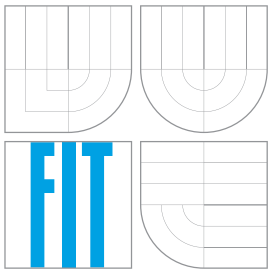
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

JAZYKY A PŘEKLADAČE V POČÍTAČOVÝCH HRÁCH

LANGUAGES AND COMPILERS IN COMPUTER GAMES

DIPLOMOVÁ PRÁCE
MASTER THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN HRANÁČ

VEDOUCÍ PRÁCE
SUPERVISOR

ZEMČÍK PAVEL, doc. Dr. Ing.

BRNO 2010

Abstrakt

Tento projekt se zabývá nasazením programovacích jazyků a kompilátorů v počítačových hrách. Cílem tohoto projektu je poskytnout mladým laikům motivaci k programování (kterou běžné programování v jazyce Pascal na úrovni středních škol a gymnázií ne vždy poskytuje). Tato idea byla aplikována na vytvořené počítačové hře, která má v sobě integrována několik jazyků, které slouží k vytváření herního obsahu této hry (dobrodružství, kampaně a objekty v nich se vyskytující). Tato hra byla vystavěna na stolní RPG hře *Dračí Doupě*™od nakladatelství ALTAR™.

Abstract

This project deals with usage of programming languages and compilers in computer games. Purpose of this project is to give young people motivation for programming, which isn't always done by usual work in Pascal at high school. This idea was applied on a developed computer game with has a few integrated languages which can be used for creation of the game's content (adventures, campaigns and objects in them). This computer game was based on a RPG desk game *Dragon's Lair* (a Czech equivalent of D&D) from ALTAR™ publishing company.

Klíčová slova

jazyky, překladače, počítačové hry, Open Inventor, Dračí Doupě™

Keywords

languages, compilers, computer games, Open Inventor, Dragon's Lair

Citace

Jan Hranáč: Jazyky a překladače v počítačových hrách, diplomová práce, Brno, FIT VUT v Brně, 2010

Jazyky a překladače v počítačových hrách

Prohlášení

Prohlašuji, že jsem tuto diplomovou vypracoval samostatně pod vedením pana docenta Pavla Zemčíka.

.....
Jan Hranáč
26. května 2010

Poděkování

Především bych chtěl poděkovat nakladatelství ALTAR^{TM}za svolení k použití jejich intelektuální vlastnictví v tomto projektu a za jednoduchost procesu jeho získání (žádost vyřizoval pan Martin Kučera).
Dále bych chtěl poděkovat panu doktoru Janu Pečivovi za poskytnutí jeho *SoSDL* tomuto projektu a také kolegům, kteří se mnou spolupracovali na vytvoření editoru map pro hru *DrdSim* (Aleš Marvan, Ivan Nejezchleb, Lukáš Pernica).
A v neposlední řadě bych chtěl také poděkovat svému vedoucímu za to, že se ujal této práce a své rodině za jejich podporu.

Obsah

Kapitola 1

Úvod

Od počátků historie vývoje počítačových her byla snaha oddělit obsah her od samotné aplikace hry. Herní světy a události v nich byly definovány odděleně od kódu hry a v lepších případech i samotné chování hry bylo skriptovatelné. V mnohých případech takto vznikly pozoruhodné programovací jazyky a prostředí. Motivací této snahy ale byla především ohebnost a zjednodušení vývoje pro vývojáře hry.

V některých případech bylo snahou částečně přiblížit vývoj herního obsahu i samotným hráčům, což se ne vždy podařilo dokonale. Tento projekt je ale od samotného počátku zaměřen právě na amatérskou tvorbu a její usnadnění pro programátorské začátečníky v kontextu využití programovacích jazyků v počítačových hrách.

Důraz je kladen na jednoduchost užití těchto jazyků a prostředí, do kterého jsou zasazeny - každý člověk, pokud nemá naprostý odpor vůči počítačům (což hráči počítačových her obvykle nemají), by měl být schopen pochopit vývojový systém hry a posléze vytvořit vlastní kampaň[1].

V rámci tohoto projektu byla vytvořena počítačová hra *Simulátor Dračího Doupěte*. Hra obsahuje několik jazyků[2] různých charakterů a zaměření. Některé mají za úkol definovat zdroje a objekty ve hře, jiné zase definují chování (události, scénáře, AI, atd.).

Téma jsem si vybral, protože mě zajímá vývoj počítačových her a zabývám se jazyky a překladači. Z hlediska kombinace těchto dvou prvků již delší dobu sleduji trendy a přístupy k vývoji herního obsahu v různých produktech. Především jsem pak byl ovlivněn svými zkušenostmi z let, když jsem ještě studoval na gymnáziu a zabýval se vývojem kampaní do počítačové hry *Operation Flashpoint*^{TM}. Tato hra nabízela vývojové prostředí, které sice mělo úspěch a kolem kterého se vytvořila velká komunita, ale které se rozhodně nedalo považovat za pěkné a navíc pokulhávalo po formální stránce. Proto jsem se rozhodl zkusit věci jinak, po svém a pokud možno lépe.

Cílem tohoto projektu však není pouze vytvořit hru s obsahem definovatelným pomocí programovacích jazyků (což by nebylo nic nového), ale také vytvořit něco, co by poskytlo mladým lidem motivaci k programování. Studenti středních škol a gymnázií se tak jako tak učí programovat, ale obvykle pracují v jazycích jako je Pascal nebo C. Problém je v tom, že tyto jazyky nejsou pro začátečníky příliš uspokojující, protože se programátor okamžitě nedočká velkých výsledků, a není tudíž motivován do dalšího experimentování. Tím se ale dostává do slepé uličky, protože programovat se člověk naučí pouze programováním.

[1] Kampaň je ucelený soubor částí jako jsou mapy/mise a případných vlastních objektů v nich a dalších náležitostí.

[2] V některých případech se však jedná o tak jednoduché záležitosti, že slovo „formát“ by bylo příhodnější.

Jedním z přínosů tohoto projektu je tedy výzkum jiného přístupu k vývoji obsahu počítačových her. Jak každý vývojář her potvrdí, vývoj herního obsahu (příběh, mapy, postavy, atd.) je dnes důležitější než vývoj samotné aplikace (i když ta zůstává základním pilířem celkové kvality produktu). Tento aspekt je v mnoha dnešních hrách dobře zvládnut pomocí propracovaných grafických toolkitů. Projekt *Simulátor DRD*[3] se však snaží o formálnější a více programátorský přístup.

Počítačová hra vytvořená v rámci tohoto projektu je založena na stolní RPG hře *Dračí Doupě*™ od nakladatelství ALTAR™, které dalo k tomuto projektu souhlas.

Již z podstaty věci je jasné, že tento projekt kombinuje několik oblastí informačních technologií a představuje opravdu velké množství práce. Krom jazyků a překladačů se především jedná i o multimediální stránku projektu. Následující kapitoly detailně popisují zkoumanou problematiku a hlavní oblasti projektu *Simulátor DRD*.

Nejprve je nutno analyzovat různá již existující řešení (kapitoly 2 a 3). Na konci kapitoly 3 je možno nalézt souhrn oblastí, na které se projekt *DrdSim* soustředí a které by chtěl zkusit jiným způsobem než v existujících řešeních.

Podrobný plán projektu a specifikace cílů je v kapitole 4. Další kapitoly se zabývají návrhem a implementací samotného SW produktu *DrdSim*. Základní struktura aplikace a řídící logika hry je popsána v kapitole 5, stejně jako použité jazyky, překladače a interprety a jejich vazba na systém. Kapitola 6 se zabývá zbytkem enginu hry (grafika, zvuk atd.).

[3] Interní zkratka *drdsim*.

Kapitola 2

Shrnutí poznatků

Tato kapitola se zabývá užitím skriptovacích jazyků ve vývoji obsahu počítačových her. Jsou zde popsány různé příklady a trendy z minulosti i současnosti. Důležitá je zejména část 2.2, která popisuje hlavní zdroj inspirace pro tento projekt.

Tato kapitola také obsahuje stručný přehled některých oblastí knihovny *Open Inventor* (respektive *Coin2*). OI se totiž stal nedílnou součástí tohoto projektu (stejně jako problémy s ním) a proto je třeba mu věnovat pár slov.

Čtenář obeznámený s tématy zde rozebíranými může tuto kapitolu přeskočit, ale na informace zde uvedené bude v dalších kapitolách odkazováno.

2.1 Lua a další jazyky

V souvislosti s tématem skriptovacích jazyků v počítačových hrách si většina lidí zabývajících se počítačovými hrami vybaví programovací jazyk *Lua*. Tento jazyk byl ve větší či menší míře použit v mnoha známých hrách, hrou *Baldur's Gate*^{TM} počínaje a dnešními hity jako *Crysis*^{TM} konče (viz [13]).

Jazyk Lua nemá C syntaxi [14, History], v tomto ohledu je podobný především jazyku *Modula*. Jedná se o víceparadigmový jazyk [14, Features] - je prototypově orientovaný a současně má podporu pro funkcionální programování. Současná verze je složitý dynamický jazyk podporující metaprogramování. Interpret jazyka je určen pro vestavění do libovolné hostitelské aplikace a je možno rozšiřovat sémantiku jazyka. Zdrojové soubory jazyka jsou přeloženy do mezikódu a pak vykonány na registrově orientovaném virtuálním stroji.

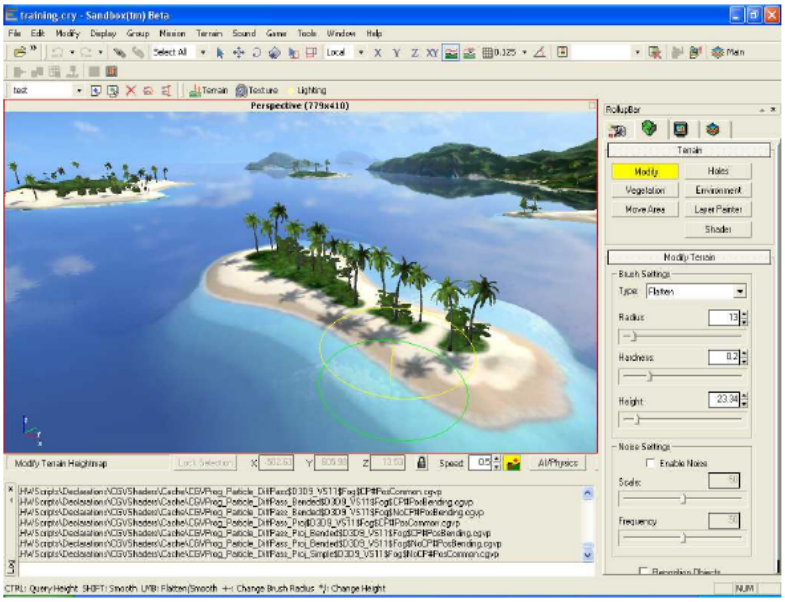
Následující příklad ukazuje vytvoření třídy a objektu v jazyce Lua:

```
-- create class
MyClass = setclass("MyClass")

-- define constructor
function MyClass.methods:init(param)
    self.attrib = param
end

-- create an object
my_object = MyClass:new("asd")
```

Jak je na první pohled vidět, jazyk je zcela odlišný od C++.



Obrázek 2.1: Editor map pro *FarCry*.

Poprvé (v počítačové hře) byl programovací jazyk Lua použit ve hře *Baldur's Gate*^{TM}. Role jazyka ale byla pouze pomocná - Lua byla použita jako debuggovací nástroj (viz [18]) a hra mohla plně fungovat i bez ní.

Oficiální podpora uživatelské tvorby byla v BG přítomna v možnosti definovat chování postaviček při boji [3]. To bylo umožněno za pomoci velice primitivního skriptovacího jazyka založeného na větvení rozhodování. Zde je například začátek AI skriptu pro klerika (převzato ze souboru `Cleric2.baf` na prvním CD prvního dílu *Baldur's Gate*^{TM}):

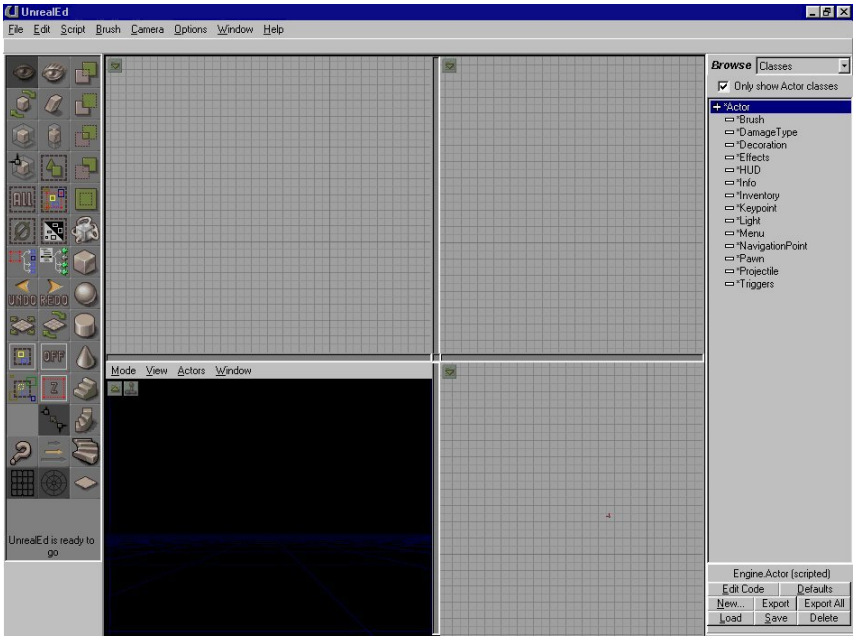
```
IF
    HPPercentLT(MostDamagedOf(),50)
    HaveSpell(CLERIC_CURE_LIGHT_WOUNDS)
THEN
    RESPONSE #100
        Spell(MostDamagedOf(),CLERIC_CURE_LIGHT_WOUNDS)
END
```

Neoficiálně také existovala možnost vytvářet vlastní obměny a rozšíření herního obsahu. Nejednalo se ale o nic přímočarého, co by se dalo srovnávat s běžným vývojem software např. v jazyce C [2]. Všechny zdroje také musely být kompilovány. Výstupem kompilace AI skriptu byl opět textový soubor obsahující jakýsi assembler AI enginu hry.

Po tomto prvním nasazení se Lua začala používat i v dalších hrách a ve větší míře (např. již hra MDK2 byla na jazyku Lua plně závislá [18]). Lua se vyvíjela a byly vydávány další a další verze tohoto jazyka [14, History] a byla nasazena do velkých hitů poslední doby jako jsou např. *The Witcher*, *Far Cry*, *Heroes of Might and Magic V* a *Crysis* (viz [13]). Podpora komunitního vývoje byla u těchto produktů různá.

Například hra *Far Cry* a další hry se stejným enginem, např. *Crysis*, obsahuje plnou podporu pro vytváření vlastních misí [8]. Jedná se ale především o vytváření map za pomoci grafického editoru (viz obrázek 2.1).

Lua byla použita především pro AI nepřátel a skripty vyžadují velkou vazbu na objekty



Obrázek 2.2: Okno aplikace *UnrealED*. Za povšimnutí stojí především výběrový strom v pravé části okna. Tato nabídka zobrazuje všechny třídy naprogramované v jazyce UnrealScript a jejich hierarchii dědičnosti.

mapy a vestavěné parametry hry. Běžný uživatel nemá k těmto záležitostem snadný přístup. Hlavním přístupem je programování za pomoci tzv. *entit*, které představují herní objekty a lze je případně umístit do herního světa v editoru map.

Jejich tvorba spočívá ve vytváření polí s množinou elementů přesně definovanou rozhraním hry, např. jakým 3D modelem má být entita zobrazena. Tato pole pak představují třídy objektů a lze jim i definovat metody (toto je Lua). Nadefinované entity jsou pak načteny enginem hry[1].

Všude je možno využívat bohatého výběru vestavěných funkcí hry, včetně nízkoúrovňových záležitostí jako je třeba přečtení uživatelského vstupu z klávesnice, grafické efekty (screen fade apod.) nebo načtení souboru z disku.

Krom jazyku Lua se proslavil například skriptovací jazyk *UnrealScript*. Jedná se o čistě objektově orientovaný jazyk podobný jazyku Java [17]. Vývoj herního obsahu kombinoval užití grafického nástroje *UnrealEd* (viz obrázek 2.2), který však poskytoval jen to nejnutnější a užíval se k editaci herních prostor a umisťování objektů do nich. Implementace chování hry a samotných herních objektů už probíhala v UnrealScriptu (ať už přímo v okně UnrealEdu, nebo v nějakém luxusnějším externím editoru). Některé hry založené na Unreal enginu (např. *Star Trek Deep Space Nine: The Fallen* [6]) se práci pokoušely zjednodušit a umožnily některé věci, které by bylo nutno textově skriptovat, definovat graficky.

UnrealScript je na rozdíl od jazyku Lua vázán na herní engine a vyznačuje se podporou pro herní čas, stavy a vlastnosti objektů a síťování [17]. To vede k výraznému zjednodušení kódu. Od Javy převzal UnrealScript bezukazatelové prostředí, automatický garbage collector a jednoduchou dědičnost. Programování v UnrealScriptu je vysokoúrovňové - pracuje se s herními objekty a interakcemi mezi nimi. Jedná se o poměrně složitý jazyk a uživatel by

[1] V tomto místě bych požádal čtenáře, aby si zapamatoval popsané rysi a povšiml si jejich podobnosti se systémem popsaným v části 2.2 a s finálním řešením v *DrdSim*.

měl mít předešlé zkušenosti s C++ a Javou a zdrojové soubory je nutno předkompilovat.

Stejně jako v Javě, programování je přísně třídně orientované - každý zdrojový soubor odpovídá jedné třídě, která je deklarována na začátku souboru [17]. Třída má své proměnné, stavy, vlastnosti, funkce, atd. Vytvořené třídy se poté dají umisťovat do mapy. Každý objekt může být kontrolován buď hráčem nebo skriptem, který pak plně určuje pohyb daného objektu.

UnrealScript a Unreal engine také již na principiální úrovni podporovaly síťování, neboť celá architektura byla již od začátku koncipována distribuovaně [17]. Virtuální stroj UnrealScriptu byl rozdělen na klienta a server. V single-player módu běžel server i klient na stejném počítači. V multi-player módu pak nebyl problém provozovat více klientů a jeden server, který rozhodoval interakce mezi objekty.

Neméně slavným se stal jazyk *QuakeC*, který je založen na jazyce C a má podobné principy jako UnrealScript [16]. Stejně jako UnrealScript, i QuakeC musí být zkompilován mimo prostředí hry (překladačem `qcc`).

Narozdíl od jazyka C, QuakeC neumožňuje definovat nové typy a struktury a nepoužívá ukazatele. Jedinou výjimkou je typ *entita*, který zastupuje objekty herního světa a je vždy referencí.

Při programování v QuakeC je možno využívat veliké množství vestavěných funkcí, které jsou nadeklarovány v jazyce QuakeC, ale jejich implementace je uvnitř enginu hry. Systém ale v jednom okamžiku udrží jen jednu návratovou hodnotu z takové funkce.

Entity a jejich vlastnosti a chování nejsou reprezentovány pomocí tříd, jak je tomu v prostředí UnrealScript. Základním prvkem je místo toho funkce (která se významem pohybuje někde mezi třídou a funkcí, ať už to zní sebepodivněji). Na jejím začátku může být vytvořena reference na novou entitu, které jsou pak přiřazeny vlastnosti a kód, který ji ovládá. Syntaxe vypadá zhruba takto (jen náznak):

```
void (<parametry>) <název> =
{
    // tělo
};
```

Obecnost, flexibilita a rozšiřitelnost enginu Quake a jazyka QuakeC byla prokázána množstvím nejrůznějším free-ware i komerčních her na této technologii založených.

Z dalších produktů by bylo dobré zmínit např. velice populární hru *Max Payne*^{TM}, která získala početnou komunitu amatérských vývojářů map a módů (celou kampaň ale nikdo nevytvořil - narozdíl od OFP, viz 2.2). Stejně jako u předchozích dvou příkladů, základem tvorby je grafický editor (*MaxEd*) [9], který je však mnohem bohatší a více se na něj spoléhá. Přímo v něm je možné použít jednoduchý skriptovací jazyk na implementaci dynamických objektů (dveří, výtahů) a předskriptovaného chování nepřátel. Jazyk je velmi orientovaný na konečné automaty.

Na závěr by chtěl podotknout, že téměř všechny skriptovací systémy zde uvedené byly postavené na jednom jazyce, od kterého se požadovalo, aby zvládl vše potřebné. Zda je tento přístup vhodný, probírají další kapitoly.

2.2 Operation Flashpoint

Počítačová hra *Operation Flashpoint*^{TM} (a jazyky v ní obsažené) byla hlavní motivací a inspirací tohoto projektu. Ačkoliv projekt *DrdSim* toho po jazykové ani systémové stránce

po hře *Operation Flashpoint* (OFP) příliš nepřebral, je třeba tomuto projektu věnovat zvláštní pozornost a prostor.

Tato žánr-definující počítačová hra vyšla 22. června roku 2001 [15] od českého vývojového týmu *Bohemia Interactive*^{TM}. Po počátečních nesnázích[2] se produkt dočkal obrovského úspěchu a to nejen ve sféře zábavního průmyslu, ale i ve vojenství [15, Reception] (VBS1 založený na OFP byl použit jako vojenský simulátor pro skutečné vojáky).

Ještě větší než úspěch mezi hráči byl úspěch mezi fanouškovskými vývojáři. Kolem hry se vytvořila obrovská komunita lidí [10], kteří začali ve svém volném čase vytvářet velké množství přídavků, misí i celých kampaní[3]. Jedním z důvodů této oblíbenosti byly jazyky užité pro vývoj obsahu hry, způsob jejich nasazení do systému a celková otevřenost, jednoduchost a zdokumentovanost prostředí. Následující text tyto záležitosti popisuje.

Informace v této kapitole pocházejí z mých vlastních zkušeností s tímto systémem. Všechno lze také ověřit na wiki stránkách OFP (viz [11]).

Herní obsah OFP je definován textovými soubory, které jsou napsány různými jazyky různých paradigmat. Kombinací těchto souborů může vzniknout mise, kampaň, nebo i celý mod, zcela odlišný od původního produktu.

Základní prvkem hry je mise. Prerekvizitou pro vytváření misí je ostrov - herní svět, ve kterém se mise odehrává. Ostrovy se obvykle definují v grafických editorech, případně s ručními úpravami v textovém editoru.

Nezbytné jsou také herní objekty - nejen stromy a domy, ale také auta, tanky, letadla a vojáci. Ty jsou definovány 3D modely a textovými soubory ve speciálním objektově orientovaném jazyce.

Samotná mise je pak adresář obsahující soubory, které tuto misi definují. Nejdůležitějším souborem je inicializační soubor, obvykle nazývaný mapa. Ten obsahuje počáteční nastavení mise jako je např. použitý ostrov (který je definován také v samotném názvu adresáře), počasí, čas, umístění jednotek, jejich vybavení, atd. Všechny uvedené věci krom první se dají dynamicky změnit nebo přidat i v průběhu mise pomocí skriptu. Dynamicky vytvářené multiplayerové mise této techniky dokonce využívají, ale je to krkolomné.

Mise dále může obsahovat popisný soubor, který definuje některá další nastavení, která se nenastavují v inicializačním souboru, např. explicitní přenastavení textu zobrazeného při načítání mise. Hlavně ale umožňuje pomocí dalšího objektového jazyka (podobného tomu dříve zmíněnému) definovat zdroje a dialogy. Tento jazyk bude zběžně popsán dále.

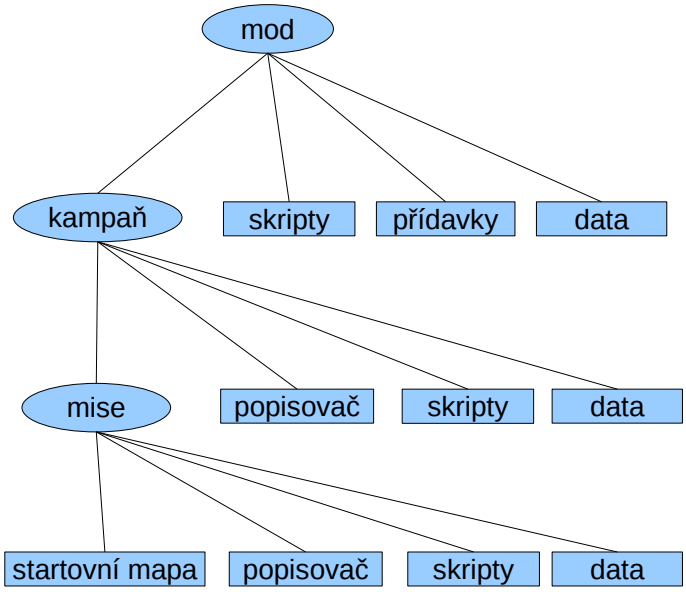
Především pak ale může obsahovat soubory napsané v tom nejdůležitějším a nejvíce používaném jazyce ve hře. Jedná se o procedurální nestrukturovaný jazyk užívaný ke skriptování událostí ve hře. Za pomoci tohoto jazyka se dá naprogramovat vše potřebné: inicializace, implementace chování jednotek a objektů, skriptování scénářů, tvorba animovaných scén nebo třeba i tvorba řídících systémů dynamických kampaní. I tento jazyk bude podrobněji popsán dále.

Adresář s misí může obsahovat audio soubory s dabingem rozhovorů mezi postavami a další soubory s daty (synchronizace rtů, obrázky, atd.).

Více misí může být spojeno speciálními popisnými soubory a superglobálními proměnnými do kampaně. Kampaň opět obsahuje popisný soubor, který definuje následnost misí v kampani nebo případně zdroje, které chceme mít dostupné ve všech misích. Také může obsahovat soubory se skripty, které se také dají spustit ze všech misí.

[2] Hra sice byla dokončena a její kvalita byla zjevná, ale žádný distributor nebyl sto pochopit, o co se vlastně jedná.

[3] Mezi tyto jsem patřil i já sám. Krom několika misí jsem v prostředí této hry naskriptoval dvě kampaně.



Obrázek 2.3: Adresářová struktura v *Operation Flashpoint*.

A nakonec, pokud je spojena kampaň, případně více kampaní, přídavky (vlastní jednotky a ostrovy) a případně vlastní definice různých menu ve hře, je možno vytvořit mod. Skripty a další zdroje definované v modu jsou dostupné ve všech kampaních patřících do daného modu.

Uvedené věci jsou přehledně znázorněny v obrázku 2.3. Obrázek nepostihuje úplně vše, jen to hlavní (nepostihuje např. lokalizační soubory a podobné drobnosti).

Následující text se podrobněji zabývá jazyky užitými ve hře.

Pro přídavky (úroveň modu - jednotky, zbraně, ostrovy, atd.) je nejdůležitější jazyk konfiguračních souborů. V každém přídavku je jeden a definuje, co všechno daný přídavek přidává. Může se jednat o zvuky, tváře, jednotky, vozidla, vybavení a další. Může být přitom využito datových souborů (audio, textury, modely) obsažených přímo v archivu modu, nebo dat již obsažených ve hře (konfigurační soubor potom pouze definuje logickou funkčnost přidávaných tříd)[4].

Jazyk je velmi podobný C++ (jeho soubory dokonce mají příponu cpp), ale je pouze deklarativní. Je třídně orientovaný, dědičnost tříd může být pouze jednoduchá. Syntaxe definice třídy vypadá zhruba takto:

[4]Například: chci přidat vojáka ze 2. světové války. Pak vytvořím 3D model tohoto vojáka a v konfiguračním souboru jej nadefinuji. Pokud bych ale chtěl přidat moderního vojáka s nějakou mou vlastní definicí (např. americký voják s ruskou zbraní), mohu toto kompletně provést na textové úrovni bez potřeby vytvoření vlastních dat.

```
class <id_třídy> [: <id_třídy>]
"{"
   { <přiřazení_do_atributu> | <vnořená_třída> }
"}"
```

Užité non-terminály netřeba detailně komentovat. Příklad práce s tímto jazykem lze nalézt v příloze B.

Soubor může obsahovat různé třídy s předdefinovanými názvy, které určují, co je v nich definováno. Uvnitř těchto tříd je pak možno definovat další třídy už s požadovaným užitečným obsahem. Na začátku soubor obsahuje třídu, která definuje, co všechno je v daném přídavku obsaženo. Následovat mohou různé jednodušší třídy jako jsou například zvuky, tváře, apod.

Nejdůležitější jsou třídy definující herní objekty (v jazyce se interně označují jako vozidla) a zbraně. Ve třídě definující objekty je nejprve nutno uvést plnou cestu dědičnosti pro jednotky a zbraně, které chceme přidat. Poté je již možno definovat nové třídy.

Třídy jednotek obsahují definice atributů dané třídy. Atributy mohou být skalární nebo vektorové. Třída dědí všechny atributy definované v mateřských třídách. Všechny atributy musí být předdefinované systémem hry, která je při startu rozpozná a načte. Třída také může uvnitř sebe obsahovat definici dalších tříd, pomocí kterých lze třídě přidat uživatelské akce, které může hráč spustit v průběhu hry, a obsluhy událostí (např. start mise).

Do třídy není možné umístit žádný procedurální kód. V případech, kdy to je zapotřebí (např. jak je výše uvedeno), je možno patřičnému atributu přiřadit řetězcovou hodnotu, která představuje jeden řádek kódu. Ten pak může např. volat nějaký externí skript, který již ošetří vše potřebné.

Popisovací soubory kampaní a misí jsou syntakticky podobné, ale mají určité sémantické odlišnosti. Je zde možno vkládat do hry vlastní hudbu a zvuky (pokud tak nechce tvůrce mise/kampaně učinit na úrovni modu) a nadabované dialogy.

Objektová orientovanost zde nastupuje v možnosti definovat zdroje a GUI dialogy. Zdroji jsou myšleny různé nápisy, titulky a podobně (pro použití např. do filmového úvodu kampaně). Opět je zde možno aplikovat jednoduchou dědičnost a podobné grafické zdroje založit na stejných třídách.

Podobným způsobem je možno definovat grafický vzhled interaktivních uživatelských rozhraní. V takovém případě je nutno ještě napsat skripty, které dané GUI budou obsluhovat. Veškeré GUI obsažené ve hře bylo autory hry vytvořeno stejným způsobem.

Nejčastěji používaným jazykem ve hře je SQS. Ten byl sice v pokračování hry nahrazen jazykem SQF [11, Scripting], ale oba jazyky jsou podobné.

Jazyk SQS je nestrukturovaný, procedurální, řádkově orientovaný skriptovací jazyk. Užívá se v OFP k implementaci všeho dynamického. Tedy například již zmíněná obsluha GUI, obsluha akcí a událostí jednotek, video scény, předskriptované chování nepřátel, řídící logika misí a kampaní a podobně.

Řádky souboru skriptu obsahují příkazy jako je přiřazení nebo volání (vestavěné) funkce. Řádek může obsahovat více příkazů oddělených středníkem. Na začátek řádku je možno umístit podmínku, zbytek řádku se provede jen pokud bude podmínka pravdivá. Dále může řádek obsahovat návěští, příkaz čekání po určitý čas a příkaz čekání do splnění nějaké podmínky (skripty mohou být spuštěny paralelně, případně může dojít ke změně pravdivosti podmínky následkem akce hráče).

Jazyk neobsahuje žádné řídící struktury typu `if-then-else`, veškeré větvení a cykly je nutno provádět pomocí návěští a příkazu `GoTo`, případně v kombinaci s podmínkou na

začátku řádku. Stejně tak neobsahuje možnost definovat funkce, je možno pouze zavolat jiný soubor pomocí příkazu `exec`. Vzhledem k tomu, že je možno při použití tohoto příkazu předat volanému skriptu i parametry, dá se soubor se skriptem považovat za alternativu funkcí ze strukturovaných jazyků. SQS dokonce umožňuje používat lokální proměnné (lišících se od obyčejných tím, že začínají podtržítkem), které platí jen v rámci skriptu a po jeho doběhnutí se smažou. Obyčejné (globální) proměnné naproti tomu platí po celý čas mise.

Největší síla jazyka je ve vestavěných funkcích. S jejich pomocí lze dosáhnout jakéhokoliv výsledku, který je ve hře možný. Je tedy například možné v průběhu mise mazat nebo přidávat jednotky nebo měnit parametry jako je čas a počasí. Není možné dynamicky vytvářet nové třídy. Krom uvedeného je možné v tomto jazyce dosáhnout všech efektů, které se od podobných jazyků očekávají - řízení chování nepřátel nad rámec jejich AI, předělová videa, změna vybavení jednotek a dokonce i speciální efekty[5].

2.3 O Open Inventoru

Jak bude popsáno v dalších kapitolách, *Open Inventor* byl zvolen jako jeden ze základních vývojových prostředků tohoto projektu. Protože byly v souvislosti s OI řešeny v rámci tohoto projektu významné záležitosti je třeba shrnout poznatky o samotném OI. Informace k OI jsou čerpány z oficiální dokumentace *Coin2* [12] a OI [19].

Open Inventor je objektová nadstavba nad OpenGL. Stejně jako u OpenGL je třeba rozlišovat dva pohledy na OI: OI jako standardní API a OI jako balík implementující toto API. Open Inventor byl vyvinut firmou SGI, která vytvořila (a standardizovala) OI API a implementovala jeho první implementaci. SGI však OI později opustila. Jedna z aktuálních implementací OI je dnes *Coin2* od norské firmy *Systems In Motion*. Tento balík je běžně dostupný v repositářích Linuxových distribucí.

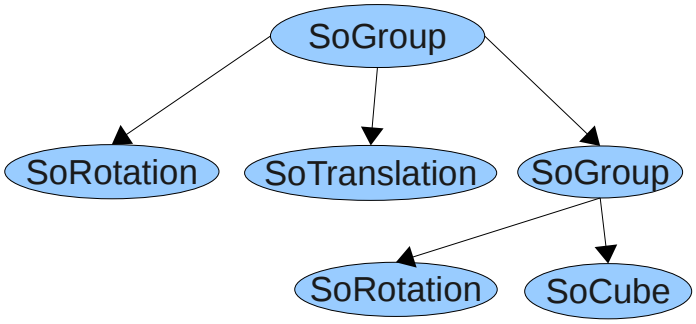
Základním jazykem, ve kterém se Open Inventor používá, je C++. Je však možné si nainstalovat i balíčky umožňující použít OI v jazycích jako je Java a Python. Fungování knihovny OI je na použitém jazyku nezávislé.

K čemu tedy vlastně Open Inventor slouží a jak funguje? OI umožňuje snadné sestavení 3D scény a její automatické vykreslování. Aplikace může pomocí této knihovny sestavit jeden či více stromů sestavených z OI objektů. Každý strom pak představuje 3D scénu. Pokud je kořen nějakého stromu předán zobrazovacímu výstupu, stává se tento strom aktuálním stromem na výstupu a je zobrazen. Zobrazovací výstup je reprezentován speciálními třídami poskytujícími rozhraní mezi aplikací a grafickým systémem operačního systému.

Zmíněné stromy se vytvářejí ze speciálních OI objektů - uzlů. Tyto objekty (v C++ slova smyslu) reprezentují uzly objektového modelu naší scény. Tyto uzly jsou provázány ukazateli. Uzly jsou buď listy (nejde k nim už připojovat žádné další uzly), nebo skupiny (uzly v pravém slova smyslu - k nim se připojují další uzly). Všechny typy uzlů jsou podtřídami třídy uzel (*SoNode*).

Po vytvoření stromu je tento předán výstupnímu rozhraní a scéna je vykreslena podle atributů uzlů které reprezentují objekty scény a podle atributů sourozeneckých a rodičovských uzlů které reprezentují změnu vlastností (poloha, materiál) dříve zmíněných uzlů. Dejme tomu, že chci například zobrazit kostku, která obíhá kolem středu scény a navíc se ještě točí kolem své vlastní osy. V OpenGL by se toto implementovalo dvěma rotačními

[5] Já sám jsem například do svých kampaní vložil možnost užívat *bullettime*^{TM} a na něm založené efekty. Při užití SQS byla implementace těchto efektů dokonce jednodušší, než by se dalo čekat. Viz příloha B.



Obrázek 2.4: Jendouchý graf scény v Open Inventoru. Uzly posunu a první rotace odsunou kostku od středu scény a zarotují s ní kolem středu scény. Druhá rotace pak otočí kostkou podle její osy. Samotný uzel kostky pak kostku vykreslí.

transformacemi, jedním posunem a samotným vykreslením kostky. V OI stačí pouze zapojit objekty do stromu popsaném v obrázku 2.4.

Renderování: Pro zobrazení scény na obrazovce jsou zapotřebí tři věci: graf scény, třída grafického rozhraní (zmíněno výše) a průchodové algoritmy.

Třídy grafického rozhraní OS je poskytována oddělenou knihovnou. V každé takové knihovně jsou dvě třídy: *SoXXX* a *SoXXXRenderArea* (XXX je název knihovny). Jejich úkolem je poskytovat komunikační vrstvu mezi aplikací a operačním systémem a vykreslovacím bufferem 3D karty. Jejich užití umožňuje aplikaci užívající OI vytvářet okna (případně vytvářet plné obrazovky a nastavovat jejich rozlišení a hloubku) a kreslit do nich.

Průchodové algoritmy pro renderování scény při žádosti o překreslení scény projdou grafem a provedou akce spojené s uzly daného grafu a parametrizované atributy jednotlivých uzlů. Tyto akce jsou uloženy ve třídách objektů představujících uzly, průchodový algoritmus je pouze volá.

Po celou dobu práce je renderovací proces skryt před programátorem, jak ostatně tvrdí jedno z hesel Open Inventoru [19, kap. 1]: *„Objects, not Drawings.“*
Toto je hlavní rozdíl mezi programováním v OpenGL a v Open Inventoru - v OpenGL programátor popisuje grafický výstup algoritmem, zatímco v Open Inventoru mu stačí pouze sestavit datovou reprezentaci zobrazované scény a algoritmicky pouze provádí její změny.

Interaktivní scény: Open Inventor umožňuje scény nejen jednoduše zobrazovat, ale je i jednoduše[6] propojit s uživatelským vstupem.

OI má totiž vestavěnou podporu pro interakci s objekty ve scéně za pomoci myši. Jádrem této podpory je operace zvaná *RayPick*, která bude popsána dále. Tuto podporu lze v OI využít třemi různými způsoby:

1. Vyžít uzlů grafu, které již mají interaktivitu kompletně naimplementovánu. Jedná se o tzv. *draggery* a *manipulátory*. Tyto uzly se ve scéně objeví jako viditelné 3D objekty, se kterými poté uživatel interaguje.

[6] S uvozovkami - viz další kapitoly.

2. Využít uzlů, které podporují interakci se scénou na poněkud obecnější úrovni a které pouze obalují části grafu scény, ale žádná tělesa do něj nepřidávají. Jedná se o *selektory* a *zvýrazňovače*.

3. Další možností je naimplementovat ošetření událostí zcela ručně. To může být učiněno dvěma způsoby:

   - Do grafu vložit uzel, který bude odchytávat události (v našem případě vstup z myši). Toto řešení je přenositelné a pracuje s OI událostmi.
   - Odchytávat události na úrovni okenního toolkitu. V tom případě budeme pracovat například s událostmi XWindow. Toto řešení je pochopitelně nepřenositelné.

Zmíněná operace RayPick je mechanismus, který je zabudovaný do OI a poskytuje výpočet průsečíku scény s paprskem. Pokud to sestavený graf scény podporuje, OI na základě kliknutí nebo pohybu myši vytvoří paprsek, který se položí do scény a vypočte se, který objekt (uzel) scény je tímto paprskem zasažen. Také je zjištěna cesta k tomuto objektu, neboť graf scény nemusí být strom, může to být i DAG. Získané informace jsou předány žadateli o RayPick.

Je jasné, že tato operace není proveditelná pouze na základě souřadnic myši a poloh jednotlivých objektů. Aby bylo možno ze 2D souřadnic vytvořit 3D paprsek, je nutno znát projekční a pohledovou matici. Tyto dvě matice jsou uloženy v uzlu kamery. Operace RayPick informaci z kamery načte a použije k vytvoření paprsku.

Shader programy v OI: V prostředí Open Inventor existují třídy uzlů, pomocí kterých je možno do grafu vložit shader program, který bude aplikován na všechny objekty pod vlivem uzlu programu. Konkrétně se jedná o třídy pro vložení vertex, geometry nebo fragment shaderů. Tyto tři typy programů jsou pak spojeny v uzlu shader programu, který všechny dílčí programy (pro každý typ ale nanejvýš jeden) spojí a aplikuje na danou část grafu.

Coin podporuje užití tří shader jazyků: ARB, CG a GLSL. CG nemusí vždy fungovat neboť na daném počítači nemusí být run-time podpora pro CG. CG programy je ale možno zkompilovat do ARB reprezentace. Tvůrci OI doporučují použití GLSL.

Kapitola 3

Zhodnocení současného stavu

Tato kapitola obsahuje zhodnocení jazyků a prostředí uvedených v předchozí kapitole z pohledu cílů této práce. Základními otázkami, které jsou zde kladeny, jsou: „Je možné se na tomto skriptovacím jazyku učit programovat a je tento jazyk pěkný?“, „Jak otevřené je toto prostředí pro vývoj vlastního obsahu obyčejnému uživateli?“, „Jak je řešeno celkové prostředí, do kterého je daný skriptovací jazyk nasazen - je dostatečně jednoduché a logické?“

Pro účely této práce je tedy zajímavé, zda daný jazyk a prostředí nabízí něco více, než např. jazyk C, ve kterém programátor vytváří aplikaci, která poběží na procesoru a bude využívat služeb operačního systému k dosažení nějakého užitečného efektu. Pro některé uživatele, kteří se teprve s programováním seznamují (obvykle studenti před maturitou), je tento přístup příliš nízkoúrovňový, pracný a nemotivující[1]. Fanouškovské skriptování herního obsahu ale přináší okamžité ovoce a motivaci k další práci a experimentování.

Tato kapitola také obsahuje dodatky k informacím uvedeným v části 2.3.

3.1 Zhodnocení jazyku Lua a dalších

Tvůrci jazyku Lua prohlašují, že se jedná o jednoduchý jazyk, avšak při pohledu na jeho současnou verzi a na jeho objektovou orientovanost, jeho podporu dalších paradigmat (funkcionální programování) a jeho podporu pro metaprogramování je vidět, že tomu tak není. Lua je jazyk poměrně pokročilý a pro začátečníky nepříliš vhodný.

Dále je o jazyce Lua známo, že je široce využíván nejrůznějšími počítačovými hrami [13]. Po projití jejich seznamu je však zvláštní, že s těmito hrami obvykle není spojován žádný komunitní vývoj obsahu v jazyce Lua. Buď proto, že jazyk Lua ani vůbec nebyl použit k vytváření herního obsahu (jak tomu bylo například v případě *Baldur's Gate*, jak bylo dříve uvedeno), nebo proto, že užití jazyka bylo rezervováno jen pro původní vývojáře hry a na záležitosti s herním obsahem (mapy, mise, videa) nesouvisejících (například jen AI). V případech jako je *Heroes of Might & Magic* nebo *Far Cry*, kdy tvorba herního obsahu byla běžnému uživateli umožněna, se převážně jednalo jen o klikání do editoru s grafickým GUI. To se rozhodně nedá studentům doporučit jako náhrada C a Pascalu.

Situace se však mění při pohledu na *Unreal* engine. Plnohodnotně je zde využíváno skriptovacího jazyka *UnrealScript*, který je otevřený běžným uživatelům a je vázán na jasně definované run-time prostředí.

[1] Z vlastní zkušenosti mohu uvést příklad mého mladšího bratra, kterého jsem se pokoušel naučit programovat a který mi řekl, že nepotřebuje Pascal k tomu aby sečetl dvě čísla.

Celá záležitost ale působí poněkud těžkotonážním dojmem. Jazyk je objektově orientovaný a nepokrytě je upozorňováno, že uživatel by už měl umět programovat [17]. Dále se uživatel musí seznámit s prostředím *Unreal* enginu a pochopit jeho systém, což je skoro stejně tak složité jako např. pochopit WinAPI. Rozhodně se jedná o jazyk a prostředí, které se uživatel musí naučit, místo toho aby se *na něm* učil. Z toho důvodu je *UnrealScript* z pohledu cílů této práce nepoužitelný. Podobné věci se dají říci i o *QuakeC*, i když tam je velikým plus podobnost s jazykem C.

Poslední zmiňované prostředí (*MaxEd*) se sice vyznačovalo velikou orientací na komunitní vývoj, ale vlastnosti hledané v této práci nemá.

3.2 Přínos Operation Flashpoint

Tato část se zabývá hodnocením přínosů OFP z hlediska této práce. Protože se jedná o věc, která mne velice motivovala, otázka zde kladená bude „Co bych udělal jinak a lépe?“ namísto „Je toto vhodné?“. V konečném důsledku ale projekt *DrdSim* po OFP téměř nic nepřebírá a řeší věci svým vlastním stylem.

První potíže přicházejí už se samotným skriptovacím jazykem SQS. Jedná se o jazyk nestrukturovaný a tudíž nevhodný pro výuku čistého a vzorného programování ve strukturovaných jazycích. Navíc se jedná o jazyk, který se jen těžko dá považovat za pěkný, spíše by se dal označit jako docela škaredý. Se svou prapodivnou syntaxí, prací s poli a dalšími podivnostmi by bylo lepší jej nedoporučovat lidem, kteří programovat neumějí. Pokud však už někdo zvládnul jazyky jako je Pascal a JavaScript, mohl by v OFP získat masivní průpravu. Navíc zde programátor může pochopit základy principů programování paralelních procesů. Pozitivní také je, že specifikace jazyka a vestavěných funkcí je obsažena přímo v nápovědě hry, což bohužel neplatí pro ostatní jazyky.

Co se týče různých třídně orientovaných jazyků v OFP obsažených, ani zde není možné nalézt nějaké pěkné věci. Třídní přístup je míchán s deklaracemi sekcí souboru a programátor musí vědět, co od něj interpret daného souboru očekává, a co by tam měl všechno napsat a kam. Navíc základní třídy dodané do hry samotnými tvůrci OFP jsou uloženy v binární podobě, a běžný uživatel si je tudíž nemůže prohlédnout a použít jako referenci. Textová podoba těchto standardních tříd byla naštěstí vydána alespoň odděleně.

Celkový interní systém hry není přehnaně složitý, ale je vidět, že nebyl navrhován s ohledem na běžné uživatele[2]. Je to zkrátka věc, která má určité chování, a kterou si komunitní vývojář musí sám vyzkoušet a přijít jí na chuť.

Další nevýhodou je, že se nejedná o čistě navržený systém a není zde možné aplikovat formální postupy návrhu. Systém také vykazuje značnou míru nedeterminismu - tvůrce skriptu může naimplementovat události jak chce, ale při jejich reálném běhu v simulovaném světě OFP má skript jen doporučující váhu[3]. Pokud chce programátor přesto dosáhnout nějakého přesného výsledku, je nutno tak učinit pomocí různých nečistých triků (tomu se nevyhnuly ani oficiální kampaně od výrobce hry). Mnoho věcí implementovaných v tomto prostředí pak působí velmi kostrbatě jak ve zdrojovém kódu tak ve hře.

[2] Vývojáře omlouvá to, že v době vývoje těchto partií nemohli obrovský úspěch hry a boom komunit předpokládat.

[3] Dejme tomu, že uživatel skriptuje animovanou scénu, ve které voják vystřelí raketu na tank. Pak je možné, že ve většině případů se tato scéna odehraje tak, jak by tvůrce skriptu očekával. Občas se ale také může stát, že se AI vojáka rozhodne napřed půl minuty plížit křovím do lepší pozice a teprve poté zaútočit. Tyto jevy není obecně možné předvídat ani eliminovat - viz teorie chaosu (malé nepřesnosti nyní vyvolají velké změny později).

I přes uvedené příklady vad měl projekt OFP veliký přínos z didaktického hlediska. Ve velice krátké době se kolem hry vytvořila veliká komunita mladých „vývojářů“, kteří začali s vervou pracovat v prostředí hry a produkovat výsledky, které mnohdy předčily původní kampaň hry. Já sám i mí spolužáci z gymnázia jsme sice v té době už znali programovací jazyk Pascal (a JavaScript, Shell a podobné) a tudíž jsme nepatřili k těm, co by se na OFP učili programovat, ale přesto i my jsme se tímto fenoménem začali zabývat. Rozhodně nemohu říci, že by nám tato zkušenost (z hlediska našich programátorských schopností) nějak ublížila, nebo že bychom z ní získali nějaké programátorské zlozvyky[4].

Dalším přínosem je, že na vývoji kampaní pro OFP je možno si vyzkoušet principy vývoje velkých projektů v týmu a řízení softwarových projektů. Hra obsahuje různé oblasti vyžadující odlišné druhy talentu, takže jakýkoliv projekt „uživí“ 6-8 lidí s různými rolemi, u větších projektů i více.

Hlavní ale je, že mnoho lidí z komunity OFP by se nikdy k programování nedostalo, případně jen v omezené míře požadované na jejich středních školách nebo až mnohem později na vysokých školách. To by pro ně znamenalo veliké mínus, pokud by se později chtěli více zabývat vývojem software. Pokud se totiž někdo nenaučí programovat v dostatečně mladém věku, může s tím později mít problémy. Sám jsem během svých vysokoškolských studií zaznamenal studenty, které s touto stránkou studia měli problémy a bez pomoci by nebyly schopni některé požadavky splnit.

Při velikém zjednodušení lze na systému OFP pozorovat koncept tří základních pilířů. Jedná se o mapu, třídně popsané objekty v ní a procedurální popis chování těchto objektů a průběhů událostí a řídící logiky. Podstata tohoto konceptu byla přejata i do projektu *DrdSim*. Základem jsou mapy uložené ve formátu XML. Dále je možno definovat třídy, které je možno do mapy umístit, a behaviorální záležitosti jsou popsány jazykem založeným na C. Oba tyto jazyky byly pro větší pohodlí programátora sloučeny - v souboru se skripty je možno nadefinovat třídu a ve třídě je možno nadefinovat obslužný kód.

3.3 Vyvození cílů

V předchozím textu byla popsána a z pohledu cílů této práce zhodnocena různá existující řešení. Některé vlastnosti byly v předchozím textu označeny jako nepříliš šťastné. Následující text uvádí předběžný souhrn oblastí, jejichž zlepšení by chtěl projekt *DrdSim* dosáhnout oproti skriptovacím jazykům jiných her i oproti běžným programovacím jazykům.

Prvním požadavkem je, aby použitý skriptovací jazyk byl jednoduchý a aby v něm bylo možné vytvářet dobře strukturovaný kód. Měl by být založen na nějakém dobře známém programovacím jazyku - konkrétně C, které má však svá komplikovaná zákoutí a proto by mělo být upraveno tak, aby bylo přísnější a čistší[5]. Prostředí jazyka by též mělo být bezukazatelové a obecně by nemělo být nízkoúrovňové. Dále by měl jazyk poskytovat dostatečné vestavěné prostředky, které umožní uživateli hned začít programovat[6].

[4] Abych byl zcela pravdivý, měl jsem nějakou dobu po svých zkušenostech s SQS sklon používat při programování v jazyce C návěští a `goto`.

[5] Příklad: v jazyce C je možno zápis `ar[2]` nahradit zápisem `2[ar]` (kde ar je pole). Jsem toho názoru, že kvintán by toto shledal matoucím.

[6] Předmaturitní programátor v C obvykle pracuje v rámci jednoho zdrojového souboru a o užívání knihoven ví pouze to, že když na píše `#include ⟨stdio.h⟩`, bude moci použít funkci printf. Použití rozsáhlých knihoven třetích stran (OpenGl, DirectX, atd.) je pro něj obvykle nemožné.

V porovnání s vývojovým prostředím jiných her by se systém *DrdSim* neměl opírat o bohaté grafické prostředí, které by zapouzdřovalo interní systém hry. Je sice rozumné umožnit uživateli naklikat mapy dobrodružství v grafické aplikaci a nenutit jej vytvářet je přímo v XML, ale vazba na objekty a funkce (reprezentované jejich zápisy v textových souborech) by měla být všudepřítomná. Rozhodně by se tvorba misí neměla sestávat z vkládání velkého množství ikonek do mapy a jejich propojování čarami [6].

Jazyky a run-time prostředí by měly být jasně a pevně definovány. Pravidla, kterými se jazyky řídí, by měla konzistentně platit v celém systému. Systém by měl mít logickou strukturu a měl by být dostatečně formálního rázu. Prostředky formálního návrhu software by zde měly být použitelné.

Při užití OO přístupu by se nemělo jednat o plnou práci s objekty jako je tomu v C++ a Javě, systém by měl obsahovat jen tolik OO paradigmatu kolik je nezbytně třeba.

Hlavním cílem by však měla být jednoduchost prostředí a snadnost jeho užívání. Mělo by být jasné, jaké prostředky má tvůrce dobrodružství k dispozici, a jak je lze použít. V rámci možností by vytváření herního obsahu mělo vyžadovat co nejnižší znalosti a dovednosti.

3.4 Dodatek k Open Inventoru

Prostředí *Open Inventor* (potažmo *Coin2*) bylo popisováno v části 2.3, která byla sepsána dle oficiální dokumentace. Tato část je naproti tomu založena na empirických zkušenostech a popisuje, jak se OI v některých případech chová doopravdy. Toto chování je v některých případech v nesouladu s dokumentací, ale většinou není dokumentováno vůbec. Důvodem není ani tak opomnění na straně tvůrců *Coin2*, jako spíše jejich nepředpokládání, že OI bude použit způsobem jako v tomto projektu. Konkrétním použitím OI v tomto projektu se zabývá kapitola 6.

RayPick při více kamerách: Jak již bylo řečeno, operace RayPick, nezávisle na tom, v jakém kontextu je použita, potřebuje ke svému správnému fungování kameru. Ve většině případů je v každém grafu jen jedna kamera a v takovém případě RayPick funguje v pořádku i bez přispění programátora.

Problémy však nastanou, když je v grafu umístěno více kamer. Dokumentace Coin2 se na pár řádcích ve FAQ zmiňuje, že použití více by nemělo působit žádné problémy a vůbec se nezabývá otázkou chování operace RayPick v takovém případě.

Co se tedy stane? Jednoduše řečeno, aplikace se zblázní a chová se nedefinovaně. Logické by bylo, kdyby objekty zobrazované jednou kamerou byly vyhodnocovány podle této kamery a objekty zobrazované druhou podle té druhé. Místo toho však Coin vezme jen jednu kameru a podle té vyhodnocuje vše. A nejedná se vždy o první, na kterou narazí - obvykle docházelo k tomu, že při kombinaci perspektivní kamery zobrazující 3D scénu a ortografické kamery zobrazující GUI se pro RayPick vždy brala informace z perspektivní kamery, ať už zapojení grafu bylo jakékoliv. Při mých pokusech toto zprovoznit se mi dokonce jednou stalo, že selektorové uzly braly při operaci RayPick informaci z jedné kamery a zvýrazňovací uzly braly informaci z druhé!

Z toho důvodu je nutno tyto situace ručně ošetřovat a to způsobem, který není nikde dokumentován ale je docela jednoduchý. V podstatě opět stačí událost z myši odchytit a ručně rozkopírovat do jednotlivých podgrafů tak, aby nemohlo dojít ke konfliktu kamer. Je však nutné dodržet určitý postup, jinak řešení nebude fungovat správně.

Užití shader programů: Jak bylo řečeno, nejjednodušší a doporučovanou volbou pro užití shaderů v OI je použít jazyk GLSL. To však s sebou nese jisté problémy. V době zavedení této podpory do Coin bylo GLSL ve verzi 1.0/1.1. V dalších verzích však došlo v GLSL ke změnám, se kterými se v OI nepočítalo a které znemožňují použití těchto novějších verzí v OI. Konkrétně se jedná o zrušení vestavěných proměnných a funkcí, které zajišťovaly datový vstup pro shadery. Když například programátor v GLSL verze 1.4 potřebuje pracovat se souřadnicí vrcholu, musí si ji z OpenGL ručně naimportovat. Pokud potřebuje transformační matici, musí ji naimportovat. Tyto věci byly v GLSL 1.0/1.1 prováděny automaticky a OI nemá žádný mechanismus, kterým by mohl pracovat s GLSL vyšší verze a jeho vysokoúrovňovost programátoru znemožňuje, aby toto nějak obešel.

Vysokoúrovňovost OI také způsobuje, že mnoho klasických shader efektů je v OI nerealizovatelných. Jedná se například o bloom a podobné post-procesové efekty nebo environmental mapping. To je způsobeno tím, že v OI je odříznut přístup k vykreslovacímu procesu, takže není například možné pracovat s buffery[7]. Následkem toho je možno v shader programech použitých spolu s OI pracovat „jen s tím co mám“ - tedy souřadnice vrcholu, barva meteriálu, přístup k aktivním texturám, světla, atd.

[7] Teoreticky by ale mělo být možné vytvořit implementaci třídy *SoXXRenderArea*, která by toto nějakým způsobem zprostředkovala.

Kapitola 4

Plán projektu

Tato kapitola obsahuje plánování, analýzu a předběžný návrh projektu (podrobné vývojové návrhy jsou obsaženy v kapitolách popisujících konkrétní oblasti projektu). Část 3.3 obsahovala předběžný nástin cílů projektu ve vztahu k ostatním produktům na trhu; tato kapitola naváže podrobnou specifikací požadavků.

4.1 Návrh hry

Při použití skriptovacích jazyků ve hrách není důležitý pouze interpret a architektura jeho run-time prostředí, ale i samotná hostitelská hra, která musí být dostatečně originální, aby se vůbec někdo obtěžoval v jejím prostředí něco vytvářet. Tato část se zabývá tím, jak by hra *DrdSim* mohla nebo měla vypadat a jak bude zhruba přistupováno k vývoji jejího obsahu. Je nutno odpovědět na otázky jako „Jaký bude žánr hry?“, „Jak bude vypadat?“, „Jakými pravidly se bude řídit virtuální svět hry?“, „Jak se bude ovládat?“, „Jak k ní budou různí uživatelé přistupovat?“ a podobné. Tyto otázky jsou v této kapitole řešeny jako první, neboť koncept hry může ovlivnit, jaké budou požadavky na integrované jazyky a virtuální prostředí.

Žánr hry: Žánr má veliký vliv na systém hry a tedy i na vývojové prostředí jejího obsahu. Dá se předpokládat, že programování misí pro letecký simulátor nebo strategii bude vypadat jinak, než pro akční hru nebo RPG. Vzhledem k cílům projektu je lepší zvolit žánr, který umožní uživateli programovat na úrovni, která je mu bližší a kde může pracovat s objekty v měřítku, jako je on sám. Dobrými volbami jsou tedy žánry jako je například akce, RPG nebo adventura. První z těchto uvedených žánrů byl použit v projektu *Operation Flashpoint*, i když žánr OFP se dá přesněji určit jako taktický simulátor. To, jak již bylo uvedeno v části 3.2, způsobuje značný nedeterminismus celého systému.

Od RPG nebo adventury se dá naopak očekávat, že systém bude v rozumné míře předvídatelný, obzvláště pokud by se jednalo o tahové RPG. Je jasné, že naprostý determinismus není žádoucí - například akce hráče nejdou předvídat a dále by všechny[1] akce a události měly mít určitou pravděpodobnost úspěchu. Bylo by však vhodné, aby měl programátor možnost naprogramovat deterministickou událost a spolehnout se na to, že akce, které naprogramoval, se opravdu stanou. O determinismu systému bude řeč v pozdějších částech dokumentu. Žánr RPG navíc přináší nejbohatší možnosti co se týče vyžití v definování

[1] Do rozumné míry - všudepřítomná náhodnost občas způsobuje opakované nahrávání hry a zkoušení náhodné akce znovu a znovu což je frustrující.

různých objektů a v implementaci řízení scénáře příběhu, zvláště pokud jsou do žánru přineseny adventurní prvky (i když to samé by se dalo říci o akčních hrách).

Konečné rozhodnutí bylo vytvořit tahovou RPG hru s diskrétním herním prostorem a časem. Obecnost užitých interpretů by navíc měla umožnit vložení adventurních prvků.

Pravidla hry: Každá hra patřící do žánru RPG vyžaduje systém pravidel, podle kterých se hra řídí. Jedná se o pravidla řídící záležitosti jako jsou vlastnosti postav ve hře a jejich vylepšování, boj, magie a další. Systém musí být logicky navržený, jednoduchý a dobře vyvážený, jinak hráčům způsobuje značné frustrace.

Hra může využívat buď vlastní systém pravidel navržený přímo pro danou hru, nebo může převzít pravidla od existující stolní hry (přesněji řečeno na ně zakoupit drahá práva). První přístup byl aplikován například ve hře *Evil Islands* [7] a ačkoliv systém pravidel této hry nebyl zcela špatný, měl daleko k dokonalosti (hlavně co se týče nevyváženosti inflace zkušeností při vylepšování postavy). Druhý přístup byl použit například v sérii *Baldur's Gate* [3] a žádný hráč si nemohl stěžovat.

Po zralé úvaze jsem se rozhodl v projektu *DrdSim* využít pravidla stolní hry *Dračí Doupě* [4] (verze 1.5) od nakladatelství ALTAR™[1]. Tato stolní hra nakonec dala tomuto projektu jméno. Pravidla Dračího Doupěte byla využita se svolením nakladatelství ALTAR[2].

Užívání systému: Dále je zapotřebí předběžně naplánovat, jak bude celý systém používán. Tedy nejen samotná aplikace hry, ale i její okolí (jazyky, adresářová struktura, pomocné aplikace, atd.).

Přísně vzato, se systémem budou pracovat dva, respektive tři, typy uživatelů:

- Hráč - spustí hlavní aplikaci, v menu nastartuje novou hru, vybere dobrodružství a hraje.
- Tvůrce herního obsahu[3]. - používá editor map a textové editory pro vytváření map, tříd a funkcí spojených do dobrodružství (kampaně). Hlavní aplikaci používá na testování a ladění svého výtvoru. Při tvorbě může využívat základní třídy a funkce poskytované hrou.
  - Tvůrce sdíleného obsahu - vytváří základní třídy a funkce zmíněné výše, které jsou definovány v textové formě (nejsou vestavěné do aplikace) a jsou dostupné všem dobrodružstvím.

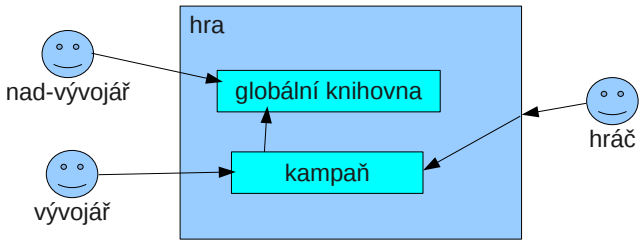
Výše uvedené je neformálně shrnuto na obrázku 4.1.

Dle hlavních cílů projektu by měla být pozornost soustředěna na tvůrce herního obsahu (pro jednoduchost dále jen PJ - pán jeskyně). Co všechno bude PJ při užívání systému potřebovat a jak bude zhruba postupovat?

- Nejprve se PJ pravděpodobně seznámí se systémem a API hry.
- Následně v adresáři hry vytvoří novou složku, ve které bude vytvářet své dobrodružství.

[2] Altar je známý zaštiťováním podobných softwarových projektů. Stačilo vyplnit jejich univerzální formulář (stejný formulář jsem před deseti lety použil k položení dotazu „Proč nám při hře neustále na kostkách padá kombinace 6+6+1?“) a v krátké době mi e-mailem přišlo svolení od pana Martina Kučery.

[3] Tedy jakýsi pán jeskyně (viz [5]), který ale pouze vytváří hru - neřídí ji, to dělá program hry.



Obrázek 4.1: Neformální schéma uživatelů DrdSim.

- Poté buď za pomoci pomocné grafické aplikace nebo ručně v textovém editoru vytvoří mapy svého dobrodružství.
- V textovém editoru vytvoří objekty (vyděděné z již předdefinovaných tříd hry), které ve vytvořených mapách použije, a napíše funkce, které budou dobrodružství řídit.
- Dá se očekávat, že PJ bude chtít, aby dobrodružství bylo lokalizovatelné. V tom případě by se všechny řetězce měly načítat ze speciálních lokalizačních souborů.
- Dále se dá očekávat, že bude nutno zavést nějaký druh definičního souboru, který bude definovat vlastnosti dobrodružství (důležité např. při zakládání nové hry).

Jak je vidět, krom samotné aplikace hry bude ještě zapotřebí vytvořit pomocnou aplikaci pro vytváření map. Bude také třeba navrhnout celkový systém hry. Ten by měl být dostatečně jednoduchý a intuitivní. Fungováním celého systému se podrobněji zabývá část 4.2 a kapitola 5.

Grafika hry: Ačkoliv dobře navržená vývojová platforma je pro hru tohoto typu důležitá, grafický vzhled samotné hry je neméně důležitý. Od počátku vývoje tohoto projektu však bylo jasné, že nebude čas vyvinout 3D engine o rozsahu a kvalitě komerčních produktů[4]. Rozhodl jsem se proto nahradit propracovanost originalitou. Protože je hra *DrdSim* je založena na stolní hře *Dračí Doupě*, měly by v ní figurovat stejné prvky jako v originále - papírová mapa a osobní deníky[5].

Nakonec jsem se rozhodl, že hra bude obsahovat dva odlišné módy pohledu na herní svět, které budou sdílet podobné koncepty ovládání a mezi kterými bude možno přepínat. První mód měl zobrazovat herní stůl a na něm rozložené papíry a teprve až na nich zobrazený herní svět. Druhý mód měl být vizualizační, zobrazující herní svět přímo ve 3D, ale v nějakým způsobem abstrahované formě která by odrážela stolní předlohu. Původní návrh šel dokonce až tak daleko, že při dobrodružství v exteriérech dostatečně velkého měřítka (kontinenty) mělo být vidět zaoblení obzoru a jevy ve vyšších vrstvách atmosféry.

Po zahájení implementace však došlo k „realistickým škrtům“ a bylo rozhodnuto, že hra bude obsahovat jen jeden mód. Protože bylo jasné, že v oblasti druhého módu by hra příliš neprorazila, byl zvolen první zmíněný přístup. V konečné implementaci tedy hra neoslňuje propracovanou 3D grafikou s fotorealistickými efekty. Místo toho má kontroverzní pojetí, které spíše zaráží. Grafikou hry se více zabývá kapitola 6.

[4] A opravdu doufám, že ode mne nikdo neočekával grafický engine o kvalitě *Dragon Age*.
[5] Papír, na kterém jsou napsány informace o postavě, za kterou hráč hraje.

Uživatelské rozhraní: Zde není třeba vymýšlet žádná kontroverzní řešení. Stejně jako ve všech RPG hrách (a nakonec i RTS, některých adventerách, atd.) i zde byla zvolena kombinace 2D GUI nataženého přes scénu, klávesnice a samotné scény obohacené o interaktivitu. Hlavním pravidlem je, aby všechno (tedy i 2D GUI) bylo renderováno a obsluhováno přímo v aplikaci - nejsou tedy využity žádné externí 2D frameworky.

Nakonec bylo rozhodnuto, že hra bude obsahovat klasickou sestavu různých obrazovek menu, přes které bude možno hru nastavovat a ze kterého bude moci hráč spustit nové tažení, a že v samotné hře bude dostupné 2D GUI sestavené z tlačítek a dalších prvků. Většina akcí proveditelných přes toto menu je proveditelná i klávesovými zkratkami. Kamera je ovladatelná myší nebo klávesnicí standardním způsobem, pouze je větší důraz na soustředění kamery na důležité prvky hry rozložené na stole (mapa, osobní deníky). Dále samotné prvky a oblasti scény mohou být interaktivní (políčko, na které se má postava přesunout, nepřítel na kterého má zaútočit, atd.).

4.2 Návrh vývojové platformy

Tento aspekt projektu byl již v předchozích částech textu dostatečně popsán a na různých příkladech byla probírána různá pro a proti. Tato část tedy pouze shrne, jak budou vypadat jazyky užité ve hře a jaká bude struktura jejich run-time prostředí. Popis konečného řešení této oblasti je možno nalézt v kapitole 5.

Použité jazyky: Jak již bylo naznačeno, ve hře by měly figurovat dva hlavní jazyky: jeden deklarativní objektově orientovaný jazyk na definici objektů vyskytujících se ve hře a druhý procedurální na vytváření skriptů řídících hru.

První jazyk je určen pro definici objektů vyskytujících se ve hře. Tedy například vybavení postav (brnění, zbraně, atd.) nebo nestvůry, které je možno během dobrodružství potkat. Jazyk umožňuje dědičnost (například ze třídy „zombie“ je možné vytvořit odvozenou třídu „radioaktivní zombie“).

Druhý zmíněný jazyk je založen na jazyce C, ale očištěný, zjednodušený a bez ukazatelů. Soubory tohoto typu budou obsahovat pouze definice funkcí - protože se jedná o interpretovaný jazyk, není třeba deklarovat prototypy funkcí nebo definovat globální proměnné. Téměř všechny sémantické kontroly, které se v C provádějí v době kompilace, v něm budou prováděny až za běhu.

OO jazyk na definici objektů měl být dle původních návrhů zcela oddělen od procedurálního. Zdrojové soubory napsané v tomto jazyce měly obsahovat jen definici objektů připomínající definici datových struktur v C. Tedy objekty měly obsahovat jen seznam členských atributů a navíc ještě jejich hodnoty[6]. Počítalo se s tím, že pokud by bylo v nějaké třídě potřeba odkázat na nějakou obslužnou funkci, byl by název této funkce uveden v řetězcovém atributu. Případné argumenty by byly uvedeny jako další atributy.

Tento přístup byl očividně okopírovaný ze hry *Operation Flashpoint* (viz 2.2). Vedoucím práce jsem byl naštěstí upozorněn na to, jak velký diskomfort by to uživateli způsobovalo. Bylo tedy umožněno, aby v atributu obsluhy události objektu šlo definovat přímo kód procedurálního jazyka, a to ne jako řetězec, ale jako blok kódu.

[6]Pokud je ve hře objekt, který představuje například skřeta, tak je informace, že tento objekt má číselný atribut určující počet životů (ekvivalent zdraví), zcela neužitečná. Je nutno vědět přesnou hodnotu tohoto atributu.

Kvůli výše uvedenému, a také pro další zvýšení uživatelské přívětivosti, byly oba zde uvedené jazyky spojeny. Jsou tedy zpracovávány stejným analyzátorem a jeden zdrojový soubor tak může obsahovat jak funkce, tak třídy. To může být užitečné, pokud by si uživatel například vytvořil třídu a hned poblíž by si chtěl nadefinovat krátkou funkci, která bude obsluhovat nějakou její událost.

Je však třeba se zeptat: nekoliduje toto rozšíření s původním záměrem učinit OO přístup co nejjednodušší (pokud bude vůbec použit)? Ve třídách by tedy správně měly být pouze atributy a žádné metody. Je však třeba mít na paměti, že uvedený návrh nepřináší do specifikace tříd skutečné metody. Jedná se pouze o registraci obslužných funkcí, které jsou volány systémem automaticky - uživatel je sám volat nebude a nebude tedy s nadefinovanými třídami pracovat jako s třídami v C++. Je sice pravda, že vložení bloku kódu do definice třídy je určitý myšlenkový skok pro středoškoláka, avšak stále je to lepší alternativa než nečistý způsob zápisu pomocí řetězcových hodnot.

Krom těchto dvou uvedených jazyků (spojených do jednoho) vyžaduje hra ještě další jazyky/formáty. Mapy herního světa budou ukládány ve formátu XML. Soubory map nebudou psány ručně, ale budou generovány za pomoci grafického editoru. Dále je zapotřebí nějaký deskriptor kampaní. Ten obsahuje informace jako je název kampaně, pro kolik postav je kampaň určena, kolik postav si může hráč na začátku vytvořit nebo naimportovat[7], apod. Dále by byl vhodný formát umožňující snadnou lokalizaci textů.

Prostředí jazyků: Předchozí úsek popisoval návrh jazyků použitých ve hře. Další text se bude zabývat prostředím, kde tyto jazyky budou běhat.

Dobrý návrh syntaxe je důležitý pro každý programovací jazyk. Ale z hlediska tohoto projektu je ještě důležitější to, do jakého prostředí budou navržené jazyky zasazeny. Profesionální programovací jazyky také nemají škaredou syntax ale laiky k sobě nepřitahují (alespoň ne jako forma zábavy). Naopak již na mnoha místech uváděný *Operation Flashpoint* měl syntax ošklivou, ale přesto díky zbytku produktu přitáhl veliké množství fanoušků.

Je proto nutné pamatovat, že jazyky a v nich napsané skripty jsou pouze ovládacím nástrojem pro „zákulisí“ herního světa. To musí být na jedné straně snadno pochopitelné a ne moc rozsáhlé a na druhé straně musí programátorovi umožnit dosáhnout velikých výsledků. Jednou z hlavních otázek také bude: jakým způsobem budou programátor s tímto zákulisím komunikovat?

Nejprve by bylo vhodné předběžně shrnout, které informace je třeba ve hře *DrdSim* uchovávat. Nebo přesněji, co by mělo tvořit stavovou informaci virtuálního světa v *DrdSim*?

Pochopitelně to jsou zejména mapy a jejich stav (po průchodu družiny). Dále tabulka všech vytvořených tříd, nadefinované funkce a v nich vytvořené globální proměnné. Některé z uvedených informací (stav map, proměnné) budou ukládány při uložení hry na disk. Může se jednat například o boolean proměnnou značící, zda nějaký politik přislíbil družině podporu.

Prostředí bylo naplánováno jako dynamické (alespoň na globální úrovni). Existence funkcí, objektů, map, globálních proměnných atd. bude tedy ověřována až za běhu. Parametry funkcí, lokální proměnné apod. budou ošetřovány během kompilace (ale jejich typ až za běhu).

[7] Například ve hře *Baldur's Gate* bylo možno mít v družině až šest postav ale na začátku si hráč vytvořil pouze jednu a ostatní musel najít v průběhu hry. Ve hře *Icewind Dale* si naopak hráč vytvořil hned při startu až všech šest.

Jednotlivé oblasti stavu hry k sobě musí mít dle potřeby přístup. Pokud bude chtít např. zloděj družiny v dialogu obelstít nějaké NPC, měl by mít skript, který daný dialog ovládá, možnost přečíst informaci o stupni inteligence daného NPC[8].

A nyní k hlavní otázce: jak bude programátor za pomoci plánovaných jazyků ovládat dění ve hře? V oblasti definovaných objektů bude nejlepší napodobit přístup OFP. Mělo by sice být možné ve třídách definovat jakékoliv atributy, které pak budou přístupné ze skriptů, ale atributy určitých předdefinovaných jmen budou rozpoznávány interním systémem hry a budou ovlivňovat způsob, jakým se budou objekty dané třídy chovat (bez přičinění programátora). Určitý atribut by měl značit, o jaký druh objektu se vůbec jedná. Další atributy mohou určovat další dělení do poddruhů a ve třídách určitých takovýchto podskupin hra zase bude automaticky rozpoznávat atributy určující konkrétní vlastnosti daného objektu.

Hra bude šířena spolu s balíkem nadefinovaných základních tříd, které tuto architekturu budou kopírovat. Tvůrci kampaně bude tedy stačit svou třídu vydědit z některé z nabízených základních tříd a upravit jen ty atributy, které potřebuje. Například dejme tomu, že PJ chce do truhly v podzemí umístit kouzelný meč s bonusem +3 k útoku. Vytvoří tedy třídu vyděděnou z již vytvořené třídy kouzelného meče a nastaví jí jediný atribut: sílu zbraně o 3 větší než kolik udává tabulka (buď vyčte z pravidel nebo z definice obyčejného nemagického meče - konstrukce typu `super.atr+3` jsou poněkud náročné). Nově nastavená hodnota se použije pro výpočet útočného čísla postavy které se pak objeví v jejím osobním deníku a které se použije při útoku na nepřátele.

V oblasti procedurálního jazyka bude tvůrce kampaně využívat vestavěných funkcí, které budou schopny spouštět operace interního systému *DrdSim* (jako je např. změna mapy). Tento přístup byl extenzivně využit v OFP a osvědčil se. Také by nebylo na škodu zavést určité předdefinované globální proměnné, které budou hrou čteny nebo do kterých bude hra zapisovat.

V oblasti map není třeba dělat žádné veliké plány neboť tato část bude ošetřována grafickou aplikací která bude kopírovat to, co umožňuje i samotná hra.

Co se týče práce v prostředí *DrdSim* na té nejnižší úrovni, tedy na úrovni adresářů a souborů, měla by být maximálně jednoduchá a srozumitelná. Tvůrci by měl stačit správce souborů (případně obyčejný průzkumník) a obyčejný textový editor.

Při založení nové kampaně bude PJ muset vytvořit v hlavním adresáři hry nový podadresář a v něm popisný soubor. Pokud bude chtít kampaň lokalizovat, tak i lokalizační soubor. Dále vytvoří tři adresáře s mapami, třídami a skripty, které budou při načtení kampaně automaticky prozkoumány. Vzhledem k propojení jazyků je toto dělení do adresářů jen doporučující - bude klidně možné v adresáři pro skripty definovat třídy a naopak.

4.3 Technologie

Tato část se zabývá plánováním, které technologie jsou použitelné pro vývoj hry (jazyky, knihovny atd.).

Co se týče volby jazyka, tak v případě aplikace samotné hry se jednalo o rozhodnutí mezi C a C++ (případně kombinace). Zde by bylo možné diskutovat o tom, že kód napsaný

[8] Nyní by se mohl někdo zeptat, proč toto nerealizovat jako obecně použitelnou speciální schopnost a zbavit tím PJ nutnosti implementovat tuto možnost ručně ve skriptu. Překážkou tomuto přístupu je obecnost, širokost a nedeterminismus pravidel DRD, které vždy potřebují živého PJ aby rozhodnul co která akce skutečně provede a nikdy se zcela nespoléhá na tabulky a vzorečky. V případě PC hry tedy musí tvůrce kampaně dopředu naprogramovat, co která akce způsobí. To je sice pro programátora pracnější, ale stále relativně jednoduché.

pouze v C je efektivnější než ten napsaný v C++. Bylo však nutno zůstat nohama na zemi a pamatovat na to, že na vývoj projektu není k dispozici neomezený čas. Implementace v C++ je pohodlnější, rychlejší a méně pracná než v C. Proto bylo rozhodnuto naprogramovat aplikaci v C++. Otázka, zda je to vůbec vhodné, vůbec nebyla řešena. Bylo by však moudré se alespoň vyhnout použití virtuálních metod, které by aplikaci ještě více zpomalily.

Krom aplikace samotné hry je zde ještě aplikace grafického editoru map. Bylo rozhodnuto, že se bude jednat o klasickou okenní aplikaci která nebude vyžadovat veliký výkon nebo běh v reálném čase. Bylo by vhodné, aby tato aplikace byla snadno přenositelná. Dále by implementace této aplikace měla být nanejvýš rychlá a pohodlná, neboť se jedná jen o pomocnou utilitu, která vůbec není předmětem tohoto projektu. Na základě výše uvedeného nebyl nalezen žádný důvod, proč by editor map (*MapBuilder*) neměl být naprogramován v jazyce Java.

Dále bylo třeba naplánovat, jaké knihovny budou v projektu použity. Bylo by vhodné, aby hra byla pokud možno (snadno) přenositelná. Dobrý způsob, jak toho docílit, je použít knihovnu SDL, která poskytuje zapouzdření zvuku, správy oken (respektive plné obrazovky), uživatelského vstupu a dalších záležitostí. Je snadno použitelná: na Linuxu je dostupná mezi standardními balíčky a na Windows je její použití otázkou přiložení jediného dll. Neposkytuje prostředky pro multiplatformní práci se soubory a adresáři, ale to není problém naimplementovat ručně pro jednotlivé cílové systémy.

Co se týče 3D knihovny pro implementaci grafiky, je volba docela jasná: OpenGL. Zajímavější otázka však je, zda bude použito jen samotné OGL, nebo zda bude použita nějaká zapouzdřující objektově orientovaná nadstavba. Zde narážíme na stejný problém jako u volby jazyka - málo času na vývoj. Bylo by nerealistické očekávat, že by bylo možné projekt stihnout při použití OGL bez nadstaveb. Je sice pravda, že použití OO nadstavby s sebou přinese své vlastní problémy, jako je např. nutnost naučit se se systémem pracovat apod. Přesto je však použití nadstavby méně pracná a proti chybám více odolná alternativa.

Existují různé nadstavby pro OpenGL s různými politikami a volnostmi užití. Jako nejjednodušší varianta se nabízí knihovna *Open Inventor*, respektive její implementace *Coin2*. Ta umožňuje vytváření scén za pomoci grafů a nabízí podporu pro interaktivitu a snadné použití 3D modelů. Problémem však je, že tato knihovna není vhodná pro vývoj počítačových her, hodí se spíše pro desktop aplikace které z nějakého důvodu potřebují 3D výstup. Navíc její síla se neshoduje se silou OGL - v Open Inventoru je možno vytvořit méně věcí než v OGL protože politika OI s sebou přináší jistá omezení. Programátor například zcela ztrácí kontrolu nad vykreslovacím procesem - knihovně pouze předává data a ta je pak automaticky vykreslí. Jedno z hlavních hesel Open Inventoru dokonce zní: „Objekty, ne vykreslování.“ To sice zní dobře a šetří programátorovi značné množství práce, ale v konečném důsledku ochuzuje aplikaci. Dále může použití OI vést na pomalejší aplikaci, ale to jen za předpokladu, že by ji programátor zvládl naimplementovat ručně efektivněji. I přes uvedené nedostatky nezbývá než se přes ně přenést a knihovnu Open Invetor (Coin2) použít.

Volba nadstavby ale s sebou přináší jeden problém. V předchozím textu bylo naplánováno, že bude použita knihovna SDL. OI však používá své vlastní okenní rozhraní, například WinAPI na Windows nebo X11 na Unixu. Nebyl by sice veliký problém naprogramovat aplikaci multiplatformně, ale použití SDL přináší jisté výhody. Naštěstí na půdě FIT VUT vznikl modul, který umožňuje použít SDL spolu s OI a to s nulovou námahou na straně programátora. Tento modul byl zvolen pro použití v projektu.

Dále je třeba rozhodnout, zda a jaké prostředky budou použity na kompilaci jazyků

vestavěných do hry. Zde se není třeba dlouho rozmýšlet, flex a bison jsou jasná volba. Na zpracování a zápis XML souborů je možno použít knihovničku *TinyXML*.

Co se týče editoru map (vyvíjeném v Javě), tak standardní knihovny JRE zcela postačí na vytvoření GUI i práci s XML. Je zde třeba jen pamatovat na možnost problémů při použití alternativního JRE (např. *IcedTea*) a naprogramovat aplikaci opatrně.

4.4 Shrnutí požadavků

Požadavky kladené na tento projekt byly v tomto dokumentu již probírány a proto je v této části uveden jen jejich shrnující výčet:

- Jazyky musí mít pěknou syntaxi.
- OO jazyk na definici objektů, procedurální jazyk na skripty.
- OO jazyk jednoduše použitelný bez nutnosti pochopení OO paradigmatu. Naopak pochopení tohoto jazyka by mělo přispět k pozdějšímu snazšímu pochopení OO přístupu.
- V procedurálním jazyku nemuset řešit nízkoúrovňové věci (např. práce s poli a ukazateli).
- Dobré propojení obou jazyků, které zvýší komfort programátora a nezkomplikuje jazyky.
- Architektura run-time prostředí jazyků dostatečně jednoduchá a pochopitelná pro laiky, snadnost použití a dobře definované vestavěné prostředky.
- Možnost použití metod (semi)formálního návrhu kampaně.
- A především: tvorba v *DrdSim* musí být zábava!

Kapitola 5

Systém a interpret hry

Zbývající část této práce se bude zabývat konečným návrhem a implementací projektu *DrdSim*.

V této kapitole bude popsána celková architektura aplikace a principy její interního systému. Tato kapitola také popisuje jazyky použité v projektu a virtuální prostředí k nim připojené. Protože se jedná o oblast důležitou z hlediska tématu projektu, budou použité jazyky popsány pokud možno co nejobsáhleji.

5.1 Návrh architektury

Nejprve je třeba definovat, jakou bude mít hra celkovou strukturu. Na aplikaci *DrdSim* a její strukturu je pro tento účel možno nahlížet následujícím způsobem:

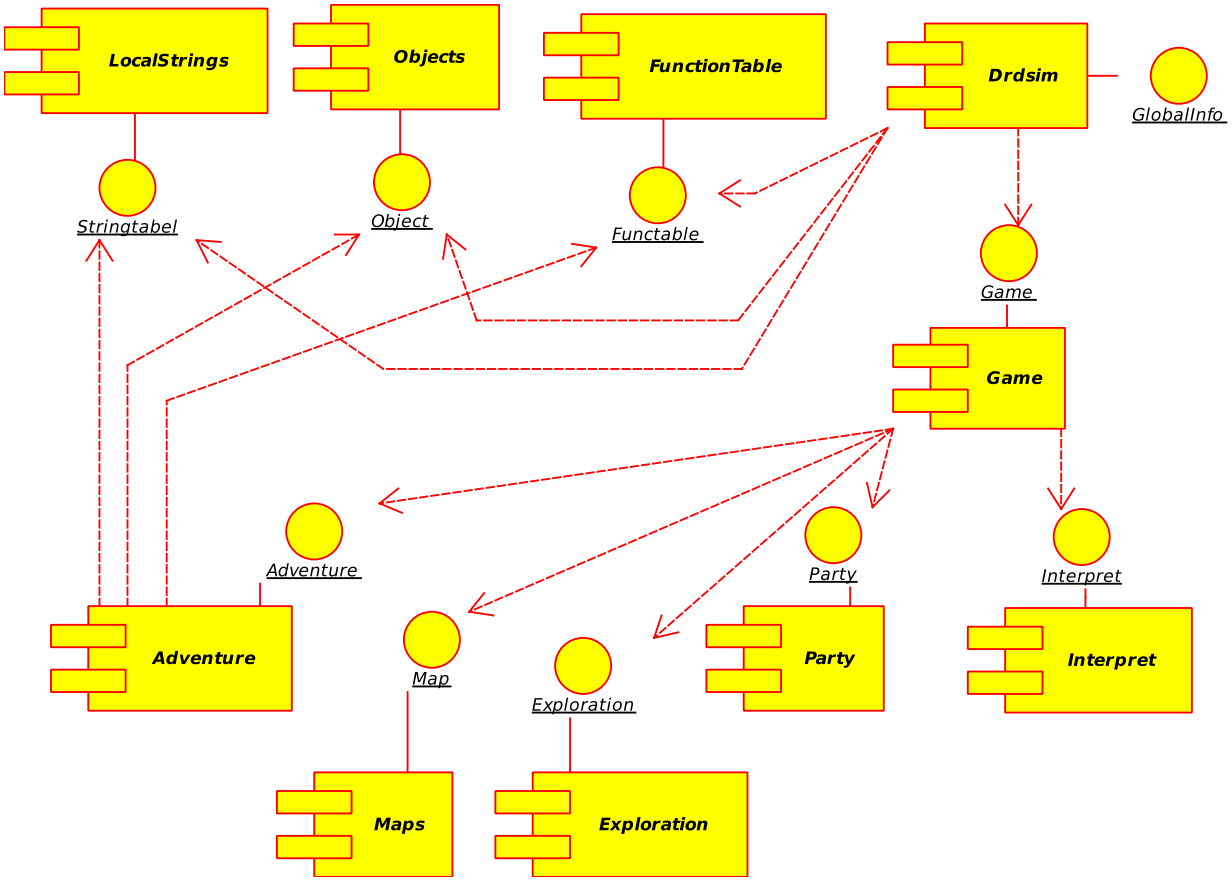
> Aplikace obsahuje data představující kompletní stav světa aktuálně rozehrané kampaně. Tento stav (respektive jeho část) je hráči prezentován za pomoci grafického a zvukového výstupu a hráč jej může ovlivňovat za pomoci ovládacího rozhraní.

Co bude tvořit tuto stavovou informaci? A co bude tvořit meta-informace (jako např. funkce)? Je jasné, že budou vrstveny do několika úrovní. Na vrcholu budou data společná všem kampaním - třídy, funkce a lokalizace poskytované v hlavním adresáři hry. To samé se bude opakovat v informacích o aktuální kampani. Krom toho to bude ještě aktuální mapa (načítat všechny mapy najednou by nebylo úsporné).

Výše zmíněné byly meta-informace, které budou načteny při startu hry a pak už se měnit nebudou. Samotná stavová informace bude složena z následujících tří částí:

- Informace o průchodu družiny herním světem (záznam průzkumu).
- Globální proměnné reprezentující dodatečná data nezaznamenatelná výše uvedeným.
- Informace o družině (úrovně zkušenosti a stupně vlastností jednotlivých charakterů atd.).

Data z těchto tří skupin definují stav hry a budou ukládána na disk při uložení hry. Krom toho je však ještě třeba pamatovat na virtuální stroj interpretu funkcí a jeho stav. Tento stav však bude pouze pomocný, po ukončení běhu funkce/funkcí se zahodí a na disk ukládán nebude. Z toho důvodu jsou globální proměnné součástí aktuální hry, ne interpretu.



Obrázek 5.1: Neúplný diagram ukazující některé komponenty nejvyšší úrovně v *DrdSim*. Rozhraní komponenty je ke ní připojeno plnou čarou. Čárkovaná šipka značí užití rozhraní.

Většina výše uvedeného je shrnuta v diagramu 5.1. Diagram by bylo možné dále zjemňovat, ale to není třeba.

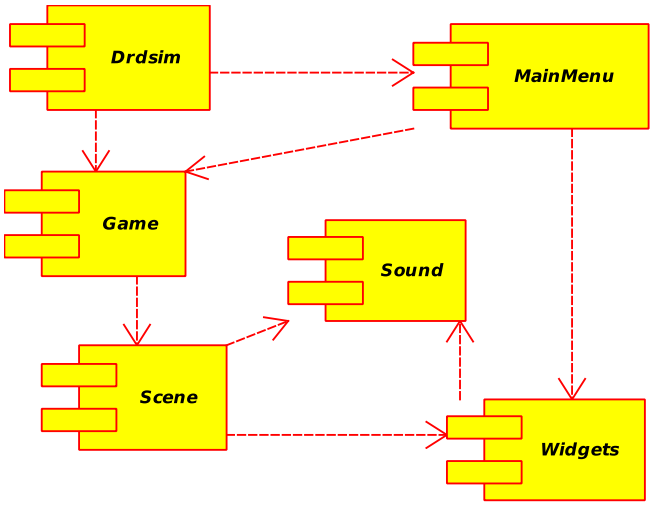
Propojení mezi virtuálním strojem a hrou bude prováděno pomocí globálních proměnných a vestavěných funkcí interpretu, které budou schopné operovat přímo s interním stavem hry.

Dále budou v aplikaci zapotřebí další komponenty, které budou mít na starosti grafiku, ozvučení a poskytnutí uživatelského vstupu. Renderování grafiky v průběhu hry bude nejlepší řídit z komponenty `Game`. Na ovládací GUI bude zapotřebí samostatná komponenta. Ta bude využívána jednak přímo ve hře a jednak v hlavním menu, které se uživateli zobrazí po spuštění aplikace. Zjednodušený návrh komponent je popsán v diagramu 5.2.

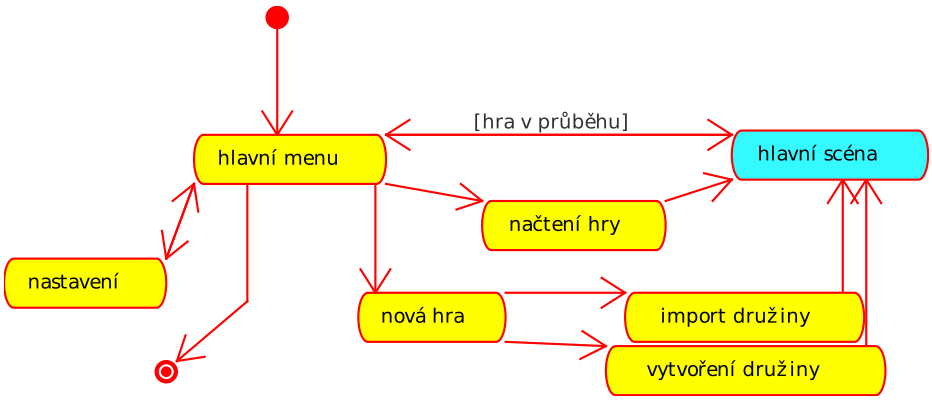
5.2 Návrh logiky

Řídící logika na nejvyšší úrovni nepřináší oproti běžným hrám nic nového. Jak již bylo zmíněno, uživateli bude po spuštění hry zobrazeno hlavní menu. Z něj bude moci přecházet do dalších obrazovek a z nich dál až do spuštění hry. Kostra logiky tohoto řízení je popsána v obrázku 5.3.

Co se týče řídící logiky v průběhu vlastního hraní, je třeba se zeptat, jaké činnosti bude



Obrázek 5.2: Hrubý a zjednodušený diagram další části *DrdSim*. Bez zobrazených rozhraní.



Obrázek 5.3: Logika přechodů mezi hlavními obrazovkami menu.

hráč provádět. Zhruba se bude jednat o následují tři skupiny:

- Prohlížení a úprava stavu družiny (prohlížení osobních deníků, změna vybavení apod.).
- Průzkum a cestování v herním světě, nebojová interakce s prostředím, atd. (v DRD nazýváno „mimoboj“).
- Boj s nepřáteli v rozšířeném soubojovém systému[1].

Uvedené položky se dají dále dělit (např. na sesílání kouzel, útok zbraní, cestování na čtverečkované mapě, cestování na hexové mapě), ale tyto tři skupiny se budou vyznačovat odlišným chováním aplikace a zobrazovaným GUI.

Hra bude (stejně jako stolní předloha) tahová - tedy ke změně stavu herního světa dojde jen následkem akce hráče. To však neznamená, že čas nebude hrát roli. Každá akce hráče je provedena za určitou časovou jednotku. Velikost jednotky záleží na měřítku, ve kterém se hráč zrovna pohybuje, a na činnosti, kterou provádí. Např. při boji trvá jedno kolo 10 sekund zatímco při cestování může přechod z jednoho políčka na druhé trvat dny až týdny. Uplynutý čas bude zaznamenáván např. pro určování dne a noci (důležité pro rychlost cestování) nebo pro tvůrcem kampaně explicitně naprogramované účely (družina by například musela něco stihnout do určitého termínu).

5.3 Pravidla světa hry

Jak již bylo řečeno v předchozí kapitole, hra bude využívat pravidla stolní RPG hry *Dračí Doupě*. Hned ze začátku je však třeba zdůraznit, že tuto hru není možné převést do počítačové podoby naprosto věrně. Pravidla hry jsou příliš rozsáhlá a popisují vše možné od soubojů postav s nestvůrami až po řízení ekonomiky států. Do hry bude tedy zanesena jen podmnožina těchto pravidel.

Navíc DRD neřeší přesně každý možný případ, v mnoha situacích se spoléhá na úsudek a cit pána jeskyně. U některých částí pravidel je dokonce napsáno, že se jedná pouze o doporučení a že záleží na úsudku PJ, jak se daná situace bude řešit.

Naproti tomu však existuje dostatek pravidel DRD, která jsou popsána exaktně a která popisují řešení daných situací pomocí tabulek, vzorečků a pokynů pro házení kostkou. Příkladem skupiny takových pravidel je například rozšířený soubojový systém.

Bylo tedy rozhodnuto, že hra *DrdSim* bude na systémové úrovni řešit všechna exaktní pravidla, jako je například boj, cestování nebo vylepšování družiny. Vše ostatní musí PJ naimplementovat explicitně, maximálně s pomocí podpůrných informací od hry.

Příklad: *Zloděj družiny se chce pokusit za pomoci svých speciálních schopností o získání důvěry podezřelé osoby. Ve skutečné hře by hráč oznámil svůj úmysl, hodil dvakrát desetistěnnou kostkou a poté by se PJ na 5 minut zamyslel a pokusil by si představit, jaký výsledek by tato akce měla správně mít. Tento výsledek však nemůže mít žádný číselný význam (např. postih k útoku). Ve PC hře však může být maximálně dostupná informace o tom, jakou úroveň má zlodějova schopnost získání důvěry a tvůrce kampaně do dialogu s osobou musí ručně přidat barevně odlišenou možnost, která bude značit použití speciální schopnosti. Akce bude mít v daném případě určitou náročnost, která hráči může a nemusí být známá. Po použití schopnosti se bude dialog dále ubírat směrem určeným na základě porovnání zlodějovy schopnosti a náročnosti dané akce*[2] *a to pokud možno bez náhodnosti*

[1] Základní soubojový systém není v tomto projektu vůbec uvažován.

(jinak by si hráč v případě neúspěchu jen nahrál dříve uloženou hru, což může být frustrující).

Při obecnějším pohledu můžeme v pravidla DRD pro potřeby projektu rozdělit do tří skupin:

1. Principiální pravidla, která určují, co se vlastně ve hře bude dělat a jak. Například, že hráč bude ovládat skupinu dobrodruhů, kterým zvolí rasu a zvláštní schopnosti, že s nimi bude cestovat po světě v šestiúhelníkové mapě a že s nimi bude bojovat v rozšířeném soubojovém systému proti nepřátelům.
   - Tato skupina pravidel se přímo podepíše na tom, jak hra bude vypadat a jaká bude její architektura a řídící logika.

2. Exaktní pravidla (viz dříve). Například je fakt, že útok jedné bytosti na druhou je proveden tak, že útočník hodí kostkou, k hodu přičte své útočné číslo zatímco obránce si také hodí a přičte své obranné číslo - rozdíl hodů určuje výsledek akce.
   - Pravidla z této skupiny je možno naprogramovat jako funkci s určitými vstupy a výstupy.

3. Neurčitá pravidla (viz vysvětlení a příklad výše).

Nyní je třeba položit otázku flexibility a upravitelnosti systému pravidel. Hra *DrdSim* je vytvářena podle pravidel verze 1.5, která jsou však přes deset let stará (dnešní verze je 1.6). Nebylo by proto vhodné zajistit, aby hra šla případně uzpůsobit pro nové verze pravidel?

Vhodný způsob, jak toho dosáhnout, je přenést části pravidel do vestavěného skriptovacího jazyka hry. To však nelze udělat všude. Například při uvažování první skupiny z výše uvedeného výčtu je jasné, že tato pravidla budou napevno vepsána nejen do kódu aplikace, ale i do jejího návrhu a architektury. Nezbývá než doufat, že tato pravidla se v dalších verzích příliš lišit nebudou.

Třetí bod z výčtu bude pochopitelně kompletně naskriptován externě. Problém je v tom, že implementace těchto pravidel nebude provedena globálně (v adresáři skriptů v hlavním adresáři hry), ale na úrovni jednotlivých kampaní a každým tvůrcem zvlášť. Zde hrozí nebezpečí nekonzistence, ale nezbývá než doufat, že tvůrce kampaně je dostatečně obeznámen s pravidly DRD. Bylo by však vhodné alespoň poskytnout podpůrné funkce pro tyto věci a zajistit tak alespoň nějakou konzistenci.

Hranice mezi napevno naprogramovaným a externě naskriptovaným se rozplývá ve druhém bodě. Bylo by poněkud podivné externě skriptovat pravidla pro útok a obranu (zvlášť když by tyto externí skripty byly volány pevným kódem hry a nikdy externě). Ale při zvážení příkladu se zlodějem a získáním důvěry je vidět, že funkce pro získání zlodějovy schopnosti pro získání důvěry by mohla být externí. Výše této schopnosti závisí na úrovni postavy a stupně jejího charisma. Mohla by tedy být vytvořena funkce, která by si pouze přečetla tyto dvě informace (pomocí napevno zakódovaných prostředků) a požadovanou hodnotu už spočítala sama dle svých vzorečků a tabulek.

5.4 Vytváření herního obsahu

Programování v jazyce DSL jsou dedikovány pozdější části této kapitoly. V této části je pouze popsáno, co všechno bude třeba udělat pro vytvoření kampaně - není zde uvedeno

[2]Přesněji řečeno, schopnost zloděje je vyjádřena procentuální pravděpodobností úspěchu a náročnost akce může způsobit postih nebo bonus k dané pravděpodobnosti.

jak. Postup byl již naznačen dříve, zde bude shrnuta celková struktura. Příloha C obsahuje návod pro začátečníky.

Jak již bylo řečeno, v hlavním adresáři hry se vyskytují tři (pro vývojáře) důležité adresáře: `scripts`, `classes` a `audio` (a ještě soubor s lokalizovanými texty, ale to není důležité). Obsahy těchto adresářů jsou při startu hry načteny a uloženy na speciálním místě (odděleně od tříd a funkcí PJ - tvůrce dobrodružství). Skripty a třídy lokální pro jednotlivá dobrodružství pak tyto zdroje mohou využívat. Je však třeba pamatovat, že systém je navržen tak, že v daném okamžiku jsou tyto třídy a funkce pouze data uložená v paměti. Fungovat mohou až ve svém run-time prostředí, které není nastartováno po startu hry, ale až po startu dobrodružství. Toto je jedna z věcí, kde se *DrdSim* odlišuje od OFP, kde byl aplikován přístup *„všechno je skriptovatelné“*. Z toho vyplývá princip: vše se odehrává jen uvnitř dobrodružství a jen pokud je dobrodružství spuštěno. To na jedné straně způsobuje jisté ochuzení a oslabení systému (zmiňované popisovače, neboli hlavičky s metadaty, by nebyly nutné, pokud by byly nahraditelné nějakým skriptem), ale na druhé straně jsem toho názoru, že to takto bude pro PJ jednodušší.

A nyní již k samotnému vytváření dobrodružství pro *DrdSim*. Co je to dobrodružství? V tomto kontextu se jedná o adresář, který je umístěný v hlavním adresáři hry a jehož jméno začíná znaky `DRD_`. Při vstupu do menu pro rozehrání nového dobrodružství jsou tyto adresáře automaticky detekovány.

V každém takovém adresáři musí být přítomen soubor `description.cfg`. Jedná se o velice prostoduchý konfigurační soubor obsahující informace o dobrodružství. Jedná se například o název a krátký popis dobrodružství. Také obsahuje informace a povaze dobrodružství a způsobu jeho nastartování. V některých RPG hrách je například zvykem začínat hru s jednou postavou (kterou si třeba ani nemůže hráč sám vytvořit) a další se mohou přidat později. Také je v různých RPG hrách možné ovládat skupiny postav (družiny) a různých maximálních velikostech. To všechno je možno nastavit v tomto souboru - kolik postav si může hráč vytvořit hned na začátku, kolik postav může může mít družina maximálně, zda může být použita importovaná družina atd.

Konfigurační soubor dále definuje, na které mapě má hra začít a na jaké pozici v ní. To se dá sice určit i skriptem, ale konfigurační soubor umožňuje bezprostřednější přístup (nemluvě o tom, že to zjednodušuje implementaci aplikace).

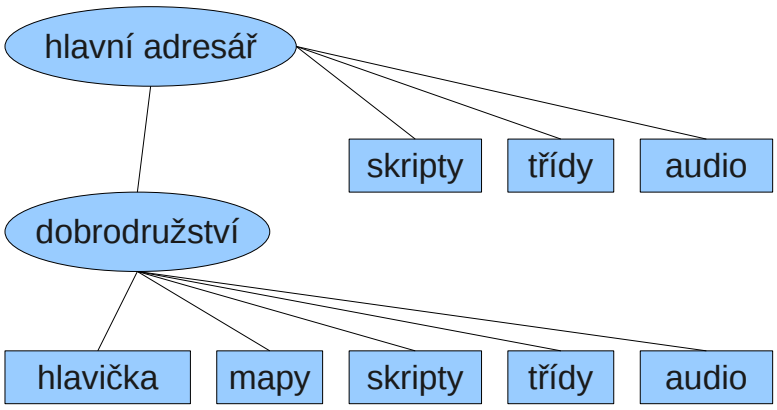
Založení adresáře a vytvoření hlavičky stačí na to, aby dobrodružství bylo rozpoznáno jako dobrodružství a zobrazeno v menu pro založení nové hry. Aby dobrodružství mohlo být spuštěno, je třeba v adresáři dobrodružství založit podadresář „`maps`“ a v něm vytvořit alespoň jednu mapu. Vytváření map, jak již bylo řečeno, se provádí speciální grafickou aplikací.

Dále je vhodné vytvořit podadresář „`scripts`“ a v něm vytvořit skript obsahující funkci `Init()`. Tato funkce, pokud existuje, bude provedena při startu dobrodružství. Může obsahovat např. inicializaci globálních proměnných důležitých pro logiku příběhu dobrodružství.

Podadresář `classes` v hlavním adresáři hry obsahuje dostatečné množství tříd objektů umístitelných do map a vybavení postav, nebo definujících kouzla, která mohou seslat. V adresáři dobrodružství si může PJ vytvořit vlastní adresář „`classes`“ a v něm si nadefinovat třídy, které potřebuje pro své dobrodružství.

Interakce hráče s objekty může vyvolat události, které jsou poté ošetřovány obslužnými funkcemi.

Do adresáře `audio` může PJ umístit nadabované dialogy, které chce v dobrodružství mít a případně vlastní hudební stopy. Tyto audio soubory může ve skriptovacím jazyce spustit pomocí vestavěných funkcí prostředí.



Obrázek 5.4: Adresářová struktura v *DrdSim*.

Na obrázku 5.4 je znázorněno to hlavní z adresářové struktury hry a dobrodružství.

5.5 Jazyky v DrdSim

V projektu se vyskytují následující jazyky a formáty souborů:

- Hlavní jazyk (DSL).
  - Deklarativní část pro popis (herních) tříd.
  - Procedurální část pro popis funkcí.
- Formát popisných souborů kampaní (cfg).
- Formát lokalizačních souborů (txt).
- XML pro ukládání map, uložených her a exportovaných družin.

Z uvedeného seznamu je nejdůležitější jazyk DSL a bude mu věnováno nejvíce místa. Ostatní položky (s výjimkou XML) mají jen pomocný charakter.

Jak je výše uvedeno, jazyk DSL se skládá ze dvou částí: deklarativní OO a procedurální. Tyto části původně tvořily samostatné jazyky, ale později bylo na doporučení vedoucího umožněno, aby oba jazyky mohly být použity v jednom a tom samém souboru.

Nyní vyvstává otázka: Proč je vůbec uvnitř jednoho systému použito více jazyků? Není to zbytečná komplikace?

V předchozích kapitolách bylo podrobně probráno, co všechno bude hra vyžadovat, a byly uvedeny různé příklady již vytvořených her. Například zmiňovaný herní engine *Unreal* používal přístup jednoho jazyka. Výsledek, jak bylo řečeno, byl ale poněkud složitý. Naproti tomu v další uvedené hře (OFP), byl aplikován přístup „jeden účel - jeden jazyk“.

Stejný přístup byl zvolen i zde. Ano, systém mohl být navržen tak, aby všechno bylo definováno jediným aparátem - ale ten by byl příliš složitý a pro předmaturitní uživatele nepochopitelný. Z toho důvodu byl vytvořen seznam věcí, které je zapotřebí nadefinovat a pro každou byl vytvořen jeden maximálně jednoduchý nástroj. Vzhledem k tomu, že tyto

nástroje jsou jen dva (pokud počítáme ty podstatné) a že do jisté míry splývají, je toto docela rozumný přístup.

5.6 DSL - deklarativní část

Předně je třeba zdůraznit, že ačkoliv tato část jazyka slouží k definici tříd objektů umístitelných do hry, nejedná se o plně OO jazyk. Jedná se pouze o definici atributů tříd (tedy jakási obdoba struktur, ale s dědičností). Je sice pravda, že tyto objekty mohou mít obslužné funkce pro události, ale tyto funkce nemohou být volány uživatelem jako v OO přístupu. Například při použití knihovny GLUT programátor také na začátku programu definuje obsluhy různých událostí, ale to ještě neznamená, že je takový GLUTovský program je objektem a jeho události metodami.

Třída v jazyce DSL je tedy záznam atributů, konkrétně řetězců a celých čísel. Krom toho je však do atributu možno přiřadit i procedurální kód z druhé části jazyka. Je možno se zeptat, zda se opravdu jedná o deklarativní jazyk, když je toto možné. Ano, jedná. Přiřazení bloku kódu do atributu je totiž pouze syntaktickým luxusem, který byl do překladače přidán po konzultaci této otázky s vedoucím. Pokud by bylo možné do atributu přiřadit pouze odkaz na obslužnou funkci, nebo dokonce jen její název v řetězci, bylo by dosaženo stejného výsledku. Pohodlnost takového přístupu by však byla nižší. O alternativě použité v OFP (přiřazení procedurálního kódu v řetězcové reprezentaci) ani nemluvě.

Zde je pro shrnutí syntaxe definice třídy v jazyce DSL:

```
<id třídy> [: <id mateřské třídy>]
"{"
   {<id atributu> = <hodnota atributu> ;}
"}"
```

Použité neterminály nejsou dále rozvinuty, protože nepotřebují další komentář ani definici. Pouze je třeba podotknout, že hodnotou atributu může být celé číslo, řetězec, nebo blok kódu. Žádná desetinná čísla nebo vektory nejsou zapotřebí. Proč? Protože DRD je postavené tak, že je nepotřebuje a *DrdSim* tuto vlastnost zdědil.

Uvedení mateřské třídy není povinné. Je ale důrazně doporučeno všechny vytvářené třídy vydědit z nějaké již existující, přinejmenším alespoň z jedné z následujících tříd: `Item`, `MapObject`, nebo `Spell`. Uvedené tři třídy jsou jediné globální třídy, které nemají mateřskou třídu a všechny ostatní jsou odvozeny z jedné z nich.

Dále je třeba se zmínit o jedné ze specialit DSL. Jedná se o lokalizované řetězce. Tyto řetězce jsou definovány ve speciálním textovém souboru v hlavním adresáři hry nebo v adresáři kampaně[3]. Na lokalizované řetězce se dá v DSL odkazovat napsáním dolaru a názvu řetězce. Tento odkaz je zcela ekvivalentní řetězcovému literálu, na který odkaz odkazuje. Při lokalizaci pak stačí vyměnit jen lokalizační soubor. Tato lokalizace probíhá za běhu aplikace, při kompilaci DSL souborů. Odkazování na lokalizované texty je možno použít v obou částech DSL.

Více o třídách objektů a jejich vazbě na prostředí hry je v části 5.8.

[3]Formát těchto souborů je docela prostoduchý - jen název řetězce, rovnítko a řetězcový literál.

5.7 DSL - procedurální část

Tato část se velice podobá definici funkcí v jazyce C. Definice funkce je také hlavním a jediným stavebním prvkem této části DSL. Definice globálních proměnných, deklarace externích proměnných a funkcí - nic z toho není třeba díky pozdní vazbě. Struktury se také nepoužívají, místo toho jsou k dispozici PHP-like pole.

Oproti C zmizely definice typů, vše se ověřuje při automatické inferenci až za běhu. Mnohé složitější konstrukce z C byly v DSL odstraněny. Tím se sice ztratily některé možnosti efektivního psaní kódu, ale nová očištěná syntaxe nutí psát programátora pěkně a přehledně.

Zde je popis syntaxe definice funkce:

```
<id funkce> "(" [ <id atributu> { , <id atributu> } ] ")"
<blok kódu>

<blok kódu> ::= "{"
                   { <příkaz> }
                "}"

<příkaz> ::= <přiřazení> | <volání funkce> | <return příkaz> |
             <if konstrukce> | <while cyklus>

<přiřazení> ::= <l-hodnota> = <výraz> ;
<l-hodnota> ::= [global] <id proměnné> { "[" <výraz> "]" }

<if konstrukce> ::= if "(" <výraz> ")" <blok kódu>
                { elif "(" <výraz> ")" <blok kódu> }
                [ else <blok kódu> ]

<while cyklus> ::= while "(" <výraz> ")" <blok kódu>
```

Opět platí, že věci dále nerozváděné jsou řešeny tak, jak by se předpokládalo. K výrazům je třeba dodat, že podporovány jsou všechny základní binární matematické a logické operace, z unárních operací pouze znaménková a boolean negace. Protože se jedná o vysokoúrovňový jazyk, nejsou k dispozici žádné operátory pro bitové operace nebo pro práci s ukazateli.

Na zápisu syntaxe je možno si povšimnout, že jediný podporovaný druh cyklu je **while**. Důvodem pro rozhodnutí nepřidat do jazyka **for** a **until** cykly jsou mé zkušenosti z gymnázia. Když jsme (já a mí spolužáci) byli poprvé seznámeni s programováním, neměli jsme většinou problém pochopit **while** cyklus. Trochu složitější už byly další dva druhy. Protože platí, že jakýkoliv druh iterace je vyjádřitelný while cyklem, rozhodl jsem se umožnit v jazyce jen tento druh cyklu. Pokud někdy zbude čas, mohl by být přidán i until cyklus a uveden v části manuálu pro pokročilé. Cyklus for není syntakticky estetický a proto přidán nebude nikdy.

Jak je také vidět, v if konstrukci nebo while cyklu není možné použít jednoduchý příkaz, pouze blok kódu. To sice může někdy způsobovat plýtvání místem, ale kód se tak stane mnohem přehlednějším. Toto rozhodnutí je založeno na mých zkušenostech z programování v C.

V jazyce neexistuje **switch** konstrukce, nebyla uznána za syntakticky estetickou.

Novinkou je klíčové slovo **global**. Protože proměnné (globální i lokální) jsou definovány přiřazením, je třeba rozlišit mezi vytvořením lokální proměnné a globální. Globální

proměnné mají v jazyce zvláštní pozici, protože nepředstavují jen místo v paměti, ale i informaci, která je uložena na disk jako součást rozehrané hry[4]. Klíčové slovo **global** se používá pouze při vytvoření globální proměnné (nebo při jejím přepisu). Při použití se již píše jen název této proměnné. Pokud není nalezena lokální proměnná takového jména, automaticky se bere jako globální (určováno v době kompilace).

V souvislosti s poli je třeba zmínit způsob práce s nimi. Pole se vytvářejí přiřazením speciálního výrazu do proměnné - prázdných hranatých závorek. V případě globálního pole je ještě třeba přidat patřičné klíčové slovo.
Příklad:

```
pole = [];
global globalni_pole = [];
```

Následně je možno pracovat s polem stejným způsobem jako v PHP.
Příklad:

```
lide[5] = [];
lide[5]["jmeno"] = "Tonda";
lide[5]["prijmeni"] = "Ucho";
```

Z uvedeného příkladu je vidět, že stejně jako v PHP jsou i zde struktury nahrazeny poli. Nepostrádá však takový přístup konzistenci (kontrolovanou za kompilace) struktur v C? Inu, pokud bude chtít programátor konzistenci zajistit, může tak učinit na úrovni algoritmu a například vytvořit specializované funkce pro práci se svými „strukturami“ (například „PřidejČlověka“ v tomto případě). Možná, že některé programátory-začátečníky takový počin přinutí přemýšlet, jestli by nebylo dobré mít danou strukturu a k ní příslušné manipulační funkce pohromadě, a vyzkoušet C++.

Jak již bylo řečeno v předchozí části, i v procedurální části DSL je možno použít odkaz na lokalizovaný řetězec. Takový odkaz bude při kompilaci nahrazen řetězcovým literálem, na který odkazuje.

Celkově se dá říci, že procedurální část DSL je v souladu s myšlenkou tohoto projektu. Jazyk je snadno použitelný, obsahuje jen jednoduché konstrukce a umožňuje nekomplikovanou práci s poli a dynamickými strukturami. Je sice pravda, že například jazyk C je silnější (díky svým pochybným konstrukcím), ale je třeba pamatovat, že na své zamýšlené použití je DSL silný více než dost.

5.8 Interpret a jeho okolí

Tato část popisuje interpretové prostředí *DrdSim*, jeho části a způsob, jakým funguje. Bude zde popsán nejen samotný exekuční stroj, ale i vše na něj navázané.

Pro začátek nechť čtenář ví, že DSL interpret je virtuální stroj zasazený do enginu hry (jak již bylo popsáno dříve). Mezi hrou a interpretem existuje množství vazeb, datových i kontrolních.

Nejprve by ale bylo dobré uvést přehled toho, co se s interpretem a jeho okolím děje v průběhu života aplikace.

1. Uživatel spustí hru.

[4] To se však v současné době týká pouze skalárů - pole jsou při ukončení hry zahazovány i když jsou globální.

(a) Je načten globální lokalizační soubor.

(b) Jsou načteny globální třídy.

(c) Jsou načteny globální funkce.

2. Uživatel vybere a spustí kampaň.

(a) Načte se lokální lokalizační soubor.

(b) Jsou načteny lokální třídy a funkce.

(c) Načte se a zobrazí mapa, na které začíná dobrodružství.

(d) Spustí se funkce `Init()`.

3. Interakce družiny s okolím spustí událost.

(a) Spustí se obslužná funkce původce události.

Co se děje při spuštění funkce v bodech 2d a 3a? Interpret byl až do toho okamžiku nečinný. V okamžiku zavolání ale ožije a převezme kontrolu nad logikou hry.

Interpret začne provádět vstupní funkci a postupně volat další funkce a plnit zásobník lokálními proměnnými. Po celou dobu může zasahovat do svého okolí změnou globálních proměnných, a voláním vestavěných funkcí. Posléze se vrátí do těla vstupní funkce a po jejím skončení předá řízení zpět hře.

Je třeba pamatovat, že na rozdíl od SQS skriptů v *Operation Flashpoint*, DSL skripty neprobíhají paralelně se hrou - vše se musí odehrát v „nulovém čase“ v okamžiku vyvolání události. Pokud není možné něco zpracovat okamžitě, je nutno rozpracovaná data uložit dejme tomu do globálního pole a pokračovat později. Příkladem takového způsobu práce by mohl být např. rozhovor družiny s NPC (cizí postavou).

Interpret jako takový obsahuje pouze exekuční stroj se zásobníkem a jedním registrem. Jeho úkolem je pouze počítat a nenese uvnitř sebe žádná relevantní data, pouze pomocné lokální proměnné. Dokonce ani samotné funkce, které jsou na něm prováděny, nejsou uloženy přímo v něm. V rámci pojmů tohoto projektu je tedy třeba rozlišovat mezi interpretem jazyka DSL a run-time prostředím *DrdSim* (což je nadmnožina předešlého).

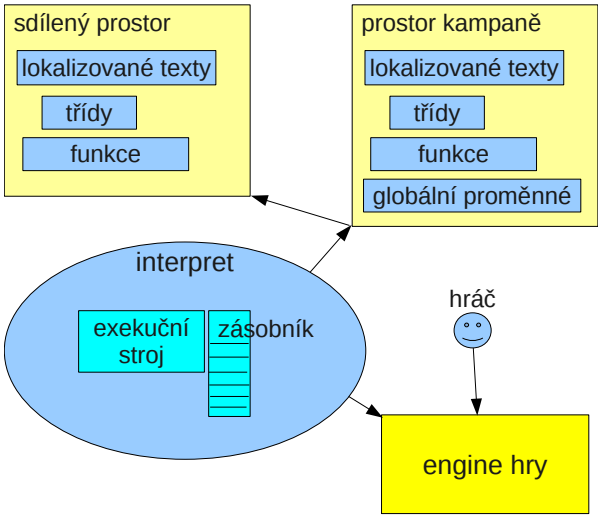
Konceptuální strukturu interpretu a jeho vazbu na okolí je možno vidět na neformální kresbě 5.5. Pokud by čtenář chtěl přesnou strukturu na úrovni tříd, je možné jej odkázat na programovou dokumentaci.

Třídy objektů popisované v části 5.6 jsou také uloženy ve speciálních prostorech. Skripty k nim budou moci (tato část ještě není naprogramována) přistupovat za pomoci vestavěných funkcí, případně za pomoci předdefinovaných globálních proměnných (takto to ale pravděpodobně řešeno nebude).

Nasazení konkrétních tříd je určeno kořenovou třídou, ze které jsou vyděděny (viz 5.6). Třídy vyděděné z `MapObject` lze umístit do herního světa. Objekty potomků `Item` se mohou vyskytovat v inventáři postav a mohou být za určitých okolností sebrány v herním světe (např. z truhly, z kapes pobitých nepřátel atd.). Potomci třídy `Spell` popisují kouzla a další speciální vlastnosti postav.

`MapObject` se od ostatních dvou liší tím, že může způsobit vyvolání funkce (přesněji řečeno, je hlavním původcem startování interpretu). Druhé dva typy objektů jsou skripty pouze využívány (pokud je engine hry neošetřuje zcela automaticky).

Nyní je třeba se zeptat: nebylo by vhodné, aby šly třídy vytvářet dynamicky uvnitř funkcí? Takový přístup by se dal použít např. k sestavení úhlavního „bosse“ na míru aktuální



Obrázek 5.5: Neformální struktura v interpretu, jeho vazby na okolí. Komponentou „engine hry“ jsou části jako je grafika, zvuk, vstup atd. V pravém slova smyslu by byl interpret jen další součást enginu.

konfiguraci družiny. Metaprogramování je ale koncept, který je pro laika těžko pochopitelný (respektive zcela nepochopitelný). Dynamická tvorba tříd by byla pro začátečníka matoucí a stejně by asi zůstala nevyužita. Navíc DRD si na nějaké dynamické upravování příliš nepotrpí. Systém hry umožňuje určit vstupní podmínky pro nastartování nové kampaně. Nebude tudíž problém, aby tvůrci kampaně uvážili a nastavili, jakou úroveň by postavy měli mít.

5.9 Shrnutí

Tato poněkud rozsáhlejší kapitola popsala základní rysy „vnitřního světa“ hry *DrdSim*. Byly popsány užité jazyky a jejich vazby na vlastní hru. Pro další informace by bylo dobré si pročíst manuál k vývojovému prostředí hry, který je více prakticky založený.

Kapitola 6

Engine hry

Tato kapitola stručně popisuje zbytek aplikace *DrdSim*. Je třeba podotknout, že ačkoliv se tato kapitola nazývá „Engine hry“, stejně tak i interpret DSL jazyka popisovaný v předchozí kapitole je součástí herního enginu. Když tedy v této kapitole bude řeč o enginu, je tím myšlena jeho podmnožina - základní herní logika, grafika, zvuk, uživatelský vstup a podobné věci.

Jak ale zmíněné věci souvisejí s tématem této práce (jazyky a překladače)? Popsaný programovací jazyk DSL a jeho interpret by byl sám o sobě poměrně bezzubý (jako C bez knihoven). Užitečnou práci může vykonávat až za pomoci vestavěných funkcí, které poskytují interface mezi interpretem a zbytkem enginu. Bude-li chtít například programátor spustit v nějakém okamžiku hudbu nebo přehrát nějaký zvuk, použije patřičnou vestavěnou funkci která přepošle požadavek zvukovému subsystému. Princip tohoto je vidět na obrázku 5.5 z předchozí kapitoly.

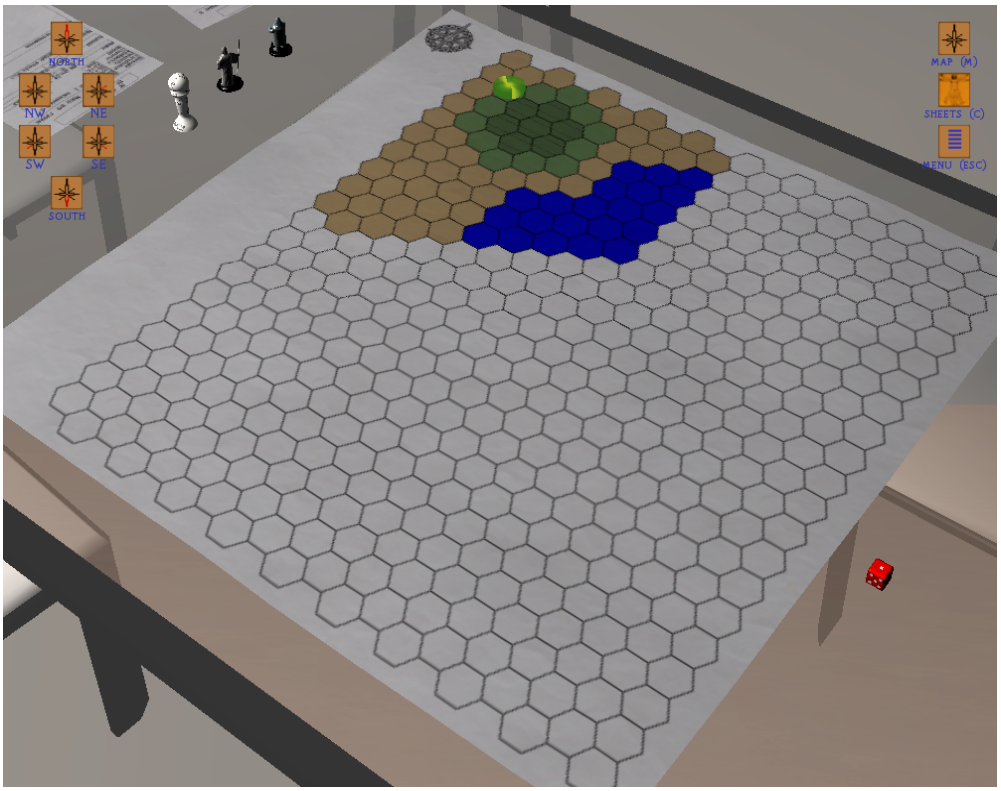
6.1 Grafika a interakce

Nejprve je třeba přesněji vymezit, jaký rozsah grafický engine v *DrdSim* má a co dělá. V počítačových hrách jsou (3D) enginy využívány k zobrazování herního obsahu. Ten je definovatelný pomocí externích zdrojů - engine je pak jakýmsi interpretem tohoto obsahu a jeho činnost je parametrizovatelná. Například u 3D enginů lze definovat pomocí speciálního formátu 3D svět, ve kterém se má hra odehrávat.

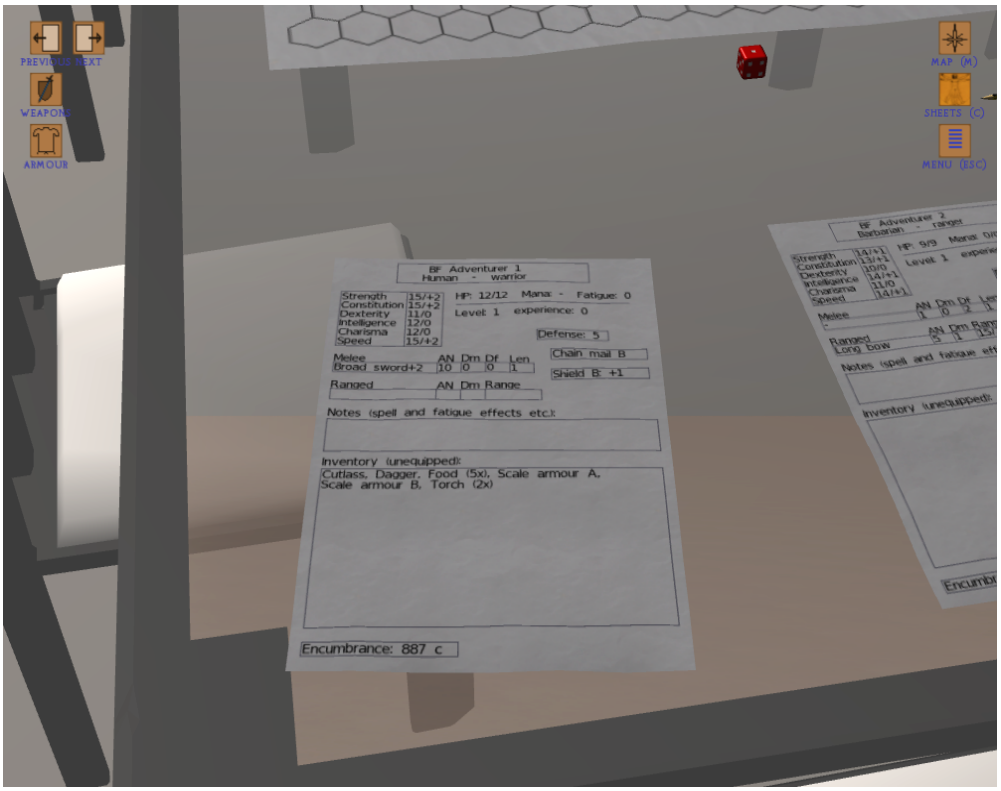
Jak je vidět na obrázcích 6.1 a 6.2, hra *DrdSim* má sice 3D grafiku, ale samotný herní svět je jen 2D a je zobrazován jako 2D. Proč? Protože tak to odpovídá způsobu hraní Dračího Doupěte. 3D scéna rozestavená kolem herní mapy (pokoj se stolem a židlemi) má jen dekorativní účel a i když je definována pomocí externího 3D modelu, její zobrazení je natvrdo naprogramováno v kódu aplikace.

Dále je možno se zeptat, zda by nebylo lepší umožnit plnou skriptovatelnost a rekonfigurovatelnost hry (jak bylo v počátcích projektu navrhováno) - tedy včetně okrasné 3D scény, menu atd. Ne, nebyl by to dobrý nápad. Cílem práce je udělat co nejjednodušší systém a v zájmu tohoto cíle je nechat programátora se soustředit jen na programování herního světa a ničeho jiného. Tvůrce herního obsahu musí jasně vědět, že oblast jeho tvorby je omezena jen na to, co se děje na papírové mapě položené na stole. Úniky z této oblasti by ho jen mátly a systému by na jednoduchosti příliš nepřidaly.

A nyní již k samotnému grafickému enginu hry. Jak již bylo řečeno v předešlých kapi-



Obrázek 6.1: Snímek hlavní scény v *DrdSim* (vypnutý antialiasing).



Obrázek 6.2: Režim prohlížení osobních deníků.

tolách, *DrdSim* využívá grafickou knihovnu *Open Inventor* (implementace *Coin2*). Veškerá grafika ve hře, včetně jednotlivých menu a prvků GUI, je pak tvořena grafy. Logika hry si po sestavení stromů ponechává odkazy na určité uzly stromu, nebo na data obsažená v těchto uzlech, a používá je k uskutečnění grafického výstupu. Například v hlavní scéně zobrazené na obrázku 6.1 je důležitý datový objekt obsahující rastr natažený na papíře s mapou. Jak družina prochází světem, mapa je postupně vykreslována zápisem do tohoto rastru.

„Papír“ s mapou a papíry s osobními deníky rozmístěné kolem ní jsou velice důležité pro hru a zobrazují většinu herní informace. Pro práci s nimi byl v rámci tohoto projektu vyvinut specializovaný framework, který uskutečňuje kreslení na tyto papíry.

Pro vytvoření 3D scén byly použity externí 3D modely. OI má vlastní souborový formát pro ukládání modelů, ale pro jejich automatické načítání vyžaduje doplňkovou knihovnu *simage*. Pro zjednodušení nasazení hry však tato knihovna použita nebyla a soubory s daty jsou načítány ručně pomocí vlastních prostředků. Nejedná se jen o 3D modely, ale i o textury, shadery apod.

Pro zlepšení vzhledu hry byly použity různé shader programy. Například kurzor, který ukazuje pozici družiny na mapě, byl animován pomocí shaderu.

Jak bylo řečeno v kapitole 3, OI neumožňuje realizovat určité oblíbené shader efekty, jako je například odraz. Pro částečnou kompenzaci tohoto faktu byl na sklo stolu ve scéně použit shader, který odraz napodobuje pomocí matematického vzorce, který na sklo dlaždicově mapuje texturu stropu místnosti v závislosti na úhlu kamery. Je však snadno poznat, že se jedná jen o podvrh.

Veškeré GUI ve hře je realizováno pomocí 3D objektů (které jsou zobrazovány ortograficky a jeví se jako 2D). To je sice neefektivní, ale při vývoji nebyl čas se takovými drobnostmi zabývat.

Jako rozhraní mezi OI a operačním systémem byl použit modul *SoSDLRenderArea*, namísto jednoho z oficiálních modulů. Tento modul byl vyvinut na VUT FIT (viz poděkování) a implementuje grafický výstup OI za pomoci knihovny SDL.

Jak bylo řečeno v kapitole 3, využití prostředků OI v této oblasti projektu bylo poněkud problematické a kvůli nedokumentovanosti tématu bylo nutno jej řešit metodou „pokus, omyl“. Cílem tohoto modulu, jak název napovídá, je umožnit uživateli interagovat s 3D scénou hry za pomoci myši.

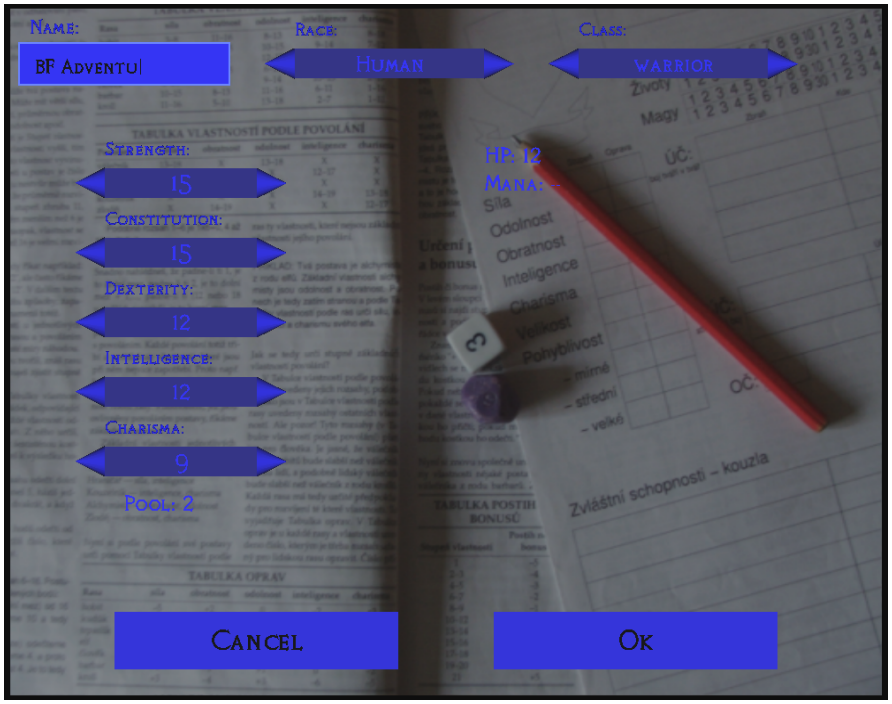
Jako základ byly použity OI grafové uzly *SoSelection* a *SoLocateHighlight*. Uzel selektoru je umístěn na kořen grafu, který obsahuje interaktivní objekty. Tyto objekty jsou obaleny speciální třídou, která je vyděděna z uzlu zvýrazňovače.

Následkem toho interaktivní objekty reagují na přejezd myší. Při kliknutí je pak zavolána univerzální obslužná funkce, která projde graf scény od nakliknutého objektu buď až ke kořenu grafu, nebo k nejbližšímu zvýrazňovači. Poté je zavolána obslužná funkce, která je v uzlu zvýrazňovače uložena (tato funkčnost byla přidána ve vytvořeném podtypu zvýrazňovače).

Byly vytvořeny prostředky, které toto dokáží vytvářet a nastavovat automatizovaně. I přes to by však práce na popsané úrovni byla příliš složitá, a proto byla vytvořena knihovna již předpřipravených widgetů, které zapouzdřují jeden a více aktivních objektů. Celá tato oblast projektu tvoří poměrně objemný framework (viz obrázek 6.3).

Popsaný systém byl použit jak na vytváření interaktivních 3D scén (režim rozšířeného soubojového systému), tak na vytváření 2D GUI. To je sice neefektivní, ale nebyl čas implementovat dva oddělené frameworky.

Při vytváření tohoto systému se vyskytl problém více kamer, popsaný v kapitole 3.



Obrázek 6.3: Menu vytváření nové postavy před startem hry.

Pro připomenutí se jedná o zmatení systému pro podporu funkce *RayPick*, pokud graf scény obsahuje více kamer (což je nutnost při kombinaci 3D scény a 2D GUI, viz obrázek 6.1). Problém byl vyřešen pomocí ruční obsluhy distribuce událostí v takových grafech (viz soubor `scene.cc`, funkce *EventHandler* na konci souboru).

Grafický engine hry dále obsahuje podporu pro načítání 3D modelů (formát *.iv) a textur (formát *.bmp, díky tomu si hra vystačí se základním modulem SDL).

6.2 Zvuk a další

Ačkoliv do knihovny *Coin2* je v poslední době přidávána podpora pro zvuk, a zvláště pak zvuk prostorový, byl zvukový systém *DrdSim* naimplementován zcela od základů za pomoci pro toto určených funkcí SDL.

Audio modul enginu hry umožňuje snadné načítání a přehrávání zvuků a hudby. Zvuky a hudba jsou od sebe odděleny, obojí je spouštěno předáním názvu souboru zvukovému systému. Hudba je pak přehrávána neustále dokola, dokud není zastavena další funkcí. Zvuk je přehrán jednou a potom se zahodí.

Systém nepoužívá SDL mixer, ale má vlastní implementaci. Hudba a zvuk jsou sloučeny do společného výstupu, pokud jsou aktivní současně. Pokud je spuštěn nový zvuk v okamžiku, když už je nějaký přehráván, je nový zvuk přimixován. Protože je však vytvořená implementace velmi primitivní, není doporučováno přehrávat větší množství zvuků současně, což se ani nepředpokládá.

Jak již bylo řečeno, přenositelnost je zajištěna použitím OpenGL, SDL a rozhraní pro platformově nezávislou práci se soubory (změna aktivního adresáře, vyhledávání souborů v adresáři, atd.). Hra je vyvíjena v Linuxu a pro něj je primárně určena (a pro překladač gcc). Je však také možné ji přeložit a spustit ve Windows, což se ale nedoporučuje tvůrcům

kampaní.

Engine hry obsahuje podporu abstraktní a platformově nezávislé práce se soubory a adresáři. Při startu je aktivní adresář automaticky přepnut na hlavní složku hry. Potom je možno pracovat s daty v adresáři jako s herními zdroji, ne jako se soubory a složkami.

Hra pracuje jen v rámci svého adresáře, kam ukládá i data hráče. Nesahá na Windows registry ani na žádné typické složky ve Windows nebo Linuxu.

Pro kompilaci na Windows je zapotřebí kombinace msys+MinGW. Kompatibilita s dalšími překladači (od Microsoftu nebo Borlandu) nebyla testována ani plánována. Zkompilovaná aplikace je již na msys nezávislá a lze spustit běžným způsobem. Je nutno k aplikaci přiložit knihovny *SDL.dll* (313 kB) a *libCoin-40.dll* (54,6 MB). Distribuce hry na Windows bude nejlepší předáním již zkompilovaných binárních souborů, kompilace hry by byla pro koncového uživatele příliš náročná (na rozdíl od Linuxu, kde vše vyřeší jediný příkaz make).

Dále by bylo vhodné krátce zmínit pomocnou aplikaci *mapBuild*, která slouží pro vytváření map pro *DrdSim*. Je napsána v jazyce Java a funguje bez problémů v Linuxu i Windows. Je doporučeno používat oficiální JRE od firmy Sun, namísto otevřené implementace *IcedTea*. Později uvedené totiž způsobuje aplikaci drobné závady, které sice neničí funkčnost aplikace, ale zato zpomalují práci v ní (toto však mizí s každou novou verzí *IcedTea*). Aplikaci *mapBuild* jsem nevytvořil sám, ale s pomocí kolegů (viz poděkování).

6.3 Srovnání, testování

Projekt byl navržen a vyvinut se zřetelem na ostatní podobné SW produkty. Oproti jiným hrám se *DrdSim* vyznačuje tím, že je více zaměřen na integrované jazyky a jejich runtime okolí. Je důležitá čistota a jednoduchost obou uvedených částí (z programátorského hlediska). Skriptovací jazyky nejsou pomůckou vývojáře hry, od samého počátku jsou navrhovány pro uživatele. Pro detailnější porovnání vlastností *DrdSim* s podobnými projekty doporučuji pročíst kapitoly 2, 3 a 4.

Hra byla testována přiloženými globálními skripty (v hlavním adresáři hry) a pokusným dobrodružstvím obsaženým v adresáři `DRD_pokus`. Toto dobrodružství obsahuje 2 mapy (jedna hexová, druhá čtverečková). K tomuto dobrodružství jsou přiloženy krátké ukázky z jazyka DSL. V současné době však *DrdSim* nemá mnoho vestavěných funkcí a proto toho není mnoho k předvedení. Není však problém nabídku vestavěných funkcí v budoucnosti rozšířit - většinou by se jednalo jen o využití již existující implementace v enginu hry.

Kapitola 7

Závěr

Cílem této práce bylo vytvořit počítačovou hru umožňující vytvářet její herní obsah ve skriptovacích jazycích, které mají kvality popsané v této práci. Tento cíl jsem splnil.

Konkrétněji jsem měl za úkol navrhnout skriptovací jazyky pro tuto hru, což jsem učinil (kapitola 5). Návrh samotné hry a prostředí, ve kterém budou zmíněné skriptovací jazyky nasazeny, je popsán v kapitole 4 (popis jazyků v kapitole 5). Implementace projektu je popsána v kapitolách 5 a 6. Testování a porovnání s ostatními hrami je uvedeno v předchozí části.

V rámci této práce se mi podařilo dosáhnout vytvoření jednoduchých, jasných a pěkných programovacích jazyků, které jsou zasazeny do logického a dobře navrženého prostředí. V současné době je hra ve fázi, kdy už funguje její engine (grafika, zvuk, interaktivita, atd.), překladače a run-time prostředí pro vestavěné jazyky hry. Stála však zbývá veliké množství práce. Vzhledem k tomu, že se jedná o počítačovou hru, na které dělám sám, která je nekomerční a u které mi tudíž nikdy žádný distributor neřekne, abych ukončil práci, nedá se předpokládat, že by se projekt někdy dostal do bodu, kdy by byl naprosto hotový.

Jako osobní postřeh bych rád uvedl, že největší množství práce a problémů nespočívalo v oblasti jazyků a překladačů, kde by snad jediným problémem bylo zorganizování použitých jazyků tak, aby byly opravdu snadno použitelné (s tímto bodem mi naštěstí pomohl můj vedoucí). Největší obtíže byly v drobných věcech technického rázu, například problémy plynoucí z interního fungování knihovny *Coin2*, která ne vždy spolupracovala s tím, co se po ní v tomto projektu chtělo.

V budoucnu budou pokračovat práce na *DrdSim* a hra bude dovedena do hratelnějšího stavu (rozšířený soubojový systém, kouzla, atd.) a bude vytvořena větší knihovna vestavěných funkcí pro skriptovací jazyk hry.

Literatura

[1] Stránky nakladatelství ALTAR. http://www.altar.cz/altar.html, 2009.

[2] Tutorials and articles. http://www.blackwyrmlair.net/articles_m.php, 2002-6.

[3] (Baldur's Gate) Scripts. http://www.planetbaldursgate.com/bg/help/customize/scripts/.

[4] Stránky Dračího doupěte. http://www.altar.cz/drd/index.html, 2009.

[5] *DRAČÍ DOUPĚ - Pravidla pro začátečníky, verze 1.5*, kapitola Řízení hry. Dračí Doupě, ALTAR, Čtvrté vydání, 1992, str. 83, iSBN 80-901078-7-7.

[6] Legacy:UnrealEd Versions. http://wiki.beyondunreal.com/Legacy:UnrealEd_Versions, 2009.

[7] Evil Islands: Curse of the Lost Soul. http://en.wikipedia.org/wiki/Evil_Islands, 2010.

[8] (Far Cry) Map Editor Tutorial Version 1.1. http://planetfarcry.gamespy.com/View.php?view=Tutorials.Detail&id=10, 2006.

[9] MaxED Tutorial. http://www.rockstargames.com/maxpayne/tutorials/MaxED/index.html, 2002.

[10] Operation Flashpoint: Community Home Pages. http://community.bistudio.com/wiki/Operation_Flashpoint:_Community_Home_Pages, 2009.

[11] OFP Wiki. http://community.bistudio.com/wiki/Main_Page, 2010.

[12] Coin Documentation. http://doc.coin3d.org/Coin-2/, 2009.

[13] Lua-scripted video games. http://en.wikipedia.org/wiki/Category:Lua-scripted_video_games.

[14] Lua (programming language). http://en.wikipedia.org/wiki/Lua_(programming_language).

[15] Operation Flashpoint: Cold War Crisis. http://en.wikipedia.org/wiki/Operation_Flashpoint:_Cold_War_Crisis.

[16] QuakeC. http://en.wikipedia.org/wiki/QuakeC.

[17] Sweeney, T.: UnrealScript (reference).
http://unreal.epicgames.com/UnrealScript.htm, 1998.

[18] Tofer, C.: Lua and Baldur's gate.
http://lua-users.org/lists/lua-l/1999-08/msg00056.html, 1999.

[19] Wernecke, J.: *The Inventor Mentor*. Silicon Graphics, 1994, iSBN 0-201-62495-8.

Dodatek A

Obsah CD

- Soubor `"README!!.TXT"`:
  Komentář a návod k obsahu CD. Nutno přečíst!

- Adresář `"report"`:
  Obsahuje elektronickou verzi tohoto textu.

- Adresář `"drdsim"`:
  Obsahuje hru drdsim (zdrojové kódy i data). Přiložena je i programová dokumentace (podadresář `"html"`).
  Přiložený Makefile funguje buď Linuxu (je třeba mít nainstalovány patřičné balíčky; na serveru **Merlin** patřičné balíčky v patřičné verzi **nejsou**, viz dále), nebo ve Windows za použití kombinace mingw+msys. To by však bylo složité a proto je přiložen již zkompilovaný exe soubor a patřičné dll knihovny. V případě užití tohoto SW ve Windows se důrazně doporučuje využít již zkompilovanou verzi. Soubory byly zkompilovány ve Windows XP, 32 bitů (měly by tedy fungovat snad všude, ale není garantováno).
  Hra vyžaduje počítač s grafickou kartou podporující shadery. Ačkoliv většina počítačů na FIT VUT toto splňuje, je občas možné narazit na počítač starší, nebo s "opotřebovanou" graf kartou (ne všechny shader jednotky funkční). Je možné hru zkompilovat tak, aby neobsahovala shadery (pak lze hra zkompilovat i na serveru Merlin).
  Bližší informace možno nalézt v readme.

Dodatek B

Ukázka práce v OFP

Následující příklad ukazuje vytvoření efektu *bullettime* v počítačové hře *Operation Flashpoint*. Příklad je zde uveden pro ilustraci (pozitivního) nepoměru mezi vynaloženým úsilím a efektností výsledku při práci v SQS.

B.1 Vytvoření jednotky

Nejprve je třeba vytvořit jednotku, která dovede *bullettime* používat. To se provede níže uvedenou třídou ze zdrojového souboru `config.cpp`. V uvedeném kódu nejsou žádné komentáře, tento konkrétní jazyk pravděpodobně ani žádné neumožňuje.

```
class timefighter: SoldierWSaboteurPipe
{
  displayName="Petr";
  model="mc vojakgo2.p3d";
  accuracy=0.7;
  precision=5
  armor=65
  armorStructural=4;
  armorHead=1;
  armorBody=1.5;
  armorHands=1;
  armorLegs=1;
  weapons[]={"Throw","Put"};
  magazines[]={};

  class UserActions
  {
    class bullettime
    {
      displayName = "bullettime";
      position = "default";
      radius = 0.1;
      sound = {};
      condition = true;
      statement = "[] exec ""\TF2_west\bullettime.sqs"";";
```

```
    };
  };

  class eventhandlers
  {
    init = "heart=false;";
  };
};
```

Atributy v první polovině třídy nejsou důležité. Požadovanou funkčnost přináší až členské třídy UserActions a eventhandlers. UserActions definuje akce, které může hráč během hry spustit (kolečkem myši z malého in-game menu). Další vložená třída bullettime už definuje jednu konkrétní akci. Atribut statement v této třídě obsahuje SQS kód (uložený jako textový řetězec), který se má při stisknutí kolečka myši zavolat.

Atribut init ve třídě eventhandlers definuje kód, který se má automaticky spustit při startu mise. Nastavení proměnné heart na false znamená, že mód *bullettime* není aktivní. Dále ještě musí tvůrce mise ručně nastavit proměnnou bt na nějakou číselnou hodnotu (případně ji mít uloženou jako kampaňovou proměnnou).

B.2 Obslužné funkce

Po vytvoření jednotky je již možno naskriptovat samotný bullettime. Zde je soubor `bullettime.sqs`, který je volán ze třídy bullettime:

```
; je bt zapnutý?
? !heart: []exec "\TF2_west\bt_on.sqs"
? heart: []exec "\TF2_west\bt_off.sqs"
```

Kód značí, že pokud je mód neaktivní, tak se zapne a pokud je aktivní, tak se vypne. Pokud řádek začíná středníkem, je ignorován (pokud je ale středník uprostřed řádku, značí oddělení příkazů).

Zde je soubor `bt_on.sqs`:

```
; zbývá ještě nějaký bt?
? bt <= 0: Hint Localize "STR_bt1"; GoTo "end"
; vstupní efekt
PlaySound "bt_on"
TitleCut ["","WHITE IN",0.5]
0.1 FadeMusic 0.1
0.1 FadeSound 0.0001
; zapnout tlukot
heart=true
[] exec "\TF2_west\heartbeat.sqs"
SetAccTime 0.25
#end
Exit
```

Skript zkontroluje, zda hráči ještě nějaký bullettime zbývá (a skočí na konec, pokud ne). Jinak přehraje zvuk spuštění režimu, udělá na obrazovce záblesk a ztiší hudbu a zvuk. Poté nastaví příznak heart, spustí skript `heartbeat.sqs` a zpomalí čas.

Zde je soubor `bt_off.sqs`:

```
; vypnout bt a ukončovací efekt
0.2 FadeMusic 0.5
0.2 FadeSound 1
heart=false
SetAccTime 1
PlaySound "bt_off"
~1
; vytisknout zbyvající bt
? !heart: Hint Format[Localize "STR_bt2", bt]
Exit
```

Skript vrátí hlasitost zvuku a hudby do původní hodnoty (0.5 je normální hlasitost pro hudbu, ne 1 jak by se dalo čekat). Poté nastaví příznak heart na false, vrátí rychlost času do normálu a přehraje zvuk vypnutí režimu.

A nakonec soubor `heartbeat.sqs`:

```
#beat
? bt <= 0: GoTo "end1"
~0.5
bt = bt-0.5
? heart: PlaySound "bt_heart"
? heart: GoTo "beat"
GoTo "end2"
#end1
[]exec "\TF2_west\bt_off.sqs"
#end2
Exit
```

Skript cykluje a přehrává zvuk tlukotu srdce a při každém cyklu snižuje a kontroluje zásobu času, po který je možno mít režim zapnutý.

Jak je vidět, předvedená implementace je velice jednoduchá. Výsledný efekt je však velice působivý (a nutno dodat, že také velice podobný efektu ze hry *Max Payne*).

Dodatek C

Průvodce k tvorbě v *DrdSim*

V této kapitole je stručný průvodce k tvorbě v systému *DrdSim* a (v současné době krátký) seznam vestavěných funkcí systému.

C.1 Vytvoření nového dobrodružství

Nejprve je nutno v hlavním adresáři hry založit novou složku dobrodružství. Její název musí začínat písmeny `"DRD_"`. Pokud první čtyři písmeny názvu budou jiná, systém složku vůbec nerozpozná, nenačte a v menu nenabídne.

Následně je nutno v tomto novém adresáři vytvořit hlavičkový soubor. Ten se musí jmenovat `"description.cfg"`. Jeho syntaxe je velice jednoduchá:

```
<název_atributu> = <řetězec> | <číslo>
```

V současné době je možno nastavit následující atributy:

TITLE	řetězec	název dobrodružství
DESCRIPTION	řetězec	stručný popis
IMPORTPARTY	číslo (1/0)	je možno importovat družinu?
NEWPARTY	číslo (1/0)	je možno vytvořit novou družinu?
STARTPARTYSIZE	číslo	maximální povolená velikost vytvořené družiny
ENTRYMAP	řetězec	název souboru startovní mapy
ENTRYX	číslo	x-ová pozice na startovní mapě
ENTRYY	číslo	y-ová pozice na startovní mapě

Jak je vidět, některé z uvedených hodnot jsou typu řetězec. Ten je možno zapsat přímo na místo, kde je použit. Pokud se však jedná o text, který bude čten hráčem, je lepší místo toho použít odkaz na lokalizační stringtable. To se provede zápisem dolaru a názvem klíče lokalizovaného řetězce.

Všechny lokalizační řetězce musí být uvedeny v souboru `"stringtable.txt"`. Ten má podobnou syntax jako hlavičkový soubor (tedy název, rovnítko, hodnota). Jeden takový soubor je vhodné vytvořit v adresáři dobrodružství, další je již obsažen v hlavním adresáři hry.

Následně je třeba založit tři podadresáře: `maps`, `classes`, `scripts` a `audio`. Stojí za povšimnutí, že složky stejných názvů jsou i v hlavním adresáři hry.

Dalším krokem je vytvoření map, které pak budou uloženy v adresáři `maps`. Editor map je přiložen v adresáři `builder` a spuští se příkazem `"java -jar mapBuild.jar"`. Nápověda k této aplikaci je obsažena přímo v aplikaci samé. Do map je možno vkládat objekty definované v adresáři `classes` (v hlavní složce hry, nebo ve složce kampaně).

Dále je nutno (/možno) nadefinovat vlastní typy herních objektů a uložit je ve složce `classes` a vlastní funkce a uložit je ve složce `scripts`.

Při definici tříd je třeba pamatovat na fakt, že soubory se načítají v abecedním pořádku. Proto, pokud by třída z jednoho souboru byla vyděděna ze třídy v jiném souboru, měl by tento soubor být pojmenován tak, aby byl načten jako první.

Pokud programátor naprogramuje funkci Init, bude tato funkce spuštěna na začátku dobrodružství.

Složka `audio` může obsahovat vlastní hudbu a zvuky (např. nadabované dialogy).

C.2 Programování v DSL

Popis gramatiky DSL je v kapitole 5, zde bude jazyk DSL vysvětlen na příkladech.

Definice třídy: Definice třídy je uvedena hlavičkou, která se sestává z názvu třídy a případně z dvojtečky a názvu mateřské třídy. Tělo třídy je uzavřeno mezi složenými závorkami a obsahuje seznam atributů a jejich hodnot (název, rovnítko, hodnota). Hodnotou může být řetězec, odkaz do stringtable nebo číslo. Navíc je možno jako atribut uvést i další složené závorky a procedurální kód z druhé části DSL (viz dále).

Příklad:

```
/// Base class for triggers.
Trigger: MapObject
{
  handler = {StdPrintLn("Base trigger triggerred.");}
}

/** Base class for NPCs - encounter won't implicitly
 * start a fight but it can still go that way. */
NPC: Trigger
{
  npc = 1;
  handler = {StdPrintLn($GOODDAY);}
}
```

Definice funkce: Definice funkce je uvozena názvem funkce, kulatými závorkami a v nich případným seznamem názvů parametrů. Poté následují složené závorky a v nich tělo funkce.

Příklad:

```
MojeFunkce(parametr1,parametr2)
{
  // tělo funkce
}
```

Tělo funkce se skládá z příkazů, složených, nebo jednoduchých. Jednoduché příkazy jsou ukončeny středníkem.

Jednoduchý příkaz může být přiřazení, volání funkce, nebo návratového příkazu. Přiřazení je provedeno napsáním názvu proměnné, rovnítka a výrazu určujícím novou hodnotu. Před název proměnné je možno připsat klíčové slovo **global**. Volání funkce je provedeno napsáním názvu funkce, kulatých závorek a v nich seznamu výrazů určujících argumenty volání.) Návrat z funkce je proveden klíčovým slovem **return** a případně hodnotou, která se má vrátit. Výraz je složený z názvů funkcí (tentokrát bez **global**), volání funkcí, čísel, řetězců a aritmetických operátorů.

Příklad:

```
global prom1 = 2*3;
prom2 = "ahoj";
MojeFunkce(prom1+1,prom2+"svete");
```

Ze složených příkazů je možno použít **if** a **while** způsobem uvedeným v příkladu:

```
n = ZiskejPocet();
if (n==7) {
  // ...
}
elif (n>3) {
  // ...
}
else {
  // ...
}

i = 0;
while (i<n) {
  // ...
  i = i+1;
}
```

Práce s poli je maximálně jednoduchá. Nejprve je nutno pole vytvořit. To se provede stejným způsobem jako přiřazení, ale napravo od rovnítka je nutno napsat hranaté závorky.

Poté je možno přiřazovat dovnitř pole. To je provedeno napsáním názvu pole, hranatých závorek s indexem a poté rovnítkem a výrazem. Index pole výraz, který má hodnotu buď celého čísla, nebo řetězce. Uvnitř pole je možno vytvářet další pole a zapisovat do nich přidáním dalších hranatých závorek. Stejným způsobem je možno z pole číst.

Příklad:

```
pole = [];
pole[2] = [];
pole[2]["jmeno"] = "Tonda";
pole[2]["prijmeni"] = "Ucho";
StdPrintLn(pole[2]["jmeno"]+" "+pole[2]["prijmeni"]);
```

Vestavěné funkce Systém má v sobě vestavěno omezené množství funkcí, které umožňují ovládat hru. Krom toho je možno v samotném jazyce DSL nadefinovat pomocné funkce a

uložit je v adresáři `scripts` v hlavním adresáři hry. Zde je jejich seznam:

int Rand6()	Vrátí číslo od jedné do šesti (včetně).
int Rand10()	Vrátí číslo od jedné do deseti.
void StdPrintLn(string)	V Linuxu vypíše na konzoli ladící hlášku. (ve Windows uloží do log souboru)
string Nmr2Str(int)	Převede číslo na řetězec.
void PrintNmr(int)	Na konzoli vypíše číslo.

V budoucnu bude nabídka vestavěných funkcí rozšiřována.