VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GENEROVÁNÍ MODELŮ DOMŮ PRO OPEN STREET MAPY

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE | ROMAN GALACZ
AUTHOR

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GENEROVÁNÍ MODELŮ DOMŮ PRO OPEN STREET MAPY

BUILDING MODEL GENERATOR FOR OPEN STREET MAPS

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE ROMAN GALACZ
AUTHOR

VEDOUCÍ PRÁCE Ing. LUKÁŠ POLOK
SUPERVISOR

BRNO 2011

Abstrakt

Tato práce se zabývá získáním dat z map poskytovaných projektem OpenStreetMap a následným převodem těchto dat z formátu zeměpisné šířky a délky do kartézské soustavy souřadnic. Dále popisuje rozpoznávání domů v zástavbách, které se na stažené mapě nacházejí. Pro demonstraci výsledků tohoto rozpoznávání je vytvořen program, který vymodeluje 3D geometrii domů a také vytvoří terén, kde dané domy leží. Vygenerovaný model je zobrazen pomocí grafické knihovny OpenGL.

Abstract

This work concerns obtaining data from the maps provided by the project OpenStreetMap. The data are converted from the format of geographical latitude and longitude to the Cartesian coordinate system. This work also concerns building type recognition in build-up area which are situated on the downloaded map. Part of the work is a demonstration application which is able to model 3D geometry of the buildings, based on the results of the recognition algorithms and also creates a terrain in which these buildings are situated. Generated model is displayed using the OpenGL graphics library.

Klíčová slova

OpenStreetMap, GIS, mapy, analytická geometrie, strojové učení, SVM, rozpoznávání domů, OpenGL

Keywords

OpenStreetMap, GIS, maps, analytic geometry, machine learning, SVM, buildings recognition, OpenGL

Citace

Galacz Roman: Generování modelů domů pro Open Street Mapy, bakalářská práce, Brno, FIT VUT v Brně, 2011

Generování modelů domů pro Open Street Mapy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Lukáše Poloka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Roman Galacz
17. května 2011

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce panu Ing. Lukáši Polokovi za jeho rady a ochotu při konzultacích práce a za poskytnutí některých zdrojových kódů. Děkuji také své rodině za podporu.

Obsah

Úvod

Každý se již určitě setkal s nějakými mapami a to buď v tištěné formě, nebo v elektronické podobě. V dnešní době patří mapy k neodmyslitelné součásti našeho života. Využíváme je na cestách, při plánování cesty, jako podklady k navigaci a v mnoha dalších případech, které si čtenář jistě domyslí. Zatímco vytištěná mapa zobrazuje pouze území určité velikosti, elektronické mohou mapovat prakticky celý svět, ke kterému tak máme přístup z jednoho místa. U těchto elektronických map se dá plynule měnit měřítko mapy a tím si daný úsek na mapě přiblížit nebo oddálit. To je možné zejména díky tomu, že tyto mapy jsou většinou vektorové, tedy složené z geometrických primitiv a při každé změně měřítka se znova přepočítávají. Navíc se již do těchto map zanášejí takové detaily, jako jsou např. půdorysy domů.

Jedním z představitelů elektronických map je i projekt OpenStreetMap, který poskytuje geografické data celého světa. Data mohou být používaná bez právních omezení a každý má možnost tato data libovolně zpracovávat a volně s nimi nakládat. V OpenStreetMap jsou zmapovány především silniční sítě, lesy, vodní plochy. Objevují se zde ale i půdorysy domů, čehož využijeme v této práci.

Cílem této bakalářské práce je na základě dat získaných z mapových podkladů OpenStreetMap rozpoznat, o jaký druh zástavby se jedná, pokud data obsahují informace o domech. Pro tento účel byl navržen nový vektor příznaků pro rozpoznáváni daného typu zástavby, což bude popsáno dále. Podle výsledků rozpoznávače je generována 3D geometrie budov. Z ostatních dat bude vytvořen 2D terén. Vše je nakonec zobrazeno pomocí grafické knihovny OpenGL 3.0, se kterou se pracuje ve forward-compatible módu. Celá aplikace je napsána v jazyce C a C++.

Úvodní část textu se zabývá vektorovou algebrou a dalšími teoretickými záležitostmi, jako je např. práce s polygony, aby mohl být generován 2D terén nebo problematiku strojového učení, což je dobré pochopit pro orientaci v dalších částech práce. Následuje kapitola zaměřená na získání dat určených ke zpracování. V další kapitole bude popsáno rozpoznávání domů. Kapitola 4 je věnována vytvoření polygonálních modelů, které se budou vykreslovat pomocí knihovny OpenGL. Kapitola 5 se zabývá základním nastavením knihovny OpenGL pro vykreslování. Předposlední kapitola shrnuje dosažené výsledky a v poslední jsou prezentovány možnosti dalších rozšíření této práce.

1 Teoretický rozbor

Jelikož jednou z částí bakalářské práce je vytvoření scény pro zobrazení, budeme se v této kapitole zabývat teorií vektorové algebry, popíšu použité algoritmy pro práci s polygony, ořezávání polygonů a jejich triangulaci. Také si zde uvedeme základní informace o mapách, o geografickém informačním systému (GIS) a o projektu OpenStreetMap. Závěr kapitoly je věnován SVM (Support Vector Machines), klasifikaci a nakonec i grafické knihovně OpenGL.

1.1 Vektorová algebra

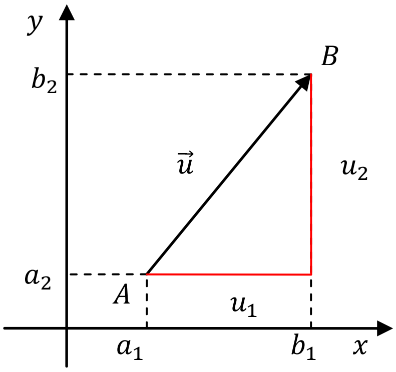
V této podkapitole se budeme zabývat vektory v analytické geometrii a to vektory v rovině, protože je ve velké části bakalářské práce využijeme při operacích s polygony, které provádíme v kartézské soustavě souřadnic roviny dané osami x a y. Popíšeme si i vektory v prostoru, které se uplatní především při práci s OpenGL. V prostoru přibude ke kartézské soustavě souřadnic tvořené osami x a y ještě další osa z. Všechny osy jak v rovině, tak v prostoru jsou na sebe kolmé.

1.1.1 Vektor

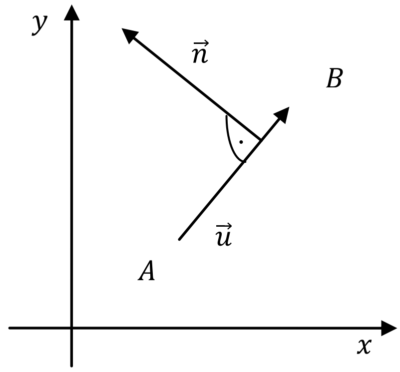
Vektor je orientovaná úsečka, která má svůj směr a velikost. Úsečka je nazývána jako orientována proto, protože je zadána svým počátečním a koncovým bodem. Nejčastěji se značí písmeny u a v, nad kterými je šipka.

Na obrázku 1.1 můžeme vidět vektor $\vec{u}$ v rovině, který má počátek v bodě $A[a_1, a_2]$ a končí v bodě $B[b_1, b_2]$. Na tomto obrázku jsou také znázorněny složky vektoru u_1 a u_2, jejichž hodnotu vypočítáme podle vzorců $u_1 = b_1 - a_1$, $u_2 = b_2 - a_2$. Výsledná podoba vektoru je tedy $\vec{u} = (u_1, u_2)$ [1]. Chceme-li z tohoto směrového vektoru $\vec{u}$ vypočítat normálový vektor $\vec{n}$, který je kolmý na tento směrový vektor, zaměníme složky u_1 a u_2 a jednu složku vynásobíme -1. Normálový vektor tedy bude $\vec{n} = (u_2, -u_1)$. Kolmost vektorů ilustruje obrázek 1.2.

V prostoru k výše uvedenému vektoru $\vec{u}$ přibude ještě jedna složka u_3, která se vypočítá stejně jako výše uvedené za předpokladu, že máme body $A[a_1, a_2, a_3]$ a $B[b_1, b_2, b_3]$, $u_3 = b_3 - a_3$ a výsledný vektor je $\vec{u} = (u_1, u_2, u_3)$.



Obrázek 1.1: Ukázka vektoru v rovině



Obrázek 1.2: Ukázka směrového a normálového vektoru v rovině

1.1.2 Operace s vektory v rovině

Zde popsané operace využijeme především u polygonů, kterým je věnována část 1.2. Jednou z důležitých operací je výpočet úhlu svíraného dvěma vektory. Nejdříve si ale vyjádříme délku jednoho vektoru a skalární součin dvou vektorů.

Pro výpočet délky využijeme obrázku 1.1, kde již umíme vypočítat vektor $\vec{u}$, tedy známe jeho složky u_1 a u_2. Poté dle znalostí Pythagorovy věty [2] odvodíme velikost vektoru, rovnice (1).

$$|\vec{u}| = \sqrt{u_1^2 + u_2^2} \qquad (1)$$

Skalární součin se provádí nad dvěma vektory a značí se tečkou. Mějme vektor $\vec{u}$ a vektor $\vec{v}$, pak se skalární součin spočte jako v rovnici (2). Je-li skalární součin roven nule, jsou vektory na sebe kolmé, viz. obrázek 1.2.

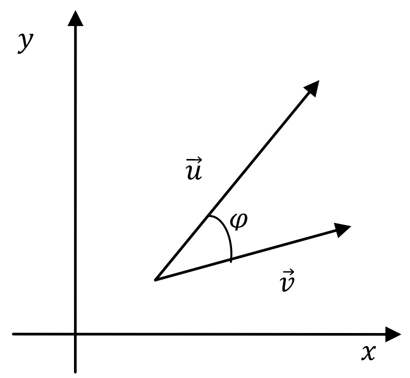
$$\vec{u} \cdot \vec{v} = u_1 v_1 + u_2 v_2 \qquad (2)$$

Nyní si ukážeme jak vypočítat úhel mezi vektory na základě výše uvedených operací. Dva vektory v rovině mohou svírat úhel pouze v rozmezí 0 až π. Vyjde-li úhel vetší než π, musíme výsledný úhel odečíst od 2 π. Výpočet úhlu svíraného dvěma vektory vidíme v rovnici (3), ve které se objevuje výpočet délky vektoru a skalární součin z rovnic (1) a (2). Tento úhel je také znázorněn na obrázku 1.3.

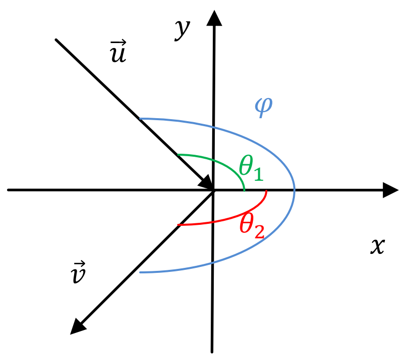
Hodně ale také budeme potřebovat vypočítat úhel svíraný dvěma stranami polygonu, který může být větší než π. Protože u polygonu víme, jak na sebe jednotlivé vektory hran navazují a konkrétně v této práci předpokládáme, že tyto vektory jsou zadány proti směru hodinových ručiček, můžeme vypočítat úhel v rozmezí 0 až 2π pomocí funkce $atan2$. Tato funkce je součástí matematické knihovny většiny programovacích jazyků. Funkci jsou předány dva parametry vektoru u_1 a u_2, kdy prvním je u_2, tedy $\varphi = atan2(u_2, u_1)$. Výsledný úhel je úhel, který vektor svírá s kladnou části osy x a je v rozmezí od -2π do 2π [3]. Takto si spočítáme oba úhly, které svírají jednotlivé vektory s osou x a odečteme je od sebe a to tak, že odečítající úhel je ten, který náleží prvnímu vektoru. Jak již bylo zmíněno výše, je potřeba vědět, jak na sebe vektory navazují. Na obrázku 1.4 vidíme, že vektor $\vec{u}$ bude svírat s osou x kladný úhel θ_1, naopak vektor $\vec{v}$ svírá s osou x záporný úhel θ_2 [4]. Pokud výsledný úhel φ vypočítaný podle rovnice (4) je větší než π anebo je menší než $-\pi$, odečteme od něj, respektive přičteme k němu 2π. Tím dostaneme úhel v rozmezí $-\pi$ až π a nakonec k tomuto úhlu přičteme π a dostaneme výsledný úhel svíraný dvěma vektory jdoucími proti směru hodinových ručiček v rozsahu 0 až 2π.

$$cos\varphi = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| \cdot |\vec{v}|} \tag{3}$$

$$\varphi = atan2(v_2, v_1) - atan2(u_2, u_1) \tag{4}$$



Obrázek 1.3: Dva vektory svírající úhel menší než π



Obrázek 1.4: Dva vektory svírající úhel vetší než π

1.1.3 Operace s vektory v prostoru

Ukážeme si, jak vypočítat úhel svíraný dvěma vektory v prostoru, k čemuž opět potřebujeme vypočítat délku vektoru a skalární součin dvou vektorů v prostoru. Vypočítáme také vektorový součin dvou vektorů, který se nám bude hodit při počítání normál povrchů modelů vykreslovaných v OpenGL.

V prostoru se délka vektoru vypočítá obdobně jako v rovině a to dle rovnice (5). Jediný rozdíl oproti počítání v rovině je ten, že nám v prostoru přibyla třetí složka vektoru u_3.

$$|\vec{u}| = \sqrt{u_1^2 + u_2^2 + u_3^2} \tag{5}$$

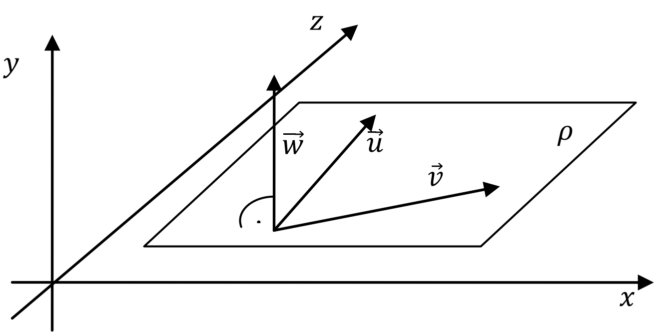
Skalární součin spočítáme dle rovnice (6), která ukazuje výpočet skalárního součinu pro vektory $\vec{u}$ a $\vec{v}$.

$$\vec{u} \cdot \vec{v} = u_1 v_1 + u_2 v_2 + u_3 v_3 \tag{6}$$

Úhel vektorů $\vec{u}$ a $\vec{v}$ vypočítáme podobně jako v rovině pomocí skalárních součinů a délky jednotlivých vektorů. V prostoru nám bude stačit vypočítat úhel ležící mezi 0 až π. Pro výpočet tedy můžeme využít rovnici (3), do které dosadíme hodnoty vypočítané díky rovnicím (6) a (5).

Poslední operaci, kterou si ukážeme je vektorový součin, jenž se počítá pouze v prostoru a značí se $\times$. Máme-li v prostoru zadány dva vektory $\vec{u}$ a $\vec{v}$, pak díky jejich vektorovému součinu $\vec{u} \times \vec{v}$, rovnice (7), vznikne nový vektor $\vec{w}$, který je na tyto dva vektory kolmý [5]. Pokud mají vektory $\vec{u}$ a $\vec{v}$ stejný počátek, výsledný vektor $\vec{w}$ je kolmý na rovinu ρ určenou těmito vektory, viz. obrázek 1.5.

$$\vec{w} = (u_2 v_3 - u_3 v_2, u_3 v_1 - u_1 v_3, u_1 v_2 - u_2 v_1) \tag{7}$$



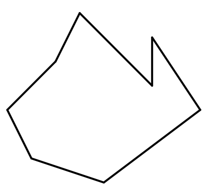
Obrázek 1.5: Vektorový součin vektorů $\boldsymbol{\vec{u}}$ a $\boldsymbol{\vec{v}}$ určujících rovinu $\boldsymbol{\rho}$

1.2 Polygony

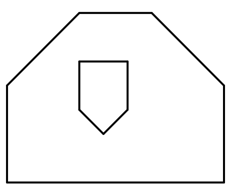
V analytické geometrii je polygon definován jako oblast ohraničená uzavřenou cestou tvořenou hranami polygonu. Obecně se polygon skládá z konečného množství 3...n hran, které mezi sebou svírají úhly $\alpha_{3...n}$. Místu, kde se dvě hrany stýkají se říká vrchol. První a poslední vrchol je z důvodu uzavřenosti polygonu totožný [6].

Nejprimitivnějším tvarem polygonu je trojúhelník. Tento polygon má jednu velkou výhodu a to tu, že je vždy konvexní. Polygony můžeme dále rozdělit podle jejich konstrukce [6] a to na jednoduchý polygon (obrázek 1.6), který je bez děr a může být konvexní i konkávní, monotónní nebo nemonotónní, viz. dále. Polygon s dírami (obrázek 1.7), který obsahuje díry, jejichž vrcholy se obvykle zadávají jako jdoucí proti směru vrcholů polygonu. Křížící se polygon (obrázek 1.8), kde jsou dvě nebo více hran polygonu, které se kříží. Konvexní polygon (obrázek 1.9), který má všechny vnitřní úhly svírané hranami polygonu menší než 180°. Konkávní polygon (obrázek 1.10), ten zase má jeden nebo více vnitřních úhlů svíraných hranami větší než 180°. Nakonec si uvedeme ještě monotónní polygon, který pokud rozdělíme přímkou procházející nevyšším a nejnižším vrcholem, tak vrcholy na jedné straně přímky jsou pouze klesající a na druhé pouze rostoucí. Jestli se jedná o pravou nebo levou stranu přímky zjistíme podle toho, v jakém směru jsou zadány vrcholy polygonu.

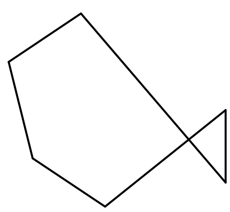
V následujícím textu budeme předpokládat polygon určený vrcholy, kdy poslední není zadán z důvodu, který je uveden výše. Vrcholy musí být zadány ve směru proti hodinových ručiček, protože všechny algoritmy popisované dále jsou pro to přizpůsobeny. Budeme pracovat pouze s polygony v kartézské soustavě souřadnic x a y.



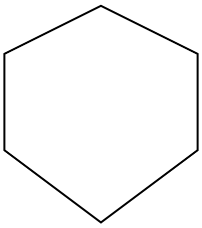
Obrázek 1.6: Jednoduchý polygon



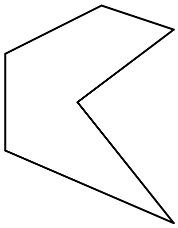
Obrázek 1.7: Polygon s dírou



Obrázek 1.8: Křížící se polygon



Obrázek 1.9: Konvexní polygon



Obrázek 1.10: Konkávní polygon

1.2.1 Triangulace polygonu

Triangulace polygonu znamená rozdělit celý polygon na trojúhelníky. Tuto operaci je třeba provést především kvůli vykreslení polygonu v OpenGL. Triangulovat se dá každý jednoduchý polygon a to tak, že polygon obsahuje přesně $n-2$ trojúhelníků, kde n je počet vrcholů trojúhelníku [7]. Popíšeme pouze triangulaci těchto jednoduchých polygonů, se kterými budeme dále pracovat. Jak jsme si již ukázali výše, jednoduchý polygon může být jak monotónní, tak nemonotónní. Rozdělíme-li nemonotónní polygon na monotónní částí, můžeme tyto části ztriangulovat v lineárním čase [7].

Monotizace obecného jednoduchého polygonu

Nejdříve musíme zkontrolovat, jestli je polygon nemonotónní. Postupně procházíme jednotlivé vrcholy p proti směru hodinových ručiček a to tak, že vezmeme vrchol p_{n-1}, p_n a p_{n+1} a ze znalosti vektorové algebry 1.1 z nich vytvoříme vektory, které na sebe navazují podobně jako je to zobrazeno na obrázku 1.4. První vektor jde od vrcholu p_{n-1} po p_n, označme jej jako $\vec{u}$ a druhý od vrcholu p_n po p_{n+1}, označme jako $\vec{v}$. Vždy ohodnocujeme vrchol p_n pomocí šesti skupin, které si označíme jako:

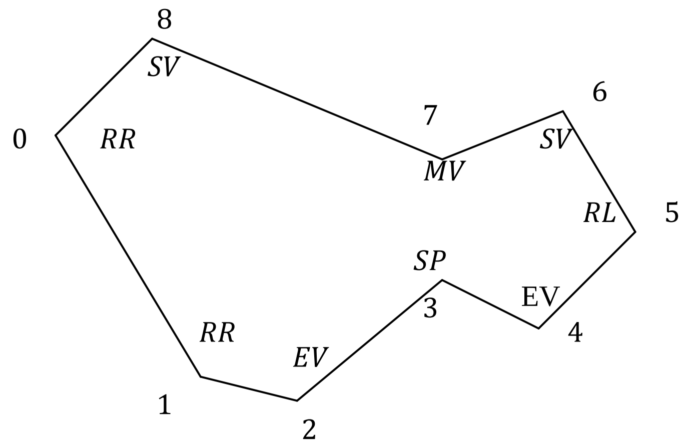
1. Start vrchol - SV
2. End vrchol - EV
3. Regular left vrchol - RL
4. Regular right vrchol - RR
5. Split vrchol - SP
6. Merge vrchol - MV

O jaký vrchol se jedná zjistíme takto (budou-li mít některé dva vrcholy stejnou souřadnici y, pak výše leží ten nejvíc vlevo, úhly vypočítáme pomocí 1.1.2, konkrétně díky rovnici (4)):

- **SV** - pokud je vypočítaný úhel mezi $\vec{u}$ a $\vec{v} < \pi \wedge p_{n-1}y < p_n y \wedge p_n y > p_{n+1} y$
- **EV -** pokud je vypočítaný úhel mezi $\vec{u}$ a $\vec{v} < \pi \wedge p_{n-1}y > p_n y \wedge p_n y < p_{n+1} y$

- **RL -** pokud $p_{n-1}y < p_n y \wedge p_n y < p_{n+1}y$, polygon leží nalevo od vrcholu p_n
- **RR -** pokud $p_{n-1}y > p_n y \wedge p_n y > p_{n+1}y$, polygon leží vpravo od vrcholu p_n
- **SP -** pokud je vypočítaný úhel mezi $\vec{u}$ a $\vec{v} > \pi \wedge p_{n-1}y < p_n y \wedge p_n y > p_{n+1}y$
- **MV** - pokud je vypočítaný úhel mezi $\vec{u}$ a $\vec{v} > \pi \wedge p_{n-1}y > p_n y \wedge p_n y > p_{n+1}y$

Ohodnocené vrcholy polygonu můžeme vidět na obrázku 1.11. Na tomto obrázku také vidíme, že polygon porušuje svojí monotónnost právě v MV a SP vrcholu. Pokud chceme polygon rozdělit, musí to být v tomto místě, tedy vedeme diagonálu z MV do SP. Polygon potom bude monotónní, pokud nebude mít žádné SP ani MV vrcholy. Algoritmicky se to dá vyřešit tak, že si seřadíme ohodnocené vrcholy podle hodnoty y souřadnice od nejvyššího, pokud mají některé vrcholy stejnou hodnotu souřadnice y, nejlevější jsou brány jako vyšší. Pak postupně procházíme tyto vrcholy a podle jejích ohodnocení je zpracováváme. Algoritmus je poměrně rozsáhlý a nebyl primárním předmětem řešení této práce, proto zde není uveden, ale je možné jej najít v [7].



Obrázek 1.11: Jednoduchý polygon s ohodnocenými vrcholy

Samotná triangulace monotónního polygonu

Hledáme diagonály, které rozdělí polygon na trojúhelníky. Poznatky čerpány z [7].

Nejdříve je potřeba si jednotlivé vrcholy rozdělit na ty, které leží na levé a pravé straně polygonu. K tomuto účelu můžeme využit metodu popsanou výše, kdy hledáme RR a RL vrcholy. Poté si vrcholy seřadíme do seznamu podle velikosti jejich y souřadnice od nejvyšší, kdy opět předpokládáme, že pokud mají některé vrcholy stejnou tuto souřadnici, pak je větší ten nejlevější. V takto seřazeném seznamu je první vrchol SV a poslední EV, což je pochopitelné, protože se jedná o monotónní polygon a ten má vždy pouze jeden SV a EV. Nyní si vytvoříme zásobník, na který budeme ukládat vrcholy určené ke zpracování. Nejdříve uložíme na zásobník první dva vrcholy ze seznamu. Dále postupně bereme jeden vrchol ze seznamu a kontrolujeme, zda tento vrchol leží na stejné straně polygonu jako vrchol na zásobníku nebo ne. Pokud ne, vyjmeme ze zásobníku všechny vrcholy a vytvoříme diagonály mezi vrcholem ze seznamu a těmito vyjmutými vrcholy, kromě posledního. Na zásobník uložíme tento vrchol ze seznamu a vrchol, který byl před vyjmutím na vrcholu zásobníku.

Pokud naopak vrchol leží na stejné straně jako vrchol na vrcholu zásobníku, vyjmeme vrchol zásobníku a odložíme jej, pak vyjmeme další vrchol zásobníku a pokud diagonála spojující tento vyjmutý vrchol a vrchol ze seznamu leží uvnitř polygonu, pak jsme našli další diagonálu. Ve vytváření diagonál pokračujeme tak dlouho, dokud daná diagonála leží uvnitř polygonu. Poté uložíme zpátky na zásobník poslední vrchol, který z něj byl vyjmut a uložíme tam také vrchol ze seznamu, se kterým se doteď pracovalo.

V provádění operací pokračujeme tak dlouho, dokud v seznamu nezbude poslední vrchol. Až se tak stane, vyjmeme ho a vedeme od něj diagonály na každý vrchol nacházející se v zásobníku kromě prvního a posledního.

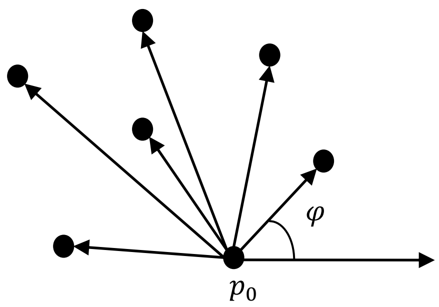
Nyní máme polygon diagonálami rozdělený na jednotlivé trojúhelníky.

1.2.2 Konvexní obálka - Graham scan

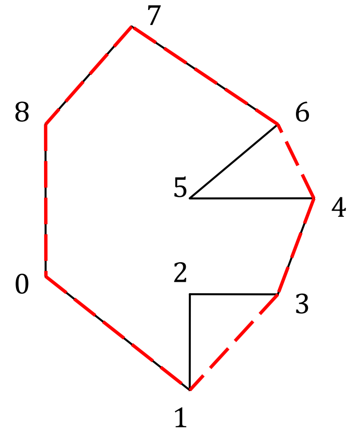
Další z důležitých operací s polygony je vytvoření jejich konvexní obálky, pokud se jedná o konkávní polygon. Pro tuhle operaci se velmi hodí algoritmus Graham scan, který je lehký na implementaci a navíc dosahuje dobrých rychlostních výsledků. Algoritmus pracuje nad množinou bodů, kde hledá jejích konvexní obálku. Vrcholy polygonu si můžeme představit jako množinu bodů.

Konvexní obálku vytvoříme tak [8], že si zvolíme pivot p_0, v našem případě to bude vrchol s nejmenší souřadnicí y ležící nejvíce vpravo, tzn. s největší hodnotou souřadnice x. Seřadíme ostatní vrcholy vzestupně podle toho, jaký úhel svírají s pivotem (výpočet úhlů mezi vektory, viz. 1.1.2), vše je patrné z obrázku 1.12. Poté postupně procházíme jednotlivé seřazené vrcholy pivotem počínaje. Vezmeme vrcholy p_n a p_{n+1} a vytvoříme z nich vektor, dále vezmeme vrchol p_{n+2} a pokud tento vrchol leží nalevo od vytvořeného vektoru, pokračujeme ve zkoumání dalších vektorů. Pokud však vrchol p_{n+2} leží napravo od vektoru, je vrchol p_{n+1}, vnitřním bodem polygonu a musíme ho z konvexní obálky vyřadit.

Konvexní obálku konkávního polygonu můžeme vidět na obrázku 1.13.



Obrázek 1.12: Pivot svírající úhel $\boldsymbol{\varphi}$ s jednotlivými vrcholy



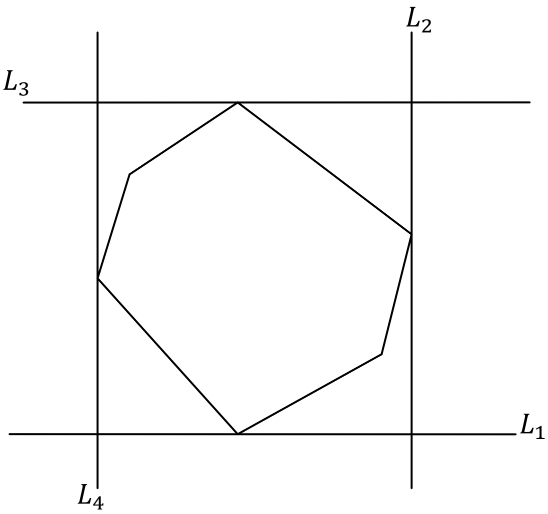
Obrázek 1.13: Konvexní obálka

1.2.3 Minimální ohraničující obdélník - MBR

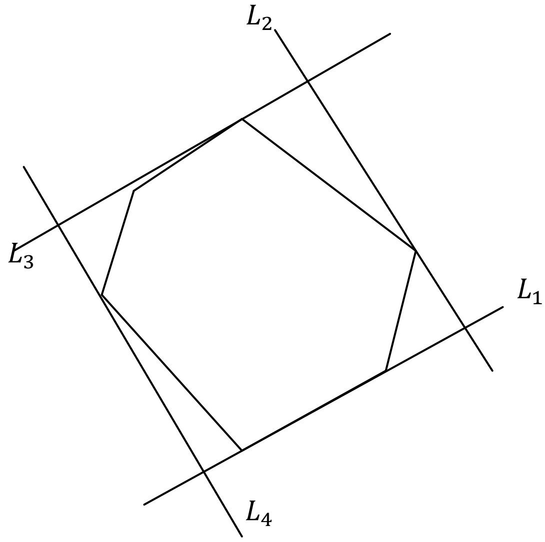
Je to obdélník s nejmenším obsahem, který ohraničuje celý polygon. Dá se jednoduše sestrojit pro konvexní polygon, proto jsme se v 1.2.2 zabývali vytvořením konvexní obálky polygonu. Budeme vycházet z důkazu [9], že vytvořený MBR se vždy dotýká nějaké hrany polygonu. V dalším textu je čerpáno z [10].

Pro sestrojení tohoto obdélníku budeme využívat čtyři přímky, které označíme L_1, L_2, L_3, L_4. Protože se jedná o obdélník, jsou dvě dvojice přímek, ve kterých jsou přímky paralelní a jedna dvojice je kolmá na druhou dvojici, zapíšeme jako $(L_1 \parallel L_2) \perp (L_3 \parallel L_4)$. Začneme tím, že přímku L_1 přiložíme na nejnižší vrchol polygonu podle osy y. Nyní hledáme ostatní vrcholy, kterých se budou dotýkat přímky L_2, L_3, L_4. Víme-li, že sestrojujeme obdélník, tudíž přímku L_2 musíme natočit od L_1 o $\frac{\pi}{2}$, L_3 od L_1 o π a L_4 od L_1 o $\frac{3\pi}{2}$. Tedy postupně procházíme jednotlivé vrcholy od prvního (ten, ke kterému přiléhá L_1) a počítáme úhly, které na těchto vrcholech svírají hrany polygonu tak dlouho, dokud jsou úhly menší než $\frac{\pi}{2}$, π respektive $\frac{3\pi}{2}$. Až jsou dané úhly větší, našli jsme hledaný vrchol pro L_2, L_3, L_4. V dalších krocích se L_1 přikládá na jednotlivé strany, úhly $\frac{\pi}{2}$, π a $\frac{3\pi}{2}$ aktualizujeme, tzn. přičteme k ním hodnotu natočení první přímky a pro hledání ostatních přímek a jejích vrcholu postupujeme jako v předchozím případě, první krok je naznačen na obrázku 1.15. Toto opakujeme pro všechny hrany polygonu.

Délky dvou kolmých stran obdélníku, které nám stačí pro výpočet obsahu MBR spočítáme ze znalosti výpočtu nejkratší vzdálenosti bodu od přímky (např. zde [11]), kde přímkou je přímka L_1 a bodem je vrchol, kterého se dotýká přímka L_3. Druhou vzdálenost spočítáme stejně.



Obrázek 1.14: Počáteční stav přímek pro MBR



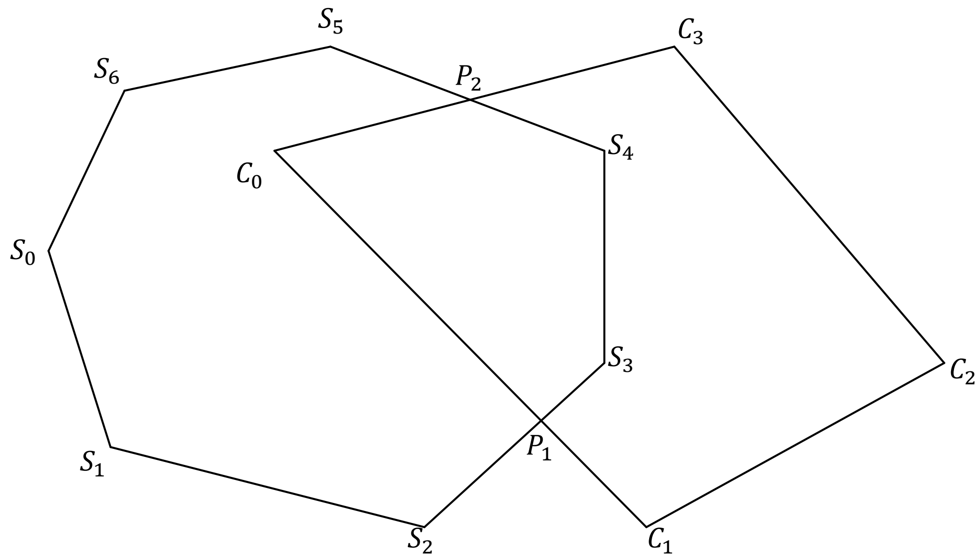
Obrázek 1.15: Stav MBR po prvním natočení

1.2.4 Ořezávání polygonu, Weiler-Atherton

Ořezávání polygonu je další důležitou operací, kterou řeší analytická geometrie a teď nemáme na mysli ořezávání přímkou, jak se to běžně dělá při vykreslování polygonu, kdy máme nějaké vykreslovací okno a chceme ořezat ty části polygonu, které nejsou vidět. Budeme se zabývat tím, jak ořezat n-stranný polygon jiným n-stranným polygonem. Další řádky textu jsou proto věnovány algoritmu Weiler-Atherton, který toto ořezávání řeší [12]. Vrcholy obou polygonů jsou zadány opět v pořadí proti hodinovým ručičkám. Algoritmus pracuje na principu hledání průsečíků dvou polygonů.

Mějme dva polygony, jeden ořezávaný a druhý ořezávající a označme je písmeny S (subject) a C (clip). Na obrázku 1.16 můžeme vidět tyto dva polygony a také zde vidíme jejich průsečíky P_1 a P_2. Tento obrázek využijeme dále. Vytvoříme si dva seznamy, jeden pro průsečíky vstupující do polygonu S, označme EN a jeden pro průsečíky vystupující z S, označme EX. Dále si vytvoříme seznam SS a CS, do kterých uložíme vrcholy polygonů S a C. Poté začneme provádět algoritmus, kdy procházíme všechny hrany polygonu S a pro každou jeho stranu testujeme, zda zde není průsečík s C, tedy u každé hrany S musíme projít všechny hrany C. Nalezneme-li průsečík, označíme si jej, uložíme ho k vrcholům do seznamů SS a CC za vrchol, kde byl průsečík nalezen a určíme zda se jedná o průsečík vystupující nebo vstupující do S a to podle znalosti hran polygonu S i C, kde leží tento průsečík. V našem případě první průsečík P_1 leží úsečce tvořené vrcholy S_2, S_3 a C_0, C_1. Vytvoříme vektor $\vec{u}$ skládající se z bodů S_2, P_1 a vektor $\vec{v}$ z bodů P_1, C_1 a vypočteme úhel, který tyto dva vektory svírají podle 1.1.2. Je-li úhel větší jak π, jedná se o průsečík vystupující z S, jinak je vstupující. U nás se tedy jedná o vystupující průsečík, uložíme jej do seznamu EN. Toto opakujeme pro další nalezené průsečíky.

Po nalezení všech průsečíku se zaměříme na seznam EN. Podle toho, kolik je v něm uložených průsečíku, tolik nových polygonů vznikne díky ořezání (při ořezávání polygonem jich může být více). Vezmeme první průsečík z EN a vyhledáme jej v SS. Tímhle průsečíkem bude začínat náš novy ořezaný polygon, uložíme si jej. Procházíme SS zleva doprava od místa, kde jsme průsečík našli a hledáme následující průsečík. Vrcholy, kterými projdeme si ukládáme. Jakmile najdeme další průsečík uložíme jej a přesuneme se do seznamu CS na místo, kde se tento průsečík také nachází. Tento seznam procházíme naopak zprava doleva od tohoto místa a ukládáme si jednotlivé vrcholy, dokud nenarazíme na další průsečík. Až se tak stane, uložíme si jej a zkontrolujeme, zda se nejedná o průsečík, kterým jsme začínali v SS. Pokud ano, našli jsme ořezaný polygon, jehož vrcholy máme uložené a můžeme skončit. Pokud to není tento průsečík, opakujeme vyhledáváni v SS a CS. Nakonec se podíváme, zda je EN prázdný, pokud ne, opakujeme znovu vyhledávání od začátku, jinak končíme, neboť jsme nalezli všechny nové polygony vytvořené ořezáváním.



Obrázek 1.16: Ukázka ořezávání polygonu

1.2.5 Výpočet plochy polygonu

Máme-li polygon zadaný N vrcholy v pořadí proti hodinovým ručičkám a to (x_0, y_0) až (x_{n-1}, y_{n-1}), pak můžeme plochu polygonu vypočítat pomocí formule The Surveyor's Formula [13], která je dána rovnicí (8).

$$A = \frac{1}{2}\sum_{i=0}^{N-1}(x_i y_{i+1} - x_{i+1} y_i) \tag{8}$$

1.2.6 Barycentrické koordináty

Tyto koordináty budeme využívat v rovině, tedy 2D prostoru, kdy hledáme u a v váhy bodu (x, y), které nám určí zda leží daný bod v trojúhelníku (koordináty v trojúhelníku nám budou stačit, protože umíme polygon triangulovat, viz. 1.2.1). Váhy u a v jsou hledané barycentrické koordináty a lze díky ním popsat každý bod v trojúhelníku [14].

Mějme trojúhelník zadaný vrcholy $A[a_1, a_2]$, $B[b_1, b_2]$ a $C[c_1, c_2]$ a bod $P[p_1, p_2]$. Zvolíme si jako počátek bod A, zde tedy budou mít váhy bodu P hodnotu $u = 0,\ v = 0$. Spočítáme si vektory $\overrightarrow{v_0} = C - A$, $\overrightarrow{v_1} = B - A$ a $\overrightarrow{v_2} = P - A$. Pokud bychom neznali bod P, ale znali bychom váhy u a v tohoto bodu, můžeme tento bod vypočítat pomocí rovnice (9). Tuto rovnici si upravíme na rovnici (10).

$$P = A + u * (C - A) + v * (B - A) \tag{9}$$

$$P - A = u * (C - A) + v * (B - A) \tag{10}$$

Do rovnice (10) dosadíme výše vyjádřené vektory a vznikne rovnice (11). Tato rovnice se dá řešit více způsoby. Jedním z nich je např. vyřešení dvou rovnic vzniklých z (11) pomocí Badouelové implementace v [15].

$$\overrightarrow{v_2} = u * \overrightarrow{v_0} + v * \overrightarrow{v_1} \tag{11}$$

Bod pak leží v trojúhelníku pokud vypočítané $u \geq 0 \wedge v \geq 0 \wedge (u + v) \leq 1$.

1.3 GIS a mapy

Pojmy v této podkapitole si popíšeme jen stručně, především se zaměříme na věci, které využijeme. V následujícím textu je čerpáno z [16].

GIS je zkratka pro Geografický Informační Systém, což je informační systém pro získávání, ukládání, analýzu a vizualizaci velkých množství geografických dat spojených se Zemským povrchem a zeměpisnou polohou. Na základě těchto dat umožňuje vytvářet modely části Země, které se pak využívají při tvorbě map, plánování výstavby silniční sítě, evidenci katastru nemovitostí, k předpovědím počasí nebo v navigačních systémech apod.

Mapa, tak jak ji budeme chápat my, je zmenšené a zevšeobecněné znázornění objektů na Zemi. Vědní obor zabývající se výrobou map se nazývá kartografie. Máme-li ale přístup k nějakému systému GIS, můžeme si tyto mapy vyrábět i sami. Základním představitelem map jsou tématické mapy, které obsahují jedno nebo více témat, které chceme zobrazit na úkor dalších objektů. Vznikají tak např. turistické mapy, kde nás zajímají především turistické stezky a jejich značení nebo autoatlasy, kde požadujeme v první řadě zobrazení silniční sítě dané oblasti apod. Důležitým parametrem při tvorbě map je volba měřítka, které volíme s ohledem na účel mapy. V závislosti na zvoleném měřítku se musí také rozhodnout, které detaily jsou ještě podstatné pro zobrazení na takové mapě. Měřítko v číselném tvaru např. 1:10 000 nám udává, že jeden centimetr na mapě je 10 000 centimetrů ve skutečnosti. V kartografii rozlišujeme tři druhy měřítka:

1. mikro měřítko (více než 1:25 000)
2. mezo měřítko (od 1:25 000 do 1:1000 000)
3. makro měřítko (méně než 1:1000 000)

Nyní si ještě řekněme něco k elektronickým mapám. Elektronické mapy můžeme většinou přibližovat, nebo oddalovat, proto jejich měřítko není fixní ale mění se podle úrovně přiblížení, resp. oddálení. Tyto mapy mohou být dvojího druhu a to rastrové nebo vektorové. Rastrová mapa je složena z obrázků, ve kterých může být zaneseno prakticky cokoliv a velmi se podobají klasickým tištěným mapám. Podrobnost mapy přitom záleží na rozlišení. Přibližujeme-li tuto mapu, jednotlivé pixely se začínají rozmazávat. Naproti tomu ve vektorových mapách jsou jednotlivé objekty popsány matematicky, např. křivkami, úsečkami atp. Při změně měřítka se tyto objekty přepočítávají a výsledné zobrazení je vždy kvalitní. A kvůli přepočítávaní je také třeba dbát na to, jaké detaily do těchto map zanášet, aby výpočet pro zobrazení mapy netrval příliš dlouho.

V souvislosti s GIS a mapami se ještě vyskytuje jeden pojem, který si popíšeme a to geografický souřadný systém, kterým můžeme popsat polohu nějakého bodu na Zemi. Souřadný systém se dělí na globální a lokální. Globální se snaží popsat celý geografický prostor Země a lokální se využívá jen pro určité území, které chceme postihnout. My budeme dále využívat pouze globální systém. Jedním z jeho představitelů je WGS 84[1], podle kterého se určuje zeměpisná šířka a délka bodu na jakémkoliv místě na Zemi. Zeměpisná šířka bodu je dána úhlem, který bod svírá s rovníkem. Rovník je počátkem souřadné osy zeměpisné šířky a má hodnotu 0°. Hodnota zeměpisné šířky může nabývat 0° až 90° jak ve směru k jižnímu, tak k severnímu pólu. Severní šířce se dává znaménko plus a jižní mínus. Rozsah šířky je tak 180°. Naproti tomu zeměpisná délka bodu se počítá pomocí poledníků. Nultý poledník byl stanoven na úroveň města Greenwitch v Anglii. Zeměpisná délka bodu se tedy vypočítá jako úhel mezi nultým poledníkem a poledníkem, kterým bod prochází a nabývá hodnot 0° až 180° na obě strany o od nultého poledníku, přičemž hodnoty úhlu pro bod ležící na západní polokouli jsou opět se záporným znaménkem. Rozsah hodnot je tedy 360°.

[1] WGS 84 - World Geodetic System, geodetický standard vydaný ministerstvem obrany USA roku 1984

1.4 Millerova projekce

Výše jsme si ukázali, jak vypočítat zeměpisnou šířku a délku bodu na Zemi. Pokud budeme ale chtít tento bod zobrazit na rovinné mapě, nebo v kartézské soustavě souřadnic, musíme tuto šířku a délku přepočítat. Pro tento přepočet se využívá především Merkatorova projekce [17], která je ale dost nepřesná pro body ležící blízko k severnímu nebo jižnímu pólu. Pokud vytvoříme pomocí této projekce mapu, bude se např. Grónsko jevit stejně velké jako Afrika. Z tohoto důvodu tuto metodu modifikoval pán Osborn Maitland Miller, podle kterého se upravená projekce taky jmenuje Millerova projekce [18]. Výpočet bodu x a y pro zeměpisnou délku, kterou označíme jako λ a zeměpisnou šířku značenou písmenem φ je znázorněn v rovnicích (12) a (13). Millerova projekce již netrpí tak velkým zkreslováním bodů ležících blízko jednoho nebo druhého pólu.

$$x = \lambda \tag{12}$$

$$y = \frac{5}{4}\ln\left[\tan\left(\frac{1}{4}\pi + \frac{2}{5}\varphi\right)\right] \tag{13}$$

1.5 OpenStreetMap

Jak již bylo uvedeno v úvodu, jedná se o svobodný projekt, kde hlavním stavebním kamenem jsou uživatelé, kteří vytvářejí geografická data o celém světě. Vytvářet data se dá více způsoby [19]. Nejčastěji se k tomu využívá GPS přístrojů, kterými uživatelé zaznamenávají pozici objektů a věcí v okolí, kde se nachází. Další možností je focení objektů, především názvů ulic, nebo zakreslovaní ulic města, kterým procházíme na papír. Dá se také použít satelitních map, ze kterých se jednotlivé objekty na mapě překreslují. Zde je ale třeba dbát na licenční podmínky těchto map. Získaná data se pak zanáší do mapy pomocí již vytvořených programů, které je možné najít na stránkách projektu [20].

Poskytované mapy jsou dvourozměrné, tedy neobsahuji údaje o nadmořské výšce ani vrstevnice. Mapy je možné exportovat na základě měřítka a geografických souřadnic v grafickém formátu (rastrově jako formát *png* nebo *jpeg*, anebo vektorově díky formátu *svg*) nebo v XML souboru. Export lze provést pomocí základního API[2] v0.6 ze stránky *www.openstreetmap.org*. U tohoto API je ale zavedeno omezení na počet uzlů (viz. níže) stažených v XML souboru při jednom požadavku a to na 50 000. Chceme-li stáhnout data s větším počtem uzlů můžeme využít XAPI [21] dostupné na *xapi.openstreetmap.org*. Nevýhodou XAPI ale je velká vytíženost serveru kvůli čemu je často nedostupný. Obě API fungují na principu HTTP požadavku a odpovědi. Požadavek na určitou část mapy se zadává jako *GET /api/0.6/map?bbox=left,bottom,right,top,* kde *left* je zeměpisná délka na levé straně výseku mapy, *bottom* je zeměpisná šířka spodního výseku mapy, *right* je opět

[2] API - Application Programming Interface

zeměpisná délka pravé strany mapy a nakonec *top* je zeměpisná šířka horního okraje požadovaného výseku mapy.

Geografická data v OpenStreetMap jsou uložena ve formátu XML. Elementy, kterými jsou data reprezentována se dělí na tři skupiny [22]:

1. **Node** - základní element, kterým se definuje další element Way (viz níže). Uchovává informace o zeměpisné šířce a délce ve formátu WGS 84. Může existovat i sám o sobě, kdy reprezentuje nějaký objekt na mapě, např. telefonní budku
2. **Way** - uspořádané spojení více bodu. Může být otevřené, pro značení silnic, železnic, potoků, atp. nebo uzavřené, označující se jako **Area** pro reprezentaci ploch na mapě jako jsou např. lesy, vodní plochy, zemědělské a obytné oblasti nebo půdorysy budov
3. **Relation** - mohou sdružovat ostatní elementy dohromady. Určují členy relace a přiřazují jím role

1.6 Strojové učení

Strojové učení je jednou z oblastí oboru umělé inteligence zabývající se algoritmy, které umožňují systému učit se [23]. Umožňují měnit znalosti inteligentního systému tak, že příště bude vykonávat stejnou nebo podobnou úlohu efektivněji [24]. Strojové učení lze rozdělit do tří kategorií:

1. Učení s učitelem
2. Učení bez učitele
3. Posilované učení

Učení s učitelem

Učení je založeno na tom, že pro každý prvek vstupující do učícího algoritmu známe jeho výsledné zařazení do třídy. Systém je tedy vždy hned informován, jak správně se přijatý vzorek dat naučil. Učení se provádí na trénovací množině příkladů.

Učení bez učitele

Při tomto učení systém nezná výsledné zařazení prvků, které vstupují do učícího algoritmu do tříd. Při učení využívá tzv. podobností mezí jednotlivými prvky a podle těchto podobností se učí. Tomuto způsobu učení se říká shlukování.

Posilované učení

Systém provádí při učení určité akce o kterých se domnívá, že povedou k výsledku. Tyto akce jsou hodnoceny a to buď pozitivně nebo negativně. Cílem systému je po čas učení dosáhnout co největšího pozitivního hodnocení.

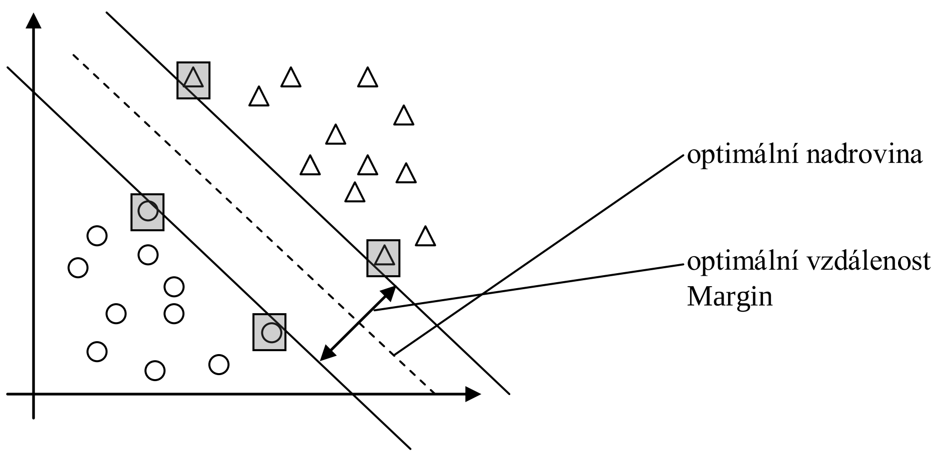
1.6.1 Rozpoznávání a klasifikace

Rozpoznávání a klasifikace spočívá v rozdělování objektů reálného světa do tříd. Máme-li nějaké objekty, můžeme u nich najít znaky, které je plně charakterizují. Znaky volíme pomocí nějaké provedené analýzy objektů reálného světa. Na základě těchto znaků, teda dat, vytvoříme číselný vektor, který nazýváme obraz a dále už nám zastupuje daný objekt. Seskupíme-li všechny vektory do prostoru, nazýváme tento prostor jako obrazový prostor. Dále se snažíme nalézt takovou funkci, která bude co nejlépe rozpoznávat tyto obrazy v obrazovém prostoru a zařadí je do správné třídy [23].

Stroj, neboli algoritmus pro klasifikaci se nazývá klasifikátor a je třeba jej naučit podle čeho má rozpoznávat, tedy musíme mu předat nějakou trénovací množinu složenou z obrazů, které již jsou rozděleny do tříd. Klasifikace je tedy založena učení s učitelem. Fáze učení se nazývá trénovací fáze. Druhou fázi je naopak testovací fáze, kdy klasifikátoru předáme data odlišná od dat použitých při učení, tedy taková, které ještě nikdy neviděl. Na základě výsledku klasifikace ověřujeme, jak dobře klasifikátor vyhodnocuje, neboli rozpoznává vstupní data [23].

1.6.2 SVM

SVM je zkratka pro Support Vector Machines a jedná se o metody strojového učení s učitelem, které tvoří kategorii jádrových algoritmů (*kernel machines*). Tyto metody hledají takovou lineární hranici (nadrovinu), která rozděluje sadu vstupních dat patřících do dvou různých tříd. Navíc dovedou pracovat i s nelineárními funkcemi, kdy vstupní prostor převádí do vícerozměrného prostoru, kde je již možné třídy rozdělit lineárně [25]. Aby klasifikátor rozděloval objekty optimálně, je třeba nalézt takový lineární oddělovač, který má maximalizovanou vzdálenost mezi sebou a pozitivními a negativními příklady. Této vzdálenosti se říká Margin [25]. Na obrázku 1.17 můžeme vidět lineárně oddělitelné data ve dvourozměrném prostoru.



Obrázek 1.17: Lineárně oddělitelné data. Support vectors jsou označeny šedým čtvercem.

Mějme trénovací množinu dat skládající se z párů $(x_i, y_i), i = 1, \ldots, l$, kde $x_i \in R^n$ jsou vstupní vektory a $y_i \in \{-1,1\}^l$ jsou třídy, do kterých jednotlivé vstupní vektory spadají. Rozdělující nadrovina se pak dá vyjádřit vztahem (14), kde w je normálový vektor kolmý na rozdělující nadrovinu, x_i je vstupní vektor a b je bias, neboli zkreslení. Pokud jsou vstupní data lineárně rozdělitelná, můžeme říci, že pro rozdělení množiny dat do jedné ze dvou tříd platí (15) a (16) [25].

$$w \cdot x_i + b = 0 \tag{14}$$

$$w \cdot x_i + b \geq 1, \text{ pokud } y_i = 1 \tag{15}$$

$$w \cdot x_i + b \leq -1, \text{ pokud } y_i = -1 \tag{16}$$

Máme-li data, která nejsou lineárně oddělitelná, vznikají při jejich klasifikaci chyby. Potřebujeme nalézt minimalizované řešení následujícího problému (17) přepsaného na (18) [26], kdy se hledá taková nadrovina, pro kterou je chyba nesprávné klasifikace minimalizována. Takováto nadrovina se nazývá Soft Margin Hyperplane.

$$\min_{w,\xi} \quad \frac{1}{2} w^T w + C \sum_{i=1}^{l} \xi_i \tag{17}$$

$$y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i, \text{ kde } \xi_i \geq 0 \tag{18}$$

V (18) vidíme vstupní vektory x_i, které jsou mapovány pomocí funkce ϕ do vyššího dimensionálního prostoru. SVM nalezne optimální rozdělující nadrovinu v tomto vyšším dimensionálním prostoru. V (17) zase vidíme parametr $C > 0$, který určuje postih za chybně určenou třídu.

Rozdělení lineárně neoddělitelných dat se dá v SVM řešit také pomocí jádrových funkcí. My se zaměříme především na jádrovou funkci RBF. RBF je zkratka pro Radial Basis Function, což je jádrová funkce, která nelineárně mapuje data do vyššího dimensionálního prostoru a zvládne rozdělit i lineárně neoddělitelné data a její popis je dán funkcí (19) [26].

$$K(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right), \gamma > 0 \tag{19}$$

Důležitým krokem při použití této jádrové funkce je správný výběr C a γ parametru. Nejlepší kombinace těchto parametrů se nejčastěji vybírá pomocí tzv. grid-search, což znamená, že zkoušíme kombinace exponenciálně se zvyšující sekvence parametrů C a γ (např. $C = 2^{-5}, 2^{-3}, \ldots, 2^{15}$, $\gamma = 2^{-15}, 2^{-13}, \ldots, 2^{3}$). Různé optimalizace jak urychlit výběr správných parametrů jsou naznačeny v [26].

1.6.3 SVM multiclass

Výše jsme si ve zkratce popsali funkčnost SVM, které umí rozdělit data do dvou tříd. My ale budeme potřebovat klasifikovat data do více tříd. Pro řešení tohoto problému existuje více technik, které dekomponují vícetřídní problém na jednotlivé binární problémy, které jsou již zpracovatelné metodami SVM. Ukážeme si dvě základní techniky z [27]. V tomto dokumentu můžeme najít i další techniky.

One-against-all (OvA)

Metoda pracuje na principu vytvoření SVM pro každou třídu. Je-li zadáno N tříd, kde $N > 2$, pak vytvoříme N SVM metod. Trénování probíhá tak, že i-té SVM klasifikuje pozitivně data z i-té třídy. Data z ostatních tříd jsou klasifikována negativně. Při rozpoznávání se pak testovací vzorek dat předloží všem N SMV klasifikátorům a výsledek je vybrán na základě maximálního ohodnocení mezi těmito klasifikátory. Nevýhodou je příliš komplexní učení při vysokém počtu tříd, protože každý klasifikátor je trénován všemi vzorky dat.

One-against-one (OvO)

Tento algoritmus vytváří $N(N-1)/2$ SVM metod využívajících všechny binární kombinace mezi jednotlivými třídami. Každý klasifikátor se učí daty z jedné třídy pro positivní příklady a daty z druhé třídy pro negativní příklady. Při rozpoznávání se využívá kombinací jednotlivých klasifikátorů. Výhodou této techniky je, že pro naučení jednoho klasifikátoru potřebujeme menší počet dat, což je výhodné proto, protože menší počet dat způsobuje menší nelinearitu. Nevýhodou je naopak to, že se každý vzorek dat musí předložit všem klasifikátorům, kterých může být mnoho. To vyúsťuje v pomalejší testování, hlavně v případě, kdy máme velký počet tříd.

1.7 Grafická knihovna OpenGL

Je to knihovna poskytující nástroje k modelování a vytváření 3D grafiky. Knihovna funguje na různých typech grafických akcelerátorů a lze ji použít na různých platformách, jako jsou unixové systémy Linux nebo naopak systém Microsoft Windows [28].

Na poli 3D grafiky je množství různých firem vyrábějících grafické akcelerátory a tyto firmy se snaží inovovat svoje výrobky a to nejen ve smyslu zvyšování výkonu a kvality, ale také přidávají nové speciální efekty a metodologie. Aby bylo možné tyto rozšíření využívat, byl do OpenGL přidán mechanismus rozšíření (extension mechanism). Tento mechanismus povoluje výrobcům přidávat do OpenGL API nové funkce, které následně mohou programátoři využívat, pokud jsou grafickým akcelerátorem podporovány. K tomu, abychom získali ukazatel na dané rozšíření budeme využívat knihovnu GLEW [28].

Knihovna OpenGL se vyvíjí již přes deset let, kdy do ní byly přidávány nové funkce v závislosti na vývoji hardware a neměnila se koncepce knihovny. Proto byly programy psané v této knihovně vždy zpětně kompatibilní. S vývojem hardware se ale mění požadavky na software a mění se mechanismy, které jsou využívání pro psaní programů na dnešní hardware, a které byly využívaný např. před 15-ti roky. Proto byla vyvinuta nová specifikace knihovny a to OpenGL 3.0, kde se začalo s odstraňováním starých funkcí, které se zde staly tzv. deprecated, což znamená, že ještě nebyly odstraněny, ale již ukazovaly, které funkce tedy nebudou v nových specifikacích knihovny přístupné.

1.7.1 OpenGL 3.0

V této knihovně lze využívat dva profily a to *compatible* nebo *core*. V *compatible* profilu lze ještě využívat staré funkce knihovny. Naopak u *core* profilu to již možné není. Tento profil budeme používat v naší aplikaci. Následující informace převzaty z [28]. Hlavním rozdílem oproti předchozím verzím OpenGL je hlavně to, že většina dat je již uložena v paměti grafické karty a nedochází tak ke zpomalování při přenosu dat z RAM do grafické karty přes sběrnici, která je rychlostně limitujícím faktorem. Dále byly také odstraněny z grafické pipeline tzv. fixed-function, tedy byly odstraněny funkce pro výpočet osvětlení, mlhy, funkce pro práci s maticemi (glMatrix*(), glTranslate*(), glRotate*(), glScale*()). Pro vytváření geometrie vrcholů se již nepoužívá příkazy glVertex*(), glColor*() nebo glTexCoord*(). Vše je nahrazeno programovatelnými shadery, které se nahrají do grafické karty jako zdrojový kód a karta si jej sama zkompiluje. Pro psaní shaderu se využívá jazyk GLSL, který je svojí syntaxí podobný jazyku C. Dále byly z knihovny odstraněny grafická primitiva rovinný čtyřúhelník, pás rovinných čtyřúhelníků a rovinný konvexní polygon. Místo toho se hojně využívá trojúhelníků.

Jak bylo již zmíněno, data se ukládájí přímo do paměti grafické karty. Jako úložiště těchto dat slouží VBO (vertex buffer object). Pro přístup k datům z VBO se využívá VAO (vertex array object), což je objekt, který si dokáže zapamatovat konfiguraci VBO. VAO poté stačí jednoduše volat při vykreslování scény. Zjednodušeně řečeno do VBO můžeme uložit geometrii jednotlivých vrcholů, např. barvu, texturu, normálu náležící vrcholu a samozřejmě pozici vrcholu. Nastavíme jednotlivé offsety pro zmíněné data, tzn. řekneme VAO odkud pokud se nachází data pro barvu vrcholu, texturu, normálu a kde se již nachází pozice vrcholu. Nakonec je třeba ještě vytvořit jeden speciální buffer object a to buffer pro uložení indexů vrcholů. Zde se uloží geometrie jednotlivých vykreslovaných objektu ve scéně. Při vykreslování se pak povolí dané VAO a pomocí jediného příkazu (např. glDrawElements) vykreslíme celou scénu.

1.7.2 Textury v OpenGL

Pomocí textur můžeme na povrchy těles nanášet místo barvy různé obrazce. Při nanášení textury se nemění geometrie tělesa. Textury se používají především tam, kde je třeba simulovat nějaký složitější povrch, např. cihlová zeď. Kdybychom u této zdi měli vykreslovat každou cihlu samostatně a poté ti obarvovat, vzniklo by nám hodně objektů pro vykreslování a zpomalila by se rychlost vykreslování celé scény. Proto využijeme texturu, kdy si vytvoříme zeď jako jednoduchý útvar, např. obdélník a na něj namapujeme jednoduše texturu zdi.

V OpenGL existují tři druhy textur a to 1D, 2D a 3D. My budeme využívat pouze textury 2D. Rozměry textur, které OpenGL dokáže načíst musí být násobkem mocniny dvou, např. 128x128 pixelů. Ve vytvářené aplikaci jsou využity textury rozměrů 512x512 pixelů.

2 Import dat z OpenStreetMap

Nyní si ukážeme formát dat, která budou sloužit jako vstup pro vytvářenou aplikaci. Popíšeme si stažení těchto dat, import a uložení do paměti ve formě použitelné pro vyvíjenou aplikaci.

2.1 Formát dat

Jak již bylo řečeno v 1.5, mapové podklady poskytované OpenStreetMap jsou ve formátu XML. Kořenový element se jmenuje `<osm>` a obsahuje dva atributy. Jeden z nich je `version`, který značí verzi API, pomocí kterého byl mapový podklad stažen. Druhým elementem je `<bounds>` jehož atributy jsou `minlat, minlot, maxlat, maxlon,` což jsou informace o zeměpisné šířce (lat) a délce (lon) výseče dané stahované mapy viz. 1.5. Poté již následují elementy `<node>`, které pomocí atributů `lat, lon` uchovávají informace o zeměpisné šířce a délce bodu. Mají také atribut `id,` jedinečný pro každý uzel. Další atributy pro nás nejsou podstatné. Následují elementy `<way>`. Tyto elementy mají také jedinečný atribut `id.` Navíc obsahují element `<nd>` tvořící uspořádané spojení bodu, viz. 1.5 a element `<tag>`. První element obsahuje odkaz na `id` elementu `<node>` v podobě atributu `ref` a druhý, tedy element `<tag>` obsahuje dva atributy `k` a `v,` kde prvně jmenovaný atribut je klíč a druhý hodnota. Čeho mohou tyto atributy nabývat nalezneme v [29]. Jako poslední element se zde vyskytuje `<relation>`, který značí relace mezi některými elementy `<way>`, jako je např. vytváření multipolygonů. V této práci ale tyto relace řešit nebudeme. Ukázka XML struktury mapy obsahující pouze vodní plochu je k vidění níže. Je dobré si povšimnout, že pokud se jedná o uzavřený objekt na mapě, je první a poslední odkaz `<nd>` v elementu `<way>` stejný.

```
<osm version="0.6" generator="CGImap 0.0.2">
<bounds minlat="49.832"
minlon="15.476"
maxlat="49.832"
maxlon="15.478"/>
 <node id="648818707" lat="49.8317431" lon="15.4773501"/>
 <node id="648818705" lat="49.8317490" lon="15.4774326"/>
 <node id="648818706" lat="49.8317452" lon="15.4773937"/>
<node id="648818713" lat="49.8317546" lon="15.4774627"/>
 <way id="50879929">
  <nd ref="648818705"/>
  <nd ref="648818706"/>
  <nd ref="648818707"/>
  <nd ref="648818713"/>
  <nd ref="648818705"/>
  <tag k="landuse" v="reservoir"/>
 </way>
     </osm>
```

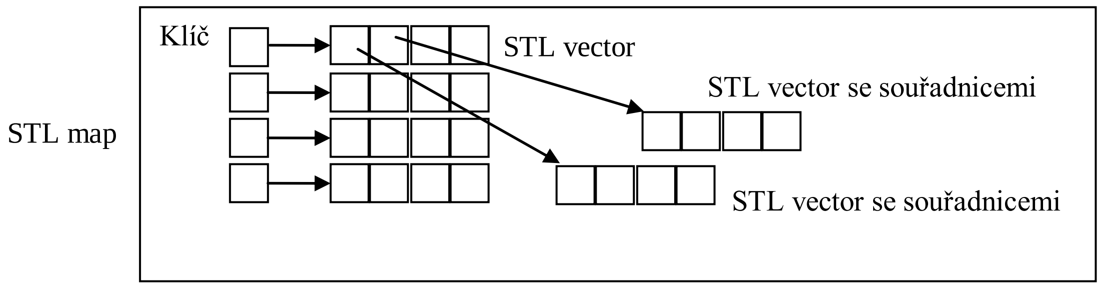
Tabulka 2.1: XML schéma mapového podkladu

2.2 Stažení, import a uložení dat do paměti

Stáhnout mapové podklady z OpenStreetMap a poskytnout je aplikaci můžeme dvojím způsobem. První je ruční stažení přímo ze stránky *www.openstreetmap.org* nebo necháme aplikaci stáhnout data automaticky a to zadáním souřadnic požadované mapy. Mapa pak bude stažena po částech pomocí API popsané v 1.5. Druhá možnost je vhodná především pro velké mapy, kde by počet uzlů mohl překročit maximální hodnotu 50 000.

Pro import dat byla vybrána knihovna libxml++, která poskytuje nástroje pro práci s XML soubory a to jak DOM[3] parser, tak SAX[4] parser. Protože stahujeme mapu s omezeným počtem uzlů (viz výše), využijeme pro práci s tímto mapovým XML souborem DOM parser. Jakmile máme načtenou celou XML strukturu v paměti, procházíme všechny elementy `<way>` a hledáme jejich podelementy `<tag>`, podle nichž rozhodneme, který element `<way>` uchovávající informaci o nějakém objektu na mapě, budeme dále zpracovávat. Protože objektů na mapě je hodně a všechny nejsou úplně podstatné pro další zpracování, byly vybrány jen některé. Úplný seznam vybraných objektů je uveden v příloze. Až najdeme jeden z požadovaných `<way>` elementů, tak podle odkazů v podelementech `<nd>` najdeme odpovídající element `<node>`, ve kterém je již uložená požadovaná poloha bodu (zeměpisná šířka a délka). Po získání všech těchto souřadnic pro daný `<way>` element můžeme přistoupit k uložení objektu.

Jednotlivé souřadnice objektu jsou převedeny pomocí Millerovy projekce 1.4 a uloženy do STL kontejneru `vector`. Pro uložení všech objektů byl zvolen STL kontejner `map`. Tento kontejner funguje na principu asociativního pole, tedy mapuje položky v kontejneru systémem klíč-hodnota. Jako klíč byl vybrán název ukládaného objektu, a protože více objektu na mapě stejného typu má stejný název, tak pro uložení hodnoty byl vybrán opět kontejner `vector`, do kterého lze uložit další kontejnery `vector` se souřadnicemi. Tuto strukturu ilustruje obrázek 2.1.



Obrázek 2.1: Ilustrace uložení objektů mapy pomocí STL kontejneru map

[3] DOM parser - XML soubor je celý načten do paměti a jeho struktura je po čas další práce s tímto XML známa. Nevýhodou jsou velké nároky na paměť v případě velkého XML souboru

[4] SAX parser - XML elementy jsou načítány a zpracovávány postupně a během toho není známá struktura XML souboru. Výhodou jsou malé nároky na paměť

3 Rozpoznávání domů

V této kapitole si ukážeme, jaké typy domů budeme rozpoznávat a jaké příznaky pro ně vybereme. Dále provedeme učení vybraného klasifikátoru následně otestujeme klasifikátor na testovací sadě.

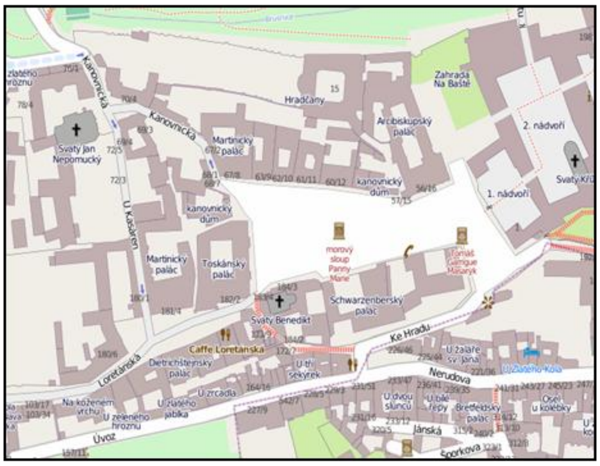
3.1 Typy domů a výběr příznaků

Pro rozpoznávání bylo vybráno šest typů domů a budov, které se ve světě vyskytují (vybráno dle subjektivního názoru autora práce).

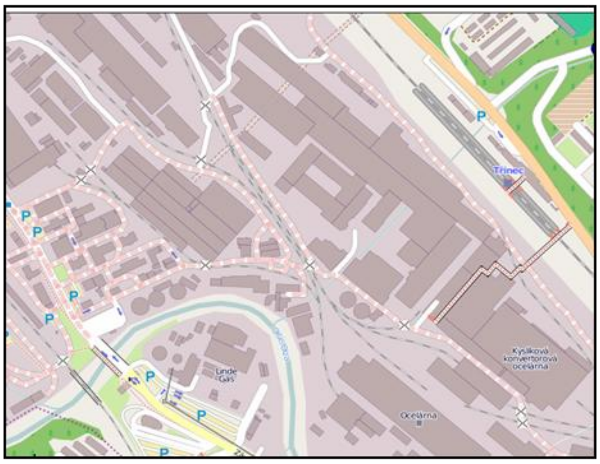
1. Rodinný dům (obrázek 3.1) - nízká stavba, výška kolem 15m, celoročně obydlená. Lze ji najít všude na světě.
2. Panelový dům (obrázek 3.2) - většinou podlouhlá stavba vybudovaná z prefabrikovaných panelů. Panelové domy lze najít hlavně ve městech východní Evropy.
3. Historický dům (obrázek 3.3) - neobyvatelná stavba, většinou starší jak 50-60 let. Historické domy tvoří pomyslný střed mnoha měst, tzv. historickou část města.
4. Továrna (obrázek 3.4) - velká průmyslová budova nebo více budov, kde lidé nebo stroje vyrábějí výrobky ve velkém. Výška budov je různá podle druhu továrny. Nacházejí se na okrajích měst.
5. Mrakodrap (obrázek 3.5) - výšková obývaná budova větší než 100m. Nachází se především ve velkých městech po celém světě.
6. Bytová jednotka (obrázek 3.6) - obyvatelná jednopodlažní jednotka. Těchto jednotek může na sebe ležet více a tvoří tak několika patrovou budovu. Najdeme ji ve většině velkých i středně velkých měst světa.

Obrázek 3.1: Rodinné domy

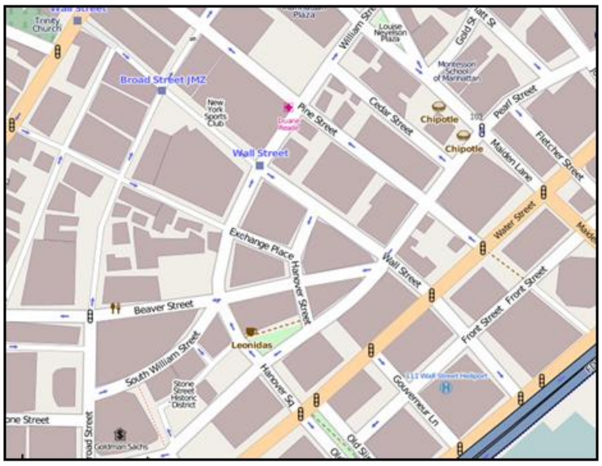
Obrázek 3.2: Panelové domy



Obrázek 3.3: Historické domy



Obrázek 3.4: Továrna



Obrázek 3.5: Mrakodrapy

Obrázek 3.6: Bytové jednotky

Na základě výše uvedených obrázku si můžeme udělat základní představu jak vypadají jednotlivé zástavby s danými typy domů. Vidíme, že např. rodinné domy mají malý obsah plochy, jejich tvar je značně rozdílný a v okolí těchto domů se nachází další podobné domy. Naopak panelové domy mají velký obsah plochy a povětšinou tvarem připomínají obdélník a vybereme-li si nějaké sídliště, tak můžeme stejné budovy najít v okolí. Historické domy jsou značně tvarově rozdílné oproti ostatním typům domů. Továrna má velký obsah, je podélná a narozdíl od panelového domu striktně nepřipomíná obdélník. Navíc jde o skupinu více budov, ze kterých se továrna skládá a tyto budovy jsou hodně blízko sebe. Mrakodrapy mají opět velký obsah, nejsou podlouhlé a v okolí najdeme tvarem a obsahem podobné budovy. Bytová jednotka má malý obsah, jednoduchý tvar a v okolí se nachází stejné nebo hodně podobné jednotky.

Po poznatcích uvedených v předchozím odstavci vybereme 15 příznaku pro každou budovu. Budovy jsou uloženy v paměti a jak již bylo řečeno v 2.1 jedná se o uzavřený objekt na mapě, budeme ho tedy reprezentovat a pracovat s ním jako s polygonem. Příznaky vybrané pro klasifikaci a rozpoznávání jsou:

- obsah budovy - spočítáme pomocí vzorce (8)
- počet stěn budovy
- minimální úhel svíraný mezi stěnami budovy - pomocí vzorce (4) a vektorové algebry
- maximální úhel svíraný mezi stěnami budovy - spočítáme stejně jako předchozí
- poměr stran minimálního ohraničujícího obdélníku - k tomu využijeme poznatky z 1.2.3

Dále si uděláme kolem dané budovy uděláme 200metrové čtvercové okolí a v tomto okolí najdeme všechny budovy (ty tvoří nějakou zástavbu) a spočítáme jim výše uvedené příznaky. Nakonec z těchto jednotlivých příznaků uděláme aritmetické průměry a přidáme je jako příznaky k dané budově a tím nám vznikne dalších 5 příznaků. Posledních pět příznaku spočítáme jako směrodatnou odchylku každého aritmetického průměru. Jak již bylo řečeno, tak u jednotlivých druhů zástavby se v okolí nacházejí stejné nebo podobné budovy, každý dům tak bude mít příznaky z aritmetických průměrů podobné v rámci dané zástavby (typu budovy). Navíc jejich směrodatné odchylky by měly být poměrně malé.

3.2 Trénování a klasifikace domů

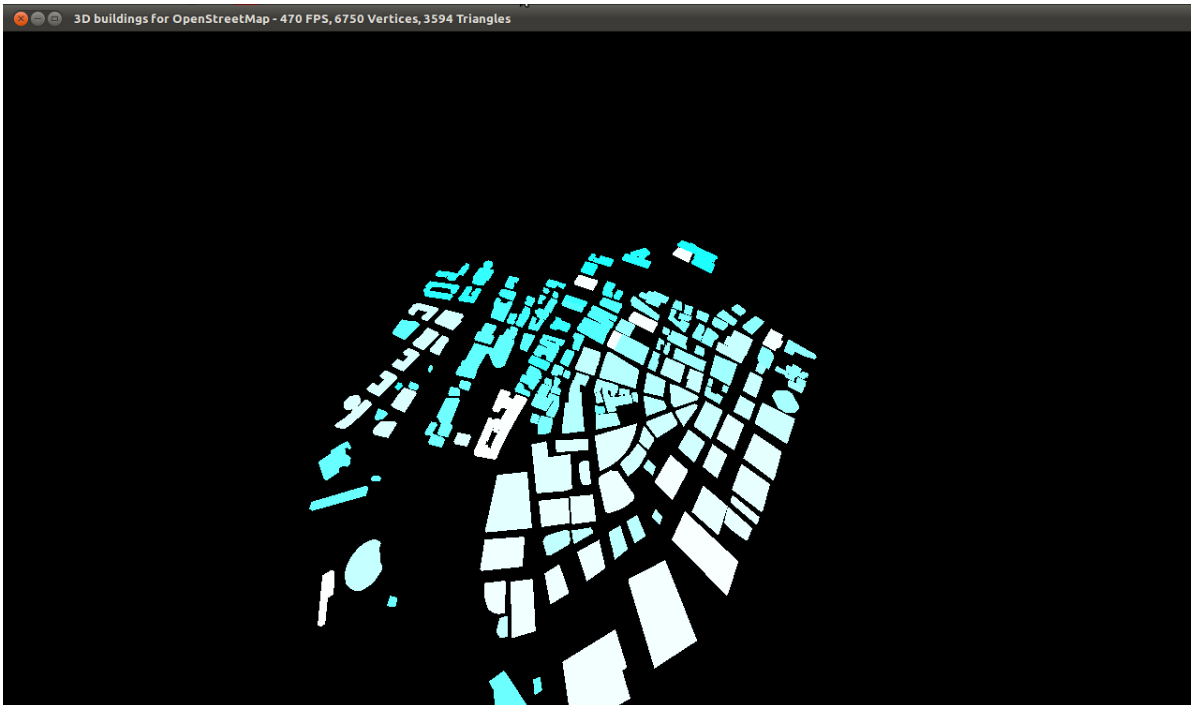
Nyní umíme vypočítat pro každý dům 15 příznaků a můžeme přistoupit k učení klasifikátoru a následovnému testování. Jako klasifikátor byl zvolen SVM^{multiclass} implementovaný panem Thorstenem Joachimsem. Klasifikátor je napsán v jazyce C a poskytován zdarma, pokud je využíván k nekomerčním účelům. Je možné jej najít na stránkách *http://svmlight.joachims.org/*.

3.2.1 Výběr trénovací množiny

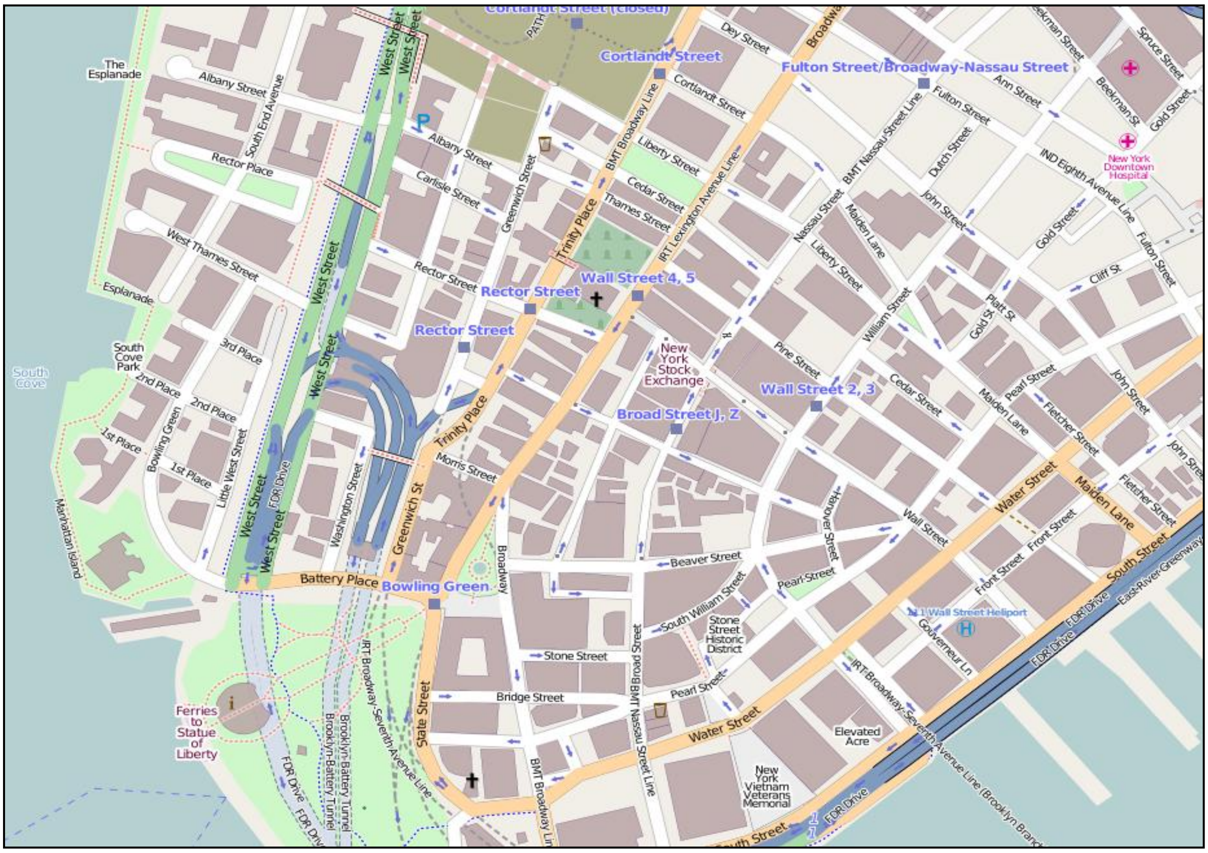
Pro trénování si musíme zvolit nějakou trénovací množinu, která bude obsahovat vektory vypočítaných atributů pro domy všech šesti typů. Vybereme tedy města, o kterých víme, že se tam nachází alespoň jeden typ dané zástavby. Pro přítomnost historického centra a bytových jednotek a rodinných domů zvolíme Prahu. Kvůli továrnímu komplexu vybereme Třinec, kde se nachází Třinecké Železárny.Také zde najdeme množství rodinných domků. Mrakodrapy určitě nalezneme v New Yorku. Nakonec potřebujeme ještě panelové domy a ty jsou mimo jiné i v Moskvě. Vybereme domy, které potřebujeme a vypočítáme pro ně příznaky. To lze provést pomocí vyvíjené aplikace.

Na obrázku 3.7 můžeme vidět jak vypadá mód aplikace, ve kterém je možné vybírat budovy a zařazovat je do jednotlivých tříd. Při výběru se orientujeme pomocí daného mapového podkladu pro tyto budovy, obrázek 3.8.

Vytvoříme si trénovací množinu 2640 budov, které mají svůj vektor příznaků. Nyní můžeme přistoupit k učení klasifikátoru.



Obrázek 3.7: Zástavba mrakodrapů ve městě New York, učící mód aplikace



Obrázek 3.8: Mapový podklad s části města New York, zástavba s mrakodrapy

3.2.2 Trénování klasifikátoru

Jako první provedeme trénování bez zapnutých jádrových funkcí, tzn. využijeme Soft Margin metodu SVM klasifikátoru viz 1.6.2. Budeme nastavovat pouze parametr C, který určuje postih klasifikátoru za chybu. Hodnota tohoto parametru musí být větší než 0. Trénování se spouští pomocí příkazu

```
svm_multiclass_learn -c [float] train model,
```

kde `train` je trénovací soubor s vektory příznaků a `model` je soubor, kde se uloží model pro rozpoznávač. Postupně tedy volíme parametr $C = 0.01, 0.1, 1, 10, 100, 200, 300$ a provádíme učení. Doba potřebná pro učení s daným parametrem C je uvedena v tabulce 3.1.

Pořadí	Parametr C	Doba učení v CPU time [s]
1	0,01	0,8
2	0,1	1,75
3	1	5,67
4	10	28,89
5	100	177,38
6	200	312,15
7	300	435,94

Tabulka 3.1: Doba trénování bez jádrových funkcí

Dále zkusíme klasifikátor natrénovat pomocí zapnutých jádrových funkcí RBF viz 1.6.2. Toto trénování je možné spustit pomocí

```
svm_multiclass_learn -t 2 -c [float] -g [float] train model,
```

kde měníme parametr C a g, který podle 1.6.2 zastupuje písmeno γ. Výsledná doba potřebná pro učení klasifikátoru je k nalezení v tabulce 3.2.

Pořadí	Parametr C	Parametr γ	Doba učení v CPU time [s]
1	2^-5	2^-15	32,1
2	2^-3	2^-13	32,98
3	2^-1	2^-11	33,84
4	2	2^-9	35,93
5	2^3	2^-7	38,36
6	2^5	2^-5	40,21
7	2^7	2^-3	41,4
8	2^9	2^-1	101,2
9	2^11	2	493,36

Tabulka 3.2: Doba trénování se zapnutou jádrovou funkcí RBF

3.2.3 Testování klasifikátoru

Trénováním jsme si vytvořili několik modelu, které nyní použijeme při rozpoznávání budov pomocí klasifikátoru SVM. Pro testování si vytvoříme testovací sadu 5093 budov z měst Ostrava, Paříž, Petrohrad, Brno a Berlín. Testování spustíme příkazem

```
svm_multiclass_classify test model prediction,
```

kde `test` je soubor s testovací sadou, `model` je vytvořený model v předchozí podkapitole a `prediction` je výsledný soubor, kde bude uložen výsledek rozpoznávání. V tabulce 3.3 můžeme vidět úspěšnost rozpoznávání testovací sady pro klasifikátor bez zapnutých jádrových funkcí, naopak v tabulce 3.4 vidíme výsledky rozpoznávání pro klasifikátor se zapnutými jádrovými funkcemi. V těchto tabulkách je také zobrazena úspěšnost rozpoznávání trénovací sady.

Pořadí	Parametr C	Doba rozpoznávání trénovací sady [s]	Přesnost rozpoznávání trénovací sady	Doba rozpoznávání testovací sady [s]	Přesnost rozpoznávání testovací sady
1	0,01	0,00	41,89%	0,00	0,00%
2	0,1	0,01	56,74%	0,00	0,00%
3	1	0,01	63,48%	0,03	0,00%
4	10	0,02	64,13%	0,02	0,00%
5	100	0,01	80,04%	0,01	20,48%
6	200	0,03	80,65%	0,05	23,35%
7	300	0,00	81,10%	0,05	24,92%

Tabulka 3.3: Rozpoznávání bez zapnutých jádrových funkcí

Pořadí	Parametr C	Parametr γ	Doba rozpoznávání trénovací sady [s]	Přesnost rozpoznávání trénovací sady	Doba rozpoznávání testovací sady [s]	Přesnost rozpoznávání testovací sady
1	2^-5	2^-15	9,62	87,27%	18,72	16,81%
2	2^-3	2^-13	9,71	91,52%	19,13	22,66%
3	2^-1	2^-11	10,04	94,43%	19,58	27,27%
4	2	2^-9	10,31	97,35%	20,06	31,34%
5	2^3	2^-7	10,89	99,47%	21,03	42,55%
6	2^5	2^-5	11,04	100,00%	21,28	55,88%
7	2^7	2^-3	11,16	100,00%	21,61	73,49%
8	2^9	2^-1	22,51	100,00%	43,76	86,20%
9	2^11	2	54,97	100,00%	106,38	96,05%

Tabulka 3.4: Rozpoznávání se zapnutou jádrovou funkcí RBF

4 Generování polygonálních modelů

Zde si popíšeme jak se bude generovat terén a budovy, které se nakonec zobrazí pomocí OpenGL.

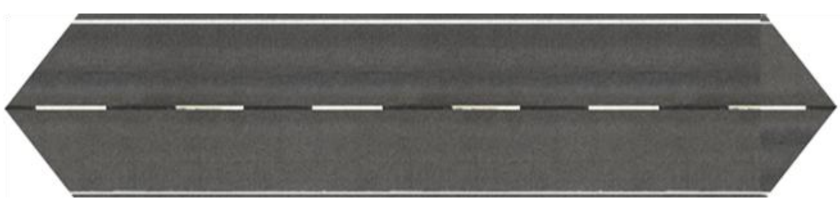
4.1 Generování terénu

Jak bylo řečeno v 1.5, mapy stažené z OpenStreetMap neposkytují informace o výšce, proto se při vytváření terénu omezíme pouze na jeho 2D podobu. Nejprve si vytvoříme obdélníkové polygony, které budou sloužit jako podklad a namapujeme na ně texturu trávy. Počet těchto obdélníků si může uživatel zvolit z omezené nabídky, nebo tento počet nechá automaticky vypočítat aplikací. Plocha vytvořená z obdélníků odpovídá rozměrům stažené mapy.

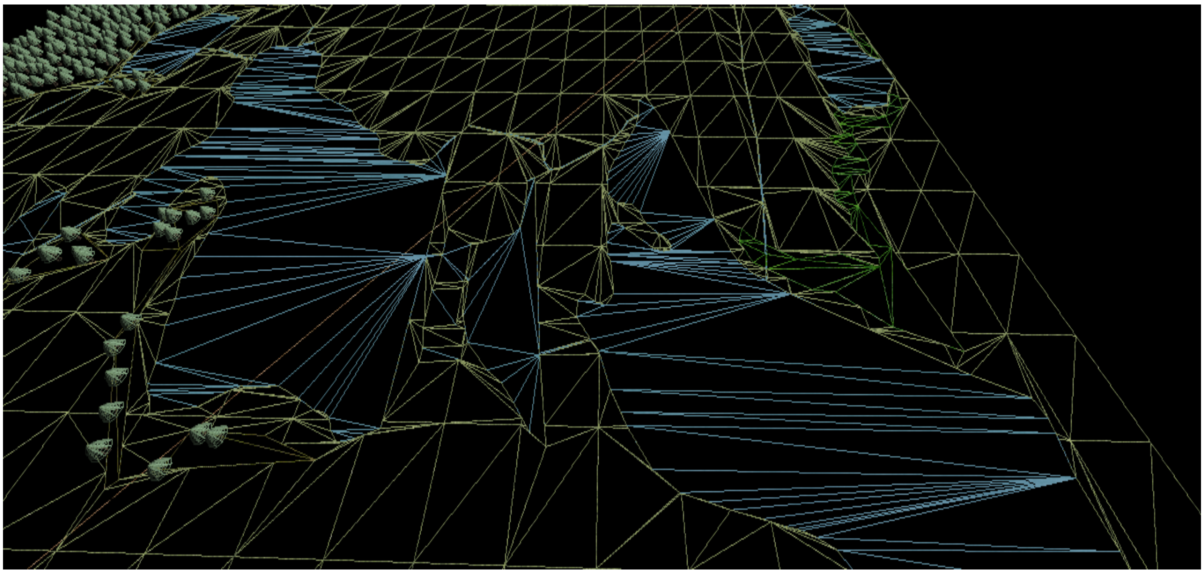
Pro reprezentaci objektů tvořících polygon je vytvořena třída `Polygon`, která ukládá jednotlivé vrcholy polygonu a pozici textury pro tyto vrcholy. Dále poskytuje metody pro triangulaci polygonu. Z paměti 2.2 si vytáhneme všechny uložené objekty mapy, a ty které tvoří polygon si uložíme pomocí třídy `Polygon`. Namapujeme jim požadovanou texturu a spočteme pozici textury pro jednotlivé vrcholy.

Dalším objektem na mapě je úsečka tvořící např. cesty, železnice nebo potoky. Tyto objekty jsou také uložené v paměti a pro jejich reprezentaci je vytvořena třída `LinePolygon`. Třída poskytuje metody pro vytvoření polygonu z úsečky. Aby se s těmito polygony dobře pracovalo, mají tvar šestiúhelníku. Opět si k těmto polygonům namapujeme textury a spočítáme pozici textury pro daný polygon. Pozice se počítá tak, aby byla textura správně orientovaná, jako např. v případě, kdy má cesta nakreslený vodící pruh, viz obrázek 4.1.

Vytvořený podklad z obdélníků osekáme výše vytvořenými polygony pomoci algoritmu Weiler-Atherton 1.2.4 a následně tímto algoritmem osekáme i polygony mezi sebou, pokud se překrývají (který polygon ořezává, a který je ořezávaný se řeší podle zvyklostí vykreslování objektu na mapě v OpenStreetMap). Při ořezávaní musíme v místech ořezu změnit nebo přidat texturové souřadnice, aby nedošlo k rozmazání a deformaci textury. Pro tenhle případ využijeme barycentrické koordináty 1.2.6. Nakonec všechny nově vzniklé polygony ztriangulujeme 1.2.1. Nyní máme vygenerovaný terén. Na obrázku 4.2 vidíme jak vypadá triangulace tohoto terénu.



Obrázek 4.1: Polygon cesty s texturou



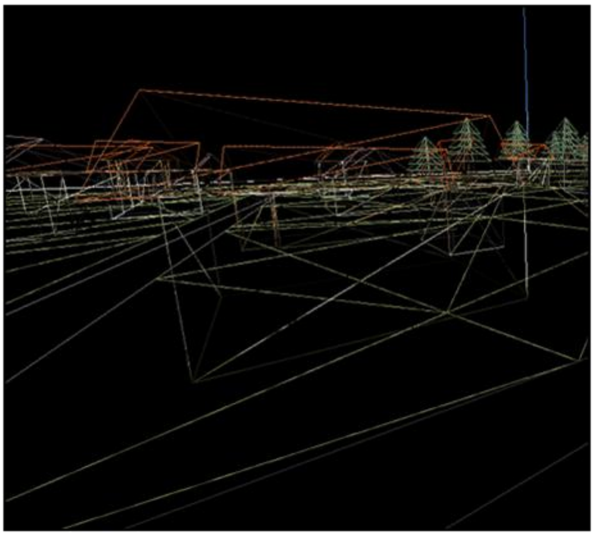
Obrázek 4.2: Síťový model terénu

4.2 Generování 3D modelů domů

Pro vytváření modelů domů byla vytvořena třída `Buildings`, které je předán půdorys budovy v podobě polygonu opět získaného z paměti a typ budovy zjištěný pomocí rozpoznávaní. Pomocí metod této třídy je možné vygenerovat polygony obvodových stěn domu a polygon střechy. Jednotlivým polygonům jsou opět přidány textury. Výška budovy závisí na typu budovy. Na typu také závisí textura, která je mapována na daný typ domu. Ukázky těchto textur nalezneme v příloze.

Pro rodinný dům je navíc možné vytvořit jednoduchou střechu, která má rozměry podle velikosti MBR 1.2.3 daného domu. Tím je zajištěno, že střecha bude pokrývat celý půdorys domu. Příklad rodinného domu je na obrázku 4.3 a na obrázku 4.4 vidíme jeho síťový model.

Obrázek 4.3: Geometrie rodinného domu



Obrázek 4.4: Síťový model rodinného domu

4.3 Generování stromů

Aby byl 2D terén alespoň nějak oživen, tak na místo kde se nachází les si vytvoříme nějaké modely stromů. Stromy mají velmi jednoduchý tvar. Skládají se pouze z kmenu, který tvoří čtyřhranný jehlan pokrytý texturou. Koruny stromu jsou pak tvořeny osmihrannými jehlany s nanesenou texturou. Pro tvorbu těchto modelů je vytvořena třída `TreeGenerator`.

5 Příprava OpenGL k vykreslování

V této kapitole si ukážeme, jak vytvořit 3.0 kontex a core profil knihovny OpenGL. Podíváme se na uspořádání dat ve VBO a naprogramuje si shadery.

Pro vytvoření scény byly využity poskytnuté třídy vedoucím práce. Tyto třídy umožňují provádět operace s maticemi (třída `Transform`), operace s vektory (třída `Vector`) a také načítání textur ve formátu Tga (třída `Tga`).

Aplikace byla vyvíjena na systémy Linux, proto se zde budou objevovat některé funkce použitelné pouze na tomto systému.

5.1 Vytvoření OpenGL 3.0 kontextu

Nejprve je třeba zjistit, zda grafická karta podporuje OpenGL ve verzi 3.0. Využijeme funkci knihovny GLEW viz. 1.7 a otestujeme, zda je podporováno rozšíření pro vytvoření kontextu pomocí `glxewIsSupported("GLX_ARB_create_context_profile")`.

Pokud je rozšíření podporováno, nastavíme si atributy pro vytvoření kontextu verze 3.0 (core profil je nastaven defaultně, proto není potřeba nastavovat daný atribut)

```
int context_attribs[] = {
        GLX_CONTEXT_MAJOR_VERSION_ARB, 3,
        GLX_CONTEXT_MINOR_VERSION_ARB, 0,
        None };
```

Nyní se pokusíme vytvořit kontext pomocí funkce

```
glXCreateContextAttribsARB(Display,GLXFBConfig,
                           GLXContext,Bool,context_attribs);
```

Pokud se to podaří máme kontext 3.0 nastaven.

5.2 Uspořádání dat ve VBO

Na obrázku 5.1 můžeme vidět uspořádání dat reprezentující jeden vrchol uložený ve VBO. Koordináty textury jsou značeny jako t_1 a t_2. Pak následují složky normálového vektoru daného vrcholu n_1, n_2 a n_3. Nakonec jsou umístěné tři souřadnice vrcholu ležícího v prostoru v_1, v_2 a v_3.

t_1	t_2	n_1	n_2	n_3	v_1	v_2	v_3	. . .	n_3	v_1	v_2	v_3

Obrázek 5.1: VBO naplněné daty daných vrcholů

V kapitole 4 jsme si vytvořili polygonální modely, které jsou dány svými vrcholy a mají ke každému vrcholu namapovanou souřadnici textury. Pro daný polygon můžeme vypočítat normálu povrchu pomocí znalostí vektorové algebry v prostoru 1.1.3, konkrétně rovnice (7). Tuto normálu přiradíme každému vrcholu tohoto polygonu a využijeme ji dále při počítání jednoduchého osvětlovacího modelu. Každý polygon má také texturu, která může být pro několik polygonů stejná. Abychom mohli jednotlivé polygony s texturou jednoduše vykreslit, musíme si vrcholy těchto polygonů ve VBO uspořádat a to tak, že polygony se stejnou texturou mají své vrcholy uloženy jdoucí po sobě. Poté následují další polygony s další stejnou texturou. My si pamatujeme, kde nám ve VBO začíná a končí každá textura a pak při vykreslování nastavíme jako aktivní požadovanou texturu a vykreslíme daný počet vrcholu s touto texturou. Toto vykreslování provedeme pro všechny textury. Výhodou takového vykreslování je, že máme všechny vrcholy uložené v jednom VBO a nemusíme se přepínat mezi jinými VBO, což by znamenalo zpomalení vykreslování.

5.3 Vertex a fragment shader

Pomocí vertex shaderu obsluhujeme jednotlivé vrcholy a můžeme pomocí něj vypočítat intenzitu s jakou se má vykreslit jeho barva. Tato barva se zase vytváří ve fragment shaderu. V našem případě je výsledná barva získána z textury.

5.3.1 Vertex shader

Jak již bylo zmíněno, vypočítáme si v něm mimo jiné intenzitu pro zobrazenou barvu každého vrcholu. Tím si vytvoříme jednoduchý osvětlovací model zvaný flat shading, kdy přivrácená strana objektu ke světlu ve scéně bude mít intenzivnější barvu než odvrácená. Nejdříve si zvolíme pozici světla ve scéně. Tato pozice je neměnná. Pro každý vrchol máme vypočítanou normálu. V shaderu si spočteme normálu směřující od zdroje světla směrem k vrcholu. Na základě vypočítaného úhlu svíraného normálou vrcholu a normálou světla je vypočtena intenzita. Příklad našeho vertex shaderu můžeme vidět níže.

```
in vec2 tex; // vstup - pozice vektoru
in vec3 normal; // vstup - normála vrcholu
in vec3 vert; // vstup - pozice vrcholu
uniform mat4 t_modelview_projection_matrix; // transformační matice
uniform mat4 mvMatrix; // modelview matice
uniform mat4 normalMatrix; // normálová matice
uniform vec3 lightSource; // pozice zdroje světla
out float diffuse; // výstup pro fragment shader
out vec2 outTex; // výstup pro fragment shader
void main(void) {
      vec4 l = mvMatrix*vec4(lightSource,1.); // světlo v eye space
      vec3 normalizeNorlmal = normalize(normal);
      vec3 vEye = normalize(mat3(normalMatrix) * normalizeNorlmal);
      // pozice vrcholu v eye space
     vec4 vPosition4 = mvMatrix * vec4(vert,1.f);
      // vektor ke zdroji světla
     vec3 vPosition3 = vPosition4.xyz/vPosition4.w;
     vec3 vLightDir = normalize(l.xyz - vPosition3);
      // hodnota intenzity barvy
     diffuse = max(0.0, dot(vEye, vLightDir));
      gl_Position=t_modelview_projection_matrix * vec4(vert,1.0f);
     outTex = tex; // souřadnice textury pro framebuffer
}
```

Tabulka 5.1: Kód vertex shaderu

5.3.2 Fragment shader

V tomto shaderu pouze spočítáme barvu vrcholu z dané textury a vynásobíme ji intenzitou vypočítanou ve vertex shaderu. K intenzitě ještě předtím připočítáme ambientní složku světla, což je světlo v okolí, které nemá žádný zdroj. Kód shaderu je uveden níže.

```
in float diffuse; // intenzita barvy vypočítána ve vertex shaderu
in vec2 outTex; // koordináty textury z vertex shaderu
uniform sampler2D textures; // aktivní textura
out vec4 frag_Color; // výstupní barva vrcholu
void main(void) {
      vec4 col;
      float ambient = 0.2; // ambientní složka světla
      // výpočet barvy z textury na základě souřadnic
      col = texture(textures, outTex);
      // změna intenzity barvy
      col.rgb = (diffuse + ambient) * col.rgb;
      col.a = 1.0;
      frag_Color=col;
     }
```

Tabulka 5.2: Kód fragment shaderu

6 Výsledky

V této kapitole zhodnotíme dosažené výsledky při rozpoznávání domů a generování terénu. Podíváme se taky na výsledky dosažené při vykreslování různě velkých scén.

6.1 Testovací sestava

Jako testovací sestava byl použit notebook značky MSI řady gx610. Technické parametry tohoto notebooku jsou vedeny v tabulce 6.1.

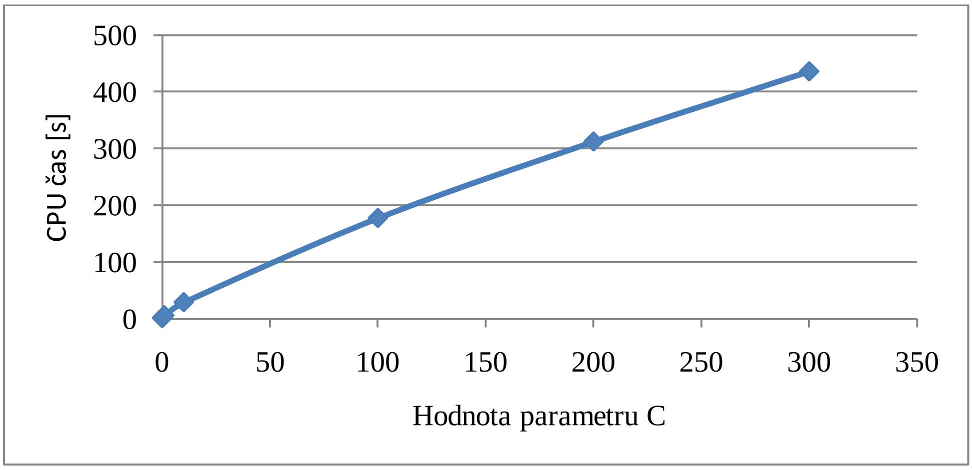
Procesor	Mobile DualCore AMD Turion 64 X2 TL-60, 2000 MHz (10 x 200)
RAM paměť	4GB, DDR2-667 (333 MHz)
Grafická karta	ATI Mobility Radeon HD 2600 (MSI), velikost paměti 256MB
Operační systém	Linux, Ubuntu 10.04

Tabulka 6.1: Testovací sestava

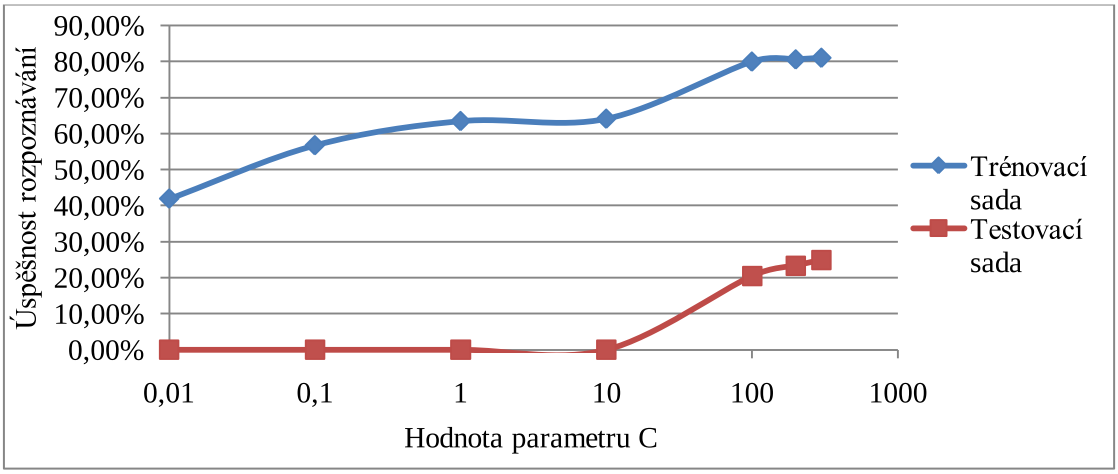
6.2 Výsledky rozpoznávání

V kapitole 3 jsme provedli učení klasifikátoru SVM a to jak lineárního, tak se zapnutými jádrovými funkcemi. Rychlost učení u lineárního je patrná z tabulky 3.1 a tyto výsledky jsou vyneseny v grafu 6.1. Při zvyšování parametru C se zvyšuje i doba učení. Je to dáno tím, že je klasifikátor při vyšším parametru C více postihován za chybnou klasifikaci třídy a hledá tak déle nejlepší lineární nadrovinu, která by rozdělila data v trénovací sadě s co nejmenším počtem chyb. V grafu 6.2 je možno vidět úspěšnost rozpoznávání jak trénovací, tak testovací sady pro jednotlivé parametry C. Doba rozpoznávání se pohybuje kolem 0 až 0,5s CPU výpočetního času.

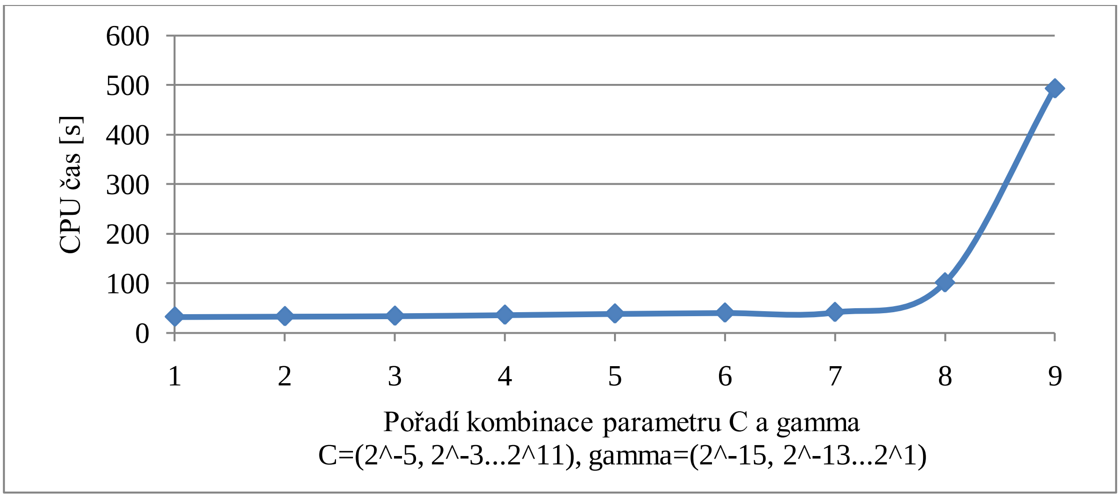
Dále jsme provedli učení klasifikátoru SVM se zapnutou jádrovou funkcí RBF. Doba učení s danými hodnotami parametrů C a γ se podle tabulky 3.2 nebo grafu 6.3 pohybuje od 32 do 493s CPU výpočetního času. Úspěšnost při rozpoznávání trénovací i testovací množiny je výrazně lepší než v případě klasifikátoru bez zapnutých jádrových funkcí. Záleží na dané kombinaci hodnot parametrů C a γ. Výsledky je možné najít v tabulce 3.4 nebo v grafu 6.4. Zvyšuje se také doba potřebná pro rozpoznávání, viz. tabulka 3.4.



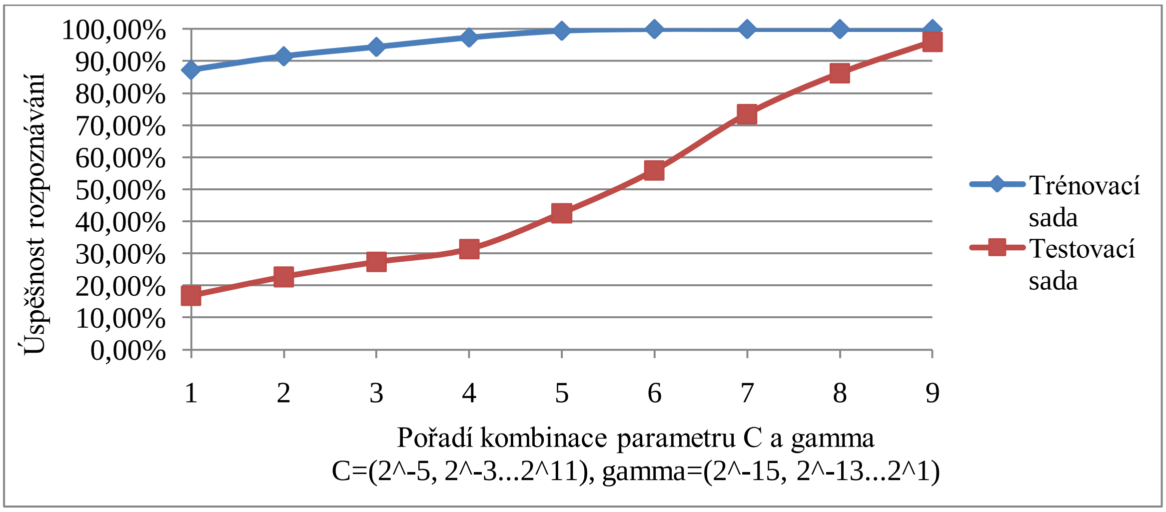
Graf 6.1: Doba učení klasifikátoru bez zapnutých jádrových funkcí



Graf 6.2: Úspěšnost klasifikace trénovací a testovací sady



Graf 6.3:Doba učení klasifikátoru s jádrovou funkcí RBF



Graf 6.4: Úspěšnost klasifikace trénovací a testovací sady

Z grafů je tedy možné vyčíst, že nejlepších výsledků při rozpoznávání dosáhneme se zapnutou jádrovou funkcí RBF a zvolenými parametry $C = 2\text{^}11$ a $\gamma = 2\text{^}1$. Úspěšnost rozpoznávání testovací množiny je 96%. Je ale jasné, že tato úspěšnost nebude stejná pro každé město na světě. Je to dáno tím, že v různých městech se může zástavba daného typu domů lišit. Nicméně pokud generujeme modely budov pro město, kde jsou výsledky rozpoznávání kolem 95-100%, tak tyto modely odpovídají typům domů, které jsme označovali v 3.2.1, kde např. pro zástavbu s rodinnými domy jsou opravdu generovány modely rodinných domů. Naopak největších nepřesností dosahuje klasifikátor při rozpoznávání zástavby s továrnami, kde se místo modelu továren většinou generují modely panelových domů. Je to dáno tím, že tyto zástavby jsou si podobné, pokud je továrna tvořena výrobními halami obdélníkového tvaru.

Vymodelované části některých měst je možno shlédnout v příloze. Pro porovnání je k nim přidán i mapový podklad na jehož základě bylo dané město modelováno.

6.3 Výsledné časy pro generování terénu a FPS při průletu nad scénou

Doba potřebná pro vytvoření terénu velmi závisí na počtu objektu na mapě, kdy se musí všechny objekty testovat mezi sebou, zda je není třeba ořezávat. Po ořezání navíc může vzniknout několik dalších objektů, které je třeba taky testovat. V prvních verzích aplikace byla doba pro vytvoření terénu enormně dlouhá, proto byla implementována optimalizace v podobě řazení objektů do kvadrantů, kde se poté ořezávají pouze objekty ležící v jednom kvadrantu. V tabulce 6.2 a 6.3 jsou uvedené časy pro generování získané při generování terénu pro některá města. Tabulka 6.2 mapuje generování terénu včetně silnic, železnic a potoků, a tabulka 6.3 naopak generování bez nich (v programu je možnost zvolit si mezi těmito dvěma případy). Je patrné, že pro první případ trvá toto generování poměrně dlouho, což je dáno tím, že se cesty, železnice i potoky skládají z úseků a proto vzniká mnoho polygonálních modelů, které je třeba testovat. V tabulce jsou navíc zobrazeny průměrné hodnoty FPS[5], dosažené při průletu nad těmito vygenerovanými městy.

	Se zapnutým vykreslováním cest, železnic a potoků			
Generování terénu	Liberec – střed města, 2000x2000 m	Praha – Střed města, Vltava, 2000x2000m	Střítež u Českého Těšína, 4500x4500m	Lille – Francie, 2000x2000m
Počet obdélníků	1024	1024	1024	1024
Objektu na mapě	1507	1903	1779	1356
Doba vytvoření terénu [s]	821,14	1091,03	1070,95	679,53
Scéna				
Počet vrcholů	168404	75240	337106	361338
Počet trojúhelníků	84106	43790	124478	179426
Průměrné FPS při průletu	200	220	180	175

Tabulka 6.2: Generování terénu 1

	S vypnutým vykreslováním cest, železnic a potoků			
Generování terénu	Šlapanice u Brna, 2000x2000m	Olomouc, okraj města, 4500x4500m	Jablunkov, 2000x2000m	Brno, střed města, 2000x2000m
Počet obdélníků	1024	1024	1024	1024
Objektu na mapě	53	45	33	200
Doba vytvoření terénu [s]	11,4	14,38	6,23	45,83
Scéna				
Počet vrcholů	54368	143377	75168	138410
Počet trojúhelníků	26616	67409	29508	79230
Průměrné FPS při průletu	455	385	400	220

Tabulka 6.3: Generování terénu 2

[5] FPS - Frame Per Second, tato hodnota udává, kolik snímku za sekundu je schopna grafická karta generovat při vykreslování určité scény.

7 Závěr

Cílem této práce bylo vytvoření algoritmu pro rozpoznávání domů pracujícího s daty získanými z mapových podkladů projektu OpenStreetMap. Následovala implementace jednoduchého modelovače, který vytvoří 3D modely domů na základě výsledku rozpoznávače a terén a pomocí grafické knihovny OpenGL zobrazí celou vytvořenou scénu. V důsledku toho musel autor nastudovat problematiku GIS systému, map a souřadnicových systému používaných pro mapování objektů na Zemi. Dále se autor seznámil se strojovým učením, především s klasifikací a s grafickou knihovnou OpenGL

Samotný program je implementován v jazyce C/C++. Jedná se čistě o konzolovou aplikaci, nezabývali jsme se tedy o vytvoření uživatelského rozhraní, které by stejně v této práci uplatnění nenašlo. Aplikace je vytvořena pro operační systém Linux, na kterém byla testována.

V aplikaci je implementován také jednoduchý algoritmus pro triangulaci polygonů, který měl původně sloužit pouze pro triangulování střech budov, ale nakonec byl rozšířen tak, že je schopen zpracovat jakýkoliv jednoduchý polygon, v důsledku čeho je tento algoritmus využit pro triangulaci celé vytvořené scény.

Během tvorby této bakalářské práce autor přicházel na řadu dalších rozšíření, které by do programu mohly být implementovány v budoucnu. Jedná se především o tvorbu terénu s výškou podle skutečného výškového profilu oblasti, která je modelována. Pro to by se dalo využít některé další GIS systémy, které poskytují informace o vrstevnicích a nadmořských výškách požadovaných oblastí. Bylo by dobré také implementovat různé efekty ve vykreslené scéně, jako je např. odraz objektů ve vodě nebo pohyb mraků na obloze. V neposlední řadě je ale třeba opravit chyby projevující se např. špatným ořezáním některých polygonů nebo problikávání textur u cest a železnic tam, kde na sebe jednotlivé úseky navazují, a také vylepšit modely stromů nebo textury domů. Dalo by se také zamyslet nad tím, jak urychlit generování terénu dané mapy.

Díky této práci si autor rozšířil obzory v oblasti pro něj dosud neznáme, kterou je zpracování map a GIS systémy. Zdokonalil se v jazyku C++ a také se naučil pracovat s grafickou knihovnou OpenGL. Plánuje se rozšíření stávající aplikace minimálně v rozsahu uvedeném výše.

Literatura

[1] HAVRLANT, L. *Vektory* [online]. c2006 [cit. 2011-04-28]. Dostupné z WWW: <http://www.matweb.cz/vektory>.

[2] HAVRLANT, L. *Pythagorova věta* [online]. c2006 [cit. 2011-04-28]. Dostupné z WWW: <http://www.matweb.cz/pythagorova-veta>.

[3] BAKER, M. *Maths - Trigonometry - Inverse trig functions* [online]. c1998 [cit. 2011-04-20]. Dostupné z WWW: <http://www.euclideanspace.com/maths/geometry/trig/inverse/index.htm>.

[4] BAKER, M. *Maths - Issues with Relative Angles* [online]. c1998 [cit. 2011-04-20]. Dostupné z WWW: <http://www.euclideanspace.com/maths/algebra/vectors/angleBetween/issues/index.htm>.

[5] KONČEL, J. *Vektorový součin* [online]. 2009 [cit. 2011-04-22]. Dostupné z WWW: <http://www.karlin.mff.cuni.cz/katedry/kdm/diplomky/jan_koncel/vektory.php?kapitola=vektorovySoucin>.

[6] Polygon. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 14.9.2001, last modified on 27.4.2011 [cit. 2011-04-29]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Polygon>.

[7] BERG, M, et al. *Computational Geometry : algorithms and applications*. Berlin : Springer, 2008. 377 s. ISBN 978-3-540-77973-5.

[8] SUNDAY, D. *The Convex Hull of a 2D Point Set or Polygon* [online]. c2001 [cit. 2011-04-30]. Dostupné z WWW: <http://softsurfer.com/Archive/algorithm_0109/algorithm_0109.htm>.

[9] EBERLY, D. *Minimum-Area Rectangle Containing a Convex Polygon* [online]. 9.2.2008 [cit. 2011-04-30]. Dostupné z WWW: <http://www.geometrictools.com/Documentation/MinimumAreaRectangle.pdf>.

[10] ARNON, D; GIESELMANN, J. Technical report : A linear time algorithm area rectangle for the minimum enclosing a convex polygon. [online]. [s.l.] : [s.n.], 7.12.1983 [cit. 2011-04-30]. Dostupné z WWW: <http://www.cs.purdue.edu/research/technical_reports/1983/TR%2083-463.pdf>.

[11] GARNER, W. Shortest *Distance from a Point to a Line* [online]. c2004 [cit. 2011-04-30]. Dostupné z WWW: <http://math.ucsd.edu/~wgarner/math4c/derivations/distance/distptline.htm>.

[12] AGOSTON, M. *Computer Graphics and Geometric Modeling : Implementation and Algorithms*. London : Springer, 2005. 907 s. ISBN 1-85233-818-0.

[13] BRADEN, B. The Surveyor's Area Formula. *The College Mathematics Journal*. 1986, vol. 17, no. 4, s. 326-337. Dostupný také z WWW: <www.maa.org/pubs/Calc_articles/ma063.pdf>.

[14] Barycentric coordinates. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 8.7.2010, last modified on 8.7.2010 [cit. 2011-05-05]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Barycentric_coordinates_%28mathematics%29>.

[15] BADOUEL, D. *Graphics Gems*. San Diego : Academic Press Inc., 1993. An Efficient Ray-Polygon Intersection, s. 390-394. ISBN 0122861663, ISBN 978-0122861666.

[16] HRUBÝ, M. *Geografické Informacní Systémy (GIS)*. [s.l.], 2006. 91 s. Studijní opora. Vysoké učení technocké, Fakulta informačních technologií.

[17] Mercator projection. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 17.12.2001, last modified on 12.4.2011 [cit. 2011-05-05]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Mercator_projection>.

[18] WEISTTEIN, E. *Miller Cylindrical Projection* [online]. c2004 [cit. 2011-05-05]. Dostupné z WWW: <http://mathworld.wolfram.com/MillerCylindricalProjection.html>.

[19] *OpenStreetMap Wiki - Mapping techniques* [online]. c2006, last modified on 6.3.2011 [cit. 2011-05-01]. Dostupné z WWW: <http://wiki.openstreetmap.org/wiki/Mapping_techniques>.

[20] *OpenStreetMap Wiki - Editing* [online]. c2006, last modified on 16.4.2011 [cit. 2011-05-01]. Dostupné z WWW: <http://wiki.openstreetmap.org/wiki/Editing>.

[21] *OpenStreetMap Wiki - Xapi* [online]. c2006, last modified on 4.5.2011 [cit. 2011-05-05]. Dostupné z WWW: <http://wiki.openstreetmap.org/wiki/Xapi>.

[22] *OpenStreetMap Wiki - Elements* [online]. c2006, last modified on 28.3.2011 [cit. 2011-05-01]. Dostupné z WWW: <http://wiki.openstreetmap.org/wiki/Elements>.

[23] MAŘÍK, V, et al. *Umělá inteligence 1*. Praha : Academia, 1993. 264 s. ISBN 80-200-0496-3.

[24] ZBOŘIL, F. *Základy umělé inteligence*. [s.l.], 2007. 142 s. Studijní opora. Vysoké učení technocké, Fakulta informačních technologií.

[25] CORTES, C; VAPNIK, V. Support-Vector Networks. *Machine Learning*. 1995, vol. 20, no. 3, s. 273-297. Dostupný také z WWW: <http://www.springerlink.com/content/k238jx04hm87j80g/fulltext.pdf>.

[26] CHIH-WEI, H; CHIH-CHUNG, CH; CHIH-JEN, L. Technical report : A Practical Guide to Support Vector Classification. [online]. Taipei : National Taiwan University, 2003, 15.4.2010 [cit. 2011-05-08]. Dostupné z WWW: <http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf>.

[27] MADZAROV, G; GJORGJEVIKJ, D; CHORBEV, I. A Multi-class SVM Classifier Utilizing Binary Decision Tree. *Informatica*. 2009, vol. 33, no. 2, s. 233-241. Dostupný také z WWW: <http://www.informatica.si/pdf/33-2/24_madzarov%20-%20a%20multi-class%20svm%20classifier%20utilizing%20bin.pdf>. ISSN 1854-3871.

[28] WRIGHT, R, et al. *OpenGL SUPERBIBLE Fifth Edition : Comprehensive Tutorial and Reference*. USA : Pearson Education Inc., 2011. 969 s. ISBN 0-32-171261-7, ISBN 978-0-32-171261-5.

[29] *OpenStreetMap Wiki - Map Features* [online]. c2006, last modified on 20.4.2011 [cit. 2011-05-08]. Dostupné z WWW: <http://wiki.openstreetmap.org/wiki/Map_Features>.

Seznam příloh

Příloha 1. Seznam vybraných objektů mapy

Název	Popis
highway	Všechny typy cest
cycleway	Cyklistická stezka
waterway	Řeky, potoky
railway	Železnice, tramvajové koleje, metro
aeroway	Letištní dráha
power	Elektrická vedení, elektrické stanice
man_made	Objekty postavené lidmi
building	Většina budov a domů na mapě
leisure	Místa určená k odpočinku nebo sportovním aktivitám
amenity	Zastupuju množství veřejných budov, např. pošta, hospoda, škola
shop	Různé obchody a prodejny
craft	Místo, kde se vyrábí nějaké výrobky
emergency	Místa, odkud je poskytována první pomoc nebo organizovány záchranné práce
tourism	Turistické oblasti, památky
historic	Oblasti, kde se konaly významné historické události. Dále hrady, zámky
landuse	Obytné oblasti, zemědělské plochy, lesy, hřbitovy, vodní plochy atp...
military	Vojenské oblasti a budovy
natural	Objekty spojené s přírodou - bažiny, rašeliniště, ledovce, pláže

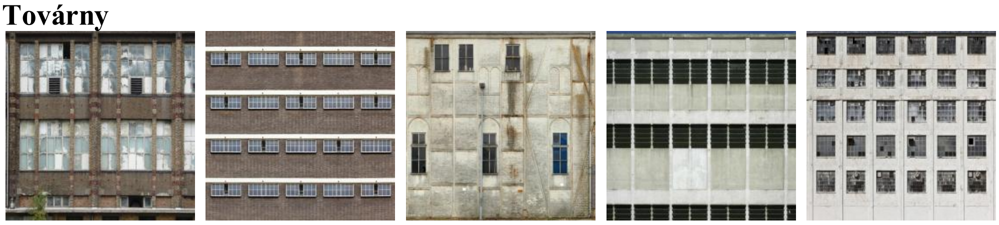
Příloha 2. Textura pro domy

Rodinné domy

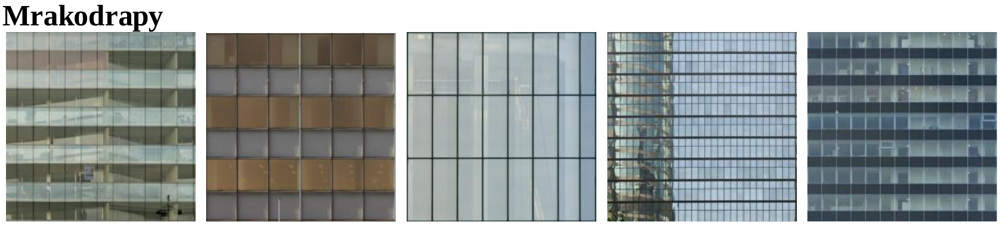
Panelové domy

Historické domy

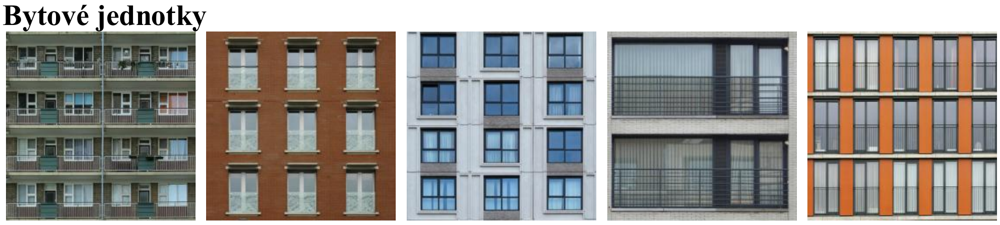
Továrny



Mrakodrapy

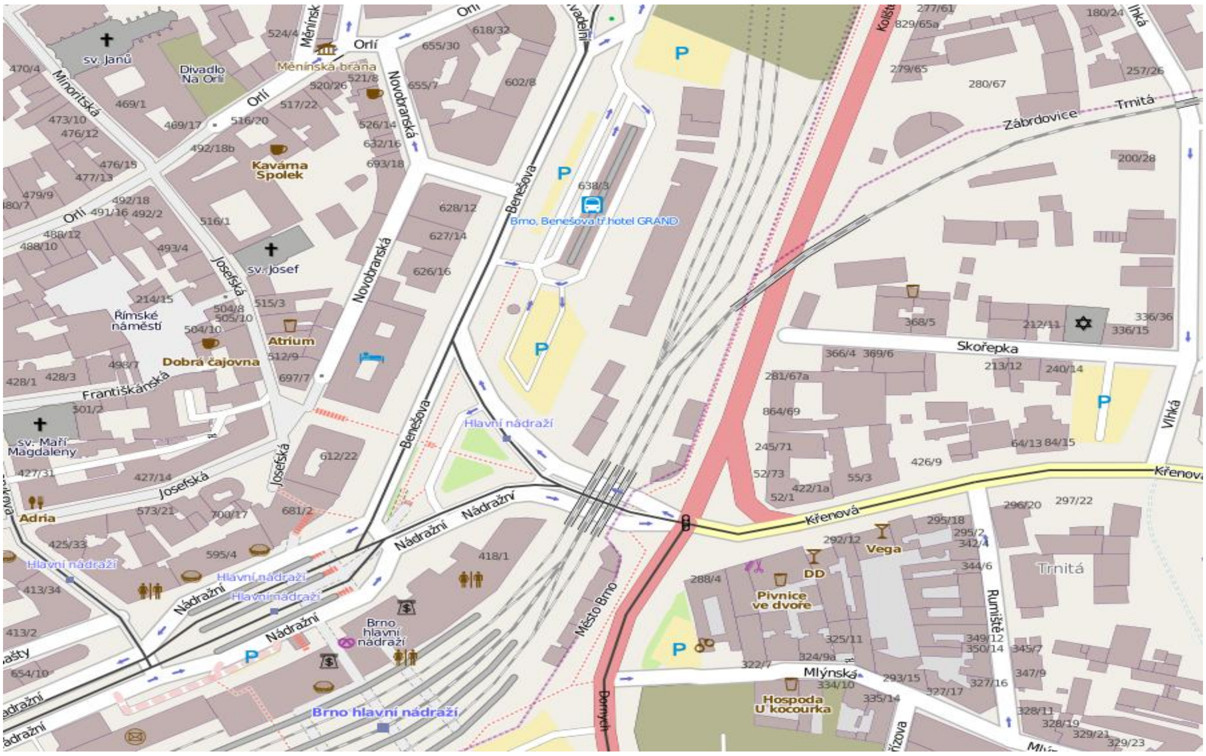


Bytové jednotky

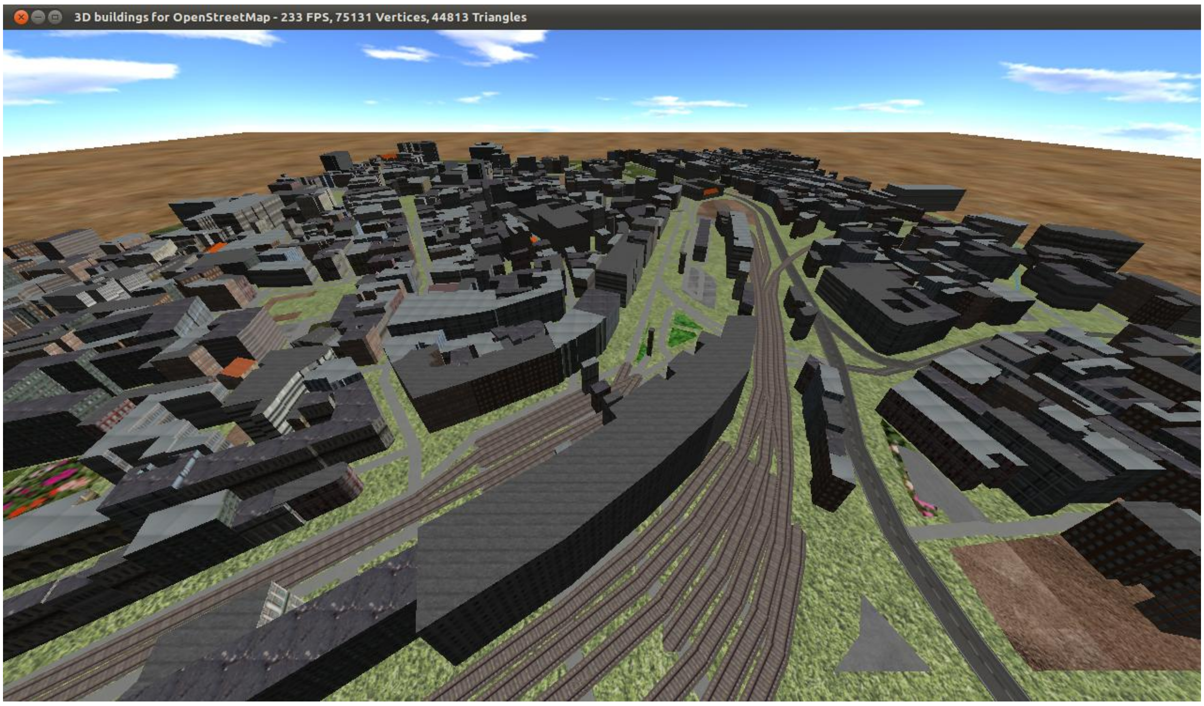


Příloha 3. Obrázky vymodelovaných měst

Brno



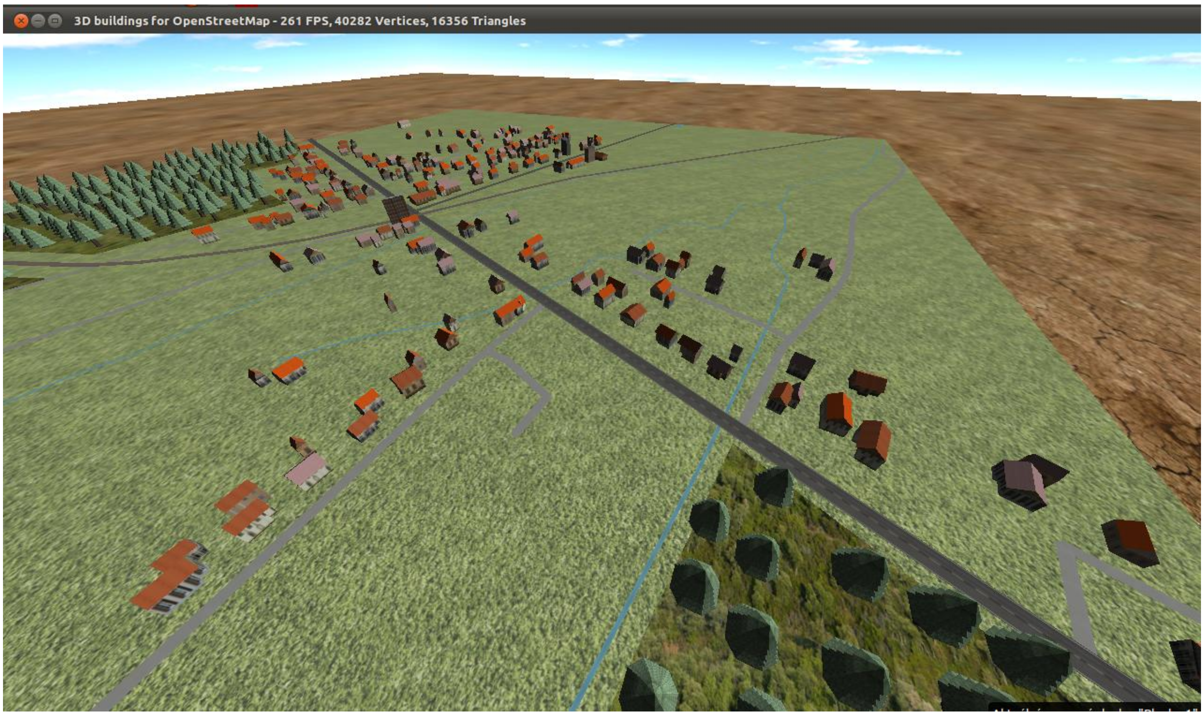
Obrázek k příloze 3: Mapový podklad středu města Brna s hlavním nádražím



Obrázek k příloze 3: Vygenerovaný model na základě výše uvedeného podkladu

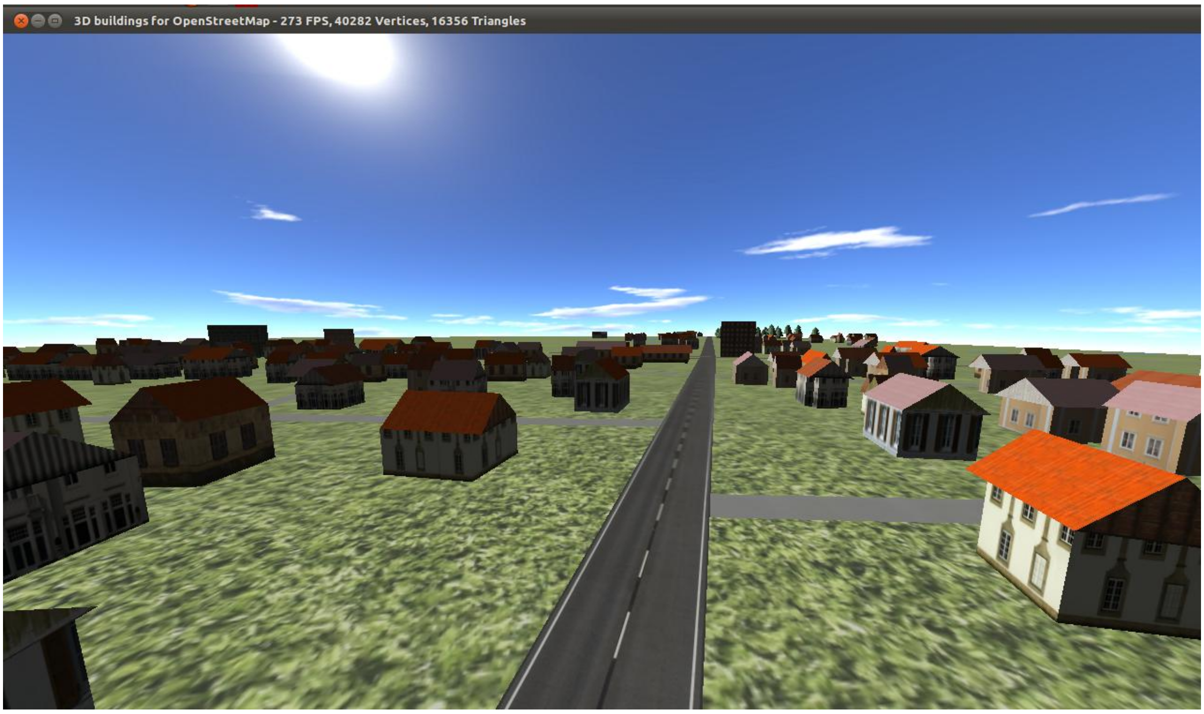
Obec Střítež u Českého Těšína

Obrázek k příloze 3: Mapový podklad obce



Obrázek k příloze 3: Vygenerovaný model obce pro výše uvedený podklad

Ukázka jednoduchého osvětlovacího modelu



Obrázek k příloze 3: Jednoduchý osvětlovací model. Budovy mají tmavší stranu odvrácenou od slunce.

Příloha 4. Instalace aplikace

Pro úspěšné přeložení kódu je třeba mít nainstalovanou knihovnu libxml++2.6 (ke stažení na *http://libxmlplusplus.sourceforge.net/* nebo lze využít správce balíčků, který je součástí distribuce operačního systému Linux), dále knihovnu OpenGL, GLEW, GLXEW a program wget. Projekt je přeložitelný na operačním systému Linux (testováno v prostředí UBUNTU 10.04) a je spustitelný pouze na počítači s grafickou kartou podporující OpenGL verze 3.0 a vyšší.

Z přiloženého CD si stáhneme složku *program*, ve které se nachází soubor Makefile. Spustíme překlad pomocí příkazu *make*. Program se přeloží a spolu sním se také přeloží zdrojové kódy SVM klasifikátoru. V kořenové složce se vytvoří výsledný spustitelný soubor **project**. Pro odstranění přebytečných souborů vzniklých při překladu slouží příkaz *make clean*.

Pro správný běh aplikace se nesmí měnit adresářová struktura aplikace.

Příloha 5. Ovládání aplikace

Popis parametrů

- --h — Zobrazí nápovědu.
- --down [l] [b] [r] [t] — Umožňuje automaticky stáhnout vybranou mapu. Parametr l - levá zeměpisná délka, b - spodní zeměpisná šířka, r - pravá délka, t - horní šířka. Hodnoty je možné najít na stránkách *www.openstreetmap.org*, kde si vybereme požadovaný úsek mapy, klikneme na *export* a uprostřed nahoře vidíme čtyři políčka seskupené do kříže. Hodnoty v nich odpovídají výše uvedeným hodnotám.
- --file [filename] — Z výše uvedených stránek projektu OpenStreetMap je možné přímo vyexportovat požadovanou mapu ve formátu XML. Tuto mapu pak můžeme přímo předat aplikaci, která ji zpracuje.
- --load_model [filename] — Pokud máme uložený model, vytvořený naší aplikací, můžeme jej pomocí tohoto parametru načíst a zobrazit.
- --learn [filename] — Pomocí tohoto parametru spustíme učící mód aplikace, kdy ji předáme soubor s mapovým podkladem, kde se nachází půdorysy budov. V tomto módu můžeme jednotlivé budovy ohodnocovat, tedy řadit je do šesti tříd. Po uložení se vygeneruje soubor, kde jsou jednotlivé budovy spolu s vektorem příznaků. Tento soubor může být použit pro učení klasifikátoru SVM.

Dále se řídíme pokyny, které jsou vypisovány do konzole.

POZOR: Pokud je stažena mapa s měřítkem menším jak 1:3400, může vytváření modelu trvat velmi dlouho, pokud ponecháme vykreslování cest, železnic a potoku (hlavně tam, kde je jich velké množství, např. nějaké větší město). Obecně platí, že pokud se v konzoli zobrazí počet objektů kolem 2000, trvá vytvoření modelu asi 20min.

Pohyb ve vytvořené scéně

Ve scéně se lze pohybovat pomocí šipek. Zatáčet se dá pomocí myši, pokud držíme zmačknuté levé tlačítko myši. Klávesami [right shift] a [right ctrl] se zrychluje resp. zpomaluje pohyb. Klávesou [l] se zobrazí síťový model scény a klávesou [esc] se program ukončí.

Učící mód

V módu se lze pohybovat po scéně stejně jako je popsáno výše. Pro označení budovy se používá pravé tlačítko myši. Ostatní instrukce jsou vypsány v konzoli.

Příloha 6. Obsah přiloženého CD

Kořenová složka CD obsahuje tři složky, `doc`, `poster`, `program`. Ve složce `doc` je možno najít technickou zprávu práce, v `poster` je vytvořený plakát v elektronické podobě a ve složce `program` se nachází zdrojové kódy a další soubory vlastního programu.

Ve složce `program` je navíc možné najít složku `doxy` s vygenerovanou dokumentací pomocí programu Doxygen a složku `examples` s ukázkovými příklady. V této složce se nachází další složka `city_models`, kde je možné najít již vygenerované modely měst a lze je načíst v programu pomocí parametru `--load_model`. Dále se zde nachází složka `SVM_models`, kde jsou uloženy jednotlivé modely vygenerované při trénování klasifikátoru. Ve složce `train_and_test_set` je možné nají trénovací a testovací sadu pro klasifikátor SVM.