Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Lukáš Hermann

Ekonomická tahová strategická hra

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Jan Lánský
Studijní program: Informatika, Programování

2009

Děkuji všem, kteří se podíleli na neprogramátorské části tvorby této hry, především Lukáši Nádvorníkovi a Jáchymu Floriánovi. Také děkuji panu Mgr. Janu Lánskému za vedení mé práce.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Lukáš Hermann

Obsah

Úvod **6**

1 Herní principy a analýza **8**
1.1 Popis pravidel a cíl hry 8
1.2 Analýza pravidel a jejich implementace 10

2 Objektový model **13**
2.1 Společné rozhraní objektů 13
2.2 Příklad implementace herního objektu 14
2.3 Propojení herních objektů 16

3 Ekonomické modely a algoritmy **18**
3.1 Snižování ceny při přebytku a algoritmus prodeje zvířat . . . 18
3.2 Zvyšování ceny při nedostatku a algoritmus skladu jídla . . . 22
3.3 Algoritmus trhu se surovinami 25
3.4 Algoritmus odstranění předmětů ve zničených budovách . . . 26

4 Editor vstupních dat a skriptování **29**
4.1 Editor vstupních dat 29
4.2 Skriptovací jazyk a rozhraní 31
4.3 Další způsoby editace dat 35

5 Grafický interface a uživatelské prostředí **37**
5.1 Základní třídy pro grafické komponenty 37
5.2 Příklad grafické komponenty 41
5.3 Skládání uživatelského prostředí 42

Závěr **45**

Literatura **47**

Seznam obrázků **48**

A Obsah přiloženého CD, kompilace a instalace **49**

Název práce: Ekonomická tahová strategická hra
Autor: Lukáš Hermann
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: Mgr. Jan Lánský
e-mail vedoucího: zizelevak@gmail.com

Abstrakt: V předložené práci se zabýváme návrhem a implementací ekonomické tahové strategie. Po popsání a analýze základních pravidel hry vytváříme objektový model, skládající se ze základního rozhraní, konkrétních tříd a jejich propojení mezi sebou. Dále podrobně rozebíráme ekonomické modely a další související algoritmy použité v jejich metodách včetně jejich zapsání v pseudokódu. Nakonec tvoříme editory vlastností a chování herních objektů a zabýváme se tvorbou originálního grafického interface s konkrétními vizuálními komponentami, z nichž sestavujeme uživatelské prostředí.

Klíčová slova: počítačová hra, tahová strategie, ekonomické modely, skriptování, vizuální komponenty

Title: Economic turn-based strategy game
Author: Lukáš Hermann
Department: Department of Software Engineering
Supervisor: Mgr. Jan Lánský
Supervisor's e-mail address: zizelevak@gmail.com

Abstract: In the thesis we occupy ourselves with the design and the implementation of an economic turn-based strategy game. After describing and analyzing basic game rules we create an object model, consisting of a common interface, specific classes and their mutual connections. Then we analyze economic models and other related algorithms used in their methods including their transcription to a pseudo-code. Finally we create editors of properties and own behaviour of game objects and we work at creating an original graphic interface with specific visual components which a user interface is made of.

Keywords: computer game, turn-based strategy, economic models, scripting, visual components

Úvod

V této práci je popsán návrh a implementace ekonomické tahové strategie. Motivací k vytvoření této hry byl nedostatek kvalitních nekomerčních tahových strategií v českém prostředí a pozitivní ohlasy na předchozí verze. Tato práce vznikla za účelem zkvalitnění a zpřehlednění celého projektu, seznámení čtenářů s projektem a předání zkušeností s jeho tvorbou.

Tvorbu hry tohoto typu lze rozdělit na několik fází, kterým se budeme postupně věnovat v jednotlivých kapitolách. Nejdříve probíranou oblast zhodnotíme obecně, poté bude popsán konkrétní způsob řešení a budou uvedeny příklady z implementace.

První, co je potřeba vymyslet, jsou základní pravidla a principy hry. Poté provedeme analýzu, v níž budeme klást požadavky na nástroje, kterými budou tato pravidla realizována. Vytvoří se z nich herní objekty s různými vlastnostmi a interakcemi s ostatními. Tato abstrakce pak bude přímo či nepřímo implementována v objektovém modelu programu. Kromě tohoto postupu, který bude uveden s konkrétními příklady, se ještě budeme zabývat společným rozhraním pro všechny objekty, které by mělo zaručit stejný způsob práce s nimi, což je důležité zejména pro rozšiřování hry zvenčí (editory dat, skriptování).

Jelikož se jedná o ekonomickou strategii, musí se některé objekty chovat podle modelů, které je třeba vymyslet tak, aby se přibližovaly realitě, byly jednoduché na vytvoření a zároveň přidaly hře na hratelnosti. Implementace některých interakcí objektů vyžadují složitější algoritmy, které se musejí navrhnout nebo s úpravami převzít. V příslušné kapitole budou uvedeny konkrétní příklady modelů a algoritmů použitých ve hře s odůvodněním jejich použití a s postupem jejich tvorby.

Pokud chceme, aby hra byla dostatečně variabilní a rozšiřitelná, tedy aby vlastnosti, ale i některé chování objektů šlo jednoduše měnit bez zásahu do původního kódu, musíme kromě zanesení těchto principů do samotných objektů také vytvořit editory, prostřednictvím nichž se budou data měnit.

V kapitole věnující se tomuto tématu bude popsáno vytváření těchto editorů, způsobu uložení dat a jejich spolupráci s programem. Pro možnost úpravy chování bude popsán návrh jednoduchého skriptovacího jazyka.

Nakonec se budeme zabývat tvorbou uživatelského prostředí. Stejně jako v klasické aplikaci se bude celé prostředí sestavovat z jednotlivých ovládacích prvků, jejichž vzhled a chování bude navrženo pro potřeby hry. Zamyslíme se nad způsobem vytvoření grafického interface a hierarchií grafických komponent a budeme se zabývat konkrétní realizaci použitou ve hře.

V závěru práce zhodnotíme dosažení cílů a zamyslíme se nad možnými dalšími rozšířeními hry. V dodatku bude popsán obsah přiloženého CD, postup kompilace zdrojových kódů, instalace a používání programu a jeho hardwarové a softwarové požadavky.

Kapitola 1

Herní principy a analýza

V této kapitole zmíníme základní pravidla a cíl vytvářené hry, na základě čehož vypracujeme analýzu její implementace. Její kompletní pravidla a ovládání jsou uvedena v uživatelské dokumentaci přiložené k této práci (viz příloha A).

1.1 Popis pravidel a cíl hry

Hlavní náplní hry je snaha hráče pomocí živočišné produkce dosáhnout pro sebe a svoji rodinu lepší životní úrovně. Začíná sám s vybraným pozemkem, několika startovními budovami a trochou peněz.

Do některých budov může nakoupit zvířata schopná se po určité době pářit a tím produkovat mláďata, která se dají prodat na jatkách nebo zpracovat na jídlo, jež se uskladní a později prodá na trhu s jídlem. Každý druh zvířete má jiné vlastnosti – dobu dožití, náklady na krmivo, prodejní cenu mláďat apod.

Další budovy slouží k různým účelům – elektrárna a vodárna produkují elektřinu a vodu nutné pro provoz budov, veterina je potřeba k ošetřování a kontrolování zvířat, v ubytovnách bydlí najmutí zaměstnanci, ve skladu je jídlo a další předměty a chrámy zvyšují pracovitost zaměstnanců a přinášejí i další výhody. Tyto budovy nemusí hráč stavět hned, protože vše se dá pořídit na trhu se surovinami, ale později to pro něj může být výhodnější.

Aby budovy fungovaly, musí být připojeny k cestě a musí je obsluhovat zaměstnanec příslušného druhu. Ze začátku budou v budovách pracovat anonymní nádeníci najatí na trhu práce, zanedlouho se ale hráči vyplatí najmout si je nastálo. Každý zaměstnanec má jiné vlastnosti ovlivňované

různými faktory (výše platu, náboženství, ...); některé jsou známy hned, jiné se projeví až časem.

Dále se hráč musí starat o založení rodiny, jelikož po nějaké době zemře a musí někomu farmu předat. Nejdříve je třeba najít vhodnou partnerku ochotnou si ho vzít. Pak se po několika pokusech narodí dítě – dcera nebo syn. Dcera se může dobře provdat a syn se může stát hráčovým nástupcem. Než dospějí, je samozřejmě nutné se o ně i o manželku postarat, a to nejenom dostatkem jídla a peněz, ale také nejrůznějšími dárky, díky kterým bude hráč v jejich oblibě.

Hra má ještě spoustu dalších možností. Nakupováním nejrůznějších předmětů v obchodech mohou hráči leccos ušetřit či vylepšit. Později také musejí přemýšlet o domácí a zahraniční politice a sledovat příběh, jelikož některá uskupení mohou v lecčem pomáhat i škodit. Při hře více hráčů se také mohou zaměřit na prodávání přebytečných surovin ostatním. Na začátku si každý hráč vybere své náboženství, což je podstatné pro celou hru – rozdíly jsou nejen ve startovních budovách a možnostech chování různých zvířat. Po zastavění prvního pozemku lze koupit další nebo lze využít různé terénní úpravy a odstraňování překážek.

Jelikož se jedná o tahovou strategii, hraje důležitou roli čas. Jednotlivé tahy představují roční období. V každém z nich lze vyvíjet různé aktivity – zvířata se v různých obdobích páří a zabíjejí, některé dárky se dají koupit jen v určitém období apod. Dalšími časovými úseky jsou rok a pětiletka, po které se uplynulé období zhodnotí a v případě hry jednoho hráče se dá hra v tuto chvíli uložit, nebo pětiletka restartovat.

Cílem hry je dosáhnout desáté životní úrovně a udržet se na ní celou další pětiletku. Životní úroveň je číslo mezi nulou a desítkou a vypočítává se z hodnoty společnosti, do které je zahrnut veškerý majetek, peníze a dluhy. Její hodnota, stejně jako množství peněz a jídla jsou pro hráče stále viditelné. V případě hry více hráčů vyhraje ten, kdo si ji udrží nejvyšší jako první celou pětiletku. V případě, že se to podaří najednou více hráčům, rozhoduje hodnota společnosti na konci pětiletky. Hra pro hráče končí v případě, že jeho dluhy přerostou hodnotu jeho společnosti a není už nikdo, kdo by mu mohl peníze darovat. Celá hra pak končí buď vítězně pro některého hráče, prohrou všech hráčů, nebo po určitém počtu let.

1.2 Analýza pravidel a jejich implementace

V pravidlech hry můžeme vysledovat konkrétní objekty, jejich vlastnosti a interakci s okolím – zaměstnanci pracující v budovách, zvířata zpracovávaná na jatkách, obchodování mezi hráči na trhu atd. Pro jejich implementaci bude tedy přirozené použít programovací jazyk podporující objektově orientované programování. Některých objektů (např. druhů zvířat či budov) bude mnoho s různými vlastnostmi, které by rozhodně měly být měnitelné mimo zdrojový kód programu. K snadnému vytvoření editorů těchto vlastností by měl programovací jazyk alespoň částečně podporovat reflexi. Jelikož těchto dat nebude mnoho a budou spolu propojena jen minimálně, bylo by databázové řešení zbytečně komplexní, a proto si vystačíme s jednoduchým sekvenčním ukládáním do souborů.

Dále budeme chtít vytvořit originální uživatelské prostředí odlišné od systémového, které si budeme moci přizpůsobit podle potřeb a které bude dostatečně rychlé a jednoduše použitelné. K vytvoření tahové strategie nám bude stačit statická 2D grafika. Jelikož nepožadujeme nezávislost na operačním systému (spokojíme se s fungováním na Windows), použijeme grafické rozhraní *DirectX*. Toto rozhraní je navíc přítomno na každém počítači s novějšími Windows, což zjednoduší distribuci programu. Protože by bylo neproduktivní zabývat se technickými detaily tohoto rozhraní, měli bychom mít možnost použití knihoven, které práci s ním usnadní.

Dalším požadavkem na programovací jazyk je možnost použití výjimek, usnadňujících ošetřování chyb v kódu a další nestandardní situace. Také by měl ve svých knihovnách obsahovat generátor pseudonáhodných čísel, která přispějí k rozšíření množství herních situací, díky čemuž se bude hráč ke hře vracet. Z pravidel také vyplývá, že jednu hru bude moci hrát více hráčů, mezi kterými bude probíhat interakce. Nám bude stačit hra více hráčů na jednom počítači, díky čemuž se nebudeme muset zabývat síťovou komunikací.

Tyto požadavky víceméně splňuje celkem velká množina moderních programovacích jazyků. Důležitým faktorem při rozhodování, jaký nakonec použít, byla již částečná implementace v jednom z nich a jeho dobrá znalost, stejně jako znalost příslušných knihoven. Přepisování do jiného modernějšího a třeba i vhodnějšího jazyka by zabralo hodně času, který může být využit na hlubší rozpracování některých podstatných částí celé hry a přidání více hráčských možností. Vybraným programovacím jazykem jsou proto *Delphi 5* s knihovnou *DelphiX* [1] jako rozhraním pro práci s *DirectX*.

Celý projekt bude rozdělen na několik částí. První z nich bude vlastní

hra, kterou dále rozdělíme na unitu definující a implementující třídy herních objektů, v nichž se realizují pravidla hry, a unitu obsahující grafické komponenty, z nichž se bude skládat uživatelské prostředí popsané v poslední unitě. Dalšími částmi budou editory vlastností a chování herních objektů. První z nich se bude používat k editaci jejich počátečních vlastností, další k modifikaci jejich chování při různých událostech a nakonec vytvoříme speciální editor pozemků, na kterých budou stát budovy hráčů. Ke všem částem bude dodána uživatelská dokumentace (viz příloha A) a jejich implementace bude popsána v této práci v kapitolách uvedených v tabulce 1.1.

Část projektu	Kapitoly
Vlastní hra	
Objektový model	2 a 3
Grafické komponenty	5.1 a 5.2
Uživatelské prostředí	5.3
Editory	
Editor vlastností objektů	4.1
Editor chování objektů	4.2
Editor pozemků	4.3

Tabulka 1.1: Obsah kapitol podle částí projektu

Zabývejme se nyní analýzou některých částí projektu. Pro externí modifikaci chování objektů je třeba použít skriptovacího jazyka schopného propojit se s programem, měnit vlastnosti objektů a volat jejich metody. Na internetu můžeme nalézt komerční[1] i nekomerční[2] skriptovací jazyky přímo pro *Delphi* a také třídy[3] umožňující použití univerzálnějších jazyků (JScript, Perl, Python, Lua, ...) přes *Windows Script Host*. V předcházejících verzích projektu byl také vytvořen jednoduchý skriptovací jazyk, lehce upravitelný pro spolupráci s novými rozhraními. Jelikož prozatím dostačuje požadovaným nárokům a jelikož jsou uvedená hotová řešení pro naše účely zbytečně komplexní a jejich propojení s programem komplikovanější, adaptujeme ho pro rozhraní zadefinované v kapitole 2.1, což popíšeme v kapitole 4.2.

Zamysleme se také nad způsobem vytvoření uživatelského prostředí. Rozhodli jsme se použít knihovny *DelphiX*, usnadňující práci s rozhraním *Di-*

[1] Dbasic (http://www.msbsoftware.it/dbasic/)
[2] TjanPico (http://jansfreeware.com/jfdelphi.htm)
[3] Windows Scripting Host Control (http://www.ekassw.com/wshctrl/)

rectX. Ty zajišťují jednoduché vykreslování elementárních grafických tvarů (čáry, obdélníky apod.), textů a obrázků. Z nich je třeba poskládat jednotlivé ovládací prvky (editační řádka, seznamy položek, tlačítka atd.), z nichž se uživatelské prostředí skládá. Jelikož neexistují žádné standardní knihovny tyto prvky obsahující a jediné nalezené řešení[4] je nevyhovující, vytvoříme vlastní grafické komponenty v kapitole 5.

[4]DxWindows (http://www.micrel.cz/Dx/download/dxwindow.rar)

Kapitola 2

Objektový model

V této kapitole se budeme zabývat objektovou implementací pravidel hry. Nejdříve se budeme věnovat společnému rozhraní všech objektů, pak si ukážeme konkrétní příklad realizace jednoho z nich a nakonec rozebereme jejich propojení mezi sebou.

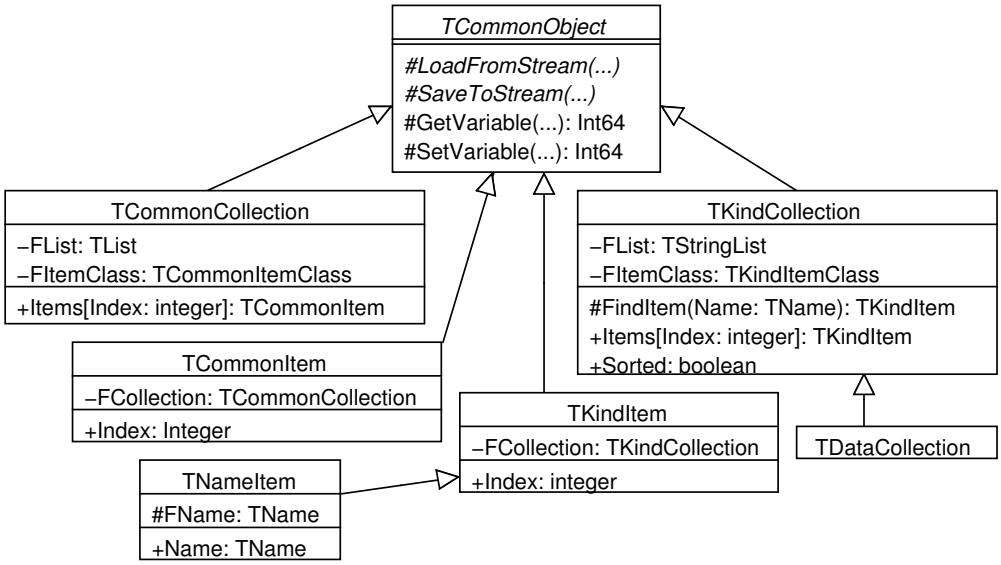
2.1 Společné rozhraní objektů

Po každém objektu vyskytujícím se ve hře požadujeme implementaci některých základních interakcí. Především by měl umět uložit svá data do proudu dat a zpětně se z nich zrekonstruovat, což je podstatné při ukládání a načítání hry ze souboru a při získání jeho počátečních vlastností (viz kapitola 4.1). Jelikož chceme, aby veškerá jejich data byla editovatelná zvenčí za běhu, zadefinujeme dvě metody, které budou čtecím a zapisovacím rozhraním mezi objektem a skriptovacím jazykem (viz kapitola 4.2). Tyto čtyři metody spojíme pod abstraktní třídu *TCommonObject*, která bude předkem pro všechny herní objekty.

Dalším společným rysem mnohých objektů je velký počet jejich instancí. Ty se uspořádají do kolekcí, přes které je bude snadné procházet popřípadě vyhledávat. Některé objekty budou mít jednoznačný identifikátor (jméno), podle kterého by je mělo jít rychle v kolekci (*TKindCollection*) vyhledat, u jiných bude stačit pouze lineární procházení celé kolekce (*TCommonCollection*). Oba typy kolekcí zajistí uložení svých objektů do proudu dat a jejich nahrání z něj voláním příslušných metod těchto objektů. Budou si také pamatovat, jakou třídu objektů obsluhují, a zajistí k nim přístup pomocí indexace podle pořadí, v případě prvního typu i podle jména.

Objekty vkládané do kolekcí budou vědět, ke které náleží, aby ji mohly informovat o svém zániku a aby přes ni mohly přistupovat k ostatním. Ty, které budou chtít rychlý přístup přes své jméno, ho budou mít jako proměnnou (*TNameItem*), nebo ho budou zjišťovat jiným způsobem (*TKindItem*). Některé takové objekty se budou na začátku hry získávat z editovatelných datových souborů, proto kolekce (*TDataCollection*) zajistí jejich korektní vytvoření.

Příkladem objektů, které nemají žádný jednoznačný identifikátor, jsou budovy, kterých může být více stejného typu, naopak každý druh dárku je ve hře jednoznačně identifikován svým jménem. Obrázek 2.1 ukazuje hierarchii základních tříd.



Obrázek 2.1: Hierarchie základních tříd použitých v objektovém modelu

2.2 Příklad implementace herního objektu

Ideálním příkladem demonstrujícím implementaci herního objektu jsou jatka. Na začátku stojí představa možnosti nakupovat a prodávat zvířata na různých místech za různé ceny měnící se v čase. K tomu se přidá požadavek, aby cena za kus byla při prodeji většího množství zvířat nižší a aby množství,

při kterém se cena začne snižovat, bylo různé a také se měnilo. Ceny by také mohly být navázány na další hráčovy aktivity. Zároveň by bylo vhodné, aby hráč tato místa nemohl libovolně měnit, jinak by se pokaždé ihned přemístil na výhodnější místo.

Z těchto požadavků se postupně začíná vynořovat objekt s konkrétními vlastnostmi, který by v reálném světě zastoupila jatka, a proto třídu popisující tento objekt pojmenuji *TSlaughter*. Jelikož jich bude víc a budou jednoznačně identifikované svým jménem, bude tato třída dědit od *TNameItem*. K ní vytvořená kolekce *TSlaughters* bude dědit od *TDataCollection*, protože jatka budou na začátku hry nahrána z datového souboru. Dále budu postupovat podle zásad tvorby tříd. Všechny proměnné třídy budou soukromé, navenek budou reprezentovány pomocí vlastností (angl. properties), které budou jen pro čtení. Veškerá modifikace proměnných se bude dít přes veřejné metody.

Název vlastnosti	Druh vlastnosti	Popis vlastnosti
Capacity	64-bitové číslo se znam.	počáteční kapacita jatek v tunách
CapacityChange	8-bitové číslo se znam.	počáteční roční změna kapacity jatek (%)
Exists	ano / ne	existují jatka na začátku hry?
(Fe)MaleBuyPrice	32-bitové číslo	nákupní cena samce / samice
(Fe)MaleBuyPriceChange	8-bitové číslo se znam.	změna ceny samce / samice za rok (%)
FodderCosts	16-bitové číslo	cena 1 kusu krmení
FodderCostsChange	8-bitové číslo se znam.	změna ceny 1 kusu krmení (%)
Group	výčtový typ	druh uskupení, kterému patří
Kind	výčtový typ	druh zvířat, která se tam zpracovávají
Name	text	název jatek
OnCanRegister	skripty	skripty vracejí, zdali se dá registrovat
OnDeregistration	skripty	skripty při zrušení registrace
OnExist	skripty	skripty při existenci jatek
OnRegistration	skripty	skripty při registraci
OverCapacityP/Q/R	8-bitové číslo	koeficienty funkce popisující pokles ceny při překročení kapacity
RegisterPrice	16-bitové číslo	základní cena za registraci
SellPrice	16-bitové číslo	prodejní cena jedné hmotnostní jednotky
SellPriceChange	8-bitové číslo se znam.	změna prodejní ceny za rok (%)

Tabulka 2.1: Vlastnosti třídy TSlaughter

V tabulce 2.1 jsou uvedeny všechny vlastnosti třídy se svými typy a stručným popisem významu. Za povšimnutí stojí zejména ty s postfixem *Change* vyjadřující, o kolik procent se každý rok automaticky změní příslušná vlastnost. Tuto změnu bude třeba implementovat v metodě, která bude volána po ukončení tahu. Nazveme ji *Refresh* a kromě těchto změn se v ní bude vypořádávat i prodej zvířat zpracovávaného druhu od všech hráčů u těchto jatek zaregistrovaných. Více se o konkrétním algoritmu pojednává v kapi-

tole 3.1. Metodu *Refresh* různými způsoby zpracovávají všechny třídy, které chtějí reagovat na konec tahu hráče nebo všech hráčů.

Dalšími důležitými vlastnostmi jsou ty s prefixem *On*, v nichž jsou uložena jména skriptů spouštějících se při provádění příslušných metod objektu. Například ty zapsané v *OnRegistration* se spustí při registraci hráče k jatkám, ty v *OnExist* zase při zavolání metody *Refresh* na konci tahu všech hráčů, pokud budou jatka existovat (proměnná *Exists*). Ty v *OnCanRegister* vracejí, zdali se k nim aktuální hráč může zaregistrovat, díky čemuž mohou mít odlišné podmínky pro registraci. Skripty mají přístup ke všem proměnným a metodám všech objektů ve hře. Je jim věnována kapitola 4.2.

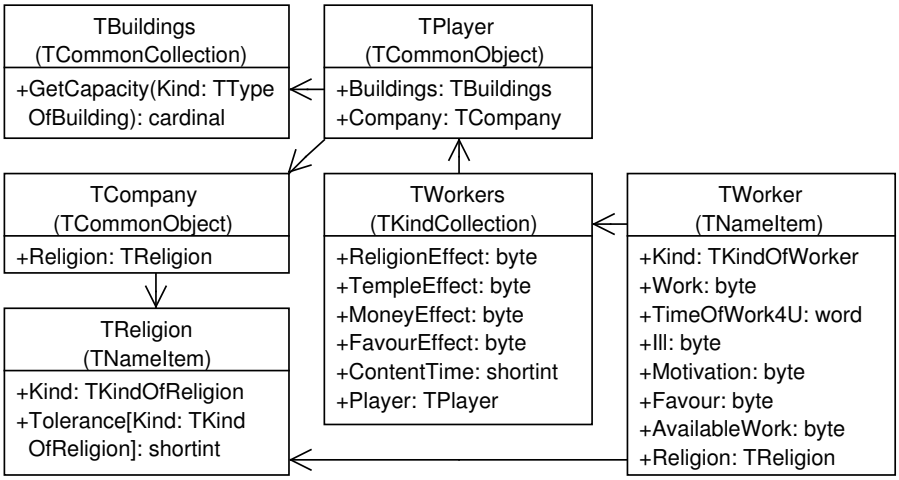
2.3 Propojení herních objektů

Po vytvoření všech tříd je velmi důležité je propojit tak, aby se respektovala logika celého modelu a aby byly mezi sebou jednoduše přístupné.

Jelikož se některé třídy týkají vlastností a vlastnictví konkrétního hráče a v jiných jsou uložena data sdílená hráči dohromady, vytvoříme třídu *TPlayer* reprezentující jednoho hráče a třídu *TGame* spojující všechna herní data. V obou budou ukazatele na instance všech příslušných kolekcí a samostatných objektů a tyto budou mít ukazatele zpět na instanci jedné z nich. Navíc *TPlayer* bude mít ukazatel na *TGame* a ta bude udržovat pole s objekty *TPlayer* a ukazatel na právě hrajícího hráče. Jelikož každý objekt v kolekci má na ni ukazatel a obráceně, mají všechny objekty přístup mezi sebou.

Tyto dvě třídy navíc při svém vytváření inicializují všechny objekty, na které ukazují, volají jejich metody pro načítání a ukládání dat, po konci tahu metodu *Refresh* u těch, které ji mají, a nakonec se postarají o jejich korektní odstranění. Instance třídy *TGame* je také jediná, která je přístupná mimo tento model, a je tedy jediným vstupem ke všem herním objektům (výjimku tvoří skripty, které mohou přistupovat přímo k aktuálním objektům).

Příkladem propojení objektů může být metoda *GetAvailableWork* třídy *TWorker* vracející, kolik daný zaměstnanec aktuálně může odpracovat. To závisí na různých faktorech, z nichž některé se získají přímo v *TWorker* (základní práce, nemoc), některé vzniknou kombinací vlastností zaměstnance a příslušných koeficientů jejich efektu, nacházejících se v kolekci zaměstnanců *TWorkers* (doba od najmutí, motivace penězi, dlouhodobá obliba), a některé se musí zjišťovat v ostatních třídách – množství chrámů ve třídě *TBuildings* obsahující kolekci hráčových budov nebo hráčovo náboženství ve třídě *TCompany* udržující základní informace o jeho postavě pro zjištění,



Obrázek 2.2: Ukázka propojení herních projektů v metodě *GetAvailableWork* třídy *TWorker*

jak je k němu zaměstnanec tolerantní. Na obrázku 2.2 je toto propojení zobrazeno. V tabulce 4.1 na straně 30 je pak zobrazen seznam většiny objektů ve hře použitých.

Kapitola 3

Ekonomické modely a algoritmy

Po vytvoření interface všech objektů je třeba doplnit implementaci jejich metod. V některých se budou používat složitější ekonomické modely a algoritmy, které budou popsány v této kapitole. Nejdříve se budeme zabývat implementací již popsané metody *Refresh* třídy *TSlaughter*, s níž je spojeno řešení problému snižování ceny za kus při zvýšeném množství prodeje zvířat. Poté budeme řešit opačný problém – jak se bude zvyšovat cena při nákupu většího množství surovin. S tím je spojen algoritmus skladu jídla, který bude spouštěn také po ukončení tahů všech hráčů. Dále se zaměříme na obchodování s dalšími herními surovinami mezi hráči a nakonec prozkoumáme implementaci jednoho známého algoritmu dobře využitelného na řešení uvedeného problému.

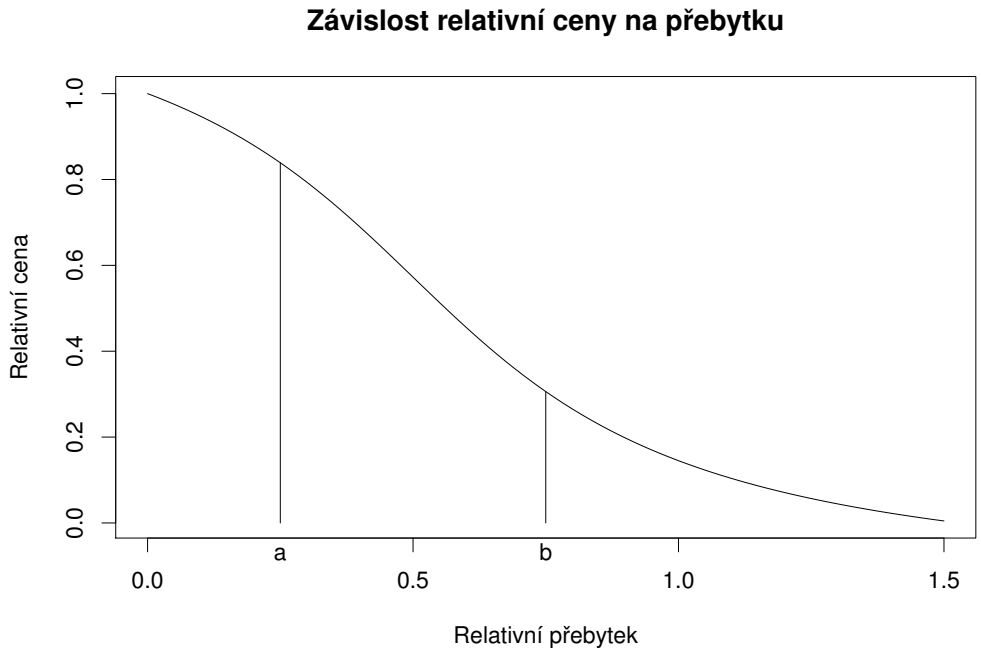
3.1 Snižování ceny při přebytku a algoritmus prodeje zvířat

Pokud je suroviny (v našem případě zvířat nebo jídla) přebytek, začne její cena klesat. U zvířat je to dáno především kapacitou jatek, která je odebírají. Pokud počet dodaných zvířat za rok překročí jejich kapacitu, jatka mají obtíže s jejich zpracováním, a proto nastavují nižší výkupní cenu. Obdobně se chová centrální sklad jídla, kterému také klesá užitek z jeho nadbytečnosti. Rozdíl oproti jatkám je v tom, že ta zpracovaná zvířata okamžitě prodají dál, kdežto jídlo se ve skladu hromadí a prodává se dál postupně. Cena tedy

závisí i na historii jeho nákupu a prodeje.

V obou případech je třeba pokles ceny při rostoucím přebytku postihnout vhodnou funkcí. Cena by měla při malém přebytku klesat pozvolna (lze jej zpracovat za malé náklady navíc), poté by se měl její pokles zrychlit (zpracování většího množství vyžaduje výrazně vyšší náklady) a nakonec znovu zvolnit (velmi nízká cena činí náklady přijatelnějšími) až k nulové ceně (větší přebytek by znamenal vyšší náklady než zisky). Tento požadavek dobře splňuje funkce $\operatorname{arctg}(-x)$, kterou upravíme tak, aby v 0 byla její hodnota 1 (při nulovém přebytku standardní cena), a doplníme ji o parametry p, q, r (na obrázku 3.1 s konkrétními):

$$f(x) = \frac{\operatorname{arctg} \frac{p-x}{q} + r}{\operatorname{arctg} \frac{p}{q} + r} \tag{3.1}$$



Obrázek 3.1: Graf zobrazuje funkci relativní ceny v závislosti na relativním přebytku $f(x)$ (3.1) pro parametry $p = 0,5$, $q = 0,4$, $r = 1,2$.

K výpočtu celkového množství peněz získaného z prodeje přebytečné suroviny potřebujeme znát $F(x)$, integrál funkce $f(x)$. Integrační konstantu zvolíme tak, aby $F(0) = 0$. Po úpravách dostaneme:

$$F(x) = \frac{2\left(rx + p\left(\operatorname{arctg}\frac{p}{q} + \operatorname{arctg}\frac{x-p}{q}\right) + x\operatorname{arctg}\frac{p-x}{q}\right) + q\ln\frac{(p-x)^2+q^2}{p^2+q^2}}{2\left(r + \operatorname{arctg}\frac{p}{q}\right)} \tag{3.2}$$

Jelikož by měl mít hráč možnosti nastavit si, za jakou cenu chce ještě zvířata prodávat a kdy si je již raději nechá na jídlo, musíme ještě spočítat funkci inverzní k $f(x)$, která nám ze zvolené relativní ceny vrátí relativní přebytek, do kterého bude ještě hráč jatka zásobovat, aby prodejní cena neklesla pod zvolenou:

$$f(y)^{-1} = p + q\operatorname{tg} r - ry - y\operatorname{arctg}\frac{p}{q} \tag{3.3}$$

Teď již tedy máme vše potřebné k sestavení algoritmu prodeje zvířat na jatkách. Pro každého hráče bude vstupem hmotnost odeslaných zvířat m_i a minimální relativní cena r_i, za niž je ochoten zvířata prodat, výstupem hmotnost zvířat p_i, která se na jatkách zpracovala (generuje peníze) a která je korigována případným poklesem ceny při přebytku, a hmotnost zvířat f_i, která se nezpracovala kvůli poklesu ceny (generuje jídlo).

V případě, že celková hmotnost zvířat od všech hráčů překračuje kapacitu jatek, vezme se nejdříve od každého poměrná část odeslané hmotnosti tak, aby součet dával kapacitu jatek, a ta se celá přičte k výsledné hmotnosti (za tuto část tedy hráč dostane plnou cenu). Nechť m_i je odeslaná hmotnost jednoho hráče, c kapacita jatek a m součet odeslané hmotnosti všech hráčů, pak počáteční $p_i = m_i \cdot c/\max(m, c)$.

Pokud kapacita překročena nebyla, jsme hotovi, jinak pro každého hráče spočítáme relativní přebytek hmotnosti e_i z kapacity c a zbytku hmotnosti zvířat po zpracování za standardní cenu $m_i - p_i$: $e_i = (m_i - p_i)/c$. Z toho dopočítáme celkový relativní přebytek t, který využijeme při dosazení do uvedených funkcí.

Jelikož si každý hráč může nastavit, při jaké ceně chce s prodejem přestat, budeme postupovat iterativně od nejvyšší relativní ceny po nejnižší, dokud bude co k prodeji a dokud bude chtít nějaký hráč prodávat ($e = \sum e_i > 0$). Před první iterací přiřadíme do levé meze $a = 0.0$, v dalších iteracích se do a dosadí předchozí pravá mez b. Pokud r je aktuální relativní cena, pak

pravá mez $b = f(r)^{-1}$. Celkově se tedy v jednom kroku bude zpracovávat $v = \min(b,t) - a$ relativního přebytku hmotnosti. Kvůli poklesu ceny získají hráči v tomto kroku peníze odpovídající hmotnosti $w = (F(\min(b,t)) - F(a)) \cdot c$. To odpovídá určitému integrálu funkce $f(x)$ s mezemi a a b, jak je vidět na obrázku 3.1, vynásobenému kapacitou.

Pak se pro každého hráče spočítá, kolik dostane peněz $(w \cdot e_i/e)$ a jaká část zpracovávané hmotnosti v této iteraci byla jeho $(v \cdot e_i/e)$, což se odečte od nezpracované hmotnosti e_i. Nakonec se z iterace odstraní ti hráči, kteří již dále nechtějí prodávat (v každé iteraci nejméně jeden), a zbytek jejich nezpracované hmotnosti se přemění na jídlo. O toto množství se také sníží celkový relativní přebytek t.

Celý algoritmus s výše popsanými proměnnými (P je množina všech hráčů vstupujících do prodeje) je uveden v algoritmu 1.

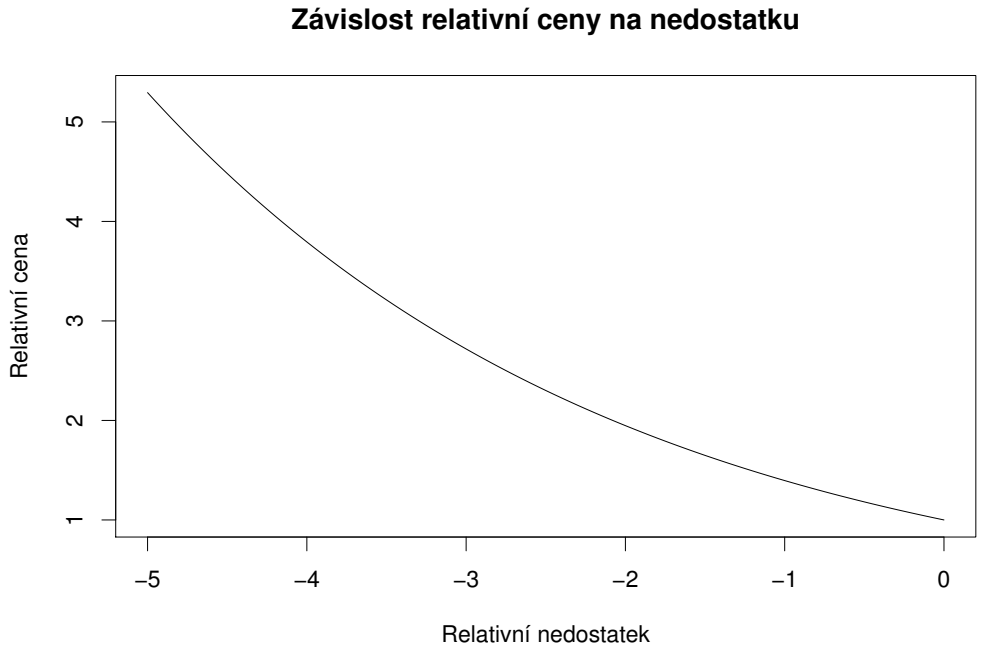
```
foreach i ∈ P do
    p_i = m_i · c/ max(m, c); // část hmotnosti za standardní cenu
    e_i = (m_i − p_i)/c;      // relativní přebytek hmotnosti
end
a = 0.0; t = (m − c)/c; // celkový relativní přebytek hmotnosti
while e = Σ e_i > 0, i ∈ P do      // dokud je co prodávat
    r = max(r_i), i ∈ P;           // aktuální relativní cena
    b = f(r)^−1;                   // pravá mez integrálu
    v = min (b, t) − a;            // zpracovaná relativní hmotnost
    w = (F(min (b, t)) − F(a)) · c;  // cenově odpov. hmotnost
    foreach i ∈ P do
        p_i = p_i + w · e_i/e;     // přičtení podílu na zisku
        e_i = e_i − v · e_i/e;     // odečtení podílu zprac. hmotnosti
    end
    a = min(b, t);                 // posunutí levé meze na pravou
    foreach i ∈ P, r_i = r do      // vyjmutí končících hráčů
        P = P \ {i};
        f_i = e_i · c;             // hmotnost ke zprac. na jídlo
        t = t − e_i;               // snížení celkové zpracovávané hmotnosti
    end
end
```

Algoritmus 1: Zpracování zvířat na jatkách

3.2 Zvyšování ceny při nedostatku a algoritmus skladu jídla

V této části se budeme zabývat opačným problémem – nedostatkem surovin, které musí hráč získávat. Půjde především o jídlo, ale i o elektřinu, vodu, námezdní práci atd. Je zřejmé, že cena se bude při nedostatku suroviny zvyšovat. Její nárůst by měl být nejdříve pozvolný a postupně se zvětšovat, proto použijeme exponenciální funkci doplněnou parametrem, která bude na intervalu $(-\infty, 0)$ klesající, větší než 1 a v bodě 0 bude nabývat hodnoty 1, jelikož relativní nedostatek suroviny bude vyjádřen zápornými čísly. Výsledkem je funkce $g(x)$ s parametrem m (na obrázku 3.2 s konkrétním):

$$g(x) = e^{-\frac{x}{m}} \tag{3.4}$$



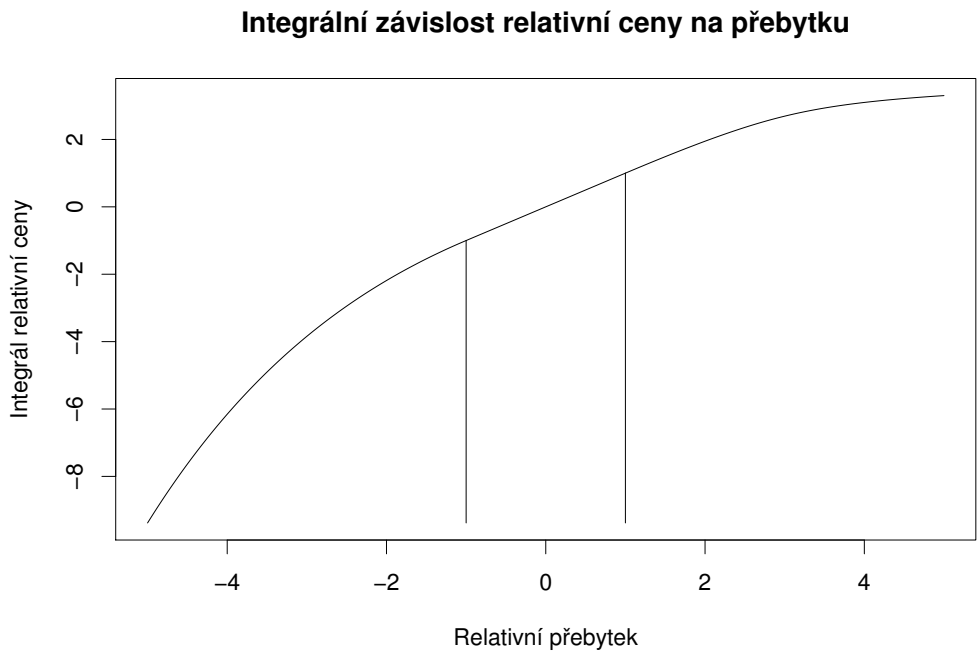
Obrázek 3.2: Graf zobrazuje funkci relativní ceny v závislosti na relativním nedostatku $g(x)$ (3.4) pro parametr $m = 3$.

Pro další výpočty potřebujeme získat integrál této funkce $G(x)$, pro nějž bude platit $G(0) = 0$:

$$G(x) = -m \cdot e^{-\frac{x}{m}} + m \tag{3.5}$$

Jelikož chceme simulovat sklad jídla, kde se jídlo nakupuje i prodává, a tudíž ho může být přebytek i nedostatek, vytvoříme si pomocnou funkci $H(x)$ složenou z funkcí $F(x)$ a $G(x)$, která bude pokrývat oba extrémy i běžný stav. Její graf s konkrétními parametry můžete vidět na obrázku 3.3.

$$H(x) = \begin{cases} G(x+1) - 1 & x \in (-\infty, -1) \\ x & x \in (-1, 1) \\ F(x-1) + 1 & x \in (1, \infty) \end{cases} \tag{3.6}$$



Obrázek 3.3: Graf zobrazuje integrální funkci relativní ceny v závislosti na relativním přebytku $H(x)$ (3.6) pro parametry $p = 2$, $q = 1$, $r = 1,5$, $m = 3$.

Vstupem algoritmu zpracování jídla bude pro každého hráče množství jídla f_i, které chce nakoupit či prodat, kapacita skladu jídla c, koeficient maximální kapacity m, standardní cena jídla p, koeficient ceny při nákupu či prodeji b, údaj s popisující přebytek či nedostatek jídla na skladu oproti optimálnímu stavu (promítá se do něj předchozí obchodování) a nakonec stabilizační koeficient k udávající, jak rychle se sklad dostává do optimálního stavu zásob. Výstupem pro každého hráče je množství peněz p_i, které dostane z prodeje nebo dá za nákup, a nová hodnota s popisující odchylku od optimálního stavu.

$t_s = \sum f_i, f_i > 0;$ `// celkový objem prodeje`
$t_b = \sum f_i, f_i < 0;$ `// celkový objem nákupu`
$t = t_s - t_b;$ `// rozdíl celkového objemu prodeje a nákupu`
$o = |t + s| - m \cdot c;$ `// přesah přes maximální kapacitu`
if $o > 0$ **then** `// úprava množství obchodovaného jídla`
  $z = \mathrm{sgn}(t + s);$ `// velký přebytek či nedostatek jídla`
  foreach $i \in P$ **do**
    if $f_i > 0$ & $z = 1$ **then** $f_i = f_i - o \cdot f_i/t_s;$
    if $f_i < 0$ & $z = -1$ **then** $f_i = f_i - o \cdot f_i/t_b;$
  end
  $t = t - z \cdot o;$
end
if $t = 0$ **then** $n = (H(\frac{s}{c} + \varepsilon) - H(\frac{s}{c} - \varepsilon))/2\varepsilon$
else $n = \left(H\left(\frac{s+t}{c}\right) - H\left(\frac{s}{c}\right)\right)\frac{c}{t};$
foreach $i \in P$ **do**
  $p_i = p \cdot n \cdot f_i;$ `// výsledné množství peněz`
  $p_i = p_i - p_i \cdot b\,\mathrm{sgn}(f_i);$ `// úprava podle nákupu či prodeje`
end
$s = s + t;$ `// zaznamenání transakce do stavu skladu`
$s = \mathrm{sgn}(s)\max(\min(|s| - c \cdot k, 0), |s|);$ `// posun k optimu`

Algoritmus 2: Obchodování se skladem jídla

Nejdříve se spočítá celkový objem prodeje $t_s = \sum f_i, f_i > 0$ a nákupu $t_b = \sum f_i, f_i < 0$ a jejich rozdíl $t = t_s - t_b$. Poté se musí upravit množství prodávaného či nakupovaného jídla, pokud by přesahovalo na jednu či druhou stranu maximální kapacity skladu $m \cdot c$. Na jedné straně sklad již nemůže i přes zvyšování ceny další sehnat a na druhé i přes její snižování nemůže další přijmout. Do tohoto množství se započítává i historie předchozích

transakcí s.

Poté se spočítá koeficient ceny za jedno jídlo n, k čemuž nám pomůže zavedená funkce $H(x)$ (3.6): $n = \left(H\left(\frac{s+t}{c}\right) - H\left(\frac{s}{c}\right)\right)\frac{c}{t}$. V případě $t = 0$ by výsledný koeficient ceny měl být derivací $H(s/c)$. Nám bude ale stačit aproximace této derivace $n = (H(\frac{s}{c} + \varepsilon) - H(\frac{s}{c} - \varepsilon))/2\varepsilon$, kde $\varepsilon = 0,05$. Tento koeficient pak použijeme pro výpočet množství peněz pro každého hráče $p_i = p \cdot n \cdot f_i$, které se ještě upraví koeficientem b zvyšujícím cenu při nákupu a snižujícím ji při prodeji.

Nakonec se upraví hodnota s o právě proběhlou transakci a v závislosti na koeficientu k se posune směrem k optimálnímu stavu (tedy k 0). Celý algoritmus s výše popsanými proměnnými (P je množina všech obchodujících hráčů) je uveden v algoritmu 2.

3.3 Algoritmus trhu se surovinami

Nyní se budeme zabývat obchodováním se surovinami, jejichž výroba musí pokrývat spotřebu v každém tahu (elektřina, voda, práce zaměstnanců, ...). Pokud má hráč nedostatek suroviny, musí ji nakoupit u jiného, který jí má přebytek, nebo u globálního dodavatele, fungujícího podobně jako sklad jídla popsaný v kapitole 3.2 (dá se ale jen nakupovat). Bylo by vhodné, aby hráč nemusel řešit obchodování každý tah, jelikož jeho nedostatek surovin bude často dlouhodobější, obchod by pro něj měl být co nejvýhodnější a měl by mít možnost ovlivnit, jakým způsobem bude obchodovat. Všechny tyto požadavky splňuje níže popsaný algoritmus.

Každý hráč si zvolí cenu p_i, za kterou je ochoten jednotku přebytečné suroviny prodat. Ta by měla být nižší než cena globálního dodavatele g, aby měl šanci něco prodat. Dále si zvolí, jakým způsobem chce tyto přebytky rozdělovat mezi ty hráče, kteří mají suroviny nedostatek. K tomu poslouží proměnná $s_{i,j}$ vyjadřující prioritu dodávek hráči j. Nastávají tyto možnosti:

$$s_{i,j} \begin{cases} = 0 & \text{hráči } j \text{ se nebude dodávat} \\ = s_{i,k} > 0 & \text{hráči } j \text{ a } k \text{ se podělí stejným dílem} \\ > s_{i,k} > 0 & \text{nejdříve se uspokojí hráč } k\text{, poté } j \end{cases} \tag{3.7}$$

Nakonec může nastavit proměnnou $d_{i,j}$ udávající, zdali chce přijímat dodávky od hráče j (true) či nikoliv (false).

Tato nastavení a přebytek či nedostatek suroviny každého hráče pro daný tah c_i tvoří vstup algoritmu. Algoritmus funguje postupným prodáváním

přebytků suroviny od těch s nejnižší cenovou nabídku k těm s nejvyšší (v případě stejné nabídky se právě prodávající vybere náhodně). Pro vybraného hráče i se zjistí všichni hráči j, kteří mají zájem o jeho dodávky – tedy $c_j < 0$ & $d_{j,i}$ & $s_{i,j} > 0$. Ti se procházejí od nejvyšší priority dodávek k nejnižší (3.7), dokud je komu co nabízet. V případě stejné priority u několika z nich se celková nabídka rozdělí na stejné díly, z nichž si vezmou, kolik potřebují. Pokud ji někteří nevyužijí celou, rozdělí se zbytek mezi ostatní se stejnou prioritou. Po uspokojení poptávky všech s touto prioritou se přejde na ty s nejbližší nižší prioritou. Výstupem algoritmu je množství peněz m_i, které hráči obchodem získají či ztratí. Popsaný algoritmus (P je množina všech obchodujících hráčů) je přehledně uveden v algoritmu 3.

Pokud po tomto obchodování některý hráč ještě stále nějaké množství suroviny potřebuje, nakoupí ji u globálního dodavatele, který má svoji cenu g a kapacitu c, neovlivněnou předchozími nákupy podobně jako u jatek.

3.4 Algoritmus odstranění předmětů ve zničených budovách

Odbočíme nyní od ekonomických modelů a algoritmů a ukážeme si využití známého problému batohu při implementaci metody zajišťující zničení skladu a jeho případného obsahu.

Ve skladu o určité kapacitě se kromě jídla mohou vyskytovat i předměty, které mají různý objem. Pokud má hráč skladů více, program si nepamatuje, v kterém jaký předmět leží, ale předpokládá se, že jejich využití je víceméně rovnoměrné. Ve chvíli, kdy se jeden z nich z nějakého důvodu zboří, je tedy nutné zjistit, jaké předměty byly zničeny.

Nejdříve se zjistí, jak velký objem zabírají předměty a jídlo ve skladech (něco se tam nemusí vejít, předměty mají ve skladu přednost před jídlem). Výsledný objem ke zničení je přímo úměrný poměru kapacity bořeného skladu a kapacity všech. Získali jsme tedy objem předmětů, který je třeba zničit. Konkrétní zničené předměty dostaneme právě implementací problému batohu, což provedeme metodou dynamického programování. Objem, který zničit nepůjde, se pak doplní příslušným množstvím jídla.

Vstupem algoritmu je celkový objem ke zničení t a množina předmětů S, jež mají nenulový objem (předměty s nulovým objemem se nenacházejí ve skladu) menší nebo roven celkovému ($\forall i \in S : 0 < i.v \leq t$). Výstupem je pak množina předmětů na zničení D a objem, který se nepodařilo naplnit.

```
S = {i ∈ P, c_i > 0 & p_i < g};                    // nabízející hráči
while S ≠ ∅ do
    foreach j ∈ P do a_j = 0;                       // aktuální prodej
    i = i ∈ S & p_i = min(p_j, j ∈ S);      // lib. hráč s min. cenou
    S = S \ {i};
    D = {j ∈ P, c_j < 0 & d_{j,i} & s_{i,j} > 0};  // poptávající se hráči
    while D ≠ ∅ & c_i > 0 do
        B = {j ∈ D, s_{i,j} = min(s_{i,k}, k ∈ D};
                                // hráči se stejnou prioritou dodávek
        while B ≠ ∅ & c_i > 0 do
            q = (c_i div |B|) + 1;              // podíl jednoho hráče
            r = (c_i mod |B|);
            k = 0;
            foreach j ∈ A do                    // rozdělení nabídky
                if k = r then q = q − 1;
                k = k + 1;
                if c_j + q ≥ 0 then         // více než je požadováno
                    a_j = a_j − c_j; // přidání požadovaného k prodeji
                    c_i = c_i + c_j; // ubrání požadovaného od nabídky
                    c_j = 0;               // dál už se nic nepožaduje
                else
                    a_j = a_j + q;            // přidání dílu k prodeji
                    c_i = c_i − q;            // ubrání dílu od nabídky
                    c_j = c_j + q;           // ubrání dílu od poptávky
                end
            end
            B = B \ {j ∈ B, c_j = 0};
        end
        D = D \ B;
    end
    foreach j ∈ P, a_j > 0 do        // zaplacení aktuálnímu hráči
        m_i = m_i + p_i · a_j;          // přičtení peněz prodávajícímu
        m_j = m_j − p_i · a_j;             // odečtení peněz kupujícímu
    end
end
```

Algoritmus 3: Obchodování na trhu se surovinami

Algoritmus používá dvě pole velikosti $t+1$ – v proměnných z_j se uchovává aktuální maximální zaplnění objemu j (inicializováno na 0), v proměnných p_j pořadí posledního přidaného předmětu do objemu j v množině S (inicializováno na -1).

Algoritmus postupně prochází předměty v množině S (pomocí proměnné $i \in S$). Každý předmět zkouší přidat do zaplnění objemu $i.v$ až t od největšího k nejnižšímu (index j), aby mohl pracovat v jednom poli z. Pokud $z_j < z_{j-i.v} + i.v$, pak $p_j = i$ a $z_j = z_{j-i.v} + i.v$. Po projití všech předmětů zrekonstruujeme optimální zaplnění objemu předměty pomocí pole p. Nastavíme j na t a dokud $p_j \neq -1$ (cyklus vždy skončí, protože $p_0 = -1$), přidáme předmět S_{p_j} do D a zmenšíme j o $S_{p_j}.v$. Nakonec budeme mít v D všechny předměty, které se mají zničit, a v j objem, který se nepodařilo zaplnit. Popsaný algoritmus s výše uvedenými proměnnými je přehledně uveden v algoritmu 4.

```
for j = 0 to t do z_j = 0;                  // inicializace zaplnění
for j = 0 to t do p_j = -1;                   // inicializace indexů
foreach i ∈ S do                            // přes všechny předměty
    for j = t downto i.v do                   // přes všechny objemy
        if z_j < z_{j-i.v} + i.v then                 // lepší zaplnění?
            p_j = i;      // uložení posledního použitého předmětu
            z_j = z_{j-i.v} + i.v;          // uložení lepšího zaplnění
        end
    end
end
j = t;
while p_j ≠ -1 do     // generování výsledné množiny předmětů
    D = D ∪ {S_{p_j}};                            // přidání předmětu
    j = j - S_{p_j}.v;
end
```

Algoritmus 4: Vybírání předmětů k zaplnění stanoveného objemu

Kapitola 4

Editor vstupních dat a skriptování

Po úspěšné implementaci všech metod tříd popisujících chování objektů ve hře je nutné nějakým způsobem zadat jejich počáteční vlastnosti, v případě kolekcí pak popsat, které budou obsahovat, což nám umožní vytvořené externí editory. Nejprve se budeme zabývat editorem zabezpečujícím přidávání objektů a úpravu jejich vlastnosti, poté skriptovacím rozhraním a jazykem, díky kterému můžeme měnit jejich chování, a nakonec se zmíníme o dalších možnostech externí úpravy dat.

4.1 Editor vstupních dat

Jak bylo popsáno v kapitole 2.1, každý objekt ve hře umí svá data načíst z proudu dat – tedy i své počáteční vlastnosti, kolekce pak všechny své objekty. Editor je tedy uloží do souborů, z kterých se při spuštění hry načtou.

Abychom nemuseli složitě vytvářet pro každý typ objektu vlastní editor, využijeme možnost reflexe, díky níž budeme moci použít již vytvořené třídy jen s minimálními úpravami. Ty se týkají hlavně odstranění pro editor nepotřebných metod (například *Refresh* či propojení se skriptováním), přidání metod pro kopírování objektů, umožnění přímého zápisu do proměnných přes vlastnosti a úpravy některých složitějších struktur (zejména polí) k jejich snazší editaci.

V editoru si uživatel vybere objekt či kolekci, kterou chce upravovat (například trh s jídlem nebo druhy zvířat), program provede reflexi na příslušnou třídu a její vlastnosti zobrazí ve speciální vizuální komponentě. V pří-

padě kolekce ještě zobrazí seznam obsažených objektů a tlačítka pro jejich přidávání a mazání. Komponenta [2] pro editaci vlastností je podobná těm z vývojových prostředí – na každém řádku je jedna vlastnost a způsob úpravy je přizpůsoben druhu vlastnosti (seznam pro výčty, textové pole se šipkami pro malé číselné hodnoty apod.). Po dokončení editace se všechny objekty uloží do souboru, který je pak distribuován společně se hrou. Kdokoliv tedy může díky tomuto editoru obohatit hru například o další jatka, dárky či předměty, aniž by upravoval či jen znal její zdrojový kód.

V tabulce 4.1 jsou zapsány všechny objekty, které je možné editovat. Jsou rozděleny na objekty patřící každému hráči (zvířata, budovy, ...), které mohou být měněny podle náboženství hráče, a na ty, které jsou společné pro všechny (předměty, jatka, obchody, ...).

Název souboru	Druh objektů	Popis objektů
Soukromé objekty		
Buildings	kolekce	budovy na mapě
Animals	kolekce	zvířata
Family	jeden objekt	konstanty týkající se rodiny
Player	asociativní pole	různé hráčovy konstanty
Společné objekty		
Religions	kolekce	náboženství
Slaughters	kolekce	jatka pro zpracování zvířat
Gifts	kolekce	dárky pro členy rodiny
PlacesToMeetPartner	kolekce	místa pro nalezení partnerky
Groups	jeden objekt	jména a vlastnosti uskupení
Markets	kolekce	trhy se surovinami
FoodMarket	jeden objekt	vlastnosti trhu s jídlem
Economy	jeden objekt	různé ekonomické konstanty
Obstacles	kolekce	překážky na pozemcích
Shops	kolekce	obchody s předměty
Items	kolekce	předměty
GroupAccidents	kolekce	náhody podle obliby u uskupení
Accidents	kolekce	náhody podle náboženství a ročního období
GlobalAccidents	kolekce	náhody ovlivňující všechny hráče
Events	kolekce	zprávy v novinách (příběh)
OnceEvents	kolekce	náhodné zprávy v novinách
Game	asociativní pole	různé společné konstanty

Tabulka 4.1: Editovatelné objekty

Pro účely editoru byly upraveny základní třídy, od kterých ostatní dědí. Místo *TCommonObject* byla vytvořena třída *TEditorObject*, která může (ale nemusí) být zároveň kolekcí. Ta si pamatuje soubor, do kterého bude zapisovat vlastnosti své či svých objektů, a typ svých objektů. Dále zavádí podobně jako *TCommonObject* metody pro práci s proudem dat a navíc implementuje ty pro uložení do souboru a načtení z něj pomocí těchto abstraktních metod

a uloženého jména souboru. Nakonec ještě obsahuje metodu pro vyhledání objektu podle jména a pro přidání objektů ze souboru do kolekce.

Od této třídy dědí *TCollectionObject* implementující metody pro práci s proudem dat ukládáním objektů nacházejících se v kolekci. Ty reprezentuje třída *TNameItem* s textovým identifikátorem. Všechny třídy představující pouze jeden objekt (trh s jídlem, rodina, ...) dědí přímo od *TEditorObject*, třídy reprezentující objekty ukládané do kolekcí (jatka, dárky, obchody, ...) dědí od *TNameItem*.

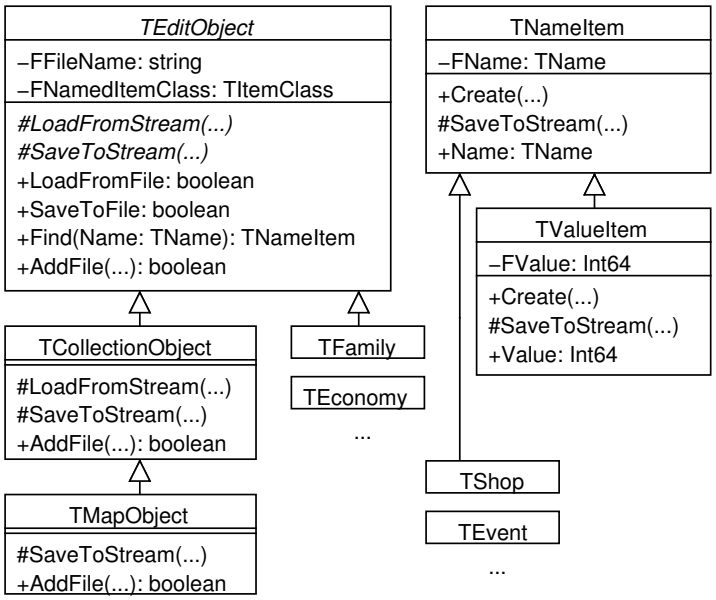
Jiný způsob uložení počátečních vlastností objektů je sdružení do jednoho souboru, kde budou uloženy dvojice identifikátor vlastnosti (řetězec) a hodnota vlastnosti (číslo). To se hodí zejména pro třídy potřebující variabilně inicializovat jen malé množství číselných hodnot, proto byl použit pro nejrůznější hráčovy a herní koeficienty. Soubor se načte na začátku inicializace objektů do asociativního pole a v konstruktorech tříd se jednotlivé vlastnosti naplňují hodnotami indexovanými příslušnými identifikátory.

Tento způsob je v editoru realizován dvěma třídami. Třída *TMapObject* dědí od *TCollectionObject* a upravuje způsob práce s proudem dat (data ukládá jako setříděné asociativní pole), zatímco třída *TValueItem* dědí od *TNameItem* a přidává proměnnou *Value* typu číslo reprezentující hodnotu vlastnosti objektu. Dědičnost základních tříd v editoru je přehledně zobrazena na obrázku 4.1.

4.2 Skriptovací jazyk a rozhraní

Aby bylo možné v rámci reakce na různé události, které se s objekty dějí, upravovat vlastnosti a chování všech objektů ve hře, je nutné vytvořit rozhraní, kterým budou skriptovací jazyk a program komunikovat. V kapitole 2.1 je toto rozhraní zadefinováno přímo v základní třídě všech herních objektů *TCommonObject* a nyní se dostaneme k jeho implementaci.

Jedná se o dvě funkce, z nichž jedna (*GetVariable*) vrací nějakou vlastnost objektu a druhá (*SetVariable*) nějakou modifikuje. Mají některé společné parametry – *Kind* (pole struktur popisující, o jakou vlastnost či metodu se jedná), *Index* (kolikátá struktura je právě aktuální) a *Rights* (stupeň oprávnění ke čtení či zápisu). *SetVariable* má navíc ještě parametry *Value* (hodnota modifikující vybranou vlastnost) a *Special* (výčtový typ popisující způsob změny hodnoty vlastnosti – přiřazení, přičtení nebo přičtení procent).

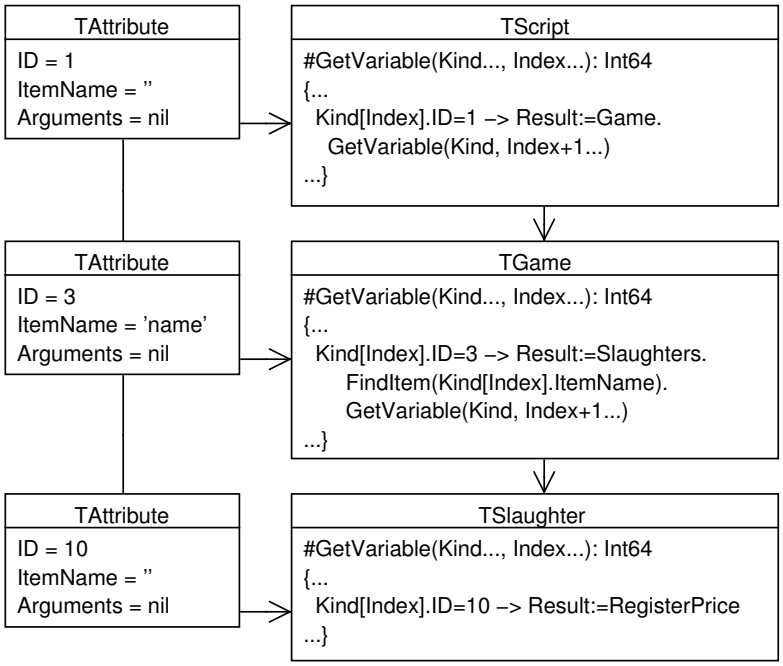


Obrázek 4.1: Hierarchie základních tříd použitých v editoru

Informace o tom, jaký objekt a jaká vlastnost se bude modifikovat, se tedy přenáší pomocí parametru *Kind*, což je pole struktur *TAttribute*, které obsahují *ID* (identifikační číslo vlastnosti či metody), *ItemName* (jméno položky v kolekci) a *Arguments* (pole čísel reprezentujících případné upřesnění vlastnosti či parametry volané metody).

Při zjišťování nebo modifikaci vlastnosti či volání metody nějakého objektu se postupuje hierarchicky. Jestliže například chceme zjistit cenu za registraci u konkrétních jatek, bude v první struktuře *ID* nastaveno na hodnotu, která reprezentuje objekt celé hry, v další bude *ID* reprezentující kolekci jatek a v *ItemName* název jatek a v poslední struktuře bude *ID* reprezentující cenu za registraci (viz obrázek 4.2). Každý objekt, u kterého bude tato metoda zavolána, tedy zjistí podle *ID* aktuální struktury (indexované parametrem *Index)*, jestli se jedná o jeho vlastnost. Pokud ano, zpracuje případné argumenty a vrátí její hodnotu, v případě, že je to odkaz na další objekt, zavolá tuto metodu na odkazovaném objektu se stejnými parametry a zvýšeným *Index* o jedna. Díky proměnné *ItemName* je pak možné přistu-

povat k jednotlivým položkám kolekce podle jména, případně ji lze použít pro přenos textové informace.



Obrázek 4.2: Příklad volání funkce skriptovacího rozhraní *GetVariable* pro zjištění ceny za registraci u konkrétních jatek – vlevo struktury uložené v parametru *Kind*, vpravo hierarchický postup volání funkcí a rozhodování se podle *ID* a *ItemName*

Nadefinovali jsme tedy rozhraní, přes které můžou být měněny objekty ve hře, teď ještě musíme vytvořit jednoduchý způsob, jak toto rozhraní zvenčí volat. Byl naprogramován jednoduchý skriptovací jazyk, který umí volaní skriptovacího rozhraní uživatelsky přívětivým způsobem, podmíněný příkaz, dočasné proměnné, základní aritmetické operace, předefinované konstanty a výstup pomocí textových zpráv. Nebudeme dopodrobna rozebírat jeho fungování, jen se zaměříme na několik důležitých vlastností.

Uživatel by si jistě při psaní skriptu nechtěl pamatovat *ID* jednotlivých objektů, a proto vytvoříme konverzní tabulku indexovanou třídou, na které je funkce zavolána, a jménem vlastnosti či třídy vlastněného objektu. Tato

jména jsou popsána v dokumentaci (viz příloha A). Například volání z obrázku 4.2 zapíšeme `getvar(Game.Slaughters['name'].RegisterPrice)`.

Skripty uživatel zapisuje ve skriptovacím editoru, kde je může přeložit i otestovat. Překladač jazyka provede pouze základní lexikální a syntaktickou analýzu skriptu pro ověření validity a jeho rychlejší spouštění v programu. Skripty jsou rozděleny do skupin podle události, na kterou reagují (například registrace na jatkách či koupení dárku). V editoru popsaném v kapitole 4.1 pak stačí u objektu do příslušné vlastnosti vyplnit jméno skriptu a ten se při vyvolání události ve hře spustí. Pokud se jmenuje stejně jako objekt, není potřeba jméno vyplňovat.

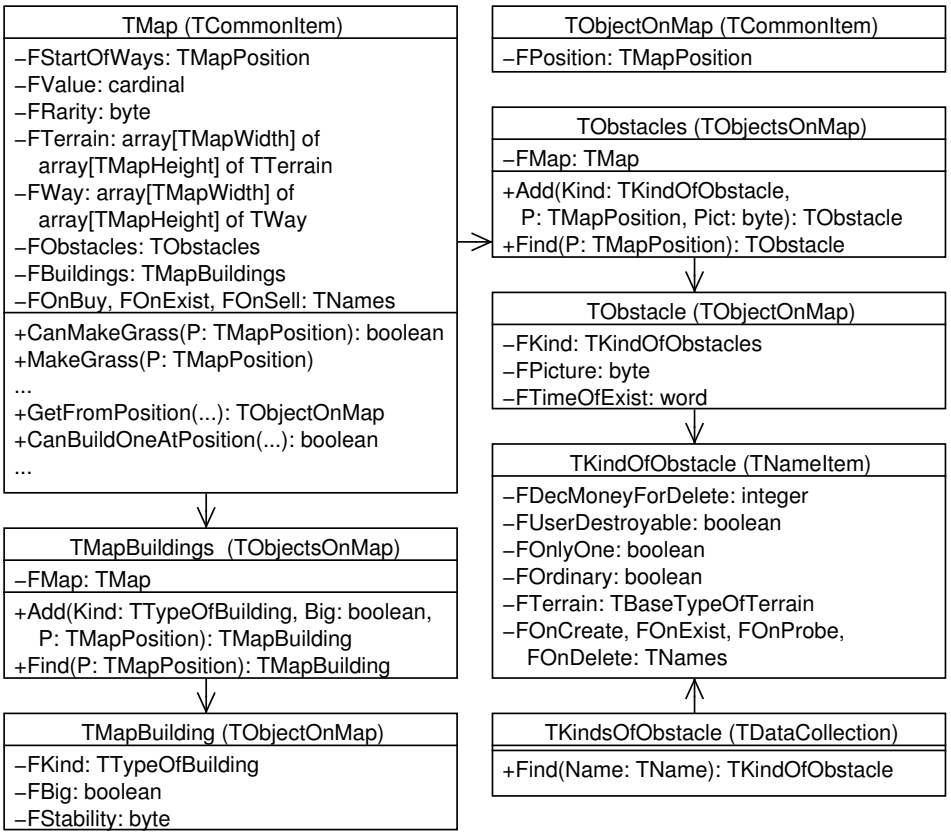
Ze skriptů je možné kromě měnění vlastností objektů také vypisovat zprávy hráči a to buď přímo oknem s jejím zněním a tlačítky pro reakci (viz kapitola 5.1), nebo nepřímo do seznamu sezonních zpráv. V prvním případě se tedy skript může hráče na něco zeptat a podle jeho odpovědi se rozhodnout, co bude dělat dál.

Pro lepší představu se podíváme na příklad skriptu, který se spouští po každém tahu, pokud má hráč na svém pozemku překážku s názvem „houpacky“. V každém ročním období kromě zimy přičte hráči tisíc korun a s pravděpodobností 1:10 sníží oblibu u obce o tři až sedm a vypíše zprávu o úrazu na houpačkách. Každé jaro pak zvýší oblibu u obce o dva.

```
if (getvar(Game.Season)<winter)
{
  setvar(Player.Money.ChangeMoney(kmTerrain), 1000);
  if (range(1, 10)=range(1, 10))
  {
    addvar(Player.GroupsRelation['community'].Favour,
      Range(-7, -3));
    msgseason(loadtext(['Obstacles'], ['houpacky'], []),
      loadtext(['Obstacles - exist'], ['houpacky'], []));
  }
}
if (getvar(Game.Season)=spring)
{
  addvar(Player.GroupsRelation['community'].Favour, 2);
}
```

4.3 Další způsoby editace dat

Některé druhy dat by se v editoru popsaném v kapitole 4.1 upravovaly jen velmi těžko. Příkladem mohou být pozemky, na kterých staví hráč své budovy. Aby byla tvorba těchto pozemků co nejjednodušší, je třeba vytvořit speciální editor, ve kterém půjde jednoduše upravovat terén, stavět překážky a budovy a zadávat skripty.



Obrázek 4.3: Třídy editoru map s jejich vlastnostmi a metodami

Na obrázku 4.3 jsou uvedeny všechny třídy, které používá editor map i program pro pozemky a objekty na nich. Terén a cesty jsou reprezentovány dvoudimenzionálním polem struktur, v nichž je obsažen druh terénu nebo cesty (výčtový typ) a číslo obrázku. Naopak budovy (*TMapBuildings*) a překážky (*TObstacles*) jsou reprezentovány jako objekty s vlastností *Po-*

sition zděděnou od třídy *TObjectOnMap*. Důvody rozdílu jsou dva – budovy a překážky se na pozemcích vyskytují méně než terén a cesty a také mají další vlastnosti, metody a odkazy na objekty představující jejich druh (*TKindOfObstacle*).

Třída reprezentující pozemek dále obsahuje metody pro úpravu terénu a cest, identifikaci objektu či možnost stavby na konkrétním políčku apod. Také obsahuje skripty reagující na koupi, existenci a prodej pozemku. Nakonec jsou zde proměnné, v nichž je uložena hodnota a rarita pozemku a umístění políčka napojující pozemek na hlavní cestu.

Všechny tyto vlastnosti a metody jsou přístupné z editoru pozemků přes rozbalovací menu, které se aktivuje kliknutím pravého tlačítka myši na políčko pozemku a jehož obsah je upraven podle toho, co se na něm nachází. Každá položka menu má také svoji klávesovou zkratku, což velmi zrychluje úpravu pozemku. Užitečným vylepšením je seznam naposledy postavených překážek, který je taktéž přístupný pomocí klávesových zkratek.

Úplně jiným způsobem se editují texty zobrazované hráči. Jelikož je žádoucí, aby se dala hra snadno přeložit do dalších jazyků, všechny texty jsou uloženy v jednom souboru, z kterého se v případě potřeby získávají. Strukturou je to *ini* soubor, tedy obsahuje pojmenované sekce, v nichž se nachází seznamy „název hodnoty=hodnota“. Z programu či ze skriptů lze k textům přistupovat zadáním názvu sekce a hodnoty. Navíc v každém textu mohou být speciální znaky, za které se pak může vložit další text ($), číslo (#) nebo množství peněz (&). Výhoda umístění textů do jednoho souboru spočívá v jejich snadné přeložitelnosti, nevýhodou je nutnost úpravy tohoto souboru při přidání jakéhokoli objektu nebo skriptu.

Funkce starající se o načítání textu se jmenuje *LoadText* a jejími parametry jsou pole řetězců sekcí (*Section*) a hodnot (*Value*) a pole čísel *Int*. Nejdříve načte text z první sekce a první hodnoty. Pokud se v něm nacházejí nějaké znaky $, pak se postupně nahradí texty podle dalších sekcí a hodnot. Jestliže je pole sekcí větší než pole hodnot, dává se místo těchto znaků obsah řetězce v poli sekcí (vhodné například pro jména dětí apod.). Znaky # a & se zase postupně nahrazují čísly z pole *Int* a příslušně se zformátují (jako číslo nebo jako počet peněz).

Uvedená funkce je ze třídy *TMessages*, kde se nacházejí další metody pro práci s textem. Jedna z nich například dává náhodné jméno a příjmení podle pohlaví ze seznamu jmen, který je setříděn podle četnosti jmen v populaci [3], a funkce je implementována tak, aby pravděpodobnost vydaného jména přibližně této četnosti odpovídala.

Kapitola 5

Grafický interface a uživatelské prostředí

V předchozích kapitolách jsme postupně vytvořili celé zázemí hry. Nyní je třeba zprostředkovat možnosti hry hráčům skrze uživatelské prostředí. To postavíme na rozhraní *DirectX*, s nímž budeme jednodušeji pracovat přes knihovnu *DelphiX* [1], pomocí níž vytvoříme vlastní vizuální komponenty. Nejdříve se budeme zabývat jejich společnými vlastnostmi a chováním, výsledkem čehož budou základní třídy, od kterých budou dědit. Poté se zaměříme na implementaci konkrétní vizuální komponenty používané v programu a nakonec promyslíme, jakým způsobem sestavit z jednotlivých komponent celé uživatelské prostředí.

5.1 Základní třídy pro grafické komponenty

Každá vizuální komponenta se především musí umět nakreslit, k čemuž slouží metoda *Paint*, která bude v základní třídě abstraktní. Ta musí vědět, kam ji má nakreslit, a proto komponenta musí mít vlastnosti *Left*, *Top*, *Width* a *Height* udávající její posun od levého a horního okraje a její šířku a výšku. Programátor by také měl mít možnost rozhodnout, že chce komponentu dočasně skrýt, a proto zavedeme vlastnost *Visible*. Skoro každá komponenta má na sobě nějaký nápis (*Text*), který je vykreslen konkrétním písmem (*Font*). Také má na pozadí barvu (*Color*) nebo nějaký obrázek (*Images*).

Tím se dostáváme k interakci komponenty s uživatelem. Ta by měla vědět, jestli nad ní je myš nebo jestli se na ni právě kliká, aby podle toho

mohla změnit svůj vzhled (*State* a příslušné obrázky v poli *Images*) a mohl na to zareagovat i programátor, který ji používá (události *OnClick*, *OnMouseEnter*, *OnMouseLeave* a *OnMouseMove*). Ten by také měl určit, zda má komponenta tyto události vyvolávat, což zajistíme vlastnosti *Enabled*. Aby se uživatel dozvěděl více o funkci konkrétního ovládacího prvku, který bude komponenta reprezentovat, přidáme ještě textovou vlastnost *Hint* a vlastnost *ShowHint* rozhodující, zdali se má zapsaná nápověda zobrazovat (více v kapitole 5.3). Nakonec ještě přidáme možnost měnit kurzor myši při najetí nad komponentu (*Cursor*).

Nyní máme sepsány společné vlastnosti vizuálních komponent starající se jen o svoje vykreslení a reakce na události myši. Vytvoříme z nich abstraktní třídu *TDxComponent* a začneme se zabývat tím, jak je sdružovat a jak by mohly reagovat na události klávesnice.

Vytvoříme třídu *TDxContainer*, která bude dědit od *TDxComponent*, a dále ji budeme nazývat kontejner. Aby mohl kontejner obsahovat komponenty, zavedeme členskou proměnnou *FCompList*, kde na ně budou uložené ukazatele. Dále vytvoříme metody pro jejich přidávání a odebírání *InsertComp* a *RemoveComp*. Aby se všechny mohly vykreslovat samy, musejí si ze svých souřadnic vůči rodičovskému kontejneru spočítat souřadnice vůči celé obrazovce, a proto musí mít na svůj rodičovský kontejner ukazatel. Do třídy *TDxComponent* tedy ještě přidáme vlastnost *Parent* a metody na přepočet souřadnic *ClientToScreen* a *ScreenToClient*, které volají samy sebe na rodičovském kontejneru, dokud nějaký existuje. Ve třídě *TDxContainer* pak implementujeme metodu *Paint* tak, že postupně zavolá stejnou metodu na všech komponentách v *FCompList*. Dále je třeba předat události myši ke správné komponentě, k čemuž slouží metoda *PointToComponent* zjišťující obsah zadaných souřadnic.

Je důležité zmínit, že seznam *FCompList* slouží pouze k zaznamenání hierarchie pro vykreslení komponent. O správu instancí jejich tříd se stará již třída *TComponent*, která je předkem *TDxComponent* a která je součástí základních knihoven vývojového prostředí. V konstruktoru vizuální komponenty tedy stačí uvést jejího vlastníka (*Owner*), který při svém zániku všechny takto vytvořené instance korektně odstraní. Ve většině případů to bude stejná komponenta jako ta starající se o vykreslování.

Nyní se dostáváme k reakci komponent na události klávesnice. Je zřejmé, že v danou chvíli na ně bude reagovat právě jedna. O ní říkáme, že má fokus, což bude uložené ve vlastnosti *HasFocus*. Programátor může nastavit, aby fokus nepřijímala (vlastnost *GetFocus*) nebo ho ihned dostala (metoda

SetFocus). Důležitou informací pro kontejner je, která z jeho komponent má právě fokus, a proto přidáme vlastnost *FocusedComp*.

Jelikož je zvykem v různých uživatelských prostředích, že klávesou TAB se přesouvá fokus na další komponentu, zavedeme seznam ukazatelů na komponenty, po kterých se bude v rámci kontejneru přeskakovat (*TabList*) a umožníme programátorovi komponentu do seznamu zařadit (*TabStop*) a určit pořadí (*TabOrder*). Jelikož se klávesou *Tab* skáče nejen po aktuálním kontejneru, ale i sousedních, vytvoříme metodu *FindNextComp*, která bude mít na starost správnou komponentu v celé hierarchii najít a bude počítat i s vlastnostmi *TabStop*, *Enabled* a *Visible*.

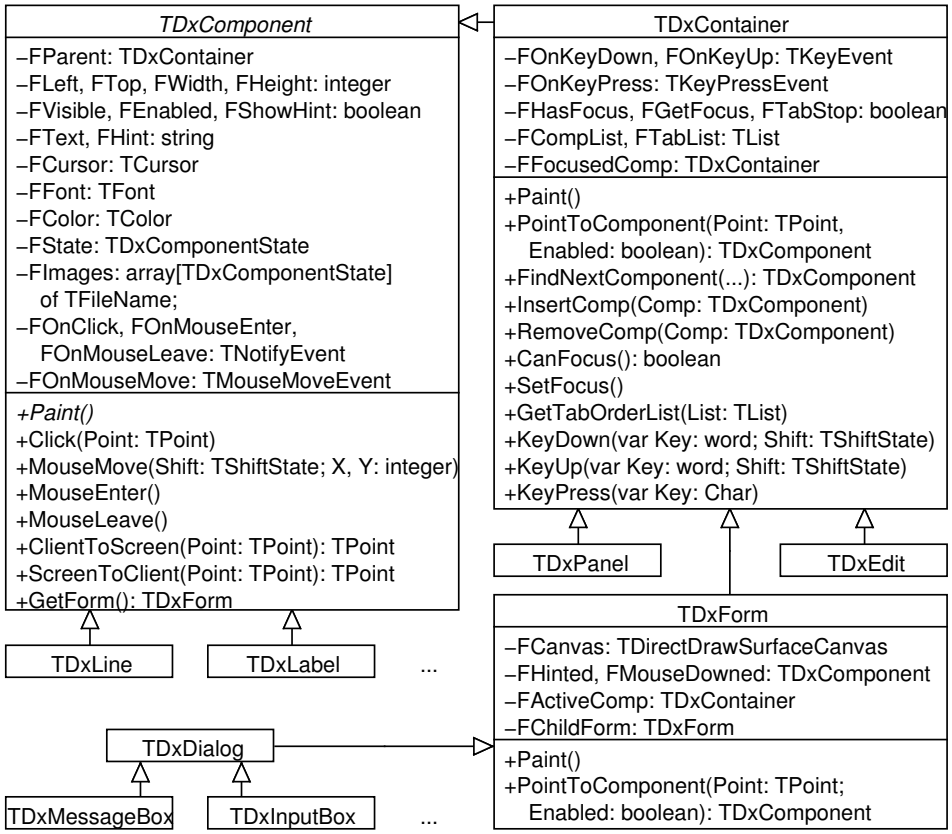
Nakonec by měl programátor mít možnost na události klávesnice reagovat. Události *OnKeyDown* a *OnKeyUp* reagují na stlačení a puštění konkrétních kláves včetně kombinace s *Shift*, *Ctrl* apod., kdežto *OnKeyPress* přináší již konkrétní znakovou interpretaci.

Díky kontejnerům teď již můžeme vytvářet libovolně složitou několikaúrovňovou hierarchii vizuálních komponent, ale abychom ji mohli z programu rozumně využívat, musíme určit jeden kontejner, který bude obsahovat všechny ostatní a na který budeme moci volat metodu *Paint* pro vykreslení všech komponent. Jelikož budeme od tohoto kontejneru požadovat ještě další služby, vytvoříme pro něj speciální třídu *TDxForm* poděděnou od *TDxContainer* a nazveme jej formulář.

Především chceme udržovat ukazatel na plátno (*Canvas*), na nějž se budou všechny komponenty vykreslovat. Každá komponenta si v metodě *Paint* zjistí, v jakém je formuláři, pomocí přidané metody *GetForm* (ta stoupá v hierarchii přes vlastnost *Parent*, dokud nenarazí na formulář) a dostane se k plátnu, na které se pomocí jeho metod vykreslí. Dále si formulář udržuje ukazatel na komponentu, pro kterou by se měla zobrazit nápověda (*Hinted*), tu, nad kterou bylo stlačeno a ještě nepuštěno tlačítko myši (*MouseDowned*), a kontejner mající právě fokus (*ActiveComp*). Nakonec obsahuje ukazatel na případný vlastněný formulář *ChildForm*, jemuž přeposílá události myši a klávesnice a který vykresluje jako poslední nad všemi svými komponentami, což umožňuje vytváření dialogů (*TDxDialog*). Formulář tedy nereaguje, dokud není dialog skryt.

Všechny vytvořené třídy jsou včetně hierarchie přehledně zobrazeny na obrázku 5.1. Navíc jsou tam uvedeny příklady konkrétních komponent, které od nich dědí.

Na závěr je třeba vyřešit správu obrázků, které budou chtít různé komponenty vykreslovat. Každá si ve vlastnosti *Images* drží názvy souborů, v nichž



Obrázek 5.1: Hierarchie základních tříd vizuálních komponent

se obrázky nacházejí. Bylo by nepraktické mít je všechny uložené přímo v ní, jelikož jsou často sdíleny a zabíraly by zbytečně místo v paměti. Na druhou stranu by bylo nesmírně pomalé nahrávat je ze souborů pokaždé, když je volána metoda *Paint*. Proto zvolíme kompromis v podobě kolekce s obrázky *TDxPictureCollection* indexované podle relativní cesty k nim, do které se obrázek při prvním použití nahraje ze souboru a z které se pak při každém dalším použití rychle vezme. Jelikož se v různých částech uživatelského prostředí (hlavní menu, vlastní hra, ...) používají různé obrázky, zavolá se při přechodu mezi nimi metoda této kolekce, která dočasně uložené obrázky odstraní.

5.2 Příklad grafické komponenty

Vytváření konkrétních grafických komponent si ukážeme na příkladu seznamu položek *TDxListBox*. Jelikož chceme, aby seznam reagoval na události klávesnice (přesun mezi položkami pomocí šipek apod.) a tedy mohl mít fokus, bude jeho třída dědit od třídy *TDxCustomButton*, která je přímým následníkem *TDxContainer* a jen lehce upravuje její chování (především je pro ni standardní mít zapnuté vlastnosti *TabStop* a *GetFocus*).

Třída seznamu rozšíří implementaci některých zděděných metod a vlastností a přidá další. Především musí mít uloženy samotné názvy položek (vlastnost *Items* typu kolekce řetězců, která navíc ke každému řetězci umí přiřadit objekt) a pořadí té, která je právě označena (*ItemIndex*). Pro vykreslení seznamu nás zajímá, jak budou jednotlivé položky vysoké (*ItemHeight*) a která má být zobrazena jako první v případě, že všechny nelze ukázat najednou (*TopIndex*). S tím souvisí také možnost posunutí na další nebo předchozí položky. V klasickém seznamu se používá posuvník, nám ale budou stačit dvě tlačítka umístěná nad a pod ním (*PageUpButton* a *PageDownButton*) zvyšující či snižující *TopIndex* o počet položek najednou zobrazitelných. Také budeme chtít vědět, zda jsou setříděné podle abecedy (*Sorted*), a případnou změnou vlastnosti je setřídit. Nakonec ještě přidáme odkaz na obrázky, které se budou zobrazovat pod nápisem vybrané položky.

Po přidání všech důležitých vlastností musíme vytvořit pomocné metody zjišťující počet položek najednou zobrazitelných v seznamu (*ItemPerPage*) a index konkrétně vybrané kliknutím (*ItemAtPos*). Poté již můžeme rozšiřovat implementaci zděděných metod. Nejdříve se zaměříme na konstruktor třídy. Zde nastavíme standardní rozměry seznamu a jména obrázků, inicializujeme seznam položek a vytvoříme tlačítka *PageUpButton* a *PageDownButton*, jejichž vlastnosti *Owner* a *Parent* nastavíme na vytvářený objekt, který se díky tomu postará o volání jejich metody *Paint* a o korektní zničení.

Poté implementujeme metodu *Paint* starající se o vykreslení seznamu. Zde si díky zavolání metody *ClientToScreen* a propojení s formulářem zjistíme, kde přesně se seznam na obrazovce nachází a na jaké plátno ho máme vykreslovat. Pak se pokusíme nahrát obrázky, které budou pozadím textů položek (vybrané a těch ostatních), přes kolekci *TDxPictureCollection*. Další postup se pak řídí tím, jestli jsou k dispozici nebo nikoliv. Nastavíme obdélník, do nějž se umístí první položka, pod horní tlačítko a postupně se budeme v cyklu posouvat přes všechny zobrazitelné položky (počet zjistíme díky metodě *ItemPerPage*). Nejdřív dáme na pozadí obrázek nebo barvu

s rámečkem a poté umístíme případný text. Na konci metody zavoláme metodu předka, jež se postará o vykreslení tlačítek.

Nyní upravíme metody reagující na kliknutí myši a na stisknutí klávesy. První z nich pomocí metody *ItemAtPos* spočítá, na kterou položku se kliklo, a označí ji. Druhá posouvá označenou položku podle stisknuté klávesy – *Home* na první, *End* na poslední, *Page Up* a *Page Down* fungují stejně jako tlačítka a šipky posouvají o jednu nahoru či dolu. Nakonec ještě nastavíme reakce na změnu velikosti seznamu a výšky položky tak, že se příslušně upraví umístění a velikost tlačítek. Tím jsme naprogramovali vše potřebné pro použití této komponenty v uživatelském prostředí.

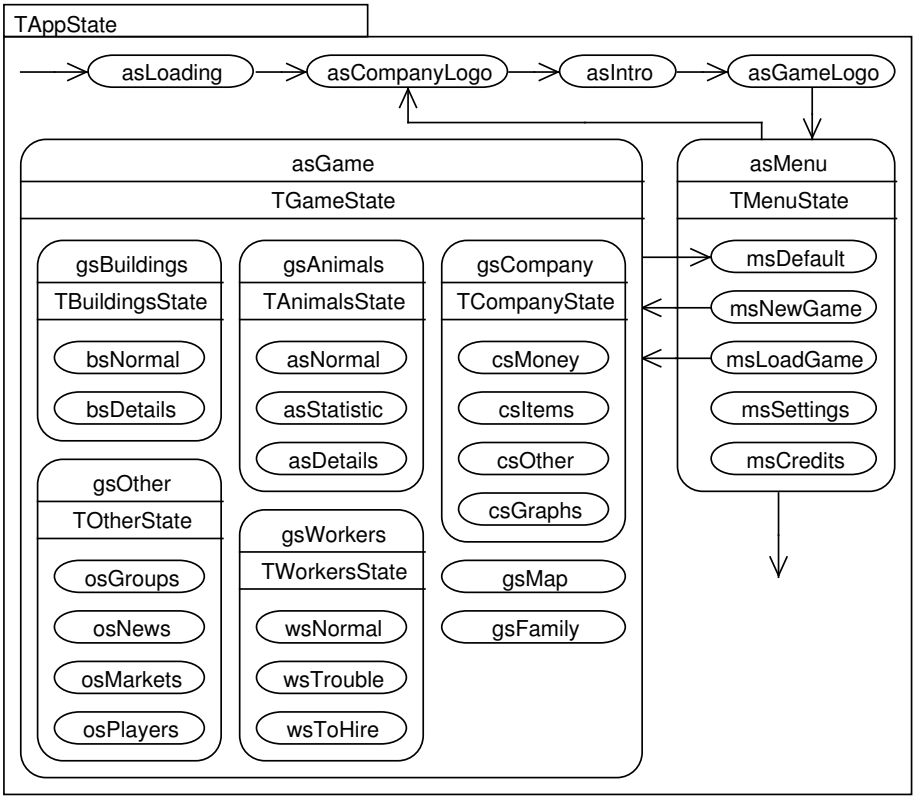
5.3 Skládání uživatelského prostředí

Vytvořili jsme všechny potřebné grafické komponenty a v této kapitole z nich budeme skládat uživatelské prostředí. Jelikož se hra sestává z různých oblastí, které musí hráč řešit (zvířata, rodina, peníze, ...), rozdělíme podle nich i celé její ovládání. V levé části obrazovky se budou vyskytovat ovládací prvky řešící konkrétní vybranou oblast a vpravo se bude nacházet menu, pomocí kterého bude moci hráč mezi jednotlivými oblastmi přepínat. Některé oblasti jsou rozsáhlejší a vizuální prvky, přes které by se daly ovládat, by se na obrazovku nevešly, proto je rozdělíme na podoblasti, mezi kterými se bude přepínat tlačítky umístěnými nahoře.

Aby program věděl, v které oblasti případně podoblasti se hráč právě nachází, a aby je mohl jednoduše měnit, vytvoříme proměnné udržující jejich aktuální stav přístupné přes vlastnosti hlavního formuláře programu. V metodě obsluhující změnu stavu se vždy odstraní všechny ovládací prvky z levé části obrazovky a vytvoří se ty obsluhující oblast odpovídající novému stavu.

Kromě samotné hry ještě musíme udělat hlavní menu, které se zobrazí po startu hry a z kterého hráč spustí novou hru, případně nahraje starou. To je další stav, v kterém se aplikace může nacházet. Tento i ostatní jsou zobrazeny na obrázku 5.2, kde jsou také naznačeny přechody mezi hlavními stavy a vstupní a výstupní stav aplikace.

Jelikož se některé vizuální prvky (texty, tlačítka, seznamy, ...) vytváří velmi často, usnadníme jejich tvorbu funkcemi, jež budou mít za parametry vlastnosti, které jsou mezi instancemi různé (poloha, velikost, text, ...), a jež vrátí požadovaný prvek s ostatními vlastnostmi nastavenými standardně. Výjimkou je vlastnost *Font* určující písmo, jímž bude napsán případný text



Obrázek 5.2: Diagram stavů uživatelského prostředí včetně přechodu mezi hlavními stavy

v prvku, která se nastaví podle globální proměnné *ActualFont*, jež se může kdekoliv změnit. Je to z toho důvodu, aby se jedním nastavením písma mohlo vyrobit více prvků bez nutnosti stále jeho instanci předávat.

Vizuální prvky je potřeba nejen vytvořit, ale také naplnit příslušnými aktuálními hodnotami získanými z objektového modelu. To zajistí metoda *ActualizeLeftScreen* vyhledávající je podle jména a dodávající do nich odpovídající obsah. Některé prvky (tlačítka, seznamy, ...) musí reagovat na operace prováděné uživatelem a vykonat podle toho nějakou akci – změnit uživatelské prostředí či zavolat metody objektového modelu. Toho docílíme přiřazením vytvořených procedur k událostem (nejčastěji *OnClick*) vizuálních komponent.

Aby celé prostředí správně fungovalo, musí se ještě naprogramovat ně-

kolik důležitých metod. Jak víme z kapitoly 5.1, pro jeho vykreslení stačí zavolat metodu *Paint* na instanci třídy *TDxForm*. Ta by se měla spouštět dostatečně často, aby uživatel nedostal pocit, že hra nereaguje. Vložíme proto její volání do metody časovače, která se bude vykonávat ve velmi malých časových intervalech. V ní se také zajistí, že kurzor myši bude vypadat tak, jak popisuje vlastnost *Cursor* komponenty, nad kterou se myš právě nalézá, a že se zobrazí nápověda nacházející se v její vlastnosti *Hint*. Ta bude umístěna v dolní části obrazovky (obdoba stavového řádku) a významně přispěje k orientaci hráče v prostředí hry.

Dále se musíme postarat o propagaci událostí myši a klávesnice ke správným vizuálním komponentám. K tomu nám pomohou již zmíněné vlastnosti třídy *TDxForm* (*Hinted*, *MouseDowned* a *ActiveComp*), z nichž se dozvíme cílovou komponentu. Nakonec napíšeme metody řešící inicializaci a finalizaci celé aplikace a metodu, která centrálně ošetřuje jinde neošetřené výjimky upozorněním uživatele, zapsáním chyby do logu a korektním ukončením aplikace.

Závěr

V práci byla na konkrétním příkladě podrobně popsána tvorba ekonomické tahové strategie od počátečního vytvoření pravidel hry až k realizaci uživatelského prostředí, díky němuž může hráč v rámci těchto pravidel hru ovládat. Nejdůležitějším přínosem práce je právě komplexnost řešeného problému. Přestože mohou být konkrétní implementace dále nevyužity, celý koncept práce může být vodítkem při přípravě vlastního podobného projektu.

Některá konkrétní řešení přeci jen stojí za vyzdvihnutí. Zmiňme především vytváření vlastních grafických komponent a jejich použití při tvorbě uživatelského prostředí popsané v kapitole 5, které musí řešit každý, kdo chce vytvořit originální interakci s uživatelem. Také různé pohledy na editaci vlastností a chování herních objektů zvenčí zmíněné v kapitole 4 umožňující hráčům jednoduše hru upravovat a vylepšovat mohou být pro někoho inspirací.

Na vývoji této hry se pracovalo před napsáním této práce a bude se v něm pokračovat i nadále. Především vytvořený skriptovací jazyk je nutné vylepšit tak, aby v něm bylo možno jednodušeji realizovat složitější chování objektů, nebo ho nahradit již hotovým řešením. S tím souvisí možnost napsání skriptu, který by reagoval na herní situace podobně jako hráči, tedy vytvoření počítačového protivníka. K tomu by bylo třeba vytvořit podpůrné algoritmy, například pro optimální stavění budov na pozemku nebo najímání zaměstnanců.

Jiným způsobem jak obohatit hru více hráčů je možnost hry po síti. Každý program by se staral o objekty svého hráče a úpravu globálních by měl na starosti jeden z nich. Vždy po konci tahu hráče by se stav hry přenesl dalšímu v pořadí. Dalo by se uvažovat i o paralelním odehrávání tahů, což by ale znamenalo zabývat se synchronizací společných objektů.

Hra se také dá obohatit o nová pravidla. Příkladem může být společná mapa pro všechny hráče, na které by kupovali pozemky a kde by viděli

postup ostatních, s čímž souvisí naprogramování generátoru pozemků. Také se může lépe využít potenciál domácích a zahraničních uskupení, která by mohla zadávat hráčům nejrůznější úkoly.

Verze hry předkládaná na CD ještě není finální, jelikož zbývá zapracovat na neprogramátorských částech vývoje, jako je příběh a další objekty a interakce vytvořené prostřednictvím editorů, jež mohou díky nim vytvářet i programování neznalí uživatelé.

Díky širokým možnostem rozšíření je hra stále otevřeným projektem pro realizaci nových nápadů.

Literatura

[1] DelphiX verze 2000.07.17 pro Delphi 5 – Hiroyuki Hori, © 1999 (http://www.yks.ne.jp/~hori/DelphiX-e.html)

[2] Rak Object Inspector pro Delphi 5 – Richard Kaspar, © 2000 (http://www.kasparsoft.de/objectinspector.htm)

[3] Tabulky četnosti jmen a příjmení – Ministerstvo vnitra České republiky (http://www.mvcr.cz/clanek/cetnost-jmen-a-prijmeni.aspx)

[4] Instalátor Inno Setup verze 5.2.3 – Jordan Russell, © 1997-2008 (http://www.innosetup.com)

Seznam obrázků

2.1 Hierarchie základních tříd programu 14
2.2 Ukázka propojení herních objektů 17

3.1 Závislost relativní ceny na relativním přebytku 19
3.2 Závislost relativní ceny na relativním nedostatku 22
3.3 Integrální závislost relativní ceny na relativním přebytku . . 23

4.1 Hierarchie základních tříd editoru 32
4.2 Příklad volání funkce skriptovacího rozhraní 33
4.3 Třídy editoru map 35

5.1 Hierarchie základních tříd vizuálních komponent 40
5.2 Diagram stavů uživatelského prostředí 43

Příloha A

Obsah přiloženého CD, kompilace a instalace

Součástí práce je přiložené CD, na němž je možno nalézt tuto práci, zdrojové kódy všech programů projektu včetně použitých nestandardních komponent, instalační balíček hry, zkompilované editory a uživatelské dokumentace ke všem programům. V tabulce A.1 se nachází popis obsahu konkrétních adresářů, který naleznete i v souboru `content.txt` v kořenovém adresáři CD.

Adresář	Obsah
`Binary`	zkompilované editory
`Components`	zdrojové kódy nestandardních komponent
`Install`	soubor s instalací hry
`Source`	zdrojové kódy hry a editorů
`Text`	tato práce a uživatelské dokumentace

Tabulka A.1: Obsah přiloženého CD podle adresářů

Ke kompilaci přiložených zdrojových kódů je nutné mít nainstalované vývojové prostředí *Delphi 5* od firmy Borland s doinstalovanými nestandardními komponenty. Pro instalaci komponent rozhraní *DelphiX* [1] spusťte soubor `Components\DelphiX\Bin\Install_for5.exe` a stiskněte tlačítko *Start*. Pro instalaci komponent *Rak* [2] využívaných v editoru vstupních dat otevřete soubor `Components\Rak\RakSrcD5.dpk` a klikněte na tlačítko *Install*, stejně postupujte se souborem `Components\Rak\RakVclD5.dpk`. Kompilaci jednotlivých projektů provedete otevřením souboru s příponou `dpr`

v příslušném adresáři a vybráním možností *Compile All Projects* a *Build All Projects* v menu *Project*.

Pro instalaci hry spusťte soubor `Install\Setup.exe` a řiďte se pokyny instalátoru. Ten byl vytvořen pomocí programu *Inno Setup* [4], vstupní skript pro tento program je uložen v souboru `Install\Setup.iss`. Hru spustíte pomocí zástupců nebo přímo přes soubor `Farma.exe` nacházející se v adresáři vybraném při instalaci. Kompletní popis ovládání programu naleznete v souboru `Text\Uživatelská dokumentace.pdf`. Pro odinstalaci hry spusťte soubor `unins000.exe` nacházející se v instalačním adresáři nebo příslušného zástupce.

K instalaci a spuštění aplikace je potřeba operační systém Windows 98 nebo vyšší s nainstalovaným rozhraním *DirectX* verze 7 nebo vyšší. Minimální hardwarové nároky jsou na úrovni počítače s 1 GHz procesorem, 128 MB operační pamětí a monitorem zvládajícím rozlišení 800×600. Celá hra kromě editorů zabírá do 7 MB místa na disku.