rohu obrazovky je vykresleno ještě množství zbývajících života jednotlivých týmů pomocí barevných proužků, k jejichž implementaci byla využita funkce `SDL_FillSurface`.

### 2.5.19. Základní systém GUI a menu

Ve hře Bazooworms jsem implementoval základní systém GUI, který mi umožnil snadno vystavět menu. Alternativním řešením mohla být volba některého GUI frameworku, to jsem však zavrhl z důvodu nízké variability grafických prvků a z důvodu nemožnosti běhu v celoobrazovkovém režimu.

MGEAN nabízí funkcionalitu zasílání zpráv, instalování uživatelských klíčů a uživatelských callbacků objektům SPRITE. Sprite je tedy v MGEANu chápán jako základní objekt, od kterého všechny objekty dědí. Z toho důvodu je možné vybudovat nad knihovnou MGEAN poměrně snadno GUI systém. Tohoto faktu využívá implementace GUI v Bazooworms.

Základním objektem GUI je objekt `WRMS_widget`, který dědí od `SPRITE` a přidává několik členských proměnných. Tou nejdůležitější je `parent`, která představuje ukazatel na rodiče. Rodič je objekt, na který je vlastní objekt navázán. Uživatelská destrukce widgetu probíhá v obsluze zprávy `on_widget_destroy`. Každý widget může obdržet zprávu `on_click`, `on_keydown`, `on_quit`. Z widgetu jsou potom odvozeny další GUI prvky, jako je text, tlačítko, textarea, kontejner, dialog, editovatelný text a další.

### 2.5.20. Práce se soubory

Konfigurační soubory obvykle patří ke každému většímu programu, Bazooworms není výjimkou. V mém projektu ukládám informace o hráčích a týmech do souboru `./bazoo_worms` na posixových platformách a do `./bazoo_worms.conf` na platformě Windows. Z konfiguračního řetězce zkompilevaného do programu dále přednačítám datové soubory. Knihovna MGEAN nabízí řešení problematiky konfiguračních souborů přes zabudovaný parser. Ten pracuje s daty podobným

lispovským seznamům. Odlišností je, že seznamy umístěné vedle sebe nemusí být odděleny mezerou. Pro ilustraci uvedu příklad souboru, který by mohl popisovat nastavení obrazovky a cestu k datovým souborům hry.

```
(:screen (:resx 1024 :resy 768 :bpp 32 :fullscreen t)
 :data-path /tmp/)
```

Po rozparsování takového řetězce dojde k rozpadu do stromu. Pro základní představu o práci se soubory uvedu kód pro modifikaci výše uvedeného konfiguračního řetězce.

```
const char * conf =
"(:screen (:resx 1024 :resy 768 :bpp 32 :fullscreen t)"
" :data-path /tmp/);"
HASH * strom = MG_text_list( conf ); /* rozparsování */
HASH * screen = MG_hash_get_key( strom , ":screen" );
HASH * data_path = MG_hash_get_key( strom , ":data-path" );
HASH * fullscreen_p = MG_hash_get_key( screen ,
                                       ":fullscreen" );

printf("fullscreen: %s data: %s\n" ,
      (char*)fullscreen_p->values,
      (char*)data_path->values );
free( fullscreen_p->values );
fullscreen_p->values = strdup( "nil" );
printf("%s\n" , MG_hash_dump( strom ) );
/* vytiskne: */
(:screen (:resx 1024 :resy 768 :bpp 32 :fullscreen nil)
 :data-path /tmp/)
```

Z přechozího příkladu je patrná jednoduchost celého přístupu. Změna hodnoty klíče je tak snadná, jako změna uzlu ve stromu. Díky typování uzlů stromu je možné jedinou vestavěnou funkcí zrekonstruovat strom do podoby řetězce. Další velkou výhodou je úspornost zápisu dat a jejich hierarchické uspořádání. Hry často používají jiné formáty pro uchování klíčů a hodnot, ale obvykle podporují pouze jednoúrovňové informace.

## 3. Dosažené výsledky

### 3.1. Stručná uživatelská příručka

V této kapitole ve stručnosti popíšu základní ovládání menu a hry Bazoworms. Součástí je také popis kompilace ze zdrojových kódů a instalace na různých platformách.



### 3.1.1. Kompilace ze zdrojových kódů

Pro kompilační proces jsem z důvodu jednoduchosti zvolil pouze nástroj Make. Díky tomu se kompilace skládá z jednoduchých úkonů. Prvním je instalace vývojových i binárních balíků knihoven SDL. V systémech z rodiny Debian je možné tyto balíky získat jednoduchým způsobem přímo z konzole. Postup je stejný jako v kapitole 2.5.1.. Stejně probíhá i kompilace.

### 3.1.2. Instalace balíků

Balíky `.deb` lze najít v adresáři `bin` na DVD, a to pro 32-bitovou i 64-bitovou architekturu. Instalaci provedeme buď jednoduchým spuštěním balíku z GUI prostředí linuxového systému nebo pomocí příkazu:

```
$ sudo dpkg -i bazoo_worms-1.0-0_i386.deb
```

Analogicky pro platformu AMD64. V případě použití tohoto konzolového příkazu je však nutné závislosti opatřit ručně.

### 3.1.3. Odinstalace balíku

Odinstalaci je možné provést opět pomocí GUI nástroje pro správu balíků. Případně pomocí jednoduchého konzolového příkazu.

```
$ sudo apt-get purge bazoo-worms
```

### 3.1.4. Spuštění hry

Spustit hru je možné opět dvěma způsoby, prvním je výběr z menu. Jelikož jsou menu na linuxových distribucích poměrně neunifikovaně řešená, uvedu způsob, jakým lze vybrat Bazooworms z menu Ubuntu 10.10. Spouštěč se nachází v menu Aplikace→Hry→bazoo\_worms.

V případě, že nechceme nebo nemůžeme použít balíček `.deb`, je nutné spustit hru pomocí binárního souboru. Ten je umístěn v adresáři `bin/` na DVD pod názvem `bazoo_worms-1.0-0_i386.tar.bz2`. Tento soubor je nutné rozbalit na lokální disk. Pokud máme splněny všechny závislosti v podobě nainstalovaných knihoven, stačí jednoduše dvojklikem z našeho oblíbeného správce souborů spustit soubor `bazoo_worms`. Případně lze využít konzolový příkaz z adresáře se hrou:

```
$ ./bazoo_worms
```

Hra však vyžaduje umístění na souborovém systému, který podporuje unixová oprávnění. Proto je nutné jej rozbalit na disk s touto podporou a přiřadit souboru `bazoo_worms` oprávnění ke spuštění. Na DVD je taktéž přiložen archiv obsahující spustitelný soubor pro systém MS Windows. Program pro fungující ukládání týmů vyžaduje povolený zápis v adresáři se hrou. Hru lze spustit pomocí souboru

bazoo\_worms.exe z adresáře se hrou. Verze pro Windows nepodporuje žádné konzolové parametry.

Pomocí konzole lze na posixových systémech předat parametr `--windowed`, který vynutí mód okna pro hru. Hra se ve výchozím stavu spustí do celoobrazovkového režimu s nejlepším možným rozlišením. Hru je dobré spouštět v módu okna pouze pro účely debugování. Případně při výskytu problémů s ovladačem grafické karty.

### 3.1.5. Ovládání menu

Menu je první, co uživatel po spuštění uvidí. Menu obsahuje čtyři tlačítka.

**PLAY** – tlačítko vedoucí na výběr týmů a následné spuštění hry.

**TEAMS** – aktivuje editaci týmů.

**CREDITS** – zobrazí dialog o autorovi, vypíše licenci a krátkou nápovědu k ovládání.

**QUIT** – vede na dialog ukončení hry.

Zmáčknutí libovolného tlačítka menu způsobí zmizení původního zobrazeného dialogu, pokud už bylo něco vybráno. Snímek 3. znázorňuje základní spuštěné menu.



Obrázek 3. Menu hry v počátečním stavu.

### 3.1.6. Dialog výběru nové hry

Tento dialog slouží pro výběr týmů, které se zúčastní následné hry. Výběr týmů je možný pomocí kliknutí na jeho název. Maximální množství vybraných týmů je čtyři. Týmy budou hrát v takovém pořadí, v jakém byly vybrány. Vybraný tým je znázorněn pomocí zeleného háčku vpravo od něj. Tým lze také odznačit pomocí kliknutí na již vybranou položku. Ve výchozím nastavení hry jsou v menu přítomny dva týmy. Hru je možné spustit pomocí tlačítka **Start game**.

### 3.1.7. Dialog editace týmů

Dialog pro editaci týmů se skládá ze tří editovatelných částí.

**TEAM NAME** – položka jméno týmu, jedná se o editovatelný text. Editaci započneme kliknutím na položku, poté běžně zeditujeme text. Až skončíme editaci stiskneme klávesu enter, tím se znovu objeví kurzor myši, který je při editaci skryt.

**TEAM TYPE** – položka pro změnu typu týmu, při kliknutí dojde k přepnutí mezi hodnotou **HUMAN** a **AI**.

**TEAMMATES** – obsahuje další čtyři položky editovatelných textů, což jsou jména jednotlivých červů.

Mezi jednotlivými týmy se lze přepínat pomocí tlačítek vespod dialogu. Jmenovitě je to tlačítko **<< Previous** a **Next >>**. Nový tým je nazván **untitled\_team**, pokud editujeme jeho libovolnou položku je tento tým uložen a je možné jej zvolit z menu **PLAY**, pokud jsou editace vráceny do původního stavu, tým se neuloží. K uložení změn v týmech dojde po kliknutí na tlačítko **OK**, kliknutím na jiné tlačítko zrušíme změny.

### 3.1.8. Dialog o autorovi

Je jednoduchý dialog informující o základních informacích o licenci a jménu autora. Ve spodní části dialogu je ještě přiložena krátká informace o ovládání hry. K jeho zrušení dojde po kliknutí na jiné tlačítko menu.

### 3.1.9. Dialog vedoucí k ukončení hry

Jedná se o jednoduchý yesno dialog, kde po kliknutí na tlačítko **YES** dojde ke korektnímu ukončení hry. Při volbě tlačítka **NO** dialog zmizí.

### 3.1.10. Ovládání hry

Možnost ovládání červa je řízeno pravidly. Červa je možno ovládat klávesnicí, pokud se jedná o červa z týmu, který je na tahu a tento hrající tým je ovládán člověkem. Pokud je červ na tahu, objeví se nad ním oscilující červená šipka. Ovládání uvnitř hry je vykonáváno vstupem z klávesnice. Konkrétní tlačítka jsou tato.

**Šipky vpravo a vlevo** – přidržením jednoho z těchto tlačítek dojde k lezení červa do strany odpovídající směru šipky. Sklon terénu však nesmí překročit jistou hranici.

**Šipky nahoru a dolů** – slouží k zaměření zbraně, šipka dolů snižuje úhel a šipka nahoru úhel zvyšuje. Zaměřování je možné pouze pokud je zobrazen zaměřovací kříž.

**Klávesa enter** – slouží ke skoku do strany, na kterou je červ právě natočen, slouží k překonávání složitějšího terénu.

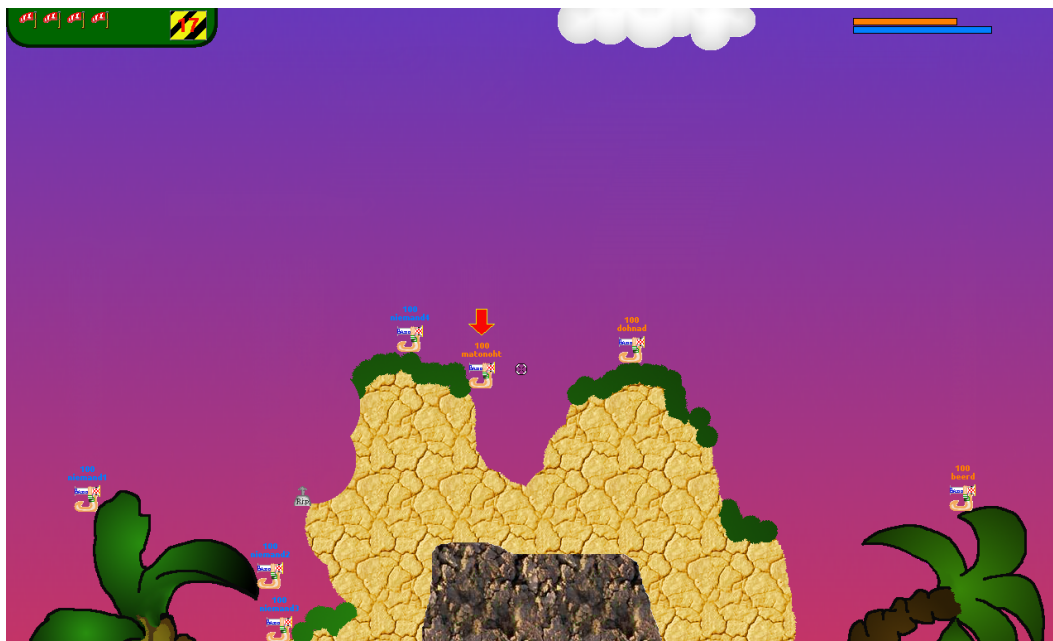
**Klávesa mezerník** – přidržením této klávesy dojde ke zvyšování natažení zbraně. Čím je déle podržena, tím rychleji vyletí náboj z ústí zbraně. Náboj vyletí, pokud hráč pustí mezeru nebo pokud natažení dosáhne maximální meze. To lze rozpoznat na indikátoru hry v levém horním rohu obrazovky.

**Escape** – stisknutím klávesy dojde k vyvolání yesno dialogu, který se uživatele ptá, zda chce ukončit rozehranou hru.

K ovládání kamery ve hře slouží kurzor myši, který posouvá kameru. K posunu kamery v daném směru dojde, pokud se myš vychýlí od středu do určité vzdálenosti. Ovládat kameru je možné pouze tehdy, když je nějaký hráč na tahu. V opačném případě přebírá řízení kamery hra. V textu je přiložen snímek z rozehrané hry 4..

### 3.1.11. Indikátory hry

K indikaci stavu hry slouží grafické objekty umístěné při horní hraně obrazovky. V levé části se nachází indikátor směru a síly větru. Pod ním se nachází indikace síly výstřelu zbraně, pokud hráč zrovna natahuje zbraň. Indikátor síly mění s velikostí natažení svoji barvu a velikost. Vedle něj se nachází odpočítávání času pro tah. Čas kola je omezen na 20 vteřin. V pravém horním rohu se nacházejí indikátory zdraví týmů. Zdraví týmu se počítá jako součet zdraví všech červů daného týmu. Indikátory života jsou rozmístěny tak, že nahoře je vždy tým který právě hraje, navíc jsou odlišeny barvou týmů.



Obrázek 4. Obrázek z rozehrané hry.

### 3.1.12. Stabilita Bazooworms

Během testování jsem nenarazil na žádný pád programu ani výskyt jiné závažné výjimky. Testování linuxové verze probíhalo na 32-bitovém sestavení v prostředí systému Ubuntu 10.10 a 11.04, verze pro Windows byla otestována v prostředí Wine a v systému Windows XP. Uživatel nesmí v průběhu hry mazat nebo jinak modifikovat obrázky a písmo. V případě chybějících nebo poškozených dat může být aplikace řízeně ukončena bez vědomí uživatele. Aplikace se vůbec nespustí, pokud nebudou všechna data na svém místě. V takovém případě bude o této skutečnosti uživatel informován formou chybového výpisu na standardní výstup. Pro úspěšné spuštění programu je také nutné mít dostatečně výkonný počítač s rozlišením obrazovky alespoň 800 na 600 pixelů.

## 3.2. Shrnutí výsledků

Přínos mé práce spatřuji ve svobodné implementaci hry Worms, která může zajistit snadný růst projektu i v budoucnosti. Navíc je Bazooworms v tuto chvíli již dobře hratelná, zábavná a fungující hra. Za druhý velký přínos považuji implementaci knihovny MGEAN. Knihovnu jsem se snažil psát co nejobecněji, aby bylo možné kdykoli bez úprav kódu knihovny začít psát vlastní projekt. V MGEAN jsem se také zaměřil na jednoduchost programového rozhraní, aby začátečník nebyl znevýhodněn neznalostí principů tvorby her. Jsem přesvědčen, že se současnou verzí knihovny lze snadno napsat adventuru, RPG, arkádu nebo strategii.

### 3.3. Licence

Licenci jsem volil zvlášť pro knihovnu a zvlášť pro projekt. Knihovna je publikována pod licencí GNU/LGPLv3, která umožňuje bezproblémové použití s uzavřenými komerčními projekty. Pro zdrojové kódy projektu Bazooworms je volena licence GNU/GPLv3, která neumožňuje uzavření projektu. Grafika a ostatní herní prostředky jsou pod licencí CC BY 3.0. Dílo Ubuntu Font je převzato od firmy Canoncial a je šířeno pod licencí Ubuntu Font Licence 1.0.

## 4. Diskuze

Kapitola diskuze zkonfrontuje moji implementaci Bazooworms s již existujícími podobnými projekty a nastíní budoucnost této hry.

### 4.1. Srovnání s jinými projekty

Během let se objevily hned dvě svobodné implementace Worms. Historicky první je hra nazývaná Warmux, dříve se jmenovala Wormux. Druhou implementací ve stylu Worms je hra Hedgewars. V obou případech jsou to však velké projekty, na kterých byly odpracovány tisíce hodin. A oba projekty disponují lidmi specializovanými na kód, grafiku, zvuk a překlady.

V současné době se rozhodně Bazooworms nemohou srovnávat s těmito implementacemi co do kvality grafiky, zvuk chybí úplně. Co si však porovnání zaslouží je kód. Podle mého soudu jsem použil poměrně inovativní přístup psaní her, který celý kód zjednodušuje a činí ho kratším. Oba konkurenční projekty mají kód velké rozsáhlý. Tím je myšleno řádově desetitisíce řádků kódu, a tak není příliš snadné se do takového projektu zapojit.

### 4.2. Možnosti dalších rozšíření

Přestože jsem do projektu Bazooworms zapracoval vše, co bylo v požadavcích k bakalářské práci, nejedná se o dokonalou hru. Nad rámec zadání bylo sice učiněno nemálo, přesto se v mé implementaci nedostalo na větší množství zbraní. Dále nejsou implementovány zvuky a pokročilejší menu. Také knihovna MGEAN by si zasloužila implementaci vrstev a dopracování zvuků. V menu by se také mělo zapracovat na internacionalizaci pomocí systému gettext a na lepších základních prvcích GUI.

### 4.3. Budoucnost projektu

Jakmile bude z mé strany dokončen počáteční kód hry a knihovny, zveřejním toto dílo na nějakém významném serveru, který hostuje otevřené projekty. Od

tohoto kroku si slibuji dotažení vizuální i programátorské části k dokonalosti za pomoci komunity.

## Závěr

Výstupem této bakalářské práce je svobodná implementace Worms z roku 1995, ale také knihovna MGEAN. Během vývoje hry jsem řešil mnoho problémů, které mě vedly na vytvoření poměrně čistého a funkčního kódu. Implementace proběhla v jazyce C a s pomocí multiplatformních knihoven SDL. Přestože celý projekt má ještě mnoho před sebou, již nyní se jedná o funkční a hratelnou hru se základní funkcionalitou. Další vývoj hry bude směřovat k otevřené komunitě, knihovna MGEAN bude uveřejněna jako samostatný projekt.



## Conclusions

The result of this thesis is a free implementation of Worms and also a library MGEAN. During the development of game I have solved many problems, which led me to create a relatively clean and functional code. Project was implemented in C and with the SDL library. The project still has open future. It is a functional and playable game with basic functionality. Further development of the game will go on as an open-source project. The library MGEAN will be published as a separate project.

## Reference

- [1] Hall, J.R. *Programming Linux Games*. No Starch Press, San Francisco, 2001.
- [2] Herout Pavel. *Učebnice jazyka C*. Kopp, České Budějovice, 1999.
- [3] Sedgewick Robert. *Algorithms in C*. Addison-Wesley, Princenton University, 2007.
- [4] *The Allegro Wiki* [online] 2010,  
[http://wiki.allegro.cc/index.php?title=pixel\\_perfect\\_collision](http://wiki.allegro.cc/index.php?title=pixel_perfect_collision)  
[cit. 2011-5-24].
- [5] *Simple DirectMedia Layer* [online] 2011,  
<http://www.libsdl.org/> [cit. 2011-5-24].
- [6] *Worms wiki* [online] 2011,  
[http://worms.wikia.com/wiki/worms\\_\(1995\)](http://worms.wikia.com/wiki/worms_(1995)) [cit. 2011-5-24].

## A. Obsah přiloženého DVD

### `bin/`

Tento adresář obsahuje několik archivů s binárními sestaveními pro různé platformy a dva balíčky deb pro 32-bitovou a 64-bitovou architekturu Intel. Soubor `bazoo_worms-1.0-0_win32-experimental.zip` je experimentální 32-bitové sestavení pro platformu MS Windows. Soubor `bazoo_worms-1.0-0_i386.tar.bz2` je 32-bitové sestavení pro distribuce GNU/Linux, mělo by však fungovat i na 64-bitových systémech za předpokladu splněných závislostí na knihovnách. Balíky `bazoo_worms-1.0-0_i386.deb` a `bazoo_worms-1.0-0_amd64.deb` jsou určeny pro systémy založené na distribuci Debian.

### `doc/`

Obsahem tohoto adresáře je text bakalářské práce včetně závazných stylů. V adresáři `images/` jsou použité obrázky. Textový soubor ve formátu PDF je nazván `work.pdf`. Sestavení textu je možné pomocí nástroje Make.

### `src/`

Obsahuje zdrojové texty hry, Makefile a několik podadresářů. Adresář `resources/` obsahuje datové soubory nutné pro spuštění hry. Podadresář `bin/` obsahuje po sestavení adresáře `posix/` nebo `win32/`, ve kterých je spustitelné sestavení programu pro danou platformu. Adresář `scripts/` obsahuje jednoduchý skript pro tvorbu deb balíků. Adresář `win32_dlls/` obsahuje dll knihovny pro platformu MS Windows. Adresář `mgean_lib/` obsahuje zdrojové kódy knihovny MGEAN a její sestavení, knihovnu lze z tohoto adresáře nezávisle přeložit pomocí Make. Sestavení knihovny pak bude obsaženo v jejím podadresáři `bin/`.

### `readme.txt`

Obsahuje instrukce pro spuštění a sestavení programu. Součástí je také základní popis ovládání a instalace.