Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Digitální zpracování obrazu

Vladislav Purchard

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Katedra informačního inženýrství

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Vladislav Purchard

Informatika

Název práce

Digitální zpracování obrazu

Název anglicky

Digital Image Processing

Cíle práce

Cílem práce je analyzovat a vybrat vhodné metody zpracování digitálního obrazu z oblasti rekonstrukce, zvýraznění i analýzy obrazu, které budou poté využity pro návrh aplikace ve formě počítačové hry. Navržená aplikace bude využívat metody digitálního zpracování obrazu pro vytvoření hrací plochy. Aplikace bude realizována pomocí programovacího jazyka C#. Jako podpůrný software bude využíván program ZODOP.

Metodika

Teoretická část práce se bude zabývat studiem a analýzou problematiky digitálního zpracování obrazu a s tím spojených metod, které by mohly být využity pro návrh a realizaci praktické části práce. Na základě analýzy bude proveden výběr vhodných metod a volba komplexnosti vstupních dat (obrazů). Praktická část práce se bude zabývat nejdříve formálním návrhem aplikace v UML, poté jeho realizací v jazyce C# a zhodnocením dosažených cílů.

Doporučený rozsah práce

60-80 stran

Klíčová slova

digitální zpracování obrazu, klasifikace, analýza obrazu, návrh aplikace v UML, C#, ZODOP, hra

---

Doporučené zdroje informací

ARLOW, Jim a Ila NEUSTADT. UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky. Vyd. 1. Překlad Bogdan Kiszka. Brno: Computer Press, 2007, 567 s. ISBN 978-80-251-1503-9.

Department of Image Processing [online]. Dostupné z: http://zoi _ zmije.utia.cas.cz/

GLYNN, Jay, Karli WATSON, Morgan SKINNER, ROBINSON, Christian NAGEL, K.Scott ALLEN, Ollie CORNES, Zach GREENVOSS a Burton HARVEY. C#: Programujeme profesionálně. 1. vyd. Brno: Computer Press, 2003, xxx, 1130 s. Programujeme profesionálně. ISBN 80-251-0085-5.

KLIMEŠOVÁ, Dana. Geografické informační systémy a zpracování obrazu. druhé, 2. dotisk. Praha: Česká zemědělská univerzita v Praze, 2008.

PRATT, William K.Digital image processing. 4th ed., Newly updated and rev. ed. Hoboken, N.J.: Wiley-Interscience, c2007, xix, 782 p., [4] p. of plates. ISBN 978-047-1767-770.

Předběžný termín obhajoby

2015/02 (únor)

Vedoucí práce

doc. RNDr. Dana Klimešová, CSc.

Elektronicky schváleno dne 30. 10. 2013

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 5. 12. 2013

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 25. 03. 2015

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Digitální zpracování obrazu" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne _____

Děkuji doc. RNDr. Daně Klimešové, CSc. Za cené rady, konzultace a trpělivost při zpracování diplomové práce.

Digitální zpracování obrazu

Souhrn:

Diplomová práce se zabývá problematikou zpracování digitálního obrazu a jejím praktickým použitím. V teoretické části je popsáno, jak je obraz reprezentován. Dále jsou vysvětleny jednotlivé fáze procesu zpracování digitálního obrazu a to konkrétně rekonstrukce, zvýraznění a analýza obrazu. V každé fázi jsou popsány příslušné metody a jejich použití.

Praktická část práce se zaměřuje na tvorbu aplikace, která využívá vybrané metody procesu zpracování obrazu pro tvorbu herní plochy ve formě bludiště. Nejdříve je navržena struktura aplikace s použitím jednotného modelovacího jazyka (UML). Poté je aplikace realizována v jazyce C# v prostředí Visual Studio 2010. Na konci praktické části jsou popsány funkce aplikace a její použití. Závěrem jsou zhodnoceny dosažené výsledky diplomové práce.

Klíčová slova:

Digitální obraz, reprezentace obrazu, Fourierova transformace, rekonstrukce obrazu, zvýraznění obrazu, analýza obrazu, bludiště, vývoj aplikace, C#, jednotný modelovací jazyk, počítačová grafika.

Digital Image Processing

Summary:

This thesis deals with problematics of the digital image processing and its practical use. Theoretical part describes how an image is represented. Next there are explained phases of digital image processing, namely image reconstruction, enhancement and analysis. In each phase, there are explained its methods and applications.

The practical part is focused on development of the application, which uses some methods of digital image processing for creating game area in form of maze. First, structure of the application is designed with use of Unified Modelling Language (UML). Then is the application implemented in C# using Visual Studio 2010. Final part describes functions of the application and its usage. Finally, results of the thesis are evaluated.

Key words:

Digital image, image representation, Fourier transformation, image reconstruction, image enhancement, image analysis, maze, application development, C #, Unified Modeling Language, computer graphics.

Obsah

1. Úvod

Digitální zpracování obrazu se díky stále většímu výkonu výpočetní techniky a její dostupnosti používá v mnoho oborech. Pomocí počítače lze snadno zlepšovat kvalitu obrazu, získávat klíčové prvky ze snímku a provádět analýzy informací získaných informací. Využití těchto technik lze nález v mikroskopii (například zkoumání genetických informací), lékařství a v neposlední řadě v dálkové průzkumu země. Metod hojně využívají geografické informační systémy (GIS), kde lze pomocí vstupního obrazu modelovat situace, jako jsou záplavy a dopravní zácpy. Novým využitím těchto metod je automatické vyhodnocování obrazu. Můžeme například číst poznávací značky a reagovat na informace získaných z nich. Satelity mohou automaticky informovat o velkých požárech, testech jaderných zbraní a dalších kritický situací pozorovatelných z orbity země. Automatické vyhodnocování obrazu ale také může přinést různá nebezpečí, a to hlavně omezení soukromí člověka. Je velice pravděpodobné, že v budoucnu bude možno v reálném čase rozeznávat například lidské tváře, což v kombinaci se stále rostoucí kamerovou sítí umožní velice jednoduché sledování pohybu obyvatelstva.

Toto téma jsem si zvolil, protože mě zajímalo, jak jsou v aplikacích implementovány filtry, jak se zjišťují hrany v obrazu a jak se provádí analýza obrazu. Zároveň jsem pomocí těchto metod chtěl vytvořit aplikaci, na které bych demonstroval jejich použití na aplikaci, která generuje hrací plochu pro bludiště za pomoci vstupního obrazu. Aplikaci je výhodné nejdříve navrhnout, pro tento návrh jsem vybral široce využívaný standard UML. Pro tvorbu aplikace jsem zvolil jazyk C#. Jedná se o moderní programovací jazyk, s obsáhlou dokumentací a mám s ním zkušenosti z předchozího studia.

2. Cíle práce a metodika

Cílem práce je navržení aplikace, která nejdříve zpracuje vstupní obraz pomocí metod digitálního zpracování obrazu a následně vygeneruje hrací plochu ve formě bludiště na základě zpracovaného obrazu.

Teoretická část práce se zabývá popisem a analýzou teorie potřebné k tvorbě aplikace. Nejdříve je popsána reprezentace obrazu, dále jsou vysvětleny metody používané v procesu zpracování digitálního obrazu. Pozornost je věnována metodám pro odstranění šumu a zaostření obrazu, metodám pro získání informací z obrazu (především hrany) a následné možnosti analýzy obrazu.

Praktická část se zabývá samotným návrhem aplikace. Aplikace je navržena pomocí UML. Nejdříve je zobrazena struktura pomocí diagramu tříd, následně je popsáno chování aplikace diagramy aktivit a stavů. Dále je zobrazena implementace pomocí jazyka C#, zde jsou vysvětleny podstatné části kódu. Praktická část je zakončena popisem aplikace a jejích funkcí.

Informace pro dosažení cílů práce jsou získány z odborné literatury citované na konci práce, odborných článků na internetu a získaných znalostí jak z nynějšího tak předchozího studia. Zde se jedná hlavně o publikace od autorů Jahneho a Pratta.

3. Digitální zpracování obrazu – teorie

Značná část technik digitálního zpracování obrazu byla vyvinuta na konci šedesátých let v USA. Vývoj byl spojen s problémem přenést snímaný reálný obraz přes digitální médium (ať už se jedná o kabelový či bezdrátový přenos) s maximální přesností a s minimem šumů a poruch.

V té době se jednalo hlavně o přenos obrazových dat ze špionážních satelitů. Aby byly zajištěny výše zmíněné vlastnosti, bylo potřeba převést snímaný obraz do číselné podoby a poté znovu rekonstruovat za požití technik zpracování obrazu [1].

Tyto operace jsou podmíněné značným výpočetním výkonem a zpočátku jejich realizace byla v řádu hodin až dní. S evolucí výpočetní techniky se však operace spojené se zpracováním obrazu stávají stále méně časově náročné. Díky zpracování téměř v reálném čase se tak můžeme setkat nejen s vylepšováním kvality obrazu ale v neposlední řadě také s automatickým vyhodnocováním obrazu, kterého se hojně využívá například v lékařství, bezpečnosti a v oborech které využívají informace získané dálkovým průzkumem [1][2].

3.1 Reprezentace digitálního obrazu

Digitální obraz je číselnou reprezentací dvourozměrného obrazu. Obrazovou funkci můžeme vyjádřit jako $f(x, y)$, pokud zahrneme i čas t, bude funkce vypadat následovně: $f(x, y, t)$[2][3]. Reprezentovaný obraz lze rozdělit na *rastrový obraz* a *vektorový obraz*.

3.1.1 Rastrový obraz

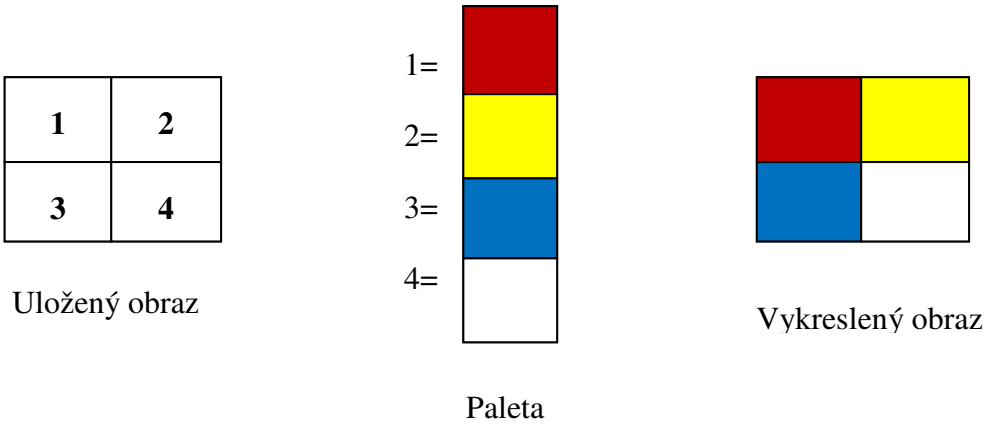
Rastrový obraz (někdy také nazývaný bitmapa), se skládá z dvourozměrné mřížky bodů (pixelů). Tyto body mohou nabývat hodnot podle typu obrazu:

Monochromatický obraz – Každý pixel je reprezentován pouze jedním bitem, může tak nabývat pouze 2 barev, ve většině případů se jedná o bílou a černou barvu.

Pokud rozšíříme obraz tak, aby každý pixel byl reprezentován 8 bity, můžeme rozšířit monochromatický obraz na obraz **šedotónový**, kdy každý pixel nabývá až 256 stupňů šedi. Tento obraz je důležitý pro potřeby práce protože všechny obrazy budou převedeny do této reprezentace z důvodu jednoduššího zpracování obrazu.

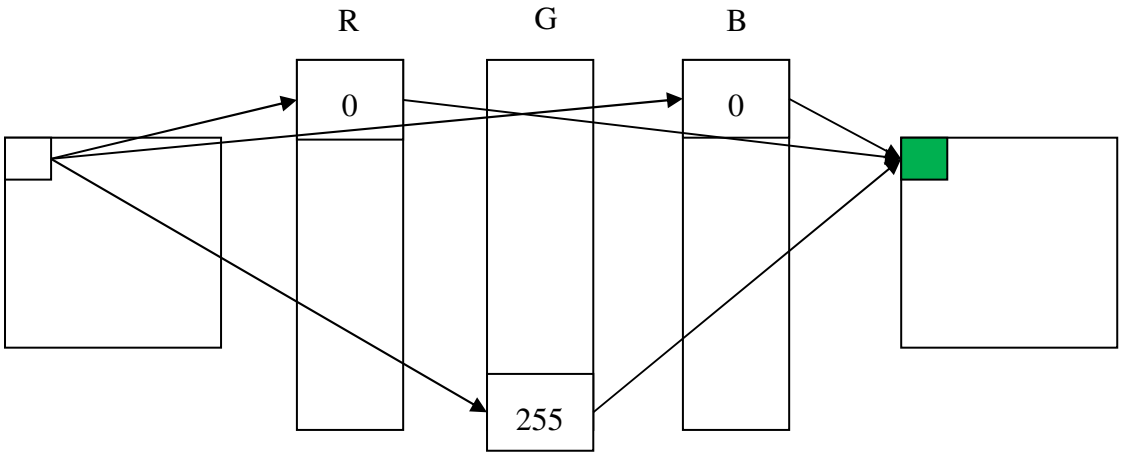
Indexový mód – Každý pixel je reprezentován jedním bytem, hodnota bytu není přímo bar-

vou, ale jedná se o odkaz do palety barev. Paleta barev je v tomto případě tabulkou o maximálně 256 řádcích. Obrázek 1 ukazuje schéma pro mapování barev v indexovém módu s paletou velikosti 4.



Obrázek 1: Indexový mód

Direct color – Tento typ zobrazení nejčastěji využívá RGB zobrazení, pixel je reprezentován třemi hodnotami (Pro každou barvu 8 z 24 bitů) je potřeba 3 palet (pro každý barevný kanál jedna). Výsledná barva je pak sumou 3 složek. Výhodou oproti indexovému módu je, že nepotřebujeme mít uloženou celou paletu barev ale pouze 1 paletu pro každý kanál [4]. Na obrázku 2 můžeme vidět vytvoření zelené barvy pro konkrétní pixel.



Obrázek 2: Direct color [4]

Rozlišení rastrového obrazu

Rozlišení je pojem, který udává počet pixelů který je schopno zobrazovací zařízení vykreslit, z pohledu dat se pak jedná o rozměry mřížky obrazu. Udává se jako počet *sloupců* * *počet*

řádků [3][4]. Uvádí se, že rastrový je závislý na rozlišení. Což znamená, že jakákoliv změna rozlišení obrazu znamená jeho zkreslení.

Výhody

Rastrový obraz je výhodnější pro zobrazovací a snímací zařízení protože odpovídá jejich technické funkci (záznam případně vykreslení jednotlivých pixelů do mřížky). Další výhodou je pak větší realističnost obrazu (například u fotografií), tato výhoda je však podmíněna dostatečným rozlišením, které obraz zaznamenává ale i vykresluje.

Nevýhody

Mezi hlavní nevýhody rastrové grafiky patří velká paměťová náročnost, která stoupá v závislosti na velikosti, vysoce kvalitní obrazy tak mohou dosahovat až několika desítek megabytů. Další nevýhodou, která plyne ze závislosti obrazu na rozlišení je zhoršení a zkreslení obrazu při jakékoliv změně velikosti obrazu. Jako poslední negativum lze uvést omezené přiblížení obrazu, kdy při velkém zvětšení lze vidět mřížku rastru [4].

3.1.2 Vektorový obraz

Vektorový obraz využívá základních geometrických primitiv, jako jsou body, přímky, křivky a tvary případně polygony k vykreslení obrazu. Poloha bodu je kódována dvojicí (případně trojicí, v případě 3D grafiky) souřadnic v konkrétním souřadném systému [1]. Body jsou poté spojeny výše zmíněnými primitivy. Spojení pak může mít další vlastnosti jako je výplň, tloušťka čáry a tvar. Veškeré informace jsou uložené v databázi, nejedná se však o konkrétní data jako u rastru ale spíše o jakýsi návod jak obraz vykreslit. Vykreslení obdélníku v jazyce C# pak může vypadat následovně:

myGraphics.DrawRectangle(myPen, 20, 10, 100, 50); [5]

kde funkce DrawRectangle určuje, že chceme vykreslit obdélník hodnota „myPen“ obsahuje vlastnosti čáry (pera), poslední čtveřice čísel pak ukazuje souřadnice bodů, které leží naproti sobě po úhlopříčce.

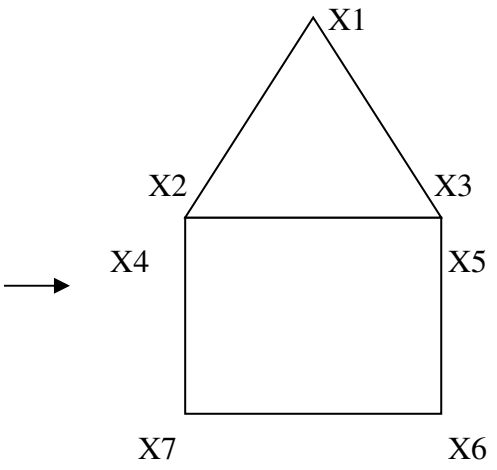
Ukládání dat vektorového obrazu je v praxi řešeno v základě dvěma modely:

Špagetový model

Jedná se o nejjednodušší datový model. Každý objekt je uložen jako řetězec souřadnic, tyto souřadnice jsou logicky seřazeny tak aby zobrazovali správnou posloupnost bodů. Tento mo-

del neposkytuje žádné informace o vztazích mezi objekty, znemožňuje tak téměř jakoukoliv analýzu obrazu (případně se musí vztahy složitě dopočítat). Další nevýhodou je jeho robustnost, v případě že 2 objekty používají stejnou čáru, je potřeba tuto čáru definovat pro oba objekty zvlášť. Na obrázku 3 vidíme schematickou reprezentaci dat pro jednoduchý obrázek. Je zde názorně vidět že i když čtverec a trojúhelník sdílej jednu hranu, je potřeba ji definovat pro oba objekty zvlášť [1][6].

Typ	**ID**	**Souřadnice**
Trojúhelník	1	X1, X2, X3
Obdélník	2	X4, X5, X6, X7



Obrázek 3: Špagetový model

Topologický model

Tento model řeší vztahy mezi objekty. Jako základní prvky se zde vyskytují spoje a uzly. Uzly mají dané souřadnice a jsou spojeny spoji. Spoj však v tomto případě není spojen se souřadnicemi ale s uzlem. Získáváme tak informace o spojení mezi objekty, nezávisle na souřadnicích. Každý spoj si dále eviduje další navazující spoj po své levé a pravé straně (vždy ten který má nejmenší rozdíl ve směru spoje). Jako uzel můžeme označit samostatný bod, bod kde se stýkají dva spoje, počáteční nebo koncový bod spoje a konečně jako bod na spoji. Tradiční topologický model se skládá z tabulky uzlů, spojů a obvykle tabulky atributů, která obsahuje informace o typu spoje, tloušťce čáry atd. Obrázek 4 ukazuje jednoduché zobrazení datové struktury topologického modelu (tabulka atributů je vynechána) [1][5][6]. Z obrázku je patrná vlastnost, že nemusíme znovu definovat uzel 2 pro každý spoj zvlášť, místo toho v tabulce spojů umístíme odkaz na korespondující uzel.

Výhody

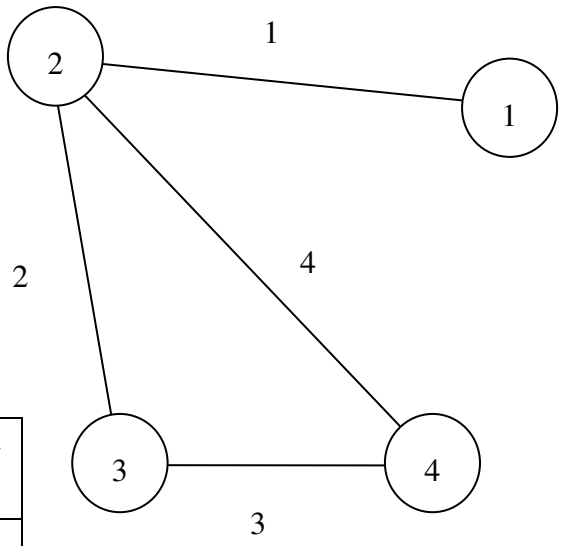
Vektorový obraz není závislý na rozlišení, při změně velikosti tak nedochází ke zkreslení obrazu, jsou pouze přepočítány souřadnice jednotlivých uzlů. Jako další výhodu lze uvést možnost teoreticky nekonečného přiblížení obrazu, teoreticky proto že při velkém přiblížení lze narazit jak na hranice konkrétního obrazu (vidíme například pouze jedinou čáru) nebo případně na hardwarové omezení kdy počítač již není schopen dále přepočítat obraz. Jako poslední výhodu lze uvést velice malou velikost souboru, vektorová grafika neobsahuje žádné konkrétní obrazové informace, pouze instrukce jak ho vykreslit.

Nevýhody

Vektorový model obrazu není vhodný pro záznam reálného obrazu, takový záznam by vyžadoval značně velký výpočetní výkon. Pokud překročíme určitou složitost obrazu, vektorová grafika se pak stává mnohem náročnější na operační paměť a výkon procesoru.

ID Uzlu	X	Y
1	X1	Y1
2	X2	Y2
3	X3	Y3
4	X4	X4

Tabulka Uzlů



ID Spoje	Počáteční uzel	Koncový uzel	Pravý spoj	Levý spoj
1	1	2	1	2
2	2	3	3	1
3	3	4	4	2
4	4	2	1	3

Tabulka spojů

Obrázek 4: Schéma datové reprezentace topologického modelu

3.2 Proces zpracování digitálního obrazu

Při pořízení snímku dochází ke zkreslení díky převodu z 3D do 2D scény. Dále pak dochází k deformaci optické informace zapříčiněné nedokonalostí prostředí a optického zařízení, tuto deformaci lze nazvat šumem, celkové pak ztrátu optických informací nazýváme primární deformací [1][4][6]. Snímek je uložen většinou ve formě rastrového obrazu, kde jemnost mřížky ovlivňuje jeho kvalitu.

Proces zpracování digitálního obrazu popisuje postup, kterým výše zmíněný snímek zpracovat tak abychom odstranili šumy, připravili ho pro naše potřeby a následně ho mohli analyzovat. Proces lze rozdělit na fáze *rekonstrukce*, *zvýraznění* a *analýzy* obrazu.

Obraz lze zpracovávat ve frekvenční oblasti nebo v prostorové oblasti. Ve frekvenční oblasti dosahujeme vyšších výkonů a není nutno zjišťovat, jakou masku použít (stačí pouze aplikovat příslušný tvar například Fourieovy transformace), nicméně samotný obraz je potřeba převádět mezi oblastmi. Dále implementace Fourierovy transformace je složitější než aplikace masek na jednotlivé pixely. Je proto nejdříve pospána Fourierova transformace a poté jednotlivé filtry pomocí masek

3.2.1 Fourierova transformace

Fourierova transformace (dále jen FT) je hojně využívána při procesu zpracování digitálního obrazu, je proto výhodné ji nejdříve definovat a popsat. FT slouží k převodu signálu (v tomto případě obrazu) z prostorové oblasti (klasická reprezentace pomocí rastru) do oblasti frekvenční (tuto reprezentaci lze nazvat jako Fourierův obraz) a pomocí zpětné Fourierovy transformace zpátky do prostorové. Frekvenční reprezentace obrazu je pak složením nekonečně mnoha signálů ve tvaru sinusoid o různých amplitudách, jež jsou fázově posunuté. Níže zmíněné metody filtrace mohou fungovat jak ve frekvenční tak v prostorové oblasti, je proto výhodné definovat možnost převodu mezi nimi protože praktická část práce bude pracovat oblasti prostorové [1][3][4][7].

Spojitá FT

Fourierův obraz jednorozměrné funkce $f(x)$ označíme jako komplexní funkci $F(u)$ získanou následující integrací

$$F(u) = \int_{-\infty}^{+\infty} f(x) e^{-i2\pi ux} dx \text{ [4]}$$

kde komplexní jednotka $i = \sqrt{-1}$. Zpětnou FT získáme pomocí vzorce:

$$f(x) = \int_{-\infty}^{+\infty} F(u) e^{+i2\pi ux} du \text{ [4]}$$

Lze vidět, že *F(u)* a *f(x)* jsou svázané relací $f(x) \Leftrightarrow F(u)$.

Diskrétní FT

Signál, který je dán omezeným počtem vzorků které se periodicky opakují lze zpracovat diskrétní FT (DFT). Dopřednou DFT časové funkce I_k lze vyjádřit jako

$$F_n = \frac{1}{N} \sum_{k=0}^{N-1} I_k \, e^{-i2\pi \frac{nk}{N}} \text{ [4]}$$

Zpětnou DFT pak duálním vztahem:

$$I_k = \sum_{k=0}^{N-1} I_k \, e^{+i2\pi \frac{nk}{N}} \text{ [4]}$$

Opět zde vidíme duální vztah mezi časovou reprezentací I_k a Fourierovým obrazem:

$F_n I_k \Leftrightarrow F_n$.

Dvourozměrná DFT

Výše zmíněná DFT definuje transformace pouze pro jednorozměrný prostor, digitální obraz je však dvourozměrný. Protože je zaveden další rozměr, výsledkem již není sinusoida ale plocha vzniklá vytažením sinusoidy do prostoru, vzniká tak efekt „vlnitého plechu“. Po zavedení druhé souřadnice bodu pak získáváme vzorec v podobě sumy sum [1][4][8]:

$$F(u,v) = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) e^{\left[-2\pi i \left(\frac{m\,u}{M} + \frac{n\,v}{N}\right)\right]} \text{ [8]}$$

Kde *u = (0, 1, …,M-1)* a *v= (0, 1, …,M-1).*

M x N je velikost mřížky.

Zpětnou dvourozměrnou DFT lze pak vyjádřit následovně:

$$f(m,n) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} F(u,v) e^{\left[+2\pi i\left(\frac{m\,u}{M}+\frac{n\,v}{N}\right)\right]}$$ [8]

3.2.2 Rekonstrukce obrazu

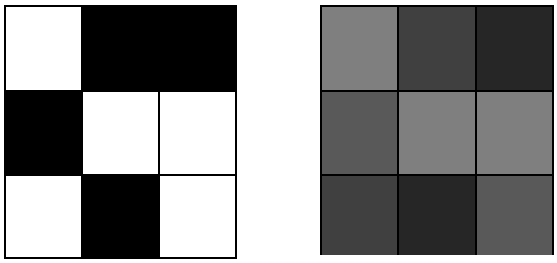
Tato fáze si klade za cíl v co nejlepší formě odstranit šumy z obrazu. Šum je informace, která byla do obrazu přidána v optickém zařízení nebo díky prostředí. Šum je vždy svázán s původní informací. Protože ale neznáme původní funkci (jako například v případě zvukového šumu), nemůže s jistotou odlišit co je původní obraz a co je šum. Postupy pro odstranění šumu tak vždy pracují s informacemi, které porovnávají rozdíly s vysokou frekvencí mezi sousedícími pixely, ostrými hranami a dalšími vysokofrekvenčními informacemi [2][3][4].

Metoda opakovaného pořízení obrazu

Tato metoda pracuje na principu několikanásobného pořízení záznamu stejné scény, porovnáním těchto snímků pak lze snadno získat původní obraz. Porovnávají se pixely o stejných souřadnicích, tam kde nalezneme největší rozdíl od základní hodnoty lze očekávat šum. Základní hodnotu lze určit pomocí průměru, mediánu, případně hodnotu jako hodnotu, která se nejčastěji opakuje. Nutno poznamenat že tato metoda se téměř nepoužívá v praxi, protože ve většině případů pracujeme pouze s jedním obrazem. Teoreticky ji lze využívat u scén, které se v čase mění jen naprosto minimálně [2][4].

Filtrace obrazu

Filtrace obrazu se zabývá odstraněním nežádoucích efektů z obrazu (dále nazývám anomálie) Jedná se například o odstranění šumu nebo zaostření obrazu. Protože původní funkce obrazu je neznámá, nevíme co je původní obraz a co je anomálie. Je proto potřeba obraz vyhodnotit z pohledu abnormalit jednotlivých pixelů a jejich okolí. Za anomálii pak lze považovat vysoké změny v hodnotách okolních pixelů. Jako základní charakteristika pro vyhodnocení změn se využívá tzv. prostorová frekvence, tu lze definovat jako počet změn v hodnotě svítivosti na délkovou jednotku, kde vysoká frekvence znamená velké rozdíly a nízká frekvence rozdíly velké (viz obrázek 5) [1][4].



Obrázek 5: Obraz s vysokou (nalevo) a nízkou (napravo) prostorovou frekvencí

Filtry se rozdělují dle frekvence, kterou propouští na *nízkofrekvenční* a *vysokofrekvenční* filtry. Opakovaným použitím filtru dochází k rozmazávání obrazu, kdy se lze limitně přiblížit až obrazu tvořeným jednou barvou. Proces odstranění šumu tak často sestává z aplikace nejprve nízkofrekvenčního filtru, který odstraní vysokofrekvenční šum, ale také zkreslí například hrany a poté vysokofrekvenční filtr který tyto hrany opět zvýrazní [1][9].

Filtraci obrazu je možno provádět ve frekvenční reprezentaci (rychlé ale a většinou se provádí během digitalizace obrazu a pro obrazy měnící se v reálném čase) nebo prostorové (pomalejší, nicméně jednodušeji realizovatelné pro již existující a statický obraz).

Nejjednodušším způsobem odstranění šumu je postupné průměrování obrazu:

Mějme bod X a jeho okolí K_i.

Pokud je hodnota svítivosti v bodě X větší než průměrná hodnota bodů v jeho okolí ($\left|X - \frac{1}{K_n}\sum_i^n K_i\right| > \varepsilon$), tak nahradíme X hodnotou průměru okolních hodnot ($X = \frac{1}{K_n}\sum_i^n K_i$) [1].

Nízkofrekvenční filtry

Tento typ filtru propouští nízké frekvence a zachycuje vysoké, dochází k odstranění šumu, ale bohužel také zkreslení ostrých detailů v obrazu, jako jsou hrany (tento efekt nicméně nemusí být nežádoucí, občas chceme odstranit příliš ostré detaily).

Pro zpracování filtrace se používá tzv. masky (případně okna, anglicky window). Použití masky lze popsat vztahem diskrétní konvoluce za použití masky g na obraz f_n [1]:

$$h_m(i,j) = \sum_{k=d}^{h}\sum_{l=d}^{h} f_m(k,l)g(i-k, j-l)[1]$$

kde $d = -\frac{s-1}{2}, h = \frac{s}{2}$, hodnota *s* je velikost masky.

Základní konvoluční jádro (maska) pro obyčejné průměrování vypadá takto [4]:

$$h = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Tato technika se *nazývá obyčejné průměrování* a patří do metod lineárního filtrování.

Jeho modifikací můžeme získat masku pro filtrování gaussova šumu [4]:

$$h = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Pokud je cílem odstranit body s velkým rozdílem oproti okolí, lze také použít modifikaci masky, kdy použijeme hodnoty průměru okolí ale bez hodnoty středového pixelu, maska pak bude vypadat takto [1][4]:

$$h = \frac{1}{8}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Součet hodnot v jádru se musí rovnat 1, při hodnotě nižší než 1 by došlo ke ztmavení obrazu a naopak při hodnotě větší než 1 dojde k zesvětlení.

Vysoké frekvence jsou těmito metodami přímo odřezány, případně je lze s určitou váhou. Tento vztah lze popsat následujícím vzorcem [3][4]:

$$h(i,j) = \begin{cases} 1 \; pro \; \sqrt{i^2 + j^2} \leq D_0 \\ 0 \; pro \; \sqrt{i^2 + j^2} > D_0 \end{cases} [4]$$

Hodnota $\sqrt{i^2 + j^2}$ je vzdálenost frekvence od počátku (ve Fourierově oblasti), ze vztahu je vidět že filtr pak nepropouští frekvence vyšší než D_0 [2][4].

Jako alternativu k lineární filtraci jsou metody pracující na bázi lokální statistiky a to konkrétně *filtrace pomocí mediánu.*

Algoritmus mediánové filtrace lze popsat takto:

1. Načti pixely obrazu do pole M.
2. Uspořádej sousední pixely na základě jejich intenzity v poli M
3. Nahraď původní hodnotu pixelu hodnotou mediánu z uspořádaného pole

Tato metoda je vhodná pro odstranění bodového šumu. Její nevýhodou je, že narušuje tenké čáry, občas je proto výhodné nepoužít čtvercové, ale křížové, případně pole ve tvaru X [9].

Vysokofrekvenční filtry

Tento typ filtru se používá pro zvýraznění hran a zaostření obrazu. Filtrovaný obraz se získává dvěma způsoby. Jednodušší variantou je odečíst hodnoty každého bodu nízkofrekvenčně filtrovaného obrazu od originálu. Případně lze použít diskrétní konvoluci obrazu s vysokofrekvenčním jádrem [1][4].

Oko vnímá hranu v diskrétním obrazu tam, kde dochází k výrazné změně sousedních pixelů. Hrana je pak určena gradientem. Protože diskrétní obraz nelze vyjádřit spojitou funkcí, je nutno gradient odhadnout [4].

Mějme funkci *s(i, j)*, tato funkce reprezentuje velikost gradientu, výsledný obraz *g(i, j)* lze získat z původního obrazu *f(i, j)*, ostřením koeficientem *c*.

$$g(i,j) = f(i,j) + c.s(i,j)[4]$$

Pro určení velikosti gradientu se používají postupy založené na konvolučních nebo jiných operátorech.

Jako nejjednodušší operátor lze uvést *operátor Robertsův*. Tento operátor není založen na konvoluci ale na prostém výpočtu pixelu z jeho tří sousedů [4]:

$$|\nabla f(i,j)| = |f(i,j) - f(i+1,j+1)| + |f(i+1,j) - f(i,j+1)|$$

Na podobném principu funguje *Sobelův operátor*, ten však již pracuje s dvojicí komplementárních konvolučních masek. Tento operátor aproximuje první derivaci. Komplementární masku se dá získat rotací původní masky o 90°, přičemž nezáleží na tom, zda doleva či doprava. Pro svislý a vodorovný směr masky mohou vypadat takto:

$$h = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \bar{h} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} [1][4]$$

Pro šikmé směry:

$$h = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}, \bar{h} = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix} [1][4]$$

Výslednou hodnotu gradientu lze pak získat následujícím způsobem:

$$|G| = \sqrt{h^2 + \bar{h}^2}\ [4]$$

Operátor pracující s druhou derivací se nazývá *Laplaceův operátor*. Lze pomocí něj najít lokální extrémy. Jádro H_1 se využívá čtyřokolí bodu H_2 osmiokolí.

$$H_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}, H_2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}\ [1][4]$$

Gradient, který tento operátor získá, se odhaduje jako součet dvou na sebe kolmých diferencí [1][4]:

$$\Delta f(x,y) = \frac{\partial^2 f(x,y)}{\partial^2 x} + \frac{\partial^2 f(x,y)}{\partial^2 y}\ [1][4]$$

Výsledkem Laplaceova operátoru mohou být i záporné hodnoty, tento problém se nejčastěji odstraňuje absolutní hodnotou.

Metoda *Statických diferencí* vyzdvihuje hrany tím, že každý bod obrazu vydělí hodnotu standardní odchylky.

$$G(i,j) = \frac{F(i,j)}{\sigma(i,j)}\ [1]$$

Odchylka se vypočítá jako: $\sigma^2(i,j) = \sum_i \sum_j [F(i,j) - \bar{F}(i,j)]^2$, $\bar{F}(i,j)$ značí průměrnou svítivost v okolí bodu $F(i,j)$.

3.2.3 Zvýraznění obrazu

Fáze zvýraznění obrazu (někdy také segmentace obrazu) je soubor metod pro zvýraznění a nalezení prvků v obrazu klíčových pro následnou analýzu. Mezi úkoly a s tím i spojené metody používané ve zvýraznění obrazu například patří: *Detekce hran a oblastí,* a jejich případné spojování. Některé metody vysokofrekvenční filtrace nalézají své uplatnění i ve fázi zvýraznění obrazu [1][7]. Na konci fáze zvýraznění již víme, který pixel patří kterému objektu [3].

Zvýraznění obrazu je výpočetně náročná operace, je proto důležité vhodně zvolit metody v závislosti na charakteru následné analýzy.

Prahování

Prahování je jedna z nejstarších ale také nejjednodušších metod. Používá se samostatně ale

často také v kombinaci s ostatními metodami. Výhodou této metody je relativně snadná implementace a malá časová náročnost.

Principem metody je že objekt a pozadí mají rozdílné hodnoty intenzity. Pixel, který má větší intenzitu než práh je objekt a pixel s intenzitou menší než práh je pozadí [7]:

$$f(i,j) = \begin{cases} 1\ pro\ g(i,j) > T \\ 0\ pro\ g(i,j) \leq T \end{cases} [7]$$

Kde *f(i, j)* je výsledný binární pixel, *g(i, j)* jsou hodnoty intenzity a *T* je práh.

Určení prahu je pro tuto metodu zásadní. Velice často nelze použít jeden práh na celou plochu obrazu (globální práh), z důvodu například nerovnoměrného osvětlení, ale je nutno použít tzv. lokálního prahu. Obraz například může rozdělit na oblasti s různými hodnotami prahu, pokud v nějaké oblasti určení prahu selže, je možno požít práh získaný interpolací hodnot prahů sousedních oblastí [9]. Modifikace prahování pak vypadá takto:

$$f(i,j) = \begin{cases} 1\ pro\ g(i,j) \ni A \\ 0\ pro\ ostatní \end{cases} [9]$$

A je množina prahů jednotlivých oblastí obrazu.

U obrazů, které obsahují výrazné vlastnosti (jako například test) lze práh určit pouhým nahlédnutím do histogramu.

Na valnou většinu obrazů je nutno však aplikovat složitější metody využívající histogramu. Každý vrchol histogramu odpovídá četnosti jednotlivých úrovní intenzity. Pokud histogram obsahuje dva výrazné vrcholy, jeden vrchol značí objekt a druhý vrchol pozadí. Výsledná prahová hodnota se určí jako minimum mezi těmito dvěma maximy. Pro multimodální histogram (více vrcholový) lze určit více hodnot prahu. Je pak možno buď použít hodnotu, která vrací nejmenší segmentační chybu, případně lze použít více prahovou metodu. Obrázek 6 ukazuje původní snímek parku (nalevo) a jeho prahovanou verzi (napravo) [7][10].


Obrázek 6: Původní a prahovaný obraz [11]

Detekce hran

Pro detekci hran lze s úspěchem použít operátorové metody používané pro vysokofrekvenční filtry. Tyto metody však byly vytvořeny více či méně „intuitivně“ a nejsou u nich formulovány požadavky, které by měli splňovat [7]. Bude proto popsán detektor hran definovaný v roce 1986 Johnem F. Cannym.

Cannyho detektor hran

Požadavky na tento detektor byly navrženy takto:

1. Minimalizace pravděpodobnosti chybné detekce,
2. Nalezení polohy reálné hrany (ne hran vzniklých pořízením) co nejpřesněji,
3. Jednoznačná identifikace bodu hrany.

Proces detekce hran pomocí Cannyho detektoru sestává z několika fází.

První krok procesu detekce hran se soustřeďuje na redukci Gaussova šumu protože Cannyho detektor je zatížen chybou vzniklou působením šumu v nezpracovaném obrazu. Pomocí vztahu pro dvourozměrné Gaussovo normální rozdělení se spočítá konvoluční maska [7][10]:

$$G(i,j) = \frac{1}{2\pi\sigma^2} . e^{-\frac{i^2+j^2}{2\sigma^2}} \text{ [7][10]}$$

Vypočtenou maskou se pak obraz zpracuje jak v ose x tak y. Maska může vypadat například takto:

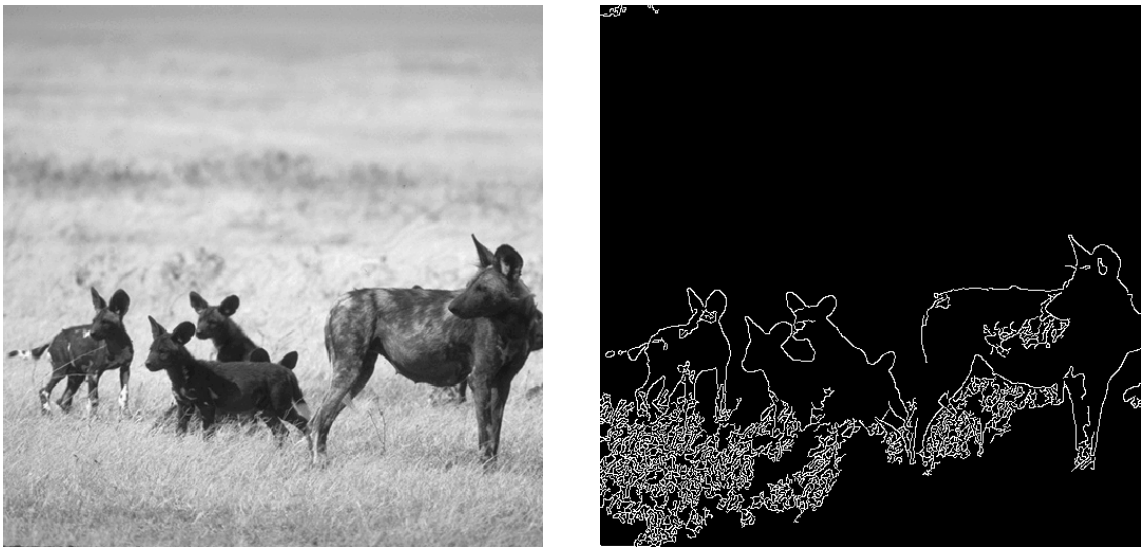
$$h = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$ [7]

Vznikne tak rozmazaný obraz. Velikostí hodnoty σ lze ovlivnit ostrost hran ve výsledném obrazu (hodnota 1 se používá jako základní pro ostrý obraz).

Ve *druhém kroku* je nutno spočítat velikost a směr gradientu, tyto hodnoty určují hledanou hranu. Zde se často používá Sobelův operátor, který není příliš citlivý na šum. Pomocí masek Sobelova operátoru pro vertikální a horizontální respektive šikmé směry, získáme hodnoty prvních derivacích dvou na sebe kolmých směrů které dosadíme do vztahu $|G| = \sqrt{h^2 + \bar{h}^2}$ [4] a získáme velikost gradientu. Pro zjištění se použije vztahu: $d = arctg2(h, \bar{h})$[7][10].

Ve *třetím kroku* probíhá tzv. ztenčení (Thinning). Pří výpočtu první derivace vznikají široké hrany, je proto potřeba najít body gradientu, které nejsou hodnotami lokálního maxima a tyto body pak odebrat [10]. Prakticky se hledají body, které mají hodnotu gradientu pravého a levého souseda (pro svislé hrany) nižší než středový pixel.

Protože v předchozím kroku byly nalezeny všechny hrany, včetně hran vyvolaných šumem, je potřeba ve *čtvrtém* a finálním kroku tyto hrany odfiltrovat. Využívá se zde tzv. prahování s hysterezí, principem je zvolit minimální a maximální prah (T_1 a T_2). Pokud je hodnota gradientu větší než T_2, je hodnota označena jako hrana. Pokud se gradient pohybuje mezi T_1 a T_2, je označen jako hranový pouze pokud sousedí s bodem, který byl již dříve označena jako hranový. V ostatních případech je odstraněn. Obrázek 7 ukazuje původní obraz (nalevo) a obraz po detekci hran Cannyho detektorem hran s hodnotami T_1=0,01 a T_2=0,5 y intervalu gradientu 0 – 1 (napravo). Za povšimnutí stojí, že celkem úspěšně byly vyfiltrovány obrysy zvířat a odstraněno rozmazané pozadí, nicméně jako hrana byla vyhodnocena také detailní tráva v popředí [7][10].



Obrázek 7: Cannyho detektor hran

Detekce oblastí

Metody detekce oblastí se nezabývají nalezením jejich hranic, ale vytvořením oblasti celku na základě vlastností, které určují jistou míru homogenity (jako je podobná barva, jas nebo pokrytí stejnou texturou). Metody pro detekci oblastí jsou méně zatížené šumy, nelze tudíž říci že jsou rovnocenné s metodami detekce hran, které naopak na šum reagují citlivě. V základě lze tyto metody rozdělit na detekce prahováním, detekce dělením oblasti a detekce plněním oblasti. Metoda prahování již byla obecně popsána, budu se proto zabývat zbývajícími dvěma.

Metoda růstu oblasti (Region Growing)

Jedná se o jednu z nejjednodušších metod zvýraznění obrazu, sousední pixely jsou zařazeny do skupin tak, aby vytvořily segmentovaný region, v praxi je nutno do algoritmu umístit různá omezení aby bylo možno získat přijatelné výsledky [2].

V první fázi, se obraz rozdělí na tzv. atomické oblasti - pixely jsou kombinovány, pokud mají stejnou amplitudu a sousedí s dalšími čtyřmi pixely této amplitudy.

V další fázi jsou slučovány sousední oblasti se slabou hranicí. Lze použít několik kritérií pro určení, zda hranice je dostatečně slabá aby bylo možno oblasti sloučit. Pro málo komplexní obrazy lze použít následujících dvou pravidel pro sloučení dvou oblastí [2][3]:

Mějme dvě sousední atomické oblasti O_1 a O_2 které mají hranice H_1 a H_2, ty mají společnou

hranici o délce *S*, proměnná *D* reprezentuje délku *S* pro kterou je rozdíl amplitud mezi hranicemi *A* menší než kritérium důležitosti ε_1. Dvě oblasti jsou poté sloučeny pokud:

$$\frac{D}{MIN\{H_1, H_2\}} > \varepsilon_2$$

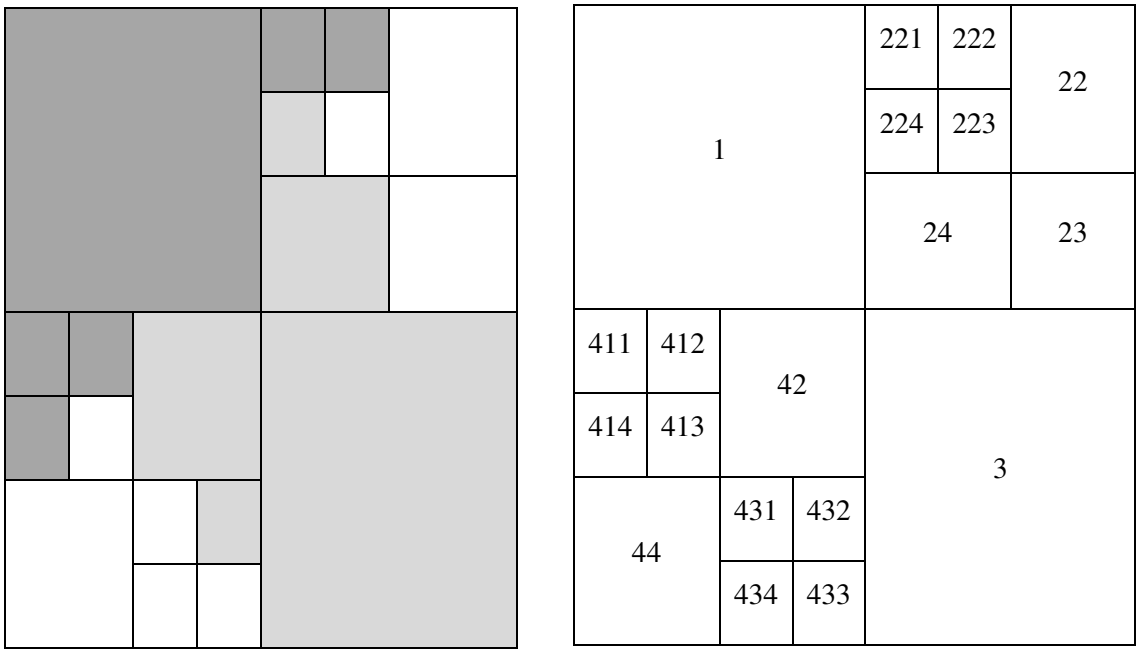
Kde ε_2 bývá konstanta o velikosti ½. Toto pravidlo zamezí sloučení oblastí o přibližně stejné velikosti a zároveň umožní začlenění malých oblastí do větších [2][3].

Aplikací pravidla $\frac{D}{C} > \epsilon_3$ kde ε_3 má většinou hodnotu ¾ se zajistí sloučení oblastí se slabou hranicí, které zůstali po aplikaci prvního pravidla. Aplikace pouze druhého pravidla může zapříčinit přílišné sloučení oblastí [2].

Pro komplexnější obrazy lze použít tzv. semínek. Tyto semínka uživatel umístí do vizuálně podobných oblastí, hodnoty bodů pak určují výchozí hodnoty, podle kterých jsou určována kritéria důležitosti. Tyto kritéria jsou pak využita pro vytvoření atomických oblastí [2].

Metoda dělení a slučování oblasti

Tato metoda se skládá ze starších metod split (rozděl) a merge (sluč). Základem je tzv. quad tree reprezentace obrazu (obrázek 8). Obraz či část obrazu je rozdělen do čtyř kvadrantů (split), pokud původní kvadrant nemá podobné vlastnosti. Pokud čtyři sousední kvadranty mají podobné vlastnosti, jsou sloučeny do jednoho kvadrantu [2][3].



Obrázek 8: Quad tree diagram

Největším problémem této metody je na jaké úrovni začít. Je možno začít s celým obrazem a inicializovat metodu split, tato varianta je však výpočetně náročná. Stejný problém nastává z opačného konce, kdy metoda začíná od jednotlivých pixelů, které se následně slučují [2][3].

Používá se proto metod, které zvolí počáteční úroveň (například podle statistického zastoupení barev v obrazu) quad tree diagramu a až poté se začne se slučováním či rozdělováním.

Metoda záplavy

Tato metoda je založena na poznatcích z oborů topografie a hydrologie. Na monochromatický obraz je zde pohlíženo jako na povrch skládající se z údolí a v vrcholů, kde údolí jsou pixely s malou amplitudou a vrcholy s amplitudou vysokou.

Voda pramení z údolí, tam kde dosáhne vrcholu a měla by si přelít do sousedního údolí, se postaví tzv. hráz. Tato hráz pak zobrazuje hranici mezi oblastmi obrazu. Segmentace končí, když hladina dosáhne nejvyššího vrcholu [2].

3.2.4 Analýza obrazu

Analýza obrazu se zabývá extrakcí dat a informací z digitálního obrazu pomocí automatických či částečně automatických metod. U nepříliš komplexních obrazů je možno použít rov-

nou analýzu obrazu, většinou však analýze obrazu předchází zvýraznění obrazu, která eliminuje šumy a anomálie ztěžující následnou analýzu. Od ostatních metod zpracování digitálního obrazu se liší tím, že výstupem je spíše numerická než vizuální reprezentace obrazu. Je potřeba také zmínit že k interpretaci není vždy potřeba složitých matematický metod. Pokud je obraz dostatečně jednoduchý, případně známý, lze ho interpretovat také na základě zkušeností s hledanou skutečností. Například víme, že voda má obvykle v mapách modrou barvu a vegetace barvu zelenou, můžeme tak oblasti obrazu zařadit do tříd pouze na základě barev [1][2].

Před samotnou analýzou je potřeba získat příznaky, které reprezentují dané objekty v obrazu, tyto příznaky se označují x_1, x_2 až x_n. Objekty je pak možno reprezentovat vektorem příznaků: $x = (x_2, x_2, \dots, x_n)^T$. Obraz je pak možno reprezentovat prostorem příznaků X_n (většinou se jedná o Eukleidovský n-rozměrný prostor). Takto definovaný obraz je nyní možno analyzovat pomocí klasifikace. Příkladem příznaku může být barva pixelu, amplituda, četnost pixelu v oblasti a další obrazové informace [1][2][7].

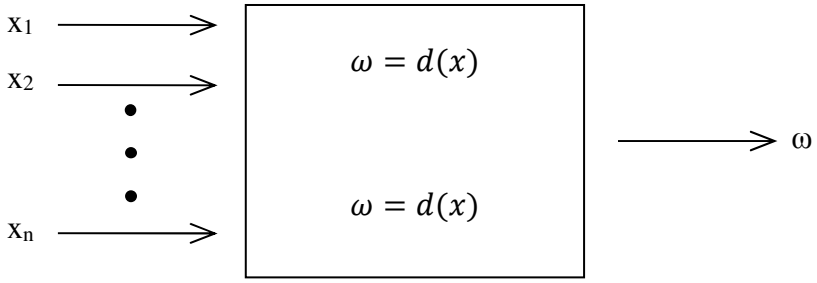
Klasifikace obrazu

Klasifikace je rozdělení objektů podle příznaků do tříd. Základní myšlenkou je že objekt *O1* mající vektor příznaků *x1* a objekt *O2* kterému odpovídá vektor příznaků *x2*, jsou podobné pokud délka vektoru *x1-x2* je nulová nebo dostatečně malá. Množinu podobných objektů je pak možno nazvat třídou. Množina tříd se označuje $\Omega = \{\omega_1, \omega_2, \dots, \omega_n\}$ [1][2][7].

Klasifikace se rozděluje na *řízenou* a *neřízenou* klasifikaci. Během řízené klasifikace se klasifikátor (viz níže) nejdříve pomocí učebního (známého) vzorku dat učí určit jednotlivé třídy, až poté jsou na vstup přivedena data, která je potřeba klasifikovat. U neřízené klasifikace se klasifikátor učí průběžně během činnosti [1][2]. Neřízená klasifikace není příliš vhodná pro účely této práce, soustředím se proto na klasifikaci řízenou.

Klasifikátor

Klasifikátor je zobrazení, které každému vektoru příznaků přiřadí identifikátor třídy, do které zpracovávaný objekt patří. Tento vztah se zapisuje následovně: $\omega = d(x)$. Klasifikátory se dají rozdělit na deterministické a nedeterministické, klasifikátor může mít omezený nebo proměnný počet příznaků [1][7].



Obrázek 9: Schéma klasifikátoru

Klasifikace diskriminačními funkcemi

Pro tento způsob klasifikace předpokládáme existenci *n* diskriminačních funkcí $g_1(x), g_2(x), \dots, g_n(x)$. Rozhodnutí do jaké třídy objekt patří, je založeno na hodnotě funkce která vrací pro daný vektor příznaků maximální hodnotu. Třídu lze pak stanovit na základě pravidla $g_i(x) > g_j(x); i \neq j$ [1][7]. Tím že máme určeny diskriminační funkce známe oblasti jednotlivých tříd, oblast lze definovat například takto [7]:

$$\chi_r = \{x|d(x)\} = \omega_r$$

Hranice mezi dvěma oblastmi lze získat pomocí řešení rovnice $g_i(x) = g_j(x)$.

Určení diskriminačních funkcí za pomoci Bayesova kritéria

Tato metoda se někdy také nazývá minimalizace rizika a většinou se využívá při *řízené* klasifikaci. Předchozí definice předpokládali jednoznačné určení třídy objektu, během klasifikace však může nastat situace, kdy jeden se objekt zobrazí jako příznak více tříd. Klasifikátor však ve svém základu dokáže objektu přiřadit pouze jednu třídu, tu to situaci lze označit jako chybu klasifikace. Utrpěnou ztrátu lze popsat funkcí $\lambda(\omega_r|\omega_s)$, hodnota funkce kvantitativně popisuje chybné zařazení *x* do ω_r místo do ω_s [7].

Mějme Bayesův vztah pro podmíněnou pravděpodobnost (modifikovaný pro potřeby klasifikace):

$$P(\omega_i|x) = \frac{P(x|\omega_i).P\omega_i}{P(x)} [1][7]$$

$P(x)$ je hustota pravděpodobnosti vektoru x a $P(\omega_i|x)$ je podmíněná pravděpodobnosti. Třídu pak vybereme podle toho kde $P(\omega_i|x)$ je maximální [1][7].

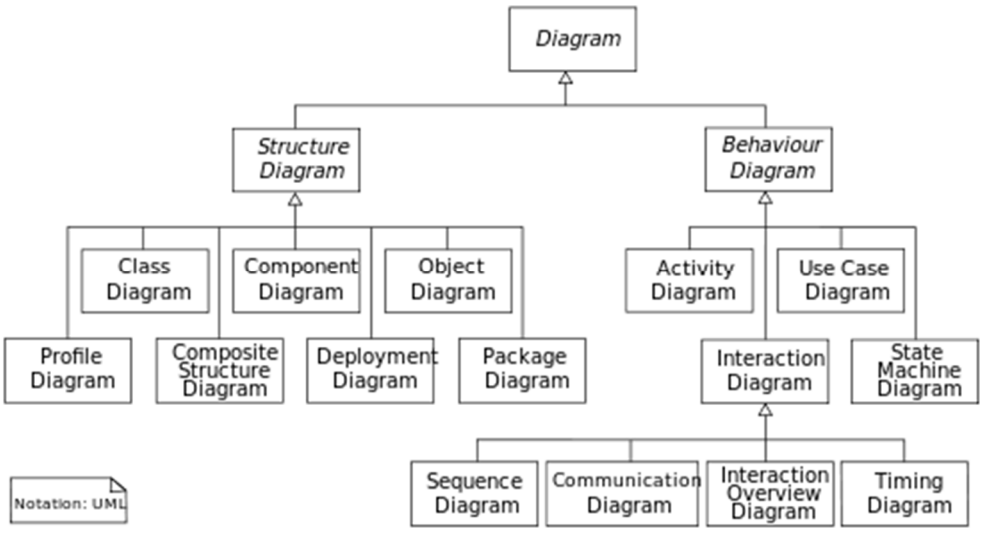
4. Návrh a realizace aplikace

Kapitola se zabývá vývojem aplikace, ta je nejdříve formálně navržena v jazyce UML, zde se jedná hlavně o diagram tříd a popis stavů aplikace. Aplikace je realizována na základě vytvořeného modelu v jazyce C# za pomocí některých metod popsaných v teoretické části práce.

Aplikace je napsaná v jazyce C#, tento jazyk je vyvíjen firmou Microsoft. Jeho první verze byla vydána v roce 2002 zároveň s .NET framework 1.0. Jedná se objektově orientovaný jazyk, vhodný pro psaní například grafických, databázových ale i mobilních aplikací. Platforma .NET slouží k vývoji aplikací primárně na počítačích se systémem Windows, lze ji však použít i pro další operační systémy [13]. Jazyk je plně objektový, podporuje rekurzy a má obsáhlou základní podporu pro práci s grafickým rozhraním a pro práci s grafikou. Další jeho výhodou je přehledná syntaxe. Jeho výhodou ale také nevýhodou je automatická práce s pamětí, kdy prostředí .NET uvolňuje a alokuje paměť na základě požadavku programu a programátor nemůže toto chování příliš ovlivnit.

4.1 Návrh aplikace

Aplikace je navržena v jazyce UML (Unified Modeling Language). Tento jazyk slouží jako standardizovaný nástroj pro modelování a vizualizaci striktury systémů (nejen informačních). Skládá se z několika úrovní diagramů (Obrázek 10), v základě se dají tyto diagramy rozdělit na diagramy popisující strukturu systému a diagramy popisující chování systému.



Obrázek 10: Diagramy UML[12]

Pro účely práce použiji následující diagramy [12]:

Diagram tříd – Zobrazuje strukturu systému pomocí jeho tříd a vztahy mezi nimi. Hlavními objekty jsou třídy, ty jsou obvykle zobrazeny jako obdélníky rozdělené na tři části. V hlavičce je název třídy, dále zde může být typ třídy a jmenný prostor. Další část obsahuje atributy třídy, zde mohou být klasické proměnné ale také metody, které pouze vrací soukromou proměnnou. Posledním prvkem je seznam metod třídy [12].

Dalším prvkem diagramu tříd jsou vztahy, graficky se jedná o spojení mezi třídami a může být čtyř druhů: **Asociace** (Třídy jsou na sobě nezávislé), zobrazuje se obyčejnou čarou. **Agregace** (vztah celek – část), čára je ukončena kosočtvercem na straně třídy, která zobrazuje celek, u agregace může třída představující část existovat sama o sobě a může být součástí více celků. **Kompozice** je podobná agregaci, nicméně třída představující část nemůže existovat sama o sobě a zaniká s celkem. Zobrazuje se plným kosočtvercem. **Generalizace** představuje dědičnost, třída děděná přebírá všechny charakteristiky předka a zároveň přidává nové, zobrazuje jako čára s trojúhelníkem na straně rodiče [12][18].

Posledním prvkem diagramu je multiplicita. Zobrazuje se nad zakončením vztahů a představuje, kolik instancí třídy může existovat ve vztahu. Například třída matka může mít žádné nebo více dětí (psáno * na straně dítěte), ale třída dítě může mít pouze jednu matku (psáno 1 na straně matky) [12][18].

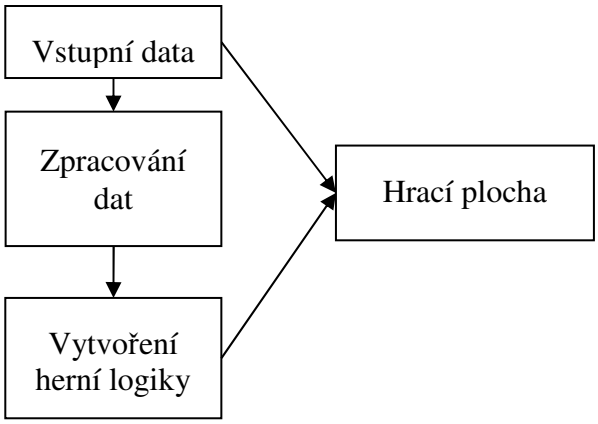
Stavový diagram – Popisuje stavy, kterými jednotlivé třídy prochází. Stavový diagram se skládá ze dvou prvků: **Stav se** skládá z názvu stavu (například „čeká na data“) a aktivit, které ve stavu probíhají. Aktivity se mohou provádět při vstupu do vztahu (označováno „entry/<název aktivity>“), během stavu („do/“) a výstupní aktivity („exit/“). Dalším prvkem jsou **přechody**, ty zobrazují přechod ze stavu do stavu. Přechod mezi stavy může být podmíněn **událostí**, ta se píše jako text nad čáru přechodu. Událost může mít i takzvaného hlídače (píše se v hranatých závorkách za událost) který kontroluje zda událost proběhla správně (například známka : [hodnota > 4]) [12][18].

Diagram aktivit – Popisuje, jak probíhají jednotlivé aktivity v systému. Tento diagram je podobný klasické vývojovému diagramu. Diagram se skládá z **aktivit** zobrazující akci, která v té chvíli proběhne. Aktivity se označují obdélníkem (klasická aktivita), přesýpacími hodinami (časově podmíněná aktivita) nebo obdélníkem trojúhelníkem na levé straně (příchozí

aktivita z jiného systému). Diagram dále obsahuje spojení a rozpojení, označované kosočtvercem [18].

4.1.1 Formulace problému

Základní myšlenkou je, že přijímá nezpracovaný obraz. Tento obraz se dále zaostří, převede do tónů šedi a poté se v něm naleznou hrany. Výsledek se poté předá třídě, která vytvoří herní logiku na základě vstupu. Vstupní data jsou dále spárována s herní logikou za účelem vytvoření výsledně hrací plochy. Základní náčrtek činnosti aplikace vypadá takto:



Obrázek 11: Náčrtek činnosti aplikace

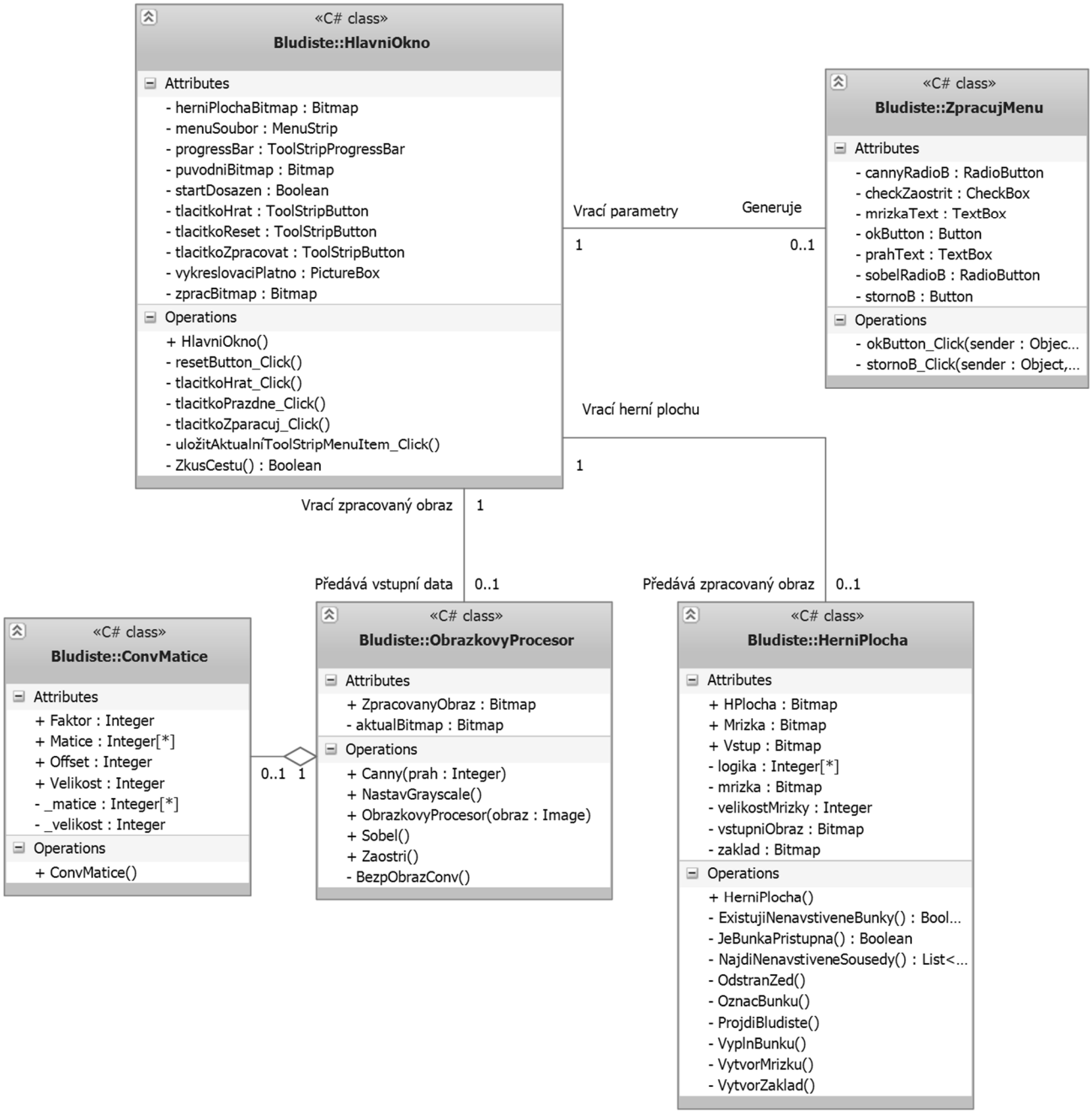
Pro dosažení nejlepšího výstupu je nutno vstupní obraz nejdříve zaostřit, převést do tónů šedi a poté z něj budou vytaženy hrany.

4.1.2 Diagram tříd

Na obrázku 12 vidíme diagram tříd, který byl navržen pro potřeby aplikace. Tento a další diagramy jsou vytvořeny přímo ve vývojovém prostředí Microsoft Visual Studio. Tento konkrétní diagram zobrazuje jakousi kostru, podle které je poté prakticky navržena fungující struktura aplikace. Nutno poznamenat že výsledná aplikace neobsahuje pouze zde zobrazené metody a atributy ale také další části, které byli přidány až během vývoje aplikace. Pro přehlednost nejsou uvedeny vstupní parametry pro konkrétní metody. Nejdříve je popsáno jaké třídy a jaké vazby aplikace obsahuje, poté jsou vysvětleny funkce jednotlivých objektů

V každé ze tříd jsou uvedeny nejdříve parametry (Attributes) a poté metody (Operations). + nebo – značí, jestli je položka veřejná nebo soukromá (soukromá znamená, že je přístupná pro

použití této třídy, veřejná pak že je k ní možno přistupovat z vnějšku). Dále následuje název položky. Text za *:* značí, o jaký datový typ se jedná, pokud zde tato část chybí, jedná se o metodu bez návratové hodnoty (přímo nevrací žádná data, pouze provede nějakou operaci) [18].



Obrázek 12: Diagram tříd

Aplikace se skládá z pěti objektů. Základní řízení aplikace probíhá v grafickém rozhraní, toto grafické rozhraní je reprezentováno třídou „HlavniOkno“. Pro nastavení parametrů pro zpracování obrazu a generování hrací plochy je k dispozici menu „ZpracujMenu“. Toto menu se

zobrazí po požadavku na zpracování obrazu v hlavním okně aplikace.

Zpracování obrazu se zajišťuje ve třídou „ObrazkovyProcessor“, tato třída získává vstupní data z grafického rozhraní a po zpracování dat je opět vrátí zpět do hlavního okna, kde se zobrazí na vykreslovacím plátně. Pro potřeby obrázkového procesoru je vytvořena třída „ConvMatice“. Tato třída je vytvořena, aby nebylo potřeba pokaždé složitě definovat novou konvoluční matici přímo v kódu procesoru.

Po zpracování obrazu grafické prostředí předá data třídě „HraciPlocha“ zde proběhne vygenerování hrací plochy a následné vrácení výsledku zpět.

V tomto stavu končí již proces zpracování dat a rozhraní čeká na pokyn ke spuštění hry. Nyní se zaměříme na obsah jednotlivých tříd, tak jak jsou zobrazeny v diagramu.

Třída HlavniOkno

Na obrázku 13 vidíme obsah této třídy. Jedná se o hlavní grafické rozhraní aplikace, v tomto okně se spouští všechny procedury potřebné pro správnou funkci aplikace. Grafické rozhraní se skládá z hlavní vykreslovací plochy, tlačítek pro ovládání aplikace a menu pro uložení a načtení obrázku. Třída obsahuje tyto parametry:

herniPlochaBitmap – V této proměnné reprezentované bitmapou je uložena vygenerovaná hrací plocha třídou „HerniPlocha“, je výhodné si tuto bitmapu uložit, pokud budu například chtít později porovnat původní obraz, zpracovaný obraz a herní plochu.

menuSoubor – Tato proměnná reprezentuje menu, ve kterém uživatel může otevírat a ukládat obrázky potřebné pro chod aplikace.

progressBar – Zobrazuje průběh zpracování dat.

puvodniBitmap – V této bitmapě je uložen vstupní obrázek, tak jak byl uživatelem otevřen.

tlacitkoHrat – Jedná se o tlačítko pro spuštění hry.

tlacitkoReset – Toto tlačítko slouží pro vyresetování aplikace do počátečního stavu.

tlacitkoZpracovat – Tlačítko pro spuštění zpracování obrazu.

vykreslovacíPlatno – Reprezentuje vykreslovací plochu na které se zobrazí požadovaný obrázek, například hrací plocha.

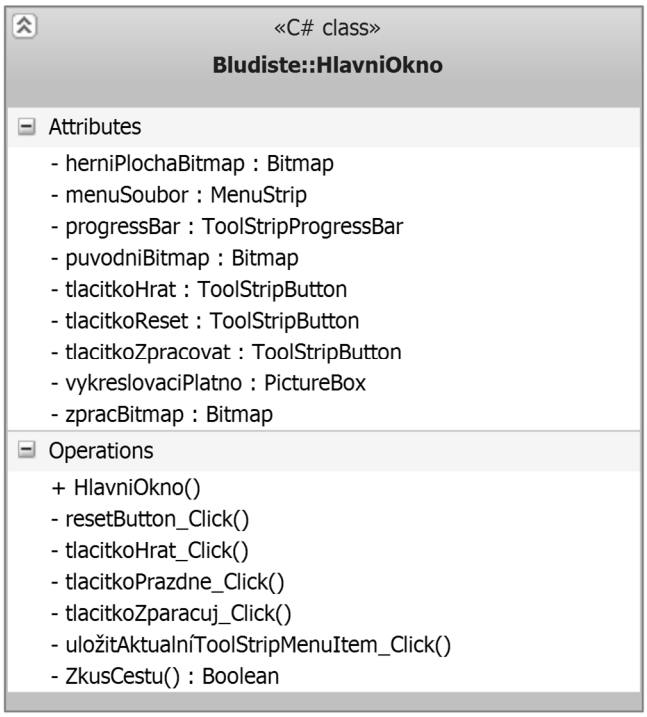
zpracBitmap – V této bitmapě je uložen zpracovaný obraz, tak jak byl vrácen obrázkovým

procesorem.

Co se týče metod (na obrázku značené jako operations), za pozornost stojí pouze metoda HlavniOkno() a ZkusCestu(). Ostatní metody které mají v názvu „Click“ nejsou prozatím důležité, jedná se pouze o reakce na kliknutí na příslušná tlačítka (viz. Parametry).

HlavníOkno() – Jedná se konstruktor této třídy. Konstruktor je metoda, kterou je vždy nutno spustit pro to, aby byla třída vytvořena, a je vždy pojmenovaná stejně jako název třídy.

ZkusCestu() – Tato metoda je aktivní po spuštění hry, zabývá se kontrolou, zda uživatel prochází bludiště správným způsobem, například že se nesnaží projít zeď.



Obrázek 13: Třída "HlavniOkno"

Třída ZpracujMenu

Toto menu se zobrazuje po kliknutí na tlačítko pro zpracování a slouží k zadání parametrů pro zpracování a vytvoření herní plochy. Konrétně se jedná o možnosti, zda má být obraz zaostřen, jak velká má být buňka v bludišti (minimální šířka chodby), pomocí jaké metody se budou detekovat hrany a jaký práh se má použít při detekci Cannyho detektorem. Okno dále obsahuje tlačítka pro potvrzení okna či pro přerušení zpracování.

Atributy tak jak jsou zobrazené na obrázku 14:

cannyRadioB a sobelRadioB – Jedná se o tlačítka určená pro volbu detektoru hran.

checkZaostrit – Zaškrtávací prvek pro volbu zda chceme obraz zaostřit nebo ne.

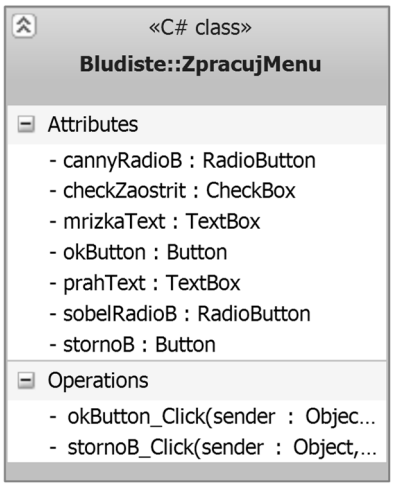
mrizkaText – Textové pole pro zadání šířky chodby bludiště.

okButton – Tlačítko pro potvrzení okna a pro spuštění zpracování.

prahText – Textové pole pro zadání práh pro Cannyho detektor hran.

stornoB – Tlačítko pro zrušení operace zpracování.

Okno obsahuje 2 metody. Metodu okButton_Click(), toto metoda se spustí po kliknutí na tlačítko okButton. Po spuštění této metody se zavře okno a spustí se zpracování dat se zadanými parametry. Metoda stornoB_Click() je triviální, jedná se pouze o zavření okna přerušení zpracování.

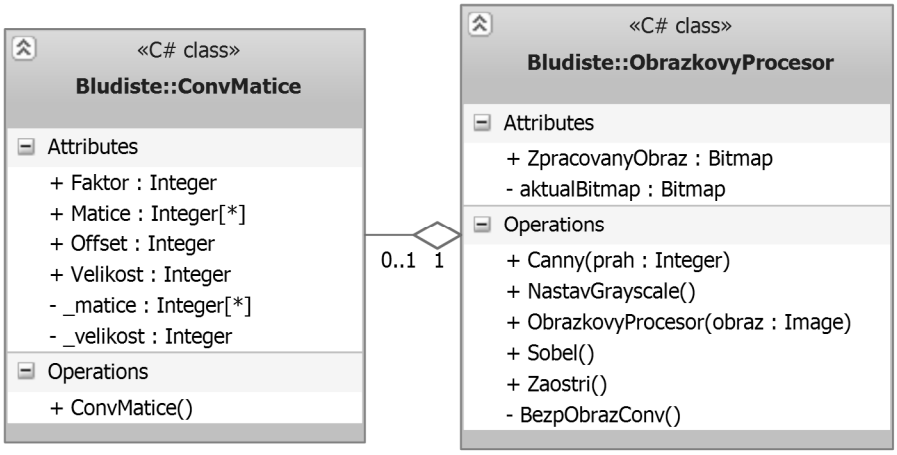


Obrázek 14: Třída "ZpracujMenu"

Třídy ObrazkovyProcesor a ConvMatice

Tyto dvě třídy jsou popsány zároveň, protože třída ConvMatice je výhradně používána procesorem. Obrázkový procesor bude přijímat bitmapu z grafického rozhraní, kterou dále zpracuje podle atributů zadaných ve „ZpracujMenu“.

Třída „ConvMatice“ definuje konvoluční matici používanou procesorem pro zaostření obrazu a v cannyho deteketoru hran. Atributy (postupně z vrchu) reprezentují nastavení a získání faktoru, offsetu, velikosti a výslednou matici podle zadání. Jedinou metodou je konstruktor.



Obrázek 15: Třída "OprazkovyProcesor"

Třída „ObrazkovyProcesor“ obsahuje tyto atributy:

ZpracovanyObraz – Tento veřejný atribut vrátí zpracovaný obraz který je uložen ve třídě.

aktualBitmap – Obsahuje dočasnou bitmapu použitou pro zpracování.

Ve třídě se nachází následující metody:

Canny() – Tato metoda představuje algoritmus pro Cannyho detektor hran.

NastavGrayscale() – Metoda která se zabývá vytvořením obrazu ve škále šedi. Tato metoda není volitelná a bude spuštěna vždy při zpracování obrazu.

ObrazkovyProcesor(obraz) – Konstruktor třídy, vstupem je bitmapa.

Sobel() – Detekce hran pomocí Sobelova operátoru.

Zaostri() – Metoda pro zaostření obrazu, tato metoda používá soukromou metodu BezpObrazConv(), která slouží k provedení bezpečné obrazové konvoluci.

Třída „HerniPlocha“

Tato třída se stará o vygenerování herní plochy po zpracování obrazu. Jedinou veřejnou metodou je konstruktor, celý proces vytvoření herní plochy proběhne zároveň s vytvořením třídy. Centrální metodou třídy je „VytvorZaklad“ a „ProjdiBludiste“. První ze zmíněných vytvoří základní plochu kombinací mřížky a vstupního obrazu, tato kombinace je pak zpracována pomocí „ProjdiBludiste“.

Obrázek 16: Třída "HerniPlocha"

Třída obsahuje tyto atributy:

HPlocha – Atribut který vrací výslednou herní plochu.

Mrizka – Pomoci tohoto atributu je možno zobrazit herní mřížku.

Vstup – Tato proměnná uchovává informace o vstupním obrazu.

Logika – Jedná o číselné vyjádření herní plochy. Toto vyjádření je použitot, protože je rychlejší než přímá práce s bitmapou. Atribut „HPlocha“ používá tuto proměnnou pro vytvoření grafické reprezentace plochy.

mrizka – Tento soukromý atribut uchovává informace o mřížce. Pro jeho zobrazení je potřeba

použít „Mřížka“.

velikostMrizky – Proměnná uchovávající informaci o velikosti chodby bludiště.

vstupniObraz – V této bitmapě je uložena informace o vstupním obrazu.

zaklad – Do této proměnné se uloží informace o vytvořeném základu pomocí „VytvorZaklad“.

Metody použité touto třídou budou:

HerniPlocha() – Konstruktor třídy

ExistujiNenavstiveneBunky() – Tato metoda kontroluje, zda v bludišti existují nezpracované buňky. Podrobný popis této a dalších metod je vysvětlen v části práce zabývající se praktickým návrhem aplikace.

JeBunkaPristupna() – Metoda kontroluej, zda zadaná buňka povolená pro zpracování.

NajdiNenavstiveneSousedy() – Kontroluje, jestli v bezprostředním okolí buňky jsou buňky přístupné pro zpracování a vrátí jejich seznam.

OdstranZed() – Odstraní zeď v zadaném směru ze zadané buňky.

VyplnBunku() – Vyplní buňku v logice hodnotami podle zadání.

ProjdiBludiste() – Jedná se o proceduru, která spustí proces vytvoření bludiště.

VytvorMrizku() – Tato metoda vytvoří mřížku, která je následně zkombinována se vstupním obrazem v metodě VytvorZaklad().

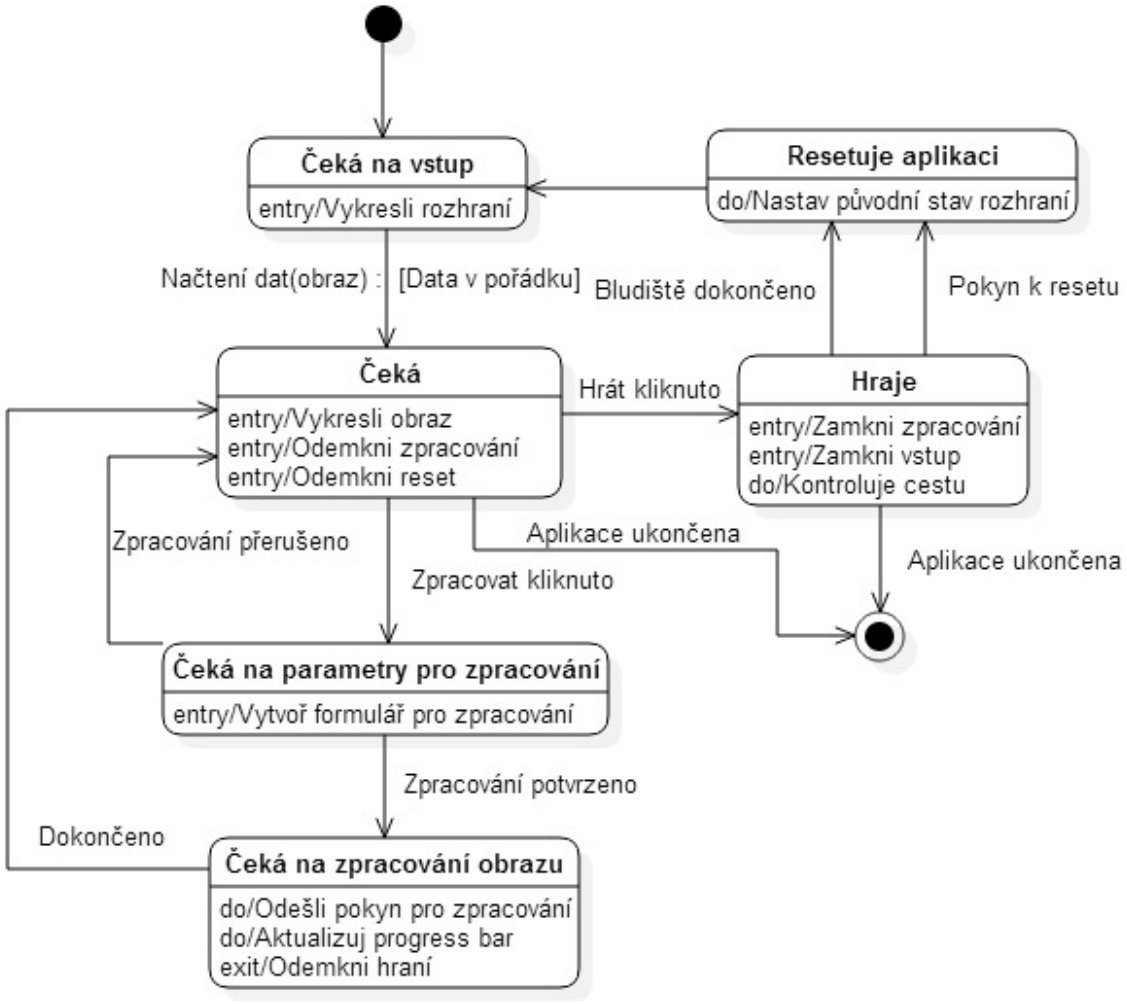
4.1.3 Stavové diagramy

Stavový diagram popisuje, jakými stavy prochází třída během své činnosti. Popsán je průběh činnosti tříd HlavniOkno, ObrazkovyProcesor a HerniPlocha.

Třída „HlavniOkno“:

Třída je hlavní řídící prvek aplikace. Třída má za úkol načítání a vykreslovaní obrazů. Dále spouští požadované procesy a ve fázi hry kontroluje, zda uživatel prochází bludiště korektním způsobem. Popíši nyní činnost třídy tak, jak je zobrazená na obrázku 17. Po spuštění aplikace se vytvoří popisovaná třída a vykreslí se grafické rozhraní, třída se tak dostane do stavu, kdy čeká na vložení vstupních dat. Po načtení obrázku, pokud jsou data v pořádku, se dostáváme

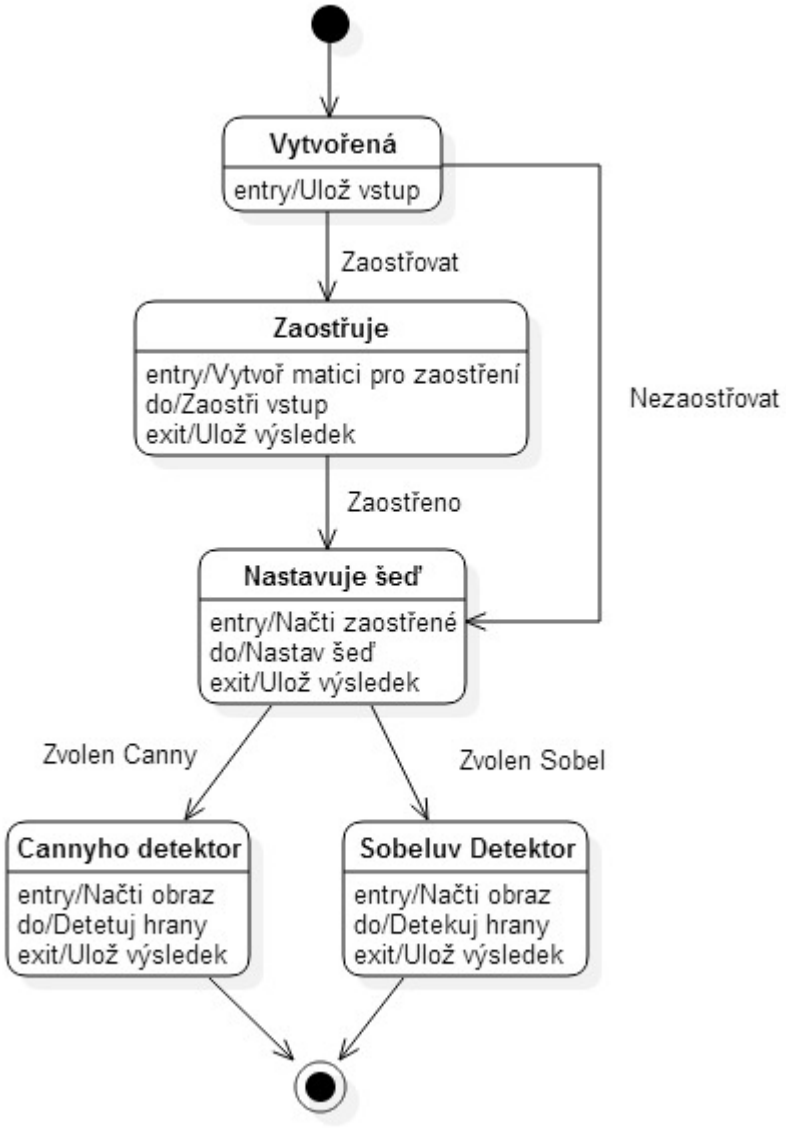
do stavu, kdy třída čeká na signál od uživatele, co má dělat. Po kliknutí na tlačítko pro zpracování obraz se vygeneruje okno pro zadání parametrů v jiné třídě. Po dobu zadávání parametrů třída čeká na potvrzení či odmítnutí zpracování obrazu. Pokud je zpracování potvrzeno, odešle se pokyn pro započnutí zpracování obrazu se zadanými parametry, a čeká se na dokončení. Během čekání třída aktualizuje progress bar. Po dokončení se odemkne tlačítko pro hraní bludiště a opět se čeká na vstup uživatele. Nyní můžeme obraz znovu zpracovat, nebo začít hrát. Po kliknutí na hrát (za předpokladu, že bylo provedeno zpracování obrazu), se dostáváme do stavu „Hraje“. Nejprve je zamknuto tlačítko pro vstup a zpracování, a poté je puštěno hraní bludiště. Během hraní je kontrolována cesta hráče. Po dokončení bludiště, nebo pokud klikneme na tlačítko pro reset se, dostaneme opět na začátek, kdy aplikace očekává vstup. Aplikaci a tím i třídu je možno ukončit během čekání a hraní.



Obrázek 17: Stavy třídy "HlavniOkno"

Třída „ObrazkovyProcesor“

Třída se stará o zpracování obraz do takové formy, aby bylo co nejjednodušší následné vytvoření herní plochy. Během vytvoření třídy se uloží vstup. Pokud bylo zvoleno, že se vstup má zaostřovat, bude nejdříve vytvořena konvoluční matice a poté bude obraz zaostřen. Výsledek bude uložen do bitmapy. Pokud bylo provedeno zaostření, případně nebylo zvoleno, obraz je nyní převeden do tónů šedi. Opět, výsledek je uložen. Nakonec je podle volby proveden buď Cannyho detekce hran nebo detekce pomocí Sobelovy masky.



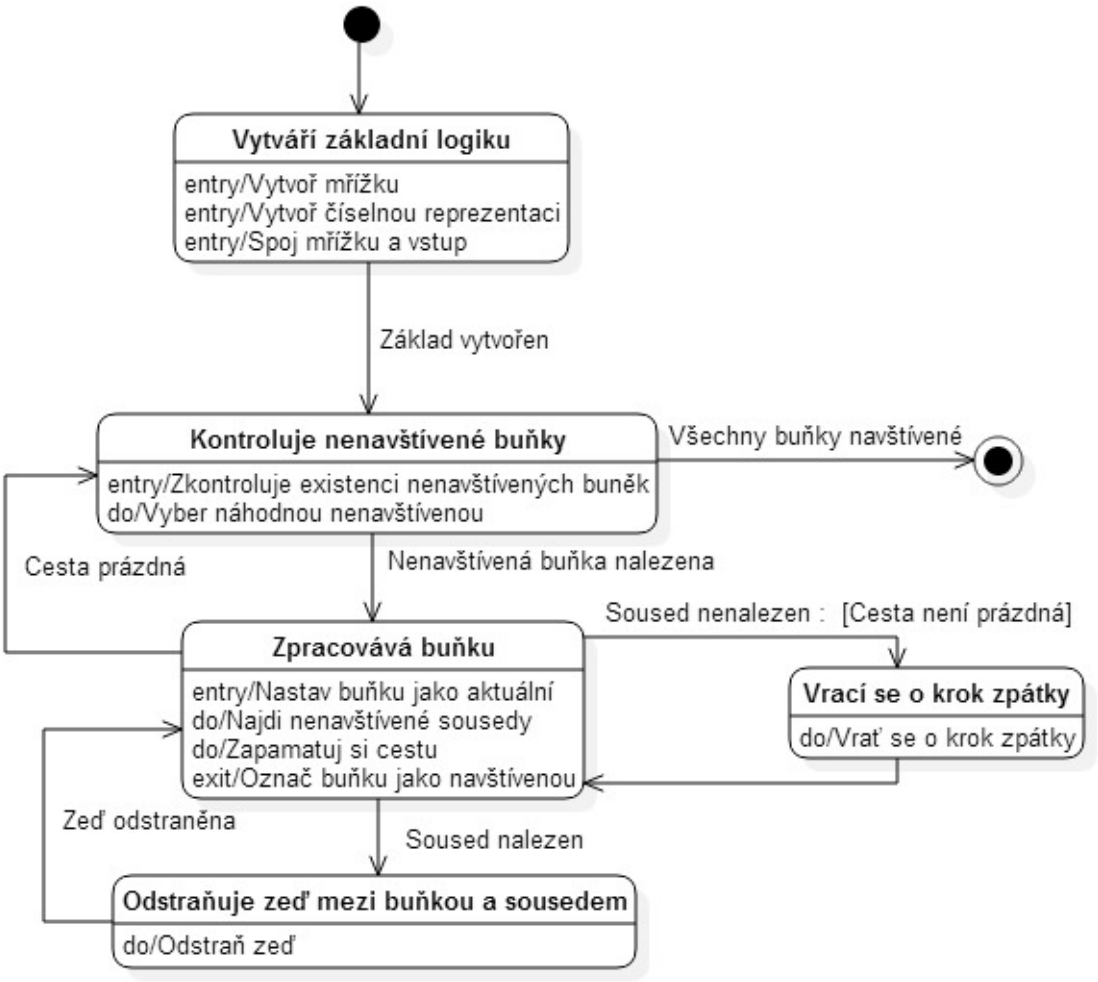
Obrázek 18: Stavy třídy "ObrazkovyProcesor"

Třída „HerniPlocha“

Třída herní plocha reprezentuje algoritmus, jakým se vytváří bludiště. Nejdříve se vytvoří

základní bitmapa a to tak, že se spojí mřížka se vstupem. Tato základní bitmapa se převede do číselné reprezentace (pro rychlejší zpracování). Nyní se začne zpracovávat bludiště, algoritmus je detailně vysvětlen v implementační části práce. V každém kroku se kontroluje, jestli v bludišti jsou nenavštívené buňky, pokud nejsou, algoritmus končí. Pokud jsou nalezeny prázdné buňky, vybere se jedna z nich náhodně a označí se jako aktuální buňka. Pokud tato buňka má nenavštívené sousedy, vybere se jeden z nich náhodně, odstraní se zeď mezi ním a náhodnou buňkou a následně se tento soused označí jako aktuální buňka. Buňka se označí jako navštívená. Předchozí buňka je uložena, aby bylo možno evidovat kroky cesty. Opět se zkontroluje, zda jsou sousedé, pokud ano opakuje se předchozí krok. Pokud ne, vracíme se o krok cesty zpátky a opět kontrolujeme sousedy. Ve chvíli kdy nejsou žádní sousedé a cesta je prázdná, je vybrána jiná náhodná nenavštívená buňka (pokud nějaká existuje).

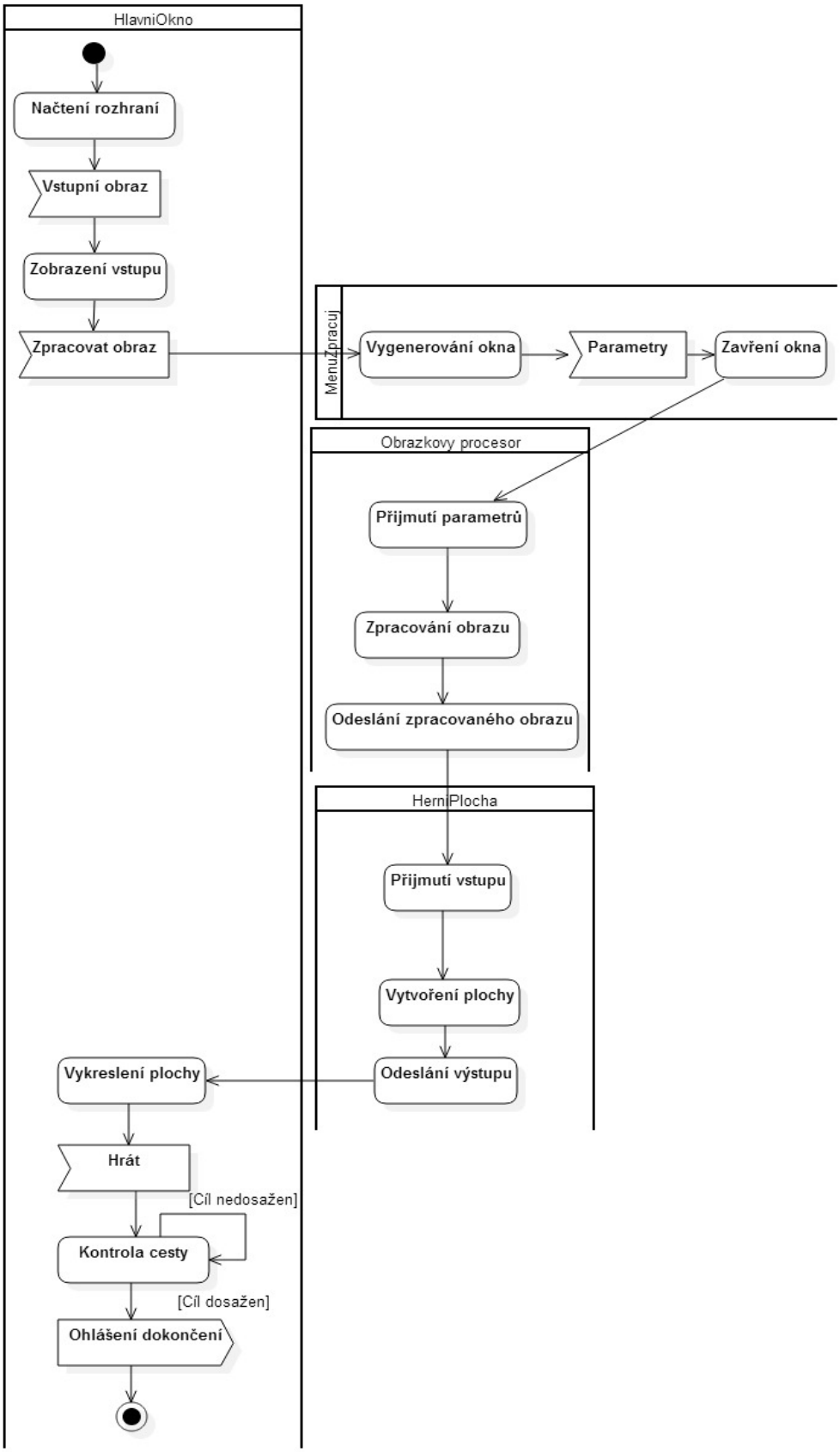
Tento algoritmu se nazývá Dept-first search pomocí zpětného trasování [15][20].



Obrázek 19: Stavy třídy "HerniPlocha"

4.1.4 Diagram aktivit

V tomto diagramu je popsán celkový pohled, jak spolu komunikují jednotlivé třídy. Po spuštění aplikace se vygenerují prvky okna, ve chvíli kdy uživatel zašle vstupní data, aplikace tyto data vykreslí. Následně se po signálu že uživatel chce vstup zpracovat, vygeneruje okno pro nastavení parametrů. Tyto parametry a vstup se předají třídě, která zpracuje obraz, ta pak předá obraz třídě herní plocha. Výsledné bludiště se vykreslí v grafickém rozhraní. Poté co uživatel odešle signál pro hraní, hlavní okno bude kontrolovat cestu uživatele. Po dosažení cíle se odešle signál potvrzující dokončení. Diagram aktivit je zobrazen na obrázku 20.



Obrázek 20: Diagram aktivit v aplikaci

4.2 Realizace aplikace

Nejdříve je popsána implementace algoritmů pro zpracování obrazu (zaostření obrazu, převedení do tónů šedi a jednotlivé detektory hran). Dále je vysvětleno, pomocí jaké logiky se tvoří herní plocha. Algoritmy jsou zobrazeny jako samostatné moduly i když v aplikaci jsou samozřejmě upravené pro potřeby jejího prostředí. Popisovaný kód je pro přehlednost zjednodušen (například o podmínky, které zabraňují přetečení pole).

4.2.1 Zaostření hran

Před vlastní implementací samotného algoritmu pro zaostření, je výhodné vytvořit si třídu, která reprezentuje masku:

```
1        public class ConvMatice
2       {
3           public int Faktor { get; set; }
4           public int Offset { get; set; }
5
6           private int[,] _matice = {  {0, 0, 0, 0, 0},
7                                       {0, 0, 0, 0, 0},
8                                       {0, 0, 1, 0, 0},
9                                       {0, 0, 0, 0, 0},
10                                      {0, 0, 0, 0, 0}
11                                  };
12
13          public int[,] Matice
14          {
15              get { return _matice; }
16              set
17              {
18                  _matice = value;
19
20                  Faktor = 0;
21                  for (int i = 0; i < Velikost; i++)
22                      for (int j = 0; j < Velikost; j++)
23                          Faktor += _matice[i, j];
24
25                  if (Faktor == 0)
26                      Faktor = 1;
27              }
28          }
29
30          private int _velikost = 5;
31          public int Velikost
32          {
33              get { return _velikost; }
34              set
35              {
36                  if (value != 1 && value != 3 && value != 5 && value != 7)
37                      _velikost = 5;
38                  else
39                      _velikost = value;
40              }
41          }
42
43          public ConvMatice()
44          {
```

```
45              Offset = 0;
46              Faktor = 1;
47          }
```

Na začátku kódu vidíme dvě proměnné, první je „offset“. Jedná se o proměnnou, která slouží k tomu, aby výsledný pixel byl v rozsahu [0, 255]. Další proměnnou je „faktor“, jedná se o sumu koeficientů matice, jeho funkcí je normalizovat výsledek. V základní podobě tato třída obsahuje matici, která vrací původní obraz. Pomocí metody int[,] Matice se deklaruje požadovaná matice, a pomocí Velikost se nastavuje její velikost [16].

Nyní, když je vytvořena třída zastupující masku pro zpracování, může být implementována třída zajišťující obrazovou konvoluci (v tomto případě bezpečnou – nehlásí chybu, pokud zpracovává okrajové pixely):

```
1    private void BezpObrazConv(ConvMatice fmat)
2           {
3               //Ochrana proti deleni nulou
4               if (fmat.Faktor == 0)
5                   return;
6
7               Bitmap tBitmap = (Bitmap)aktualBitmap.Clone();
8
9               int x, y, filterx, filtery;
10              int s = fmat.Velikost / 2;
11              int r, g, b;
12              Color docasPix;
13
14              for (y = s; y < tBitmap.Height - s; y++)
15              {
16                  for (x = s; x < tBitmap.Width - s; x++)
17                  {
18                      r = g = b = 0;
19                      for (filtery = 0; filtery < fmat.Velikost; filtery++)
20                      {
21                          for (filterx = 0; filterx < fmat.Velikost; filterx++)
22                          {
23
24                              docasPix = tBitmap.GetPixel(x + filterx - s, y + filtery - s);
25
26                              r += fmat.Matice[filtery, filterx] * docasPix.R;
27                              g += fmat.Matice[filtery, filterx] * docasPix.G;
28                              b += fmat.Matice[filtery, filterx] * docasPix.B;
29                          }
30                      }
31
32                      r = Math.Min(Math.Max((r / fmat.Faktor) + fmat.Offset, 0), 255);
33                      g = Math.Min(Math.Max((g / fmat.Faktor) + fmat.Offset, 0), 255);
34                      b = Math.Min(Math.Max((b / fmat.Faktor) + fmat.Offset, 0), 255);
35
36                      aktualBitmap.SetPixel(x, y, Color.FromArgb(r, g, b));
37
38                  }
39              }
40          }
```

V prvních 10 řádcích jsou definovány proměnné, které používá algoritmus. Proměnné x a y představují pixely, které jsou procházeny v bitmapě. Filterx a filtery jsou proměnné masky. Proměnná slouží jako počáteční bod pro cyklus procházení bitmapy (počítá se jako polovina

velikosti matice – tím se zajistí, že nikdy nepřeteče pole). R, g a n pak představují jednotlivé složky barvy pixelu. DocasPix představuje pracovní proměnnou, která se používá pro uložení barvy zkoumaného pixelu.

Samotný algoritmus funguje tak, že pro každý pixel obrázku se zpracuje okolí podle vstupní masky. Nejdříve je do proměnných r, g a b uložena hodnota pixelu po vynásobení příslušnou složkou filtru. Následně jsou z každé složky vybrána minima z maximálních hodnot podle vzorce například na řádku 34. Nakonec jsou příslušné pixely bitmapy nastaveny podle hodnot r, g a b [16].

Celá procedura se spouští pomocí této metody:

```
1           public void Zaostri()
2           {
3               ConvMatice matr = new ConvMatice();
4               matr.Velikost = 3;
5               matr.Matice = new int[3, 3] {
6                                       { 0 , -2 , 0 },
7                                       { -2 , 11 , -2 },
8                                       { 0 , -2 , 0 }
9                                   };
10
11              BezpObrazConv(matr);
12
13          }
```

Nejedná se o nic jiného, než že se nejdříve nadefinuje maska pro zaostření:

$$\begin{bmatrix} 0 & -2 & 0 \\ -2 & 11 & -2 \\ 0 & -2 & 0 \end{bmatrix}$$

Dále je spuštěna BezpOnrazConv, s parametrem této matice.

4.2.2 Převedení obrazu do škály šedé

Jedná se o velice jednoduchý algoritmus:

```
1            public void NastavGrayscale()
2           {
3               Bitmap tBitmap = (Bitmap)aktualBitmap.Clone();
4               Color c;
5               for (int i = 0; i < tBitmap.Width; i++)
6               {
7                   for (int j = 0; j < tBitmap.Height; j++)
8                   {
9                       c = tBitmap.GetPixel(i, j);
10                      byte gray = (byte)(.299 * c.R + .587 * c.G + .114 * c.B);
11                      tBitmap.SetPixel(i, j, Color.FromArgb(gray, gray, gray));
12                  }
13              }
14              aktualBitmap = tBitmap;
15          }
```

V každém kroku je načtena barva pixelu do proměnné c. Z této proměnné se získávají barevné složky, které se použijí pro výpočet hodnoty šedi pomocí vzorce [2][3]:

$$Y = 0{,}299R + 0{,}588G + 0{,}144B$$

Kde R, G, B jsou barevné složky a jejich příslušné koeficienty.

4.2.3 Cannyho detektor hran

Kód pro Cannyho detekci hran je dlouhý s opakujícími se prvky, jsou zde proto popsány pouze klíčové části kódu. Nejdříve se do polí načtou jednotlivé barevné složky vstupního obrazu (bitMap1). Tyto pole se budou jmenovat aPixR, aPixG a aPixB [20]:

```
1               for (int i = 0; i < sirka; i++)
2               {
3                   for (int j = 0; j < vyska; j++)
4                   {
5                       aPixR[i, j] = bitMap1.GetPixel(i, j).R;
6                       aPixG[i, j] = bitMap1.GetPixel(i, j).G;
7                       aPixB[i, j] = bitMap1.GetPixel(i, j).B;
8                   }
9               }
```

Dále je provedena aplikace Gaussova filtru na každou tuto složku, aby byl odstraněn případný šum, cimž je se zredukuje chyba detekece. Zde je příklad pro zelenou složku obrazu, kde i a j jsou složky vstupního obrazu:

```
1   int green = (
2                               ((aPixG[i - 2, j - 2]) * 1 + (aPixG[i - 1, j - 2]) * 4 + (aPixG[i, j
    - 2]) * 7 + (aPixG[i + 1, j - 2]) * 4 + (aPixG[i + 2, j - 2])
3                               + (aPixG[i - 2, j - 1]) * 4 + (aPixG[i - 1, j - 1]) * 16 + (aPixG[i,
    j - 1]) * 26 + (aPixG[i + 1, j - 1]) * 16 + (aPixG[i + 2, j - 1]) * 4
4                               + (aPixG[i - 2, j]) * 7 + (aPixG[i - 1, j]) * 26 + (aPixG[i, j]) * 41
    + (aPixG[i + 1, j]) * 26 + (aPixG[i + 2, j]) * 7
5                               + (aPixG[i - 2, j + 1]) * 4 + (aPixG[i - 1, j + 1]) * 16 + (aPixG[i,
    j + 1]) * 26 + (aPixG[i + 1, j + 1]) * 16 + (aPixG[i + 2, j + 1]) * 4
6                               + (aPixG[i - 2, j + 2]) * 1 + (aPixG[i - 1, j + 2]) * 4 + (aPixG[i, j
    + 2]) * 7 + (aPixG[i + 1, j + 2]) * 4 + (aPixG[i + 2, j + 2]) * 1) / 273
7                               );
```

Nyní je potřeba zjistit velikost a směr gradientu, nejdříve se tedy použije Sobelův operátor na každou barevnou složku obrazu, zde je opět příklad pro zelenou složku:

```
1   rc = aPixRn[i + hw, j + wi];
2   n_rx += gx[wi + 1, hw + 1] * rc;
3   n_ry += gy[wi + 1, hw + 1] * rc;
```

Hodnota rc zastupuje původní obraz, hodnoty gx a gy jsou složky sobelova operátoru pro jednotlivé směry. N_rx a n_ry jsou složky potřebné pro další výpočet. Spočítané hodnoty dosadím do vzorce pro velikost gradientu pro zelenou složku [20]:

```
1   gradG = (int)Math.Sqrt((n_gx * n_gx) + (n_gy * n_gy));
```

Výraz Math.Sqrt je funkce jazyka C# pro odmocninu. Nyní, když je známa velikost gradientu,

lze spočítat jeho směr:

```
1   atanG = (int)((Math.Atan((double)n_gy / n_gx)) * (180 / Math.PI));
```

Nyní je potřeba zaokrouhlit směr gradientu do 4 směrů, vertikální, horizontální a dvou diagonál. Tímto je zajištěno, že nalezené hrany budou v požadovaných směrech:

```
1                       if ((atanG > 0 && atanG < 22.5) || (atanG > 157.5 && atanG < 180))
2                       {
3                           atanG = 0;
4                       }
5                       else if (atanG > 22.5 && atanG < 67.5)
6                       {
7                           atanG = 45;
8                       }
9                       else if (atanG > 67.5 && atanG < 112.5)
10                      {
11                          atanG = 90;
12                      }
13                      else if (atanG > 112.5 && atanG < 157.5)
14                      {
15                          atanG = 135;
16                      }
```

Nicméně výsledný obraz je rozmazaný a výsledné hrany mohou být širší, než jsou ve skutečnosti, je proto potřeba porovnat sousedící pixely a vybrat z nich ten, který má hodnotu gradientu vyšší (viz teorie). V kódu algoritmu je hledání maxim implementováno takto:

```
1                       if (tanG[i, j] == 0)
2                       {
3                           if (graidientG[i - 1, j] < graidientG[i, j] && graidientG[i + 1, j] <
    graidientG[i, j])
4                           {
5                               allPixGs[i, j] = graidientG[i, j];
6                           }
7                           else
8                           {
9                               allPixGs[i, j] = 0;
10                          }
11                      }
```

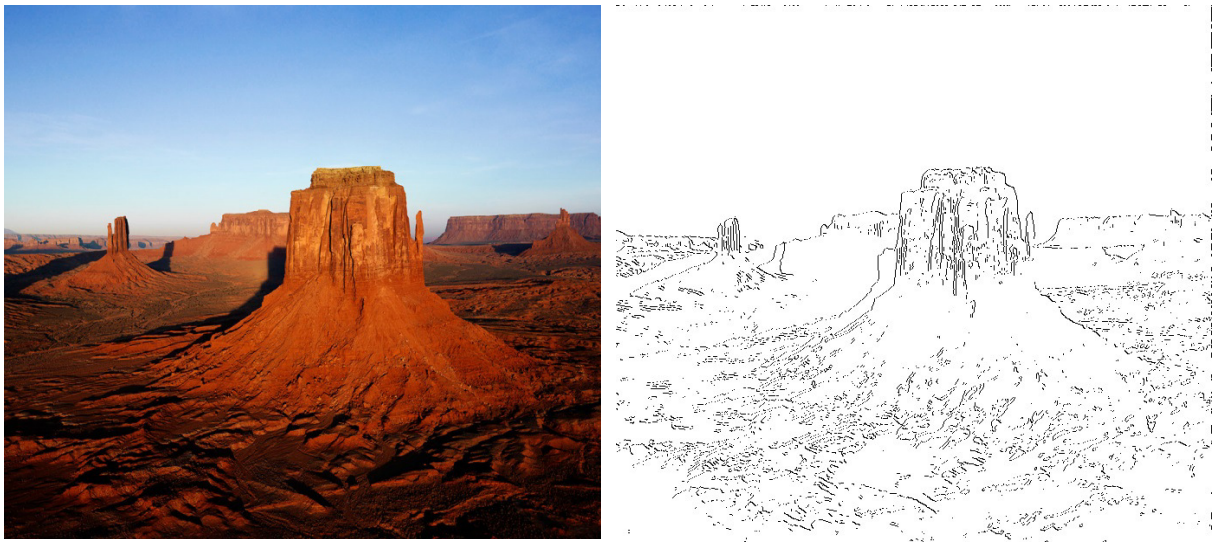
První podmínka kontroluje směr v jakém je gradient kde 0 je 0 stupňů, 1 je 45, 2 je 90 a 3 znamená 135. Další podmínka porovnává maxima, pokud sousední pixel má větší gradient, než je označen jako hrana, pokud nemá gradient větší, je tomuto pixelu ponechána původní hodnota. Nyní je aplikován práh na dosud zpracovaný obraz, touto hodnotou lze ovlivnit, s jakou přesností jsou detekovány hrany. Zde je použita jednoduchá podmínka [20]:

```
1                       if (allPixGs[i, j] > threshold)
2                       {
3                           allPixGf[i, j] = 1;
4                       }
5                       else
6                       {
7                           allPixGf[i, j] = 0;
8                       }
```

Pokud je hodnota dosud zpracovaného pixelu vyšší než práh, je označena jako 1, pokud je menší, je označena jako 0. Podle těchto hodnot se nyní vykreslí finální obraz:

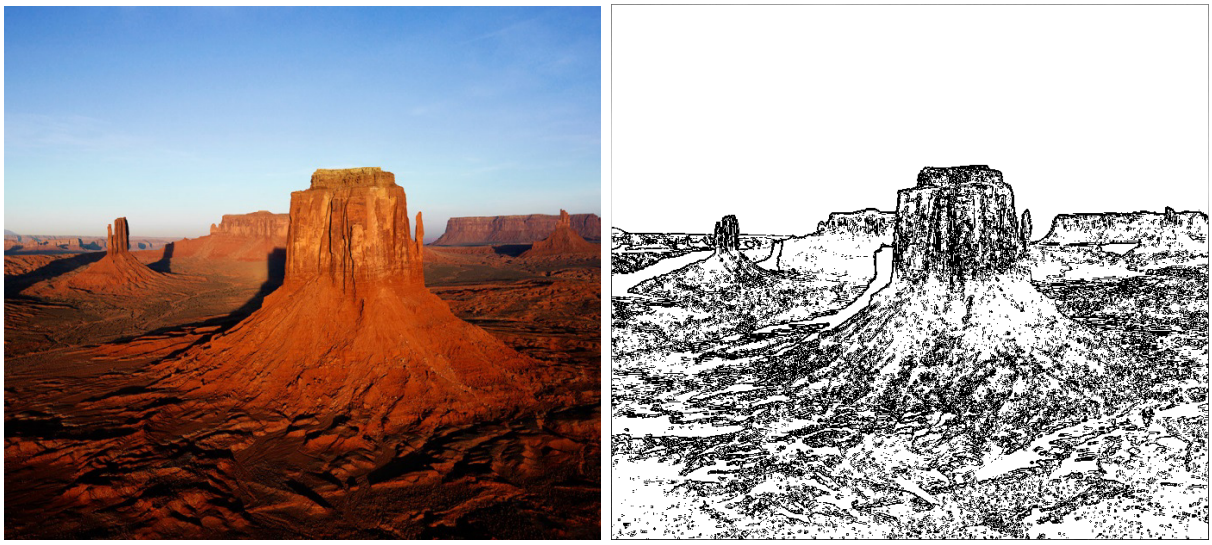
```
1                       if (allPixRf[i, j] == 1 || allPixGf[i, j] == 1 || allPixBf[i, j] == 1)
2                       {
3                           bb.SetPixel(i, j, Color.FromArgb(255, 0, 0, 0));
4                       }
5                       else
6                       {
7                           bb.SetPixel(i, j, Color.FromArgb(255, 255, 255, 255));
8                       }
```

Hrany jsou zobrazeny černou barvou a volná plocha barvou bílou. Toto rozdělení je důležité pro další algoritmy. Na obrázku 21 je zobrazen výstup z Cannyho detektor za použití prahu s hodnotou 100.



Obrázek 21: Cannyho detektor

Pro porovnání je v obrazovém procesoru implementováno i použití Sobelova operátoru jako samostatné detekce. Zdrojový kód je téměř stejný, pouze upraven tak, aby se dal spustit samostatně. Sobelův operátor jako samostatná funkce vrací poměrně dobré výsledky, nicméně je na první pohled vidět menší přesnost detekce hran (Obrázek 22). Některé hrany jsou příliš široké, výrazné pixely jsou vyhodnoceny jako hrany. Celkově Cannyho deteketor poskytuje „čistější“ výstup, navíc mohu ovlivňovat přesnost s jakou algoritmus najde hrany.



Obrázek 22: Sobelův detektor

4.2.4 Algoritmus pro vytvoření bludiště

Nyní, když je obraz zpracován, je možno přistoupit k vytvoření samotné herní plochy. Zde je použito modifikovaného algoritmu Depth-First search se zpětným trasováním [15].

Ještě před tím než započne samotná tvorba bludiště, je provedeno několik přípravných kroků. Jako první je vytvořena mřížka, která bude reprezentovat stěny bludiště.

```
1           private void VytvorMrizku(int velikostMrizky)
2           {
3               mrizka = new Bitmap(vstupniObraz.Width, vstupniObraz.Height);
4               int velikostX = velikostMrizky;
5               int velikostY = velikostMrizky;
6
7               for(int i=0;i<mrizka.Width;i+=velikostX)
8               {
9                   for (int j = 0; j < mrizka.Height; j++)
10                  {
11                      mrizka.SetPixel(i, j, Color.Black);
12                  }
13              }
14
15              for(int i=0;i<mrizka.Height;i+=velikostY)
16              {
17                  for (int j = 0; j < mrizka.Width; j++)
18                  {
19                      mrizka.SetPixel(j, i, Color.Black);
20                  }
21              }
22          }
```

Nejdříve je vytvořena prázdná bitmapa o stejné velikosti jako je vstupní obraz, dále jsou v cykly změněny pixely ve sloupci, či řádku na černou barvu tak aby vytvořily mřížku. Velikost kroku cyklu je závislá na velikosti mřížky. Takto vytvořená mřížka je nyní sloučena se vstupním obrazem. Nyní je vytvořena základní bitmapa, která je pomocí metody „VytvorZa-

klad()“ převedena do číselné reprezentace. Černá barva bude zastoupena hodnotou 1. Buňka, která obsahuje černou barvu mimo, mřížku bude označena číslem 5, což bude reprezentovat informaci, že se nejedná o stěnu bludiště ale o původní obrazovou informaci. Číslem 8 a 9 bude označen start a cíl.

Modifikovaný Depth-first search algoritmus

Algoritmus používá rekurzi a zpětné trasování. Rekurze znamená, že metoda opakovaně volá sama sebe (ať už přímo nebo nepřímo) [13]. Zpětné trasování spočívá v tomto případě v principu, že bude evidováno kterou cestou se prošlo, čímž se dosáhne zaprvé lepší kvality bludiště a za druhé se zamezí přetečení pole. Celý algoritmus je implementován v metodě „ProjdiBludiste()“ [16]:

1. Vyber náhodnou nenavštívenou buňku a označ ji jako navštívenou.
2. Opakuj, dokud jsou nenavštívené buňky a není dosažen cíl:
    a. Pokud aktuální buňka má nenavštívené sousedy:
        i. Vyber náhodného souseda
        ii. Přidej aktuální buňku do zásobníku
        iii. Odstraň stěnu buňky ve směru vybrané buňky
        iv. Nastav vybranou buňku jako aktuální
    b. Pokud zásobník není prázdný:
        i. Odeber buňku ze zásobníku
        ii. Nastav ji jako aktuální
    c. Ostatní případy:
        i. Nastav náhodnou nenavštívenou buňku jako aktuální.

Implementace algoritmu je zobrazena v tomto kódu:

```
1           private void ProjdiBludiste(int lRX, int lRY)
2           {
3               Random r = new Random();
4               Stack<int> cestaX = new Stack<int>();
5               Stack<int> cestaY = new Stack<int>();
6               logika[lRX, lRY] = 3;
7               int aktualniBunkaX = lRX;
8               int aktualniBunkaY = lRY;
9
10              cestaX.Push(aktualniBunkaX);
11              cestaY.Push(aktualniBunkaY);
12
13              while (ExistujiNenavstiveneBunky() == true)
14              {
15                  int rV = r.Next(NajdiNenavstiveneSousedy(aktualniBunkaX, aktualniBunkaY).Count);
16                  if (NajdiNenavstiveneSousedy(aktualniBunkaX, aktualniBunkaY).Count > 0)
```

```
17                  {
18                      if (NajdiNenavstiveneSousedy(aktualniBunkaX, aktualniBunkaY)[rV] == smer.Sever)
19                      {
20                          cestaX.Push(aktualniBunkaX);
21                          cestaY.Push(aktualniBunkaY);
22                          OdstranZed(aktualniBunkaX, aktualniBunkaY, smer.Sever);
23                          aktualniBunkaY = aktualniBunkaY - velikostMrizky;
24                          logika[aktualniBunkaX, aktualniBunkaY] = 3;
25                      }
26                  }
27                  else if (cestaX.Count > 0)
28                  {
29                      aktualniBunkaX = cestaX.Pop();
30                      aktualniBunkaY = cestaY.Pop();
31                  }
32                  else
33                  {
34                      int rX = (r.Next(0, (logika.GetLength(0)) / velikostMrizky) * velikostMrizky);
35                      int rY = (r.Next(0, (logika.GetLength(1)) / velikostMrizky) * velikostMrizky);
36
37                      while (logika[rX, rY] != 1)
38                      {
39                          rX = (r.Next(0, (logika.GetLength(0)) / velikostMrizky) * velikostMrizky);
40                          rY = (r.Next(0, (logika.GetLength(1)) / velikostMrizky) * velikostMrizky);
41                      }
42
43                      aktualniBunkaX = rX;
44                      aktualniBunkaY = rY;
45
46                      logika[rX, rY] = 3;
47                  }
48              }
```

V řádcích 3 až 11 jsou definovány, proměnné používané algoritmem. Je potřeba definovat nahodnou *r*, ta bude implementována pomocí datového typu *Random*. Tento typ generuje pseudonáhodná čísla pomocí algoritmu navrženém Donald E. Knuthem [13].

Dále se eviduje cesta jakou algoritmus prochází, toho je dosaženo pomocí proměnných *cestaX* a *cestaY*. Tyto proměnné jsou reprezentovány pomocí datového typu *Stack*, jedná se o pole s logikou LIFO (Last In First Out – poslední dovnitř, první ven). Dále je na řádcích 29 a 30 eviduvána aktuální buňka bludiště, do těchto proměnných se na začátku dosadí parametry metody *lRX* a *lRY*, které představují levý roh buňky v příslušném směru. Následně je buňka označena jako navštívená (v logice je navštívená buňka zastoupena číslem 3)[19].

Nyní je v každém kroku použita metoda „ExistujiNenavstiveneBunky()“, jedná se o triviální metodu, která v cyklu projde logiku a hledá buňky označené 1. Pokud takovou buňku najde, je na řádku 37 vygenerováno náhodné číslo v závislosti na počtu platných sousedů (pomocí metody „NajdiNenavstiveneSousedy(int lRX, int lRY)“, tato metoda kontroluje sousední buňky, pokud najde nenavštívenou, uloží si ji do pole). Nyní je zvolen náhodný soused (zde je pro zjednodušení zobrazena pouze volba severního souseda). Aktuální buňka je přidána do evidence cesty a dále je pomocí metody „OdstranZed(int lRX, int lRY, směr s)“ odstraněna zeď mezi aktuální buňkou a vybraným sousedem. Jako aktuální buňka je poté nastaven zvole-

ný soused a nakonec je buňka označena jako navštívená.

Pokud v předchozím kroku nebyl zvolen žádný soused a v evidenci cesty jsou hodnoty, je odstraněna z evidence vrchní hodnota (pomocí metody „.Pop()“). Odstraněná hodnota je pak nastavena jako aktuální buňka.

Pokud nebyla nalezena v evidenci žádná hodnota, ale existují nenavštívené buňky, je zvolena náhodná nenavštívená buňka a označena jako aktuální.

Pokud již nejsou žádné nenavštívené buňky, algoritmus končí.

4.2.5 Hraní bludiště

Poté co je vytvořena hrací plocha, je v grafickém rozhraní odemčeno tlačítko „Hrát“. Po kliknutí na něj se jsou zamčeny ostatní prvky grafického rozhraní kromě tlačítka „Reset“ a je aktivován herní mód aplikace.

Při každém kliknutí se provede procedura, která zajistí, že hráč začíná na startovním poli, že se nesnaží projít stěnou bludiště (jako stěna se počítá i detekovaná hrana v obrazu) a že postupně dosáhne cílového pole. Procedura se dá popsat takto:

1. Start nedosažen:
    a. Čekej na kliknutí na start
2. Start dosažen ale nedosažen cíl:
    a. Pokud je krok platný:
        i. Vykresli přímku mezi předchozí souřadnicí a kliknutou.
        ii. Oznam úspěch
        iii. Započítej krok
    b. Pokud je krok neplatný:
        i. Oznam chybu
        ii. Čekej na platný krok
3. Cíl dosažen:
    a. Vykresli přímku mezi předchozí souřadnicí a cílovou
    b. Oznam dokončení bludiště.

Kontrola platného kroku je realizována pomocí Bresenhamova algoritmu pro vykreslení přímky. I když se jedná algoritmus pro vykreslení přímky, žádná přímka se nekreslí, pouze je

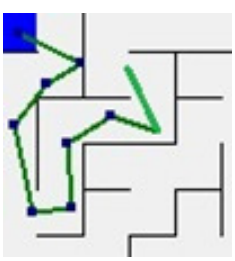
kontrolována barva každého pixelu po cestě teoretické přímky, pokud je nalezena černá barva, je krok označen jako neplatný. Modifikace algoritmu je v aplikaci implementována takto [4][17]:

```
1           private bool ZkusCestu(Point a, Point b)
2           {
3               bool platna = true;
4
5               Bitmap bm = (Bitmap)mainDrawingArea.Image;
6
7               int dx = Math.Abs(b.X - a.X);
8               int dy = Math.Abs(b.Y - a.Y);
9               int rozdil = dx - dy;
10
11              int posun_x, posun_y;
12
13              if (a.X < b.X) posun_x = 1; else posun_x = -1;
14              if (a.Y < b.Y) posun_y = 1; else posun_y = -1;
15
16              while ((a.X != b.X) || (a.Y != b.Y))
17              {
18                  int p = 2 * rozdil;
19
20                  if (p > -dy)
21                  {
22                      rozdil = rozdil - dy;
23                      a.X = a.X + posun_x;
24                  }
25                  if (p < dx)
26                  {
27                      rozdil = rozdil + dx;
28                      a.Y = a.Y + posun_y;
29                  }
30                  if(bm.GetPixel(a.X,a.Y).ToArgb()==Color.Black.ToArgb())
31                  {
32                      platna = false;
33                  }
34              }
35              return platna;
36
37          }
```

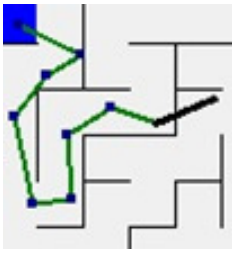
V základu je každý krok považován za platný. Nejdříve se z hrací plochy zkopíruje dočasná bitmapa (Tuto transformaci je nutno provést, protože datový typ Image, kterým je reprezentován obraz v C# nezná metodu „GetPixel()“, kterou lze získat hodnotu pixelu).

Dále jsou zjištěny na řádku 7 a 8 vzdálenosti mezi první a koncovou souřadnicí. Celkový rozdíl je pak spočítán z těchto dvou složek. Na řádku 13 a 14 je zvoleno, jestli se přímka bude vykreslovat v kladném či záporném směru, podle toho jsou dosazeny hodnoty posunu v jednotlivých souřadnicí.

Dále dokud nedosáhneme koncových souřadnici, se v každém kroku se posuneme po cestě přímky a kontrolujeme, zda momentální pozice obsahuje černou barvu.



Platný krok

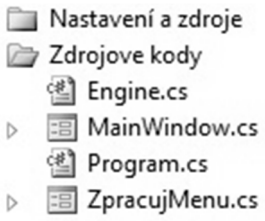


Neplatný krok

Obrázek 23: Platný a neplatný krok

4.3 Popis aplikace a jejích funkcí

Aplikace je navržena tak, aby její ovládání bylo co nejjednodušší a zároveň aby poskytovala možnost, jak ovlivňovat svojí funkčnost. Ovládacím prvkem aplikace je grafické rozhraní, bez tohoto rozhraní není možno aplikaci ovládat, nicméně všechny třídy pro zpracování obrazu a pro vytvoření hrací plochy lze použít samostatně za předpokladu, že budou zkompilovány například jako knihovna nebo vloženy do jiného projektu. Prostředí je navrženo pomocí knihovny Windows.Forms, která obsahuje veškeré ovládací prvky systému Windows. Pro zajištění kompatibility jsou v aplikaci zakázány vizuální prvky. Obsah projektu v prostředí Visual Studio je uspořádán takto:



Obrázek 24: Uspořádání projektu

Složka *Nastavení a zdroje* není pro účely práce důležitá a její obsah je proto vynechán.

Složka *Zdrojove kody* obsahuje, jak název napovídá, funkční část aplikace. V souboru *Engine.cs* jsou uložené zdrojové kódy pro zpracování obrazu a vytvoření herní plochy. Tento soubor je možné přidat do jakéhokoliv projektu a jeho třídy samostatně používat. Dále je možno ho zkompilovat jako knihovnu a poté ho používat jako jmenný prostor.

MainWindow.cs obsahuje zdrojový kód pro hlavní okno, dále také logiku pro hraní bludiště. Tato logika je bohužel závislá na tomto grafickém rozhraní a nelze ji použít samostatně.

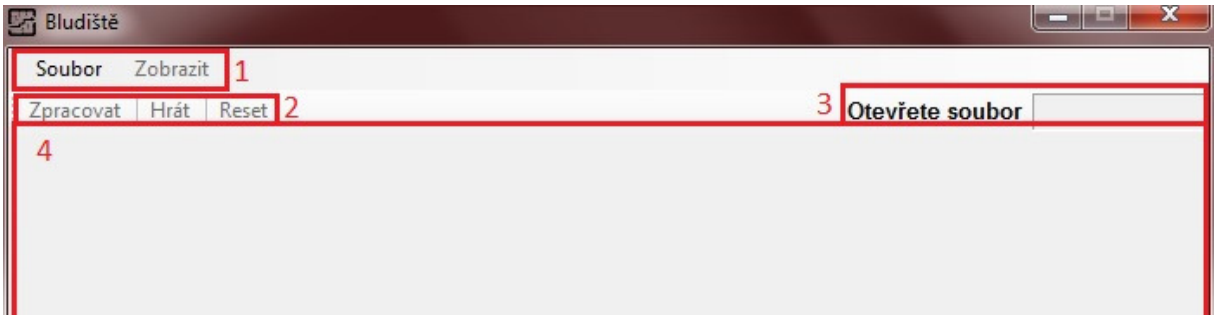
Program.cs obsahuje inicializační rutiny aplikace, prakticky se jedná pouze o spuštění hlavního okna.

ZpracujMenu.cs obsahuje kód pro zobrazení menu na zadání parametrů.

4.3.1 Popis aplikace

Aplikaci je možno spustit v operačním systému Windows, na jiných platformách je potřeba znovu zkompilovat zdrojový kód aplikace. Pro správný běh aplikace je nutno mít nainstalovaný .NET Framework alespoň verze 4.0. Windows 7 a vyšší by měli tuto verzi obsahovat v základní instalaci. Ve starších verzích systému Windows bude pravděpodobně nutné tento Framework nainstalovat manuálně.

Samotná aplikace se spustí pomocí souboru *bludiste.exe*. Pokud spuštění proběhne v pořádku, zobrazí se hlavní okno aplikace:

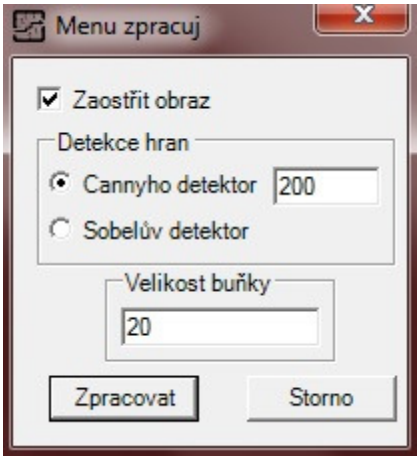


Obrázek 25: Hlavní okno

Na obrázku 24 je zobrazeno hlavní okno aplikace, červeně s příslušnými číslicemi jsou zobrazeny hlavní prvky okna. Číslo 1 představuje panel menu, a obsahuje dvě rozevírací menu: Menu soubor, které obsahuje otevření vstupu, možnost uložit aktuální obraz a také tlačítko pro vytvoření prázdného obrazu. To využijeme, pokud chceme vytvořit čisté bludiště. Dále je zde vidět menu Zobrazit, toto menu slouží přepínání mezi vstupním a zpracovaným obrazem a výslednou herní plochou.

Menu zpracuj

Číslo 2 označuje ovládací tlačítka aplikace. Tlačítko Zpracovat zobrazí menu pro nastavení parametrů:



Obrázek 26: Menu zpracuj

Zde je možno nastavit zda chceme obraz zaostřit (doporučuji nechat zaškrtnuto). Dále je možno nastavit zda chceme použít Cannyho detektor hran nebo Sobelův operátor. V případě Cannyho detektoru je možno nastavit citlivost. Aplikace je testována na práh 1 až 400, nicméně doporučuji pohybovat se v rozsahu 150 – 250, vyšší nebo nižší hodnoty mohou negativně ovlivňovat algoritmu vygenerování herní plochy. Jako poslední je možno nastavit velikost buňky bludiště s tím, že čím je hodnota nižší, tím komplexnější bludiště je vygenerováno. Velikost přesahující hodnotu 100 generuje značně triviální bludiště.

Pod číslem 2 se dále nachází tlačítko Hrát, po jeho použití se spustí herní mód aplikace (podrobněji bude popsáno dále). Tlačítko Reset vrátí aplikaci do stavu, kdy byla spuštěna.

Číslo 3 představuje stavovou část okna, zde se zobrazuje, jaká činnost momentálně probíhá a jaký je její postup.

Nakonec číslo 4 představuje vykreslovací plátno.

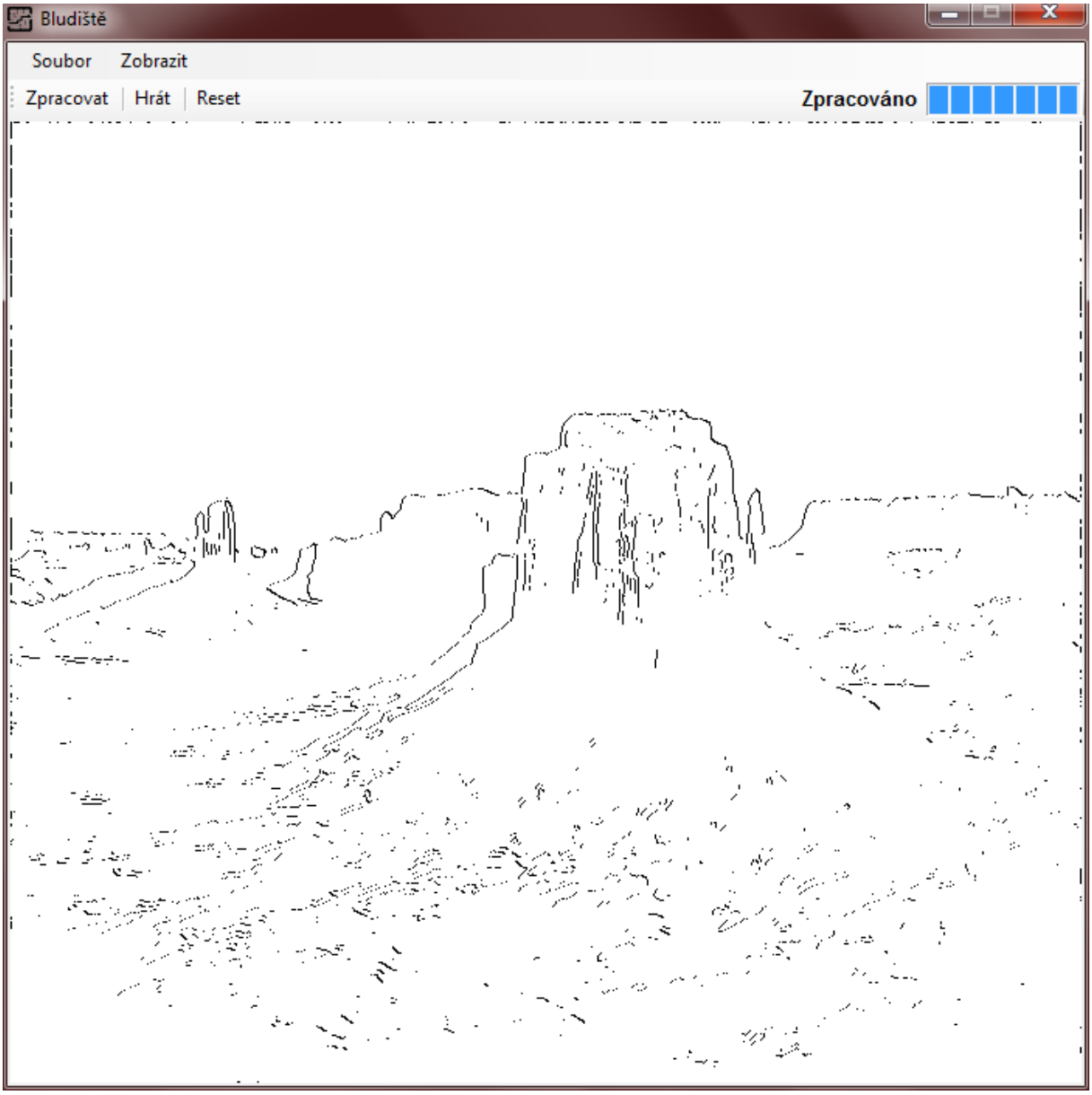
4.3.2 Funkce aplikace

Aplikace se dá funkčně rozdělit do dvou hlavních částí, do části zpracování obrazu a do herního módu. Postupná činnost aplikace bude znázorněna na tomto obrázku:

Obrázek 27: Počáteční stav

Zpracování obrazu

Obraz pracujeme pomocí těchto parametrů: Bude zaostřen, použijeme Cannyho detektor s prahem 150 a velikost buňky bludiště 30. Po proběhnutí zpracování a zobrazení zpracovaného obrazu, získáme zpracovaný obraz, který vypadá takto:

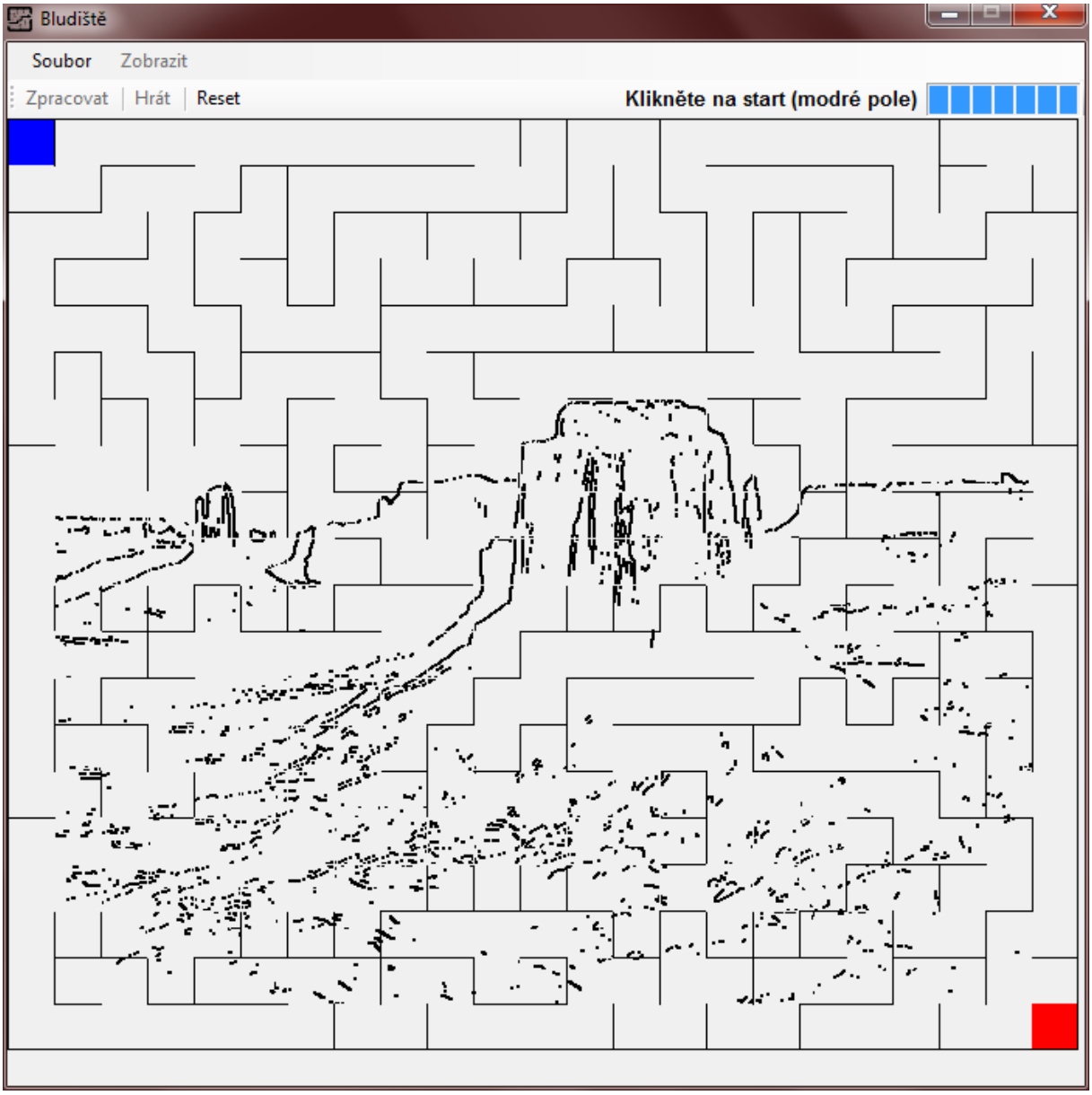


Obrázek 28: Zpracovaný obraz

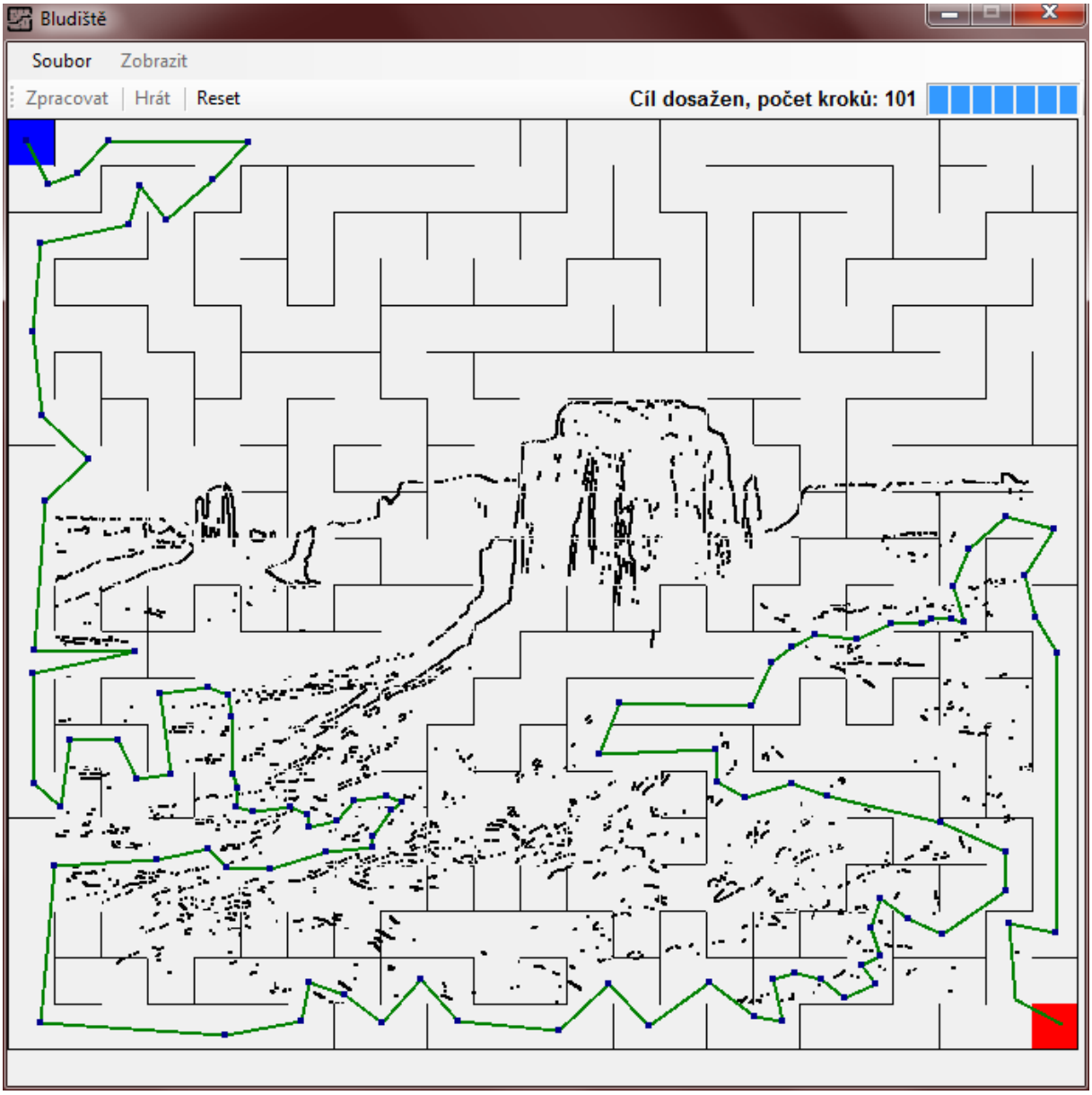
Je vidět, že detektor našel důležité hrany ale také shluk pixelů (hlavně ve spodní části obrazu) které příliš neodpovídají hranám. Jejich hustota je však zanedbatelné a neměla by příliš ovlivnit vytvoření herní plochy.

Herní mód

Na obrázku 28 je vidět vygenerovaná herní plocha v okamžiku klinutí na tlačítko Hrát. V této fázi je potřeba kliknout na modrou buňku představující start (vždy se nachází v levém horním rohu). Poté již můžeme procházet bludiště, opakovaným klikáním do plochy se přesouváme po požadovaných pozicích v bludišti (za předpokladu, že se jedná platný krok). Bludiště po jeho dokončení je vidět na obrázku 29.



Obrázek 29: Bludiště na počátku hraní



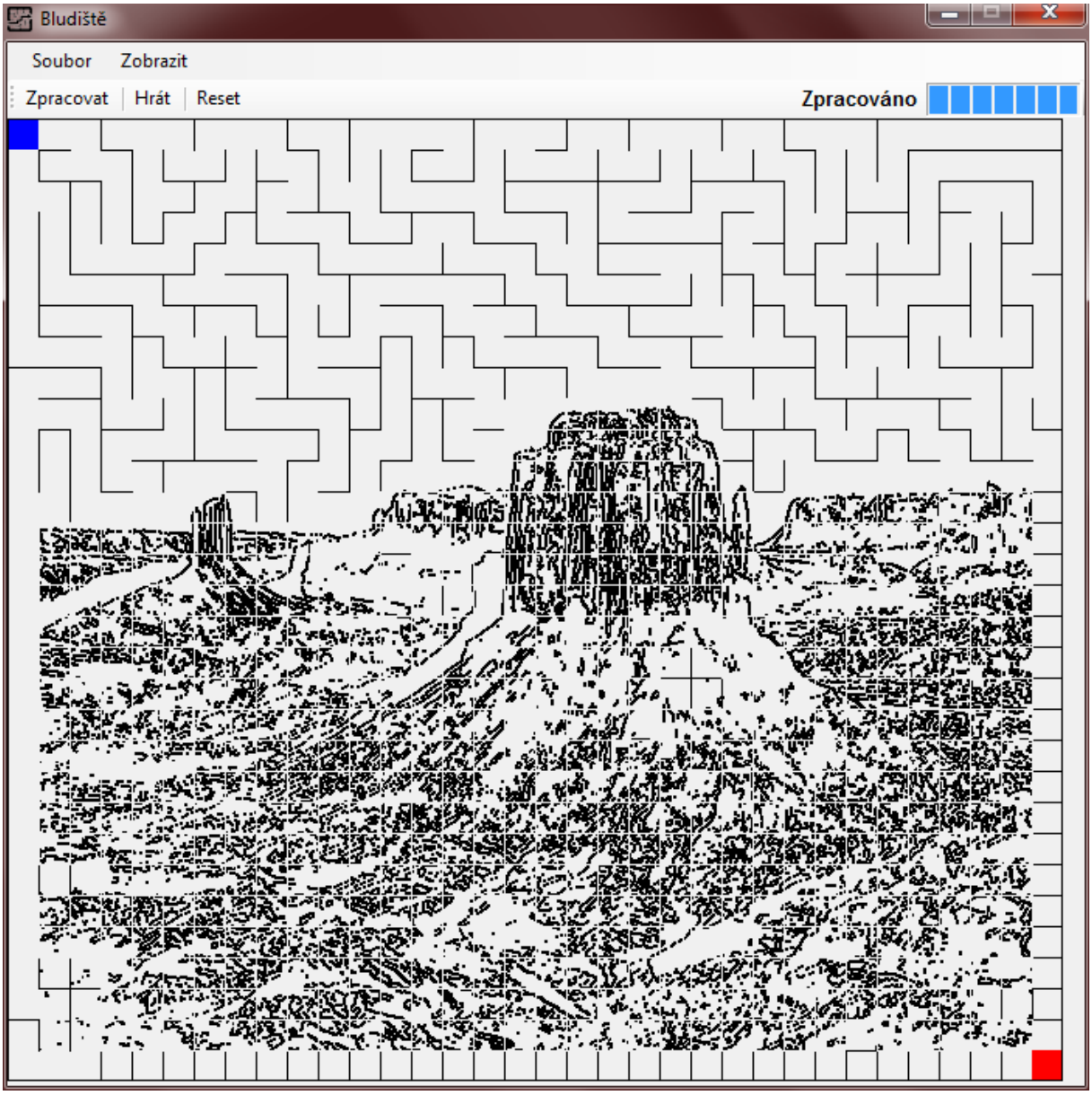
Obrázek 30: Bludiště na konci hraní

Nedostatky aplikace

Mezi první zřetelné nedostatky je poměrně vysoká výpočetní náročnost zpracování obrazu. To je způsobeno částečně tím, že aplikace pracuje v prostorové oblasti místo frekvenční. Hlavní příčinou nízkého výkonu je však používání metod GetPixel() a SetPixel() u bitmapy. Toto by se dalo obejít pomocí požití tzv. „unsafe“ kódu [16]. Unsafe kód představuje množinu metod, které jsou jazykem C# sice podporované ale nejsou doporučené z různých důvodů. V tomto případě by se dali pomocí metody UnlockBits() odemknout bity bitmapy a tak přímo přepisovat hodnoty pixelů. Aplikace však pracuje se statickým obrazem a není proto příliš nutné pou-

žívat tento kód.

Dalším nedostatkem je, že ačkoliv algoritmus pro vytvoření bludiště je nastaven tak aby vždy existovala cesta ze startu do cíle, může selhat. Selhání nastává, pokud je použit příliš nízký práh u Cannyho detektoru. Nalezne se příliš vysoká hustota hran u okrajů obrazu a algoritmus nenajde cestu. Tato závada bohužel nebyla v době odevzdání práce opravena. Selhání algoritmu je zobrazeno na obrázku 30.



Obrázek 31: Selhání algoritmu

5. Závěr

Práce byla věnována problematice zpracování digitálního obrazu a jako cíl si kladla vytvoření aplikace, která využívá vstupní obraz vy vytvoření herní plochy.

Výsledná aplikace byla nejdříve navržena pomocí UML. Zde bylo použito diagramu tříd, který zobrazil strukturu této aplikace. Dále byly vysvětleny prvky zobrazených tříd. Chování jednotlivých tříd bylo vysvětleno pomocí stavových diagramů a chování aplikace jako celku bylo zobrazeno pomocí diagramu aktivit. Po formálním návrhu byla aplikace implementována pomocí jazyka C#. Pro zpracování vstupního obrazu se využívá nejdříve vysokofrekvenčního filtru, který obraz zaostří. Obraz je dále převeden do odstínů šedi, což zjednoduší následnou detekci hran, kterou je možno provést pomocí Cannyho detektoru hran nebo samostatného Sobelova operátoru. Cannyho detektor jsem zvolil, protože poskytuje kvalitní výsledky a také proto, že je možno ovlivňovat jeho rozlišovací schopnosti. Samostatný Sobelův operátor byl implementován, aby bylo možno porovnat výsledky s Cannyho detektorem. Zpracovaný obraz poté využívá Depth-first search algoritmu pro vygenerování bludiště. Celá aplikace se ovládá pomocí grafického rozhraní, které umožnuje různé nastavení parametrů pro zpracování obrazu a pro tvorbu bludiště.

Při testování aplikace jsem zjistil, že algoritmus pro vytvoření selhává u příliš hustých obrazů při Cannyho detekci s hodnotou prahu menší než 50 až 75. Dalším nedostatkem aplikace je poměrně pomalé zpracování obrazu. Výkon aplikace by bylo možno zvýšit zpracováním obrazu ve frekvenční oblasti, případně použít jinou reprezentaci obrazu než je bitmapa.

Zdrojové kódy pro zpracování obrazu a vytvoření bludiště lze použít jako samostatné moduly v jiných aplikacích. Aplikaci lze v budoucnu rozšířit například o další metody detekce hran nebo možnost používat uživatelem zvolené konvoluční masky. Dále je aplikaci možno rozšířit o více parametrů, kterými by uživatel mohl ovlivňovat vzhled bludiště.

6. Seznam použitých zdrojů

1. KLIMEŠOVÁ, Dana. *Geografické informační systémy a zpracování obrazů*. Vyd. 2., 2. dotisk [i.e. 3. vyd.]. V Praze: Česká zemědělská univerzita v Praze, 2008, 92 s. ISBN 978-80-213-1933-2.
2. PRATT, William K. *Digital image processing*. 4th ed., Newly updated and rev. ed. Hoboken, N.J.: Wiley-Interscience, c2007, xix, 782 p., [4] p. of plates. ISBN 978-047-1767-770.
3. JAHNE, Bernd. *Digital Image Processing*. 6th ed. 2005 edition. Heidelsberg: Springer, 1997. ISBN 978-3540240358
4. ŽÁRA, Jiří, Bedřich BENEŠ a Petr FELKEL. *Moderní počítačová grafika*. Vyd. 1. Praha: Computer Press, 1998, xvi, 448 s. ISBN 80-722-6049-9.
5. Vector Graphics Overview. *MSDN magazine* [online]. San Francisco, CA: CMP Media Inc., c2000-2014 [cit. 2014-02-07]. Dostupné z: http://msdn.microsoft.com/en-us/library/t5c9b4dt(v=vs.110).aspx
6. *Úvod do geografických informačních systémů: Datové modely používané v GIS pro ukládání prostorových dat* [online]. 2010 [cit. 2014-02-07]. Dostupné z: http://www.gis.zcu.cz/studium/ugi/elearning/msgisu02s04cz/default.htm
7. SOJKA, Eduard. *Digitální zpracování a analýza obrazů*. 1. vyd. Ostrava: VŠB - Technická univerzita, 2000, 133 s. ISBN 80-707-8746-5. Dostupné z: http://www.cs.vsb.cz/licev/skripta_dzo.pdf
8. Fourierova transformace ve 2D. VUT. *Vysoké učení technické v Brně, FEKT* [online]. 2009 [cit. 2014-02-09]. Dostupné z: http://www.uamt.feec.vutbr.cz/~richter/vyuka/0910_mpov/tmp/integral_tr_2DFT.html.en
9. JÄHNE, Bernd. *Practical handbook on image processing for scientific and technical applications*. 2nd ed. Boca Raton: CRC Press, 2004, xiii, 610 s. ISBN 08-493-1900-5.
10. E-learning: Segmentace obrazu. *Technická univerzita v Liberci 2000/2001* [online]. 1. vyd. Liberec: TU v Liberci, 2001 [cit. 2014-02-21]. Dostupné z: http://e-learning.tul.cz
11. KRIZHANOVSKY, Andrew. Pavlovsk park. In: [online]. 1. vyd. 2005-2006 [cit. 2014-02-21]. Dostupné z: http://en.wikipedia.org/wiki/File:Pavlovsk_Railing_of_bridge_Yellow_palace_Winter.jpg
12. OBJECT MANAGEMENT GROUP, Inc. *Unified Modeling Language (UML)* [online]. 1997 [cit. 2015-02-08]. Dostupné z:http://www.uml.org/
13. GLYNN, Jay, Karli WATSON, Morgan SKINNER, ROBINSON, Christian NAGEL, K.Scott ALLEN, Ollie CORNES, Zach GREENVOSS a Burton HARVEY. *C#: Programujeme profesionálně*. 1. vyd. Brno: Computer Press, 2003, xxx, 1130 s. Progra-

mujeme profesionálně. ISBN 80-251-0085-5.

14. PRAKASH, Saleth. Image Processing using C#. *CodeProject* [online]. 2009 [cit. 2015-02-14]. Dostupné z:http://www.codeproject.com/Articles/33838/Image-Processing-using-C
15. Think Labyrinth: Maze Algorithms. *Astrolog.org* [online]. 2011 [cit. 2015-03-12]. Dostupné z: http://www.astrolog.org/labyrnth/algrithm.htm#perfect
16. Creating an Image Processing Library with C#. *Studentguru.gr* [online]. 2011 [cit. 2015-03-12]. Dostupné z: http://studentguru.gr/b/jupiter/archive/2009/10/14/creating-an-image-processing-library-with-c-part-1
17. The Bresenham Line-Drawing Algorithm. FLANAGAN, Colin. *Helsinky.fi* [online]. 2001 [cit. 2015-03-17]. Dostupné z: http://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html
18. ARLOW, Jim a Ila NEUSTADT. UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky. Vyd. 1. Překlad Bogdan Kiszka. Brno: Computer Press, 2007, 567 s. ISBN 978-80-251-1503-9.
19. Maze generation. *Institute of Computer Science University of Wrocław* [online]. 2010 [cit. 2015-03-25]. Dostupné z:http://www.ii.uni.wroc.pl/~wzychla/maze_en.html
20. Canny Edge Detection. *Code Project* [online]. 2013 [cit. 2015-03-25]. Dostupné z:http://www.codeproject.com/Articles/93642/Canny-Edge-Detection-in-C

7. Seznam obrázků