CORWX – THE CORE WAR SIMULATOR

PROGRAMÁTORSKÁ DOKUMENTACE

VERZE 1.1

ANOTACE

Program je simulátorem souboje několika programů v abstraktním paměťovém poli. Těmto soubojům se někdy říká *Core Wars*. Program je jedním ze zástupců tzv. *MARS* (*Memory Array Redcode Simulator*), které byly inspirovány článkem od A. K. Dewdney v květnovém čísle Scientific American z roku 1984. Jednotlivé programy (dále bojovníci) jsou napsány ve speciálním jazyce *Redcode*, který je podobný zjednodušenému assembleru. Poté co jsou bojovníci nahráni do paměťového pole, simulátor začne střídavě spouštět jejich instrukce. Existuje jedna speciální instrukce, kterou nelze spustit. Bojovníci se snaží tuto instrukci přímo či nepřímo podstrčit soupeři tak, aby ji v příštím cyklu spustil. To by totiž vedlo k jeho zničení a vyřazení z bitvy. Bojovníci se proti zničení brání kopírováním sebe sama na jiná místa v paměťovém poli nebo spouštěním více instancí svého procesu. Existuje mnoho bojových strategií – někteří bojovníci jsou malí a agresivní, jiní jsou složeni z velkého počtu instrukcí a dokážou měnit své chování. Od vydání zmíněného článku v roce 1984 se začaly objevovat snahy pravidla Core Wars ujasnit a standardizovat. První *International Core Wars Standard* (*ICWS*) z roku 1986 byl upraven v roce 1988 (ICWS ’88) a následně v roce 1994 rozšířen dodnes rozpracovaným konceptem, který byl nakonec ustálen na verzi 3.3. Od té doby se jednotlivé simulátory začaly od standardu odchylovat implementací nejrůznějších extenzí. Program v sobě implementuje co nejpřesněji standard ICWS ‘94 3.3.

UPŘESNĚNÍ

Program na vstupu předpokládá textové soubory obsahující zdrojový kód bojovníků napsaný v Redcode podle ICWS ‘94 3.3. Poskytnutý zdrojový kód se pokusí přeložit a v případě úspěchu jej nahraje do paměťového pole. Pokud je zdrojový kód chybný nebo neodpovídá standardu, je odmítnut a uživatel je upozorněn o chybě. Uživatel si přes uživatelské rozhranní může zdrojový a přeložený kód jednotlivých bojovníků prohlédnout. Uživatel má v programu k dispozici mapu celého paměťového pole a lupu, kterou může na zvolených místech zjišťovat přesný obsah paměťových buněk. Uživatelské rozhranní dále obsahuje přehled naplnění úlohových front bojovníků a ovládací prvky pro startování a pozastavování simulace. Parametry simulace lze změnit přes vestavěné menu. Po spuštění simulace jsou postupně v cyklech střídavě spouštěny úlohy jednotlivých bojovníků. Výsledné efekty spuštěných instrukcí jsou okamžitě vidět v mapě paměťového pole. Bitva probíhá tak dlouho, dokud alespoň dva bojovníci mají nenulový počet úloh ke spuštění. Bitvu vyhrává poslední živý bojovník. Pokud bitvu přežije více bojovníků, je to považováno za remízu mezi přeživšími.

REDCODE

Jak již bylo řečeno v anotaci, program je implementací standardu ICWS ‘94 3.3 ve své originální formě bez použití jakýchkoliv extenzí. Objemnost zmíněného standardu neumožňuje jeho doslovný překlad z angličtiny a vložení do této dokumentace. Navíc by tím utrpěla srozumitelnost originálu. Pro porozumění Redcode a této dokumentaci je však nutné předpokládat alespoň zběžnou znalost standardu. Pro rychlé seznámení se základní sémantikou Redcode slouží následující odstavce textu. Pro opravdové ovládnutí Redcode na úrovni programátora bojovníků je však nutné přečíst přímo standard, který sémantiku a syntaxi vysvětluje mnohem detailněji.

Soubor bojovníka se skládá z komentářů a instrukcí. Komentáře mohou být na samostatném řádku nebo následovat za instrukcí. Soubor musí začínat komentářem *;redcode*. Každý bojovník se skládá z jedné a více instrukcí. Každá instrukce je na samostatném řádku. Instrukce je složena z instrukčního kódu, modifikátoru a dvou operandů oddělených čárkou. Instrukční kód a modifikátor jsou odděleny tečkou. Každý operand je složen z adresního režimu a aritmetického výrazu, který může obsahovat závorky, unární minus, sčítání, odčítání, násobení, celočíselné dělení a operaci modulo. Všechny adresní režimy jsou relativní, takže adresa je vzdáleností od současné instrukce. Před instrukcí může být volitelně napsán textový řetězec bez bílých znaků, který při použití v aritmetickém výrazu operandu libovolné instrukce zastupuje číselné pořadí instrukce, před

kterou je zapsán ve zdrojovém kódu. Dále se mohou ve zdrojovém kódu nacházet pseudo instrukce a rezervované textové řetězce, jejichž význam bude vysvětlen dále.

Instrukční kódy:

DAT Nelze spustit. Pokus o její spuštění vyřadí jeden bojovníkův proces z bitvy.

MOV Obsah adresy, na kterou ukazuje druhý operand, nahradí obsahem adresy, na kterou ukazuje první operand. Následující instrukce je přidána do fronty.

ADD Obsah adresy, kam ukazuje druhý operand, přepíše součtem obsahu adresy, kam ukazuje druhý operand a obsahu adresy, kam ukazuje první operand. Následující instrukce je přidána do fronty.

SUB Dtto, ale zapisuje se rozdíl.

MUL Dtto, ale zapisuje se součin.

DIV Dtto, ale zapisuje se celočíselný podíl. Dělení nulou má stejný efekt jako pokus o spuštění instrukce DAT.

MOD Dtto, ale zapisuje zbytek po celočíselném dělení. Dělení nulou má stejný efekt jako pokus o spuštění instrukce DAT.

JMP Zařadí do fronty instrukci na adrese, kam ukazuje první operand.

JMZ Otestuje, zda na adrese, kam ukazuje druhý operand, je nula. Pokud ano, tak do fronty zařadí instrukci na adrese, kam ukazuje první operand. Jinak do fronty zařadí následující instrukci.

JMN Dtto, ale testuje nenulovost.

DJN Dtto, ale před posouzením nenulovosti hodnotu dekrementuje.

SEQ Porovná obsah adres, na které ukazují oba operandy. Pokud se hodnoty shodují, přeskočí následující instrukci a do fronty zařadí až jejího následníka. Jinak do fronty zařadí následující instrukci. Z důvodů zpětné kompatibility lze zapsat i jako CMP.

SNE Dtto, ale přeskakuje, když se hodnoty nerovnají.

SLT Dtto, ale přeskakuje, když je první hodnota menší než druhá.

SPL Zařadí do fronty následující instrukci a instrukci, která se nachází ve vzdálenosti odpovídající obsahu adresy, na kterou ukazuje první operand. Přidává bojovníkovy jeden proces.

NOP Zařadí do fronty následující instrukci.

Modifikátory:

A Za obsah adresy, na který je odkazováno operandy, je považován pouze aritmetický výraz prvního operandu instrukce na dané adrese.

B Dtto, ale za obsah je považován druhý operand.

AB Za obsah adresy, na který je odkazováno prvním operandem, je považován pouze aritmetický výraz prvního operandu instrukce na dané adrese. Za obsah adresy, na který je odkazováno druhým operandem, je považován pouze aritmetický výraz druhého operandu instrukce na dané adrese.

BA Za obsah adresy, na který je odkazováno prvním operandem, je považován pouze aritmetický výraz druhého operandu instrukce na dané adrese. Za obsah adresy, na který je odkazováno druhým operandem, je považován pouze aritmetický výraz prvního operandu instrukce na dané adrese.

F Za obsah adresy, na který je odkazováno operandy, je považován aritmetický výraz prvního a druhého operandu instrukce na dané adrese.

X Za obsah adresy, na který je odkazováno prvním operandem, je považován aritmetický výraz prvního a druhého operandu instrukce na dané adrese (v tomto pořadí). Za obsah adresy, na který je odkazováno druhým operandem, je považován aritmetický výraz druhého a prvního operandu instrukce na dané adrese (v tomto pořadí).

I Za obsah adresy, na který je odkazováno operandy, je považována celá instrukce na dané adrese.

Adresní režimy:

\# *Immediate*
Adresa operandu je nulová. Operand tedy ukazuje sám na sebe.

$ *Direct*
Hodnota aritmetického výrazu v operandu je adresa, na které se již nachází cíl. Výchozí režim při neuvedení adresního režimu před operandem.

* *A Indirect*
Výslednou adresou je součet hodnoty aritmetického výrazu operandu a hodnoty aritmetického výrazu prvního operandu instrukce, na kterou tento operand ukazuje.

@ *B Indirect*
Dtto, ale přičítá se hodnota aritmetického výrazu druhého operandu.

{ *A Indirect Predecrement*
Výslednou adresou je součet hodnoty aritmetického výrazu operandu a hodnoty aritmetického výrazu prvního operandu instrukce, na kterou tento operand ukazuje, snížené o jedna.

< *B Indirect Predecrement*
Dtto, ale přičítá se hodnota aritmetického výrazu druhého operandu snížená o jedna.

} *A Indirect Postincrement*
Výslednou adresou je součet hodnoty aritmetického výrazu operandu a hodnoty aritmetického výrazu prvního operandu instrukce, na kterou tento operand ukazuje, která je po provedení součtu zvýšena o jedna.

> *B Indirect Postincrement*
Dtto, ale přičítá se hodnota aritmetického výrazu druhého operandu, která je po provedení součtu zvýšena o jedna.

Pseudo instrukce:

ORG Je následována číslem nebo aritmetickým výrazem, jehož hodnota udává inicializační instrukci bojovníka. Pokud je tato pseudo instrukce vynechána, bojovník je inicializován nultou instrukcí.

EQU Je předcházena textovým řetězcem bez bílých znaků a následována textovým řetězcem, ve kterém již bílé znaky mohou být. Všechny výskyty prvního řetězce ve zdrojovém kódu budou při překladu nahrazeny druhým textovým řetězcem.

END Nepovinná pseudo instrukce označuje konec zdrojového kódu bojovníka. Volitelně může být následována číslem nebo aritmetickým výrazem se stejným významem jako u ORG.

Rezervované textové řetězce:

CORESIZE	Velikost paměťového pole.
MAXCYCLES	Počet cyklů bitvy před vyhlášením remízy.
MAXLENGTH	Maximální počet instrukcí, ze kterých je složen bojovník.
MAXPROCESSES	Maximální počet spuštěných instancí jednoho bojovníka.
MINDISTANCE	Minimální počet instrukcí mezi prvními instrukcemi dvou bojovníků.

Další pojmy:

Initial instruction	Instrukce, kterou je inicializováno paměťové pole. Může být nastavena na RANDOM nebo NONE, což znamená, že pole nebude po předchozích bitvách čištěno.
Separation	Pevný počet instrukcí mezi prvními instrukcemi dvou bojovníků. Navzájem se vylučuje s MINDISTANCE, které umožňuje náhodnou vzdálenost mezi bojovníky.
Read Distance	Maximální vzdálenost, ze které může být prováděno čtení při vyhodnocování operandů instrukce. Pokud adresa ukazuje za tento limit, je ohnuta pomocí operace modulo tak, aby ke čtení došlo v povolené vzdálenosti.
Write Distance	Dtto, ale limituje zápis.

Jako příklad je vhodné uvést jednoduchého bojovníka Dwarf, který byl navržen samotným A. K. Dewdneym v původním článku v Scientific American. Jeho strategií je zapsat nulu do druhého operandu každé čtvrté instrukce v paměťovém poli.

```
;redcode
       ORG    start                ;Druhá instrukce je inicializační.
step   EQU    4                    ;Všechny výskyty step budou nahrazeny číslem 4.
target DAT.F   #0,    #0           ;Nic nedělá. Slouží k ukládání cílových pozic.
start  ADD.AB  #step, target       ;Přičte hodnotu step ke druhému operandu target.
       MOV.AB  #0,    @target      ;Do druhého operandu instrukce, kam ukazuje druhý
                                   ;operand target, uloží nulu.
       JMP.A   start               ;Skočí zpátky na start.
       END                         ;Ukončení bojovníka.
```

OVLÁDÁNÍ

Kliknutím na tlačítko *Add warrior* lze docílit načtení bojovníka do paměťového pole. Po kliknutí na tlačítko dojde k zobrazení dialogového okna pro výběr souboru. Lze vybrat pouze soubory s příponou *.red*, což je tradiční přípona souborů obsahující Redcode instrukce. Po potvrzení výběru souboru dojde k pokusu o jeho překlad. Je kontrolováno, zda soubor začíná řetězcem ;*redcode*, který potvrzuje, že se jedná o Redcode soubor. Pokud je v souboru chyba, neodpovídá standardu nebo vůbec neobsahuje validní instrukce Redcode, je zobrazena chybová hláška. Pokud vše proběhlo v pořádku, bojovník se objeví na mapě paměťového pole a do seznamu bojovníků v pravé části okna programu jsou přidány jeho ovládací prvky. Jedná se především o tlačítko, jehož barva je stejná jako barva paměťových buněk, které byly bojovníkem ovlivněny. Popisek tlačítka odpovídá pořadí, v jakém byl bojovník přidán do pole. Po stisknutí tlačítka se zobrazí zdrojový kód a přeložené instrukce daného bojovníka. Dále má každý bojovník vedle tlačítka zobrazeno naplnění jeho úlohové fronty. Před spuštěním simulace má každý bojovník ve frontě jednu úlohu.

Simulaci lze spustit, jakmile jsou do paměťového pole nahráni alespoň dva bojovníci. Ke spuštění simulace dojde buď po stisknutí tlačítka *Start*, nebo tlačítka *Step.* Po stisknutí tlačítka Start bude simulace probíhat tak dlouho, dokud neskončí bitva. Tlačítko Step provede pouze jeden cyklus bitvy, tedy spuštění jedné instrukce od každého bojovníka. Bitva může být kdykoliv pozastavena tlačítkem *Stop* a následně opět spuštěna. Rychlost bitvy lze regulovat posuvníkem nad tlačítky – čím více je nastaven doprava, tím rychleji bude bitva probíhat. Pod tlačítky je zobrazen počet cyklů, které již byly spuštěny, a počet cyklů, které zbývají do konce bitvy. Bitva však může skončit i dříve, pokud zůstane pouze jeden živý bojovník.

Mapa paměťového pole je pokryta malými čtverci, které odpovídají jednotlivým buňkám paměťového pole. Čtverce mění svoji velikost podle toho, jak rozsáhlé je paměťové pole. Černý čtverec reprezentuje prázdnou buňku paměti, tedy takovou, která doposud nebyla ovlivněna žádným bojovníkem. Obarvené buňky odpovídají svojí barvou bojovníkovi, který je naposledy ovlivnil. Existují dvě rezervované barvy, které nejsou používány pro bojovníky. První z nich je červená, která ukazuje pro každého bojovníka instrukci, která bude v příštím cyklu spuštěna. Druhou barvou je modrá, která po najetí myši na bojovníkovo barevné tlačítko obarví jeho příští instrukci. Protože v dané chvíli budou příští instrukce ostatních bojovníků červené, lze takto nalézt konkrétní místo, kde se daný bojovník zrovna nachází.

V pravé horní části okna programu se nachází podrobný náhled na obsah paměťového pole na zvolené adrese. Adresa lze zvolit buď zadáním čísla do políčka nad náhledem, nebo kliknutím myši do mapy paměťového pole. V černém rámečku je vždy zobrazeno 10 instrukcí od této adresy. Barvy instrukcí odpovídají obarvení buněk v mapě paměťového pole. Pro rychlejší orientaci lze aktivovat pomocný *Slider*, který červeně obarví buňky, které jsou zrovna zobrazeny v náhledu. Slider je viditelný pouze pokud neběží simulace.

Parametry simulace lze přenastavit po kliknutí na tlačítko *Setup*. V dialogu, který se zobrazí, je možné nastavit hodnoty buď manuálně, anebo kliknutím na jeden ze tří výchozích profilů načíst nejpoužívanější konfigurace. Význam jednotlivých položek je vysvětlen ve standardu. Po potvrzení dialogu tlačítkem *OK* jsou hodnoty zkontrolovány a načteny. Pokud je některá z hodnot odmítnuta, objeví se informační hláška. Po stisknutí tlačítka OK dojde k resetování aplikace podle nového nastavení – dojde k vyčištění paměťového pole a seznamu bojovníků.

Pro resetování aplikace po bitvě lze použít také tlačítko *Wipe*, které resetuje aplikaci podle současného nastavení hodnot v Setup menu. Pro značné urychlení simulace lze dočasně vypnout renderování paměťového pole odškrtnutím *Render*.

ALGORITMY

Vstupní textový soubor je předán vestavěnému parseru, který má za úkol zdrojový kód bojovníka převést do strojově srozumitelnější podmnožiny jazyka, která již neobsahuje žádné substituce, výrazy nebo textové proměnné. Takto zpracovaný zdrojový kód je následně již jednoduchým loaderem nahrán do paměťového pole. Po spuštění simulace bere za chod programu zodpovědnost funkce, která postupně spouští instrukce na adresách, které odpovídají úlohám jednotlivých bojovníků. V následujících odstavcích bude popsán algoritmus parseru a zmíněné spouštěcí funkce.

Parser překládá kód podle gramatiky definované ve standardu. Kód je postupně na požádání převáděn na proud tokenů dané gramatiky. Je třeba zmínit, že některé tokeny jsou rovnou zahazovány – příkladem jsou komentáře nebo opakující se výskyt bílých znaků po sobě. Zpracování vstupního souboru probíhá v 5 průchodech. V *prvním průchodu* jsou nalezeny substituční pseudo instrukce EQU, jejichž operandy jsou společně s klíči uloženy do hashovací tabulky. Do druhého průchodu již řádky s EQU instrukcemi nepostupují. Ve *druhém průchodu* jsou všechny řetězce v souboru porovnány s klíči v hashovací tabulce a pokud je nalezena shoda, daný řetězec je substituován řetězcem z tabulky. Ve *třetím průchodu* jsou nalezeny a odstraněny textové ukazatele na řádky – ukazatele jsou uloženy do hashovací tabulky jako klíče k jednotlivým číslům řádků. Dále jsou ve třetím průchodu expandovány chybějící části instrukcí, které jsou dle standardu volitelné. Pokud je zjištěno, že daná instrukce je zapsána ve starším standardu, je převedena na novější (standardy jsou zpětně kompatibilní). Ve *čtvrtém průchodu* jsou vyčíslovány hodnoty výrazů v operandech instrukcí. Operandy jsou po vyčíslení nahrazeny výslednou hodnotou. Na vyčíslení výrazů je použito klasické schéma do sebe vnořených funkcí *Expression*, *Term* a *Factor*. Ve výrazech mohou být použity závorky a textové proměnné, jejichž hodnoty byly uloženy do hashovací tabulky ve třetím průchodu. Čtvrtý průchod také slouží k detekci pseudo instrukcí ORG a END za účelem zjištění instrukce, kterou je inicializován bojovníkův proces. Pseudo instrukce jsou po zjištění požadovaných informací smazány. Úkolem *pátého průchodu* je pouze vyčištění textového streamu od zbytečných mezer a jiných bílých znaků, které vznikly při manipulacích s textem v předchozích průchodech.

Hlavní spouštěcí funkce dostane na vstup adresu instrukce, kterou je potřeba spustit. Tato instrukce je nahrána z paměťového pole a uložena v instrukčním registru. Druhým krokem je vyhodnocení obou operandů na základě jejich adresních režimů a modifikátoru instrukce. Pokud je adresní režim indirect, jsou použity další dočasné proměnné na uložení mezikroků k získání finálních hodnot operandů. Po vyhodnocení operandů je podle zvoleného instrukčního kódu provedena příslušná manipulace paměťového pole. Funkce je napsána tak, aby u instrukčních kódů s podobným chováním nebylo potřeba opakovat velké množství zdrojového kódu, ve kterém by bylo změněno třeba jen znaménko operace. Za tímto účelem funkce používá několik pomocných funkcí na aritmetické operace a porovnávání.

ARCHITEKTURA

Program je rozdělen do tří hlavních tříd – *Engine*, *Parser* a hlavní okno.

Základem je Engine, který kromě seznamu instrukčních kódů, modifikátorů, adresních režimů a proměnných pro uložení simulačních parametrů obsahuje také všechny datové struktury potřebné pro uložení paměťového pole a úlohových front bojovníků. Obsahuje hlavní funkci pro spouštění instrukcí společně s jejími pomocnými funkcemi. Další důležitou částí Engine je funkce *Loader*, která má za úkol nahrávat bojovníky do paměťového pole. Loader musí zajišťovat, že bojovníci se navzájem nepřepíšou a jejich počáteční pozice budou odpovídat nastaveným odstupům, které mohou být buď pevné, nebo náhodné. Poslední důležitou funkcí v Engine je funkce *Start*, která na požádání zvenčí spustí simulaci tím, že v cyklu připravuje a předává data hlavní spouštěcí funkci. Engine je navržen tak, aby pro veškerou komunikaci s okolím používal interface. Je tedy nezávislý a v budoucnu by kdykoliv mohl být napojen na jiné uživatelské rozhranní.

Třída Parser zastřešuje podtřídu TokenStream, která zpracovává požadavky parsovací funkce a hledá pro ni ve vstupním souboru tokeny použité gramatiky. Samotná parsovací funkce potom využívá tokeny z TokenStream v pěti průchodech, ve kterých z původního zdrojového kódu bojovníka připraví zjednodušený zápis instrukcí pro Loader v Engine. Parser je ve své podstatě také nezávislou třídou, ovšem aby bylo možné instrukční kódy, modifikátory a adresní režimy spravovat ve zdrojovém kódu pouze na jednom místě, načítá je Parser z Engine.

Instance obou těchto tříd jsou vytvářeny a ovládány hlavním oknem, které se soustřeďuje hlavně na implementaci grafických prvků a ovládání. Okno je tedy naprosto funkčně závislé na obou výše zmíněných třídách. Když chce uživatel načíst do paměťového pole nového bojovníka, klikne v okně na příslušné tlačítko a v dialogu vybere soubor k otevření. Soubor je následně předán Parseru, který se jej pokusí přeložit a v případě neúspěchu emituje výjimku, která je zachycena hlavním oknem a zobrazena uživateli ve formě informační zprávy. Pokud byl zdrojový kód úspěšně přeložen, je dále hlavním oknem předán Loaderu v Engine. Loader se

pokusí instrukce nahrát do paměťového pole. Na rozdíl od Parseru oznamuje chyby místo výjimkami vrácením chybového stavu. Okno chybový stav interpretuje a opět zobrazí uživateli. Nakonec jsou pro daného bojovníka přidány a inicializovány jeho ovládací prvky.

Jakmile uživatel nahraje do paměťového pole bojovníky, může spustit simulaci. Pro zajištění odezvy uživatelského rozhranní je Engine spuštěn na vedlejším vlákně. Uživatel může regulovat rychlost simulace tím, že mění hodnotu času, po který je vedlejší vlákno v každém průchodu cyklem uspáno. Engine veškeré vnitřní události oznamuje pomocí interface, který je implementován hlavním oknem. Protože však hlavní okno běží na jiném vlákně než Engine, je potřeba dbát vláknové bezpečnosti při vzájemném volání funkcí. Pokud tedy Engine volá pomocí svého interface nějakou funkci v hlavním okně a chce přitom měnit datové struktury okna, musí to řešit pomocí .NET funkce *Dispatcher.Invoke*, která za pomocí delegáta požádá o změnu datových struktur přímo vlákno, které je vlastníkem těchto struktur. Takto je řešeno veškeré obnovování stavu mapy paměťového pole, náhledového okna instrukcí a informačních progress barů.

Grafické ztvárnění paměťového pole je renderováno do bitmapy, která je přiřazena jako zdroj dat náhledu v uživatelském rozhranní. Po vykonání každé instrukce v Engine je do bitmapy vykreslena změna. Protože renderování bitmapy je výpočetně i paměťově drahá operace, lze volitelně vypnout, což vede ke značnému urychlení simulace.

Uživatel má dále k dispozici okno náhledu na krátký úsek po sobě jdoucích instrukcí v paměťovém poli počínaje zvolenou adresou. Instrukce v tomto okně jsou obnovovány stejnou funkcí jako bitmapa, takže v každou chvíli obsahují aktuální stav. Pokud je tedy vypnuté renderování bitmapy, okno náhledu se neaktualizuje. Pokud uživatel aktivuje slider, který kreslením do bitmapy znázorňuje, jaká oblast pole je zobrazena v okně náhledu, je nutné si pamatovat původní obsah bitmapy pod sliderem. Tato informace je uložena za pomocí barev jednotlivých instrukcí zobrazených v náhledovém okně. Jakmile uživatel polohu slideru změní, původní obsah bitmapy je obnoven, do náhledového okna jsou načteny instrukce z nového umístění a nakonec je přesunuto i grafické znázornění slideru.

Posledním složitějším prvek uživatelského rozhranní je menu, pomocí kterého lze přenastavit vnitřní parametry Engine i samotného paměťového pole. Většina zdrojového kódu menu se nachází v hlavním okně. Samotné menu tedy obsahuje jen základní vnitřní logiku. Po kliknutí na příslušné tlačítko je napřed okno menu připraveno pomocí současných parametrů Engine. Následně je menu zobrazeno uživateli jako modální dialog. Uživatel provede úpravy a dialog potvrdí. Veškeré hodnoty jsou nabídnuty Engine, který zkontroluje, zda jsou navrhované změny bezpečné a pokud ano, tak je provede. Aby byl uživatel informován o případném zamítnutí jím navrhovaných hodnot, jsou následně porovnány navrhované hodnoty s hodnotami, které skutečně Engine uložil. V případě neshody se uživateli zobrazí upozornění. Ve chvíli kdy jsou všechny hodnoty schváleny a uživatel informován o změnách, je nutné resetovat většinu datových struktur uvnitř Engine a v hlavním okně. Tyto operace jsou zapouzdřeny v několika nezávislých funkcích, které se volají mimo jiné i při počáteční inicializaci programu. Resetování ovládacích prvků, grafiky a datových struktur Engine je tedy odděleno a připraveno na použití na více místech v programu.

Za zmínku ještě stojí mechanismus použitý pro znepřístupnění ovládacích prvků v situacích, kdy je nemožné dané prvky použít. Mechanismus je shrnut do jedné funkce, která je volána při významných změnách stavu programu – např. při spuštění simulace. Funkce si zjistí základní informace o stavu programu a uloží je do boolovských proměnných. Jednotlivé ovládací prvky jsou potom (de)aktivovány podle výsledku logického výrazu sestávajícího ze stavových proměnných programu.

ROZŠÍŘENÍ

Základním rozhodnutím bylo, zda a do jaké míry implementovat zmíněný standard. Vzhledem k tomu, že standard z roku 94 se nikdy nedostal do své finální podoby a jeho koncept skončil v roce 95 na verzi 3.3, vzniknul během doby prostor pro vytváření různých doplňků a rozšíření standardu. Tyto extenze však nejsou nikde pořádně shrnuty a popsány. Nakonec tedy padla volba na implementování čisté verze standardu bez doplňků. Program jsem se každopádně snažil psát tak, aby se dal v budoucnu jednoduše rozšířit o podporu nových funkcí. Protože zmíněný standard nebyl dotažen úplně do konce, je v něm několik sporných míst. Především se jedná o implementaci *Read* a *Write* limitů, kterou jsem oproti standardu mírně zpřísnil, aby bylo zachováno předpokládané chování bojovníků při jejich použití. Dále jsem úplně vynechal vyhodnocování argumentu u komentáře *;assert*, který má za úkol kontrolovat, zda jsou pro daného bojovníka správně nastaveny vnitřní parametry Engine. Případná budoucí implementace doplňků a *;assert* si bude žádat značné rozšíření Parseru. Úpravy v Engine a formulářích by neměly být tak dramatické.

Dalším aspektem programu, který by šel dále vylepšovat, je výkon vykreslovacích funkcí, které v současné podobě značně brzdí rychlost průběhu bitvy a spotřebovávají velké množství operační paměti.

Způsob renderování změn do bitmapy, který nabízí technologie WPF, se příliš nehodí k častému volání v cyklu a je několikanásobně paměťově náročnější než u WinForms.

Kromě implementace extenzí standardu a optimalizace vykreslování by šlo program rozšířit o další funkce. Do menu by se zase dalo přidat nastavování barev paměťového pole a bojovníků. Správa bojovníků by se dala rozšířit o editor zdrojového kódu s vestavěným debuggerem. Uživatelské rozhranní by šlo obohatit o správce sad soubojů, bodování, vytváření žebříčků a export výsledků bitvy. Program by se mohl rozšířit o možnost veškerou činnost (od načtení bojovníků až po export výsledků bitvy) provést automaticky pomocí jednoduchého skriptu. Tato úprava by také pro maximalizaci rychlosti zahrnovala vypnutí veškerých vizuálních prvků programu.