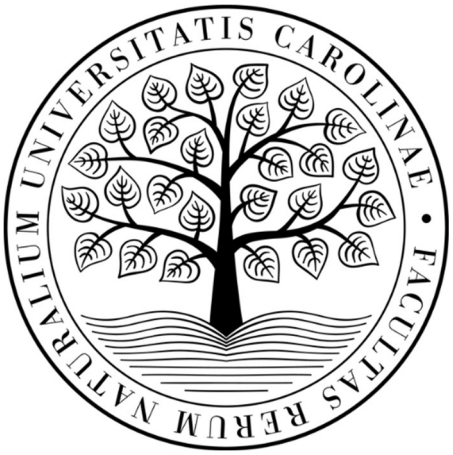
Univerzita Karlova

Přírodovědecká fakulta



Algoritmy počítačové kartografie

Geometrické vyhledávání bodu

Josef Vojkovský, Jan Prýmek

1. ročník N-GKDPZ

Praha 2024

Úloha č. 1: Geometrické vyhledávání bodů

Vstup: Souvislá polygonová mapa n polygonů $\{P_1, ..., P_n\}$, analyzovaný bod q.

Výstup: P_i, $q \in P_i$.

Nad polygonovou mapou implementujte Ray Crossing Algorithm pro geometrické vyhledání incidujícího polygonu obsahujícího zadaný bod q.

Nalezený polygon graficky zvýrazněte vhodným způsobem (např vyplněním, šrafováním, blikáním). Grafické rozhraní vytvořte s využitím frameworku QT.

Pro generování nekonvexních polygonů můžete navrhnout vlastní algoritmus či použít existující geografická data (např. mapa evropských států).

Polygony budou načítány z textového souboru ve Vámi zvoleném formátu. Pro datovou reprezentaci jednotlivých polygonů použijte špagetový model.

Hodnocení:

Krok	**Hodnocení**
Detekce polohy bodu rozlišující stavy uvnitř, vně polygonu.	10b
Analýza polohy bodu (uvnitř/vně) metodou Winding Number Algorithm.	*+10b*
Ošetření singulárního případu u Winding Number Algorithm: bod leží na hraně polygonu.	*+5b*
Ošetření singulárního případu u Ray Crossing Algorithm: bod leží na hraně polygonu.	*+5b*
Ošetření singulárního případu u obou algoritmů: bod je totožný s vrcholem jednoho či více polygonů.	*+2b*
Zvýraznění všech polygonů pro oba výše uvedené singulární případy.	*+3b*
Rychlé vyhledávání potenciálních polygonů (bod uvnitř min-max boxu).	*+10b*
Řešení pro polygony s dírami.	*+10b*
Max celkem:	**55b**

= implementovaná funkcionalita

Point-in-Polygon Problem

"Point in Polygon Problem" (PIPP) je základní geometrický problém v počítačové grafice a geografických informačních systémech (GIS). Spočívá v určení, zda se daný bod nachází uvnitř nebo vně polygonu (či na jeho hranici). Tento problém má mnoho praktických využití, včetně geografické analýzy, detekce kolizí ve hrách, plánování cest a dalších. Možnosti řešení se pro konvexní a nekonvexní útvary liší zejména v náročnosti algoritmů potřebných pro zohlednění specifických případů polohy vrcholů jednotlivých polygonů (Rourke 2005). Bayer (2024) rozlišuje 2 základní techniky řešení PIPP – planární dělení roviny a převedení problému na vztah bodu a mnohoúhelníku. První technika rozděluje rovinu na množinu pásů či lichoběžníků a vzniká tzv. trapezoidální mapa. Tento přístup je výrazně rychlejší, avšak jeho implementace je náročnější. Druhá technika opakovaně určuje polohu bodu vzhledem k mnohoúhelníku. Tento koncept je pomalejší, ale snadněji implementovatelný.

Detekce vztahu mezi bodem a konvexním útvarem může být prováděna prostřednictvím jednoduchých algoritmů, jako je test polohy bodu vzhledem k jednotlivým hranám útvaru (nazývaný "Half-plane test") s časovou složitostí O(n), kde n je počet hran útvaru. Tyto algoritmy jsou často využívány v triangulačních procesech, avšak mnoho z těchto algoritmů nelze samostatně použít pro nekonvexní útvary. Pro detekci vztahu mezi bodem a nekonvexními útvary proto existují dva základní algoritmy: Ray Crossing Algorithm a Winding Number Algorithm. Oba tyto algoritmy mají časovou složitost O(n), kde n je počet hran polygonu. Nicméně jeden z nich se v praxi ukazuje jako výrazně rychlejší (Rourke 2005).

Každý z algoritmů má své výhody a nevýhody a vhodnost pro použití závisí na konkrétní situaci, například na složitosti polygonu, požadované přesnosti a požadavcích na výpočetní výkon.

Ray Crossing Algorithm

Ray Crossing Algorithm je jedním z nejpopulárnějších algoritmů. Vzájemný vztah polohy bodu vzhledem k polygonu zjišťuje tak, že se počítá počet průsečíků, které tvoří polopřímka vedená z daného bodu s hranami polygonu. Pokud je počet průsečíků sudý, bod se nachází vně polygonu, pokud je lichý, nachází se uvnitř. Je-li s uvažovaným bodem totožný právě jeden průsečík, bod se nachází na hraně polygonu. Tento algoritmus je relativně efektivní a snadno implementovatelný.

Podstata algoritmu

Mějme uzavřenou oblast *O* v R^2, jejíž hrany jsou tvořené množinou bodů $P = \{P_1, ..., P_n, P_1\}$ a bod *q*. Uvažujme vodorovnou testovací přímku *r* (paprsek, tzv. ray) procházející bodem *q* takovou, že

$$r(q): y = yq.$$

Počet průsečíků *k* přímky *r* s oblastí *O*, pak určuje polohu bodu *q* vůči oblasti *O* tak, že

k % 2 = 1 → bod $q \in$ oblasti *O*

k % 2 = 0 → bod $q \in$ oblasti *O*

Takovýto algoritmus je o řád rychlejší než Winding Number Algorithm představený níže, ale neošetřuje možné singularity – když *r(q)* prochází vrcholem *Pi* nebo hranou a neumí detekovat případ, když *q* leží na hraně *δ*. Z těchto důvodů je vhodné provést modifikaci algoritmu s redukcí ke *q* a s rozdělením *r* na dvě polopřímky r_1 a r_2 s opačnou orientací, kde r_1 je levostranná a r_2 pravostranná polopřímka. Zavedeme-li si lokální souřadnicový systém s počátkem v bodě *q* a osami *x'*, *y'* a polopřímky $r_1(q)$, $r_2(q)$ ztotožníme s osou *x'* tak, že je možné je popsat rovnicí (Bayer 2024)

$$y' = 0,$$

můžeme provést redukci bodů *pi = [xi, yi]* ke *q*:

$$x'i = xi - xq,$$

$$y'i = yi - yp.$$

Pokud existuje průsečík *M* = [*x'm, y'm* = 0] hrany oblasti *O* a osy *x'* (jedné z polopřímek *x', y')*, můžeme ho určit ze vztahu

$$x'_m = \frac{x'_{i+1}y'_i - x'_i y'_{i+1}}{y'_{i+1} - y'_i}$$

Podmínky existence průsečíku M s jednou z polopřímek $r_1(q)$, $r_2(q)$ udávají vztahy:

pro levou polorovinu: $t_l = y_{i+1} < y_q \neq y_i < y_q$

pro pravou polorovinu: $t_r = y_{i+1} > y_q \neq y_i > y_q$,

kde t_l, t_r nabývají hodnoty *True* nebo *False*. Průsečík *M* se pak vypočítá pro každou polopřímku zvlášť:

pokud $t_l = True \wedge x'_m < 0$, inkrementujeme k_l,

pokud $t_r = True \wedge x'_m > 0$, inkrementujeme k_r.

Průsečíky pro levou a pravou část tedy budeme započítávat v případě, že počáteční a koncový bod protnuté hrany leží v jiné polorovině (vrchní nebo spodní). Pak *q*

$$\in \delta O, k_l \,\%\, 2 \neq k_r \,\%\, 2,$$

$$\in O, k_r \,\%\, 2 = 1,$$

$$\notin O, \text{jinak.}$$

Speciální případy algoritmu

Při řešení PIPP pomocí Ray Crossing Algorithm může docházet k singulárním případům, které je nutné dodatečně ošetřit.

Pokud je zvolený bod *q* totožný s jedním z bodů *pi*, pak je možné prohlásit, že $q \in \delta O$.

Pokud přímka *r(q)* prochází vrcholem oblasti *O*, může nastat detekce dvou průsečíků (koncový bod jednoho segmentu a počáteční bod druhého segmentu). Situaci je možné ošetřit započtením tohoto vrcholu jako průsečíku jenom jednou (Rourke 2005).

Pokud platí, že

$$y_{i+1} - y_i = 0,$$

pak body p_{i+1} a $p_i$tvoří horizontální hranu a výpočet průsečíku *M* ze vztahu výše nebude možný. Ve vlastní implementaci algoritmu se v tomto případě pokračuje následují iterací.

Může nastat situace, když se bod p_i chybně zařadí do vrchní nebo spodní poloroviny oblasti *O*, pokud tento bod leží velmi blízko testovací přímky *r(q)*. Je proto vhodné zavést prahovou hodnotu ε, která tento případ ošetří:

$$|y_i - y_q| \leq \varepsilon.$$

Pseudokód

V metodě rayCrossingAlgorithm v souboru algorithms.py se nachází vlastní implementace Ray Crossing Algorithm. Níže je shrnuta v pseudokódu.

kr= 0 → inicializace počtu pravostranných průsečíku
kl = 0 → inicializace počtu levostranných průsečíku
n = len(pol) → délka polygonu

for i in range(n): → pro všechny vrcholy v polygonu v rozsahu *n* udělej:
    xi = pol[i].x() - q.x() → spočítej x_i
    yi = pol[i].y() - q.y() → spočítej y_i

    if xi == 0 and yi == 0: → pokud bod leží na vrcholu
        return -1 → vrať hodnotu -1 (bod leží na hraně)

    xi1 = pol[(i+1) % n].x() - q.x() → spočítej x_{i+1}
    yi1 = pol[(i+1) % n].y() - q.y() → spočítej y_{i+1}

    if (yi1 – yi) == 0: → pokud (y_{i+1} - y_i) == 0 (horizontální segment) pak:
        continue → pokračuj novou iterací

    *xm = (xi1 * yi - xi * yi1) / (yi1 - yi)* → spočítej průsečík x'_m

    if (yi1 < 0) != (yi < 0): → pokud (y_{i+1} < 0) ≠ (y_i < 0) (spodní segment), pak:
        if xm < 0: → pokud x'_m < 0 (průsečík v levé polorovině), pak:
            kl += 1 → inkrementuj k_l

    if (yi1 > 0) != (yi > 0): → pokud (y_{i+1} > 0) ≠ (y_i > 0) (vrchní segment), pak:
        if xm > 0: → pokud x'_m > 0 (průsečík v pravé polorovině), pak:
            kr += 1 → inkrementuj k_r

    if kl % 2 != kr % 2: → pokud k_l % 2 ≠ k_r % 2, pak:
        return -1 → vrať hodnotu -1 (bod leží na hraně)

    elif kr % 2 == 1: → pokud k_r % 2 == 1, pak:
        return 1 → vrať hodnotu 1 (bod leží uvnitř polygonu)

    else: → jinak
        return 0 → vrať hodnotu 0 (bod leží vně polygonu)

Winding Number Algorithm

Winding Number Algorithm využívá konceptu úhlu otočení mezi bodem a hranami polygonu. Je založen na sčítání a odčítání úhlů, které vznikají při rotaci vektoru od testovaného bodu k vrcholům polygonu. Pokud je tento součet úhlů 2π, bod se nachází uvnitř polygonu. Tento algoritmus je přesný, ale může být náročnější na implementaci.

Podstata algoritmu

Mějme uzavřenou oblast *O* v R^2, jejíž hrany jsou tvořené množinou bodů $P = \{P_1, ..., P_n, P_1\}$ a bod *q*. Výsledkem Winding Number Algorithm je hodnota vzniklá součtem všech úhlů, které opíše průvodič v polygonu, přičemž tedy platí vztah

$$\Omega(q, P) = \frac{1}{2\pi} \sum_{i=1}^{n} \omega(p_i, q, p_{i+1}),$$

Pro výpočet celkového úhlu pro daný polygon je nejprve nutné analyzovat polohu bodu *q* vůču každé přímce. Každá přímka je definovaná dvěma po sobě následujícími vrcholy p_i a p_{i+1}, jež tvoří hranu polygonu. Vzájemná poloha bodu a dané přímky se vyšetří pomocí Half-plane testu, kde podle Bayera (2024) mohou nastat 3 situace – bod *q* leží vpravo od přímky, bod *q* leží vlevo od přímky, nebo bod *q* leží leží na přímce. Jako testovací kritérium se použije vztah pro výpočet determinantu matice, která se skládá z vektorů $\underset{p}{\rightarrow} = (p_x, p_y)$ a $\underset{s}{\rightarrow} = (s_x, s_y)$:

$$\underset{p}{\rightarrow} = p_{i+1} - p_i,$$

$$\underset{s}{\rightarrow} = q - p_i,$$

Vztah pro výpočet determinantu je následující: $det = (p_x * s_y) - (s_x * p_y)$. Pokud:

det > 0 → bod q leží vlevo od přímky

det < 0 → bod q leží vpravo od přímky

det = 0 → bod q leží na přímce

Následně je potřeba procházet jednotlivé hrany v polygonu a sčítat, či odečítat všechny úhly $\omega_i + \omega_{i+1} + ... + \omega_n$ ve směru hodinových ručiček (nebo v opačném směru) v závislosti na výsledku determinantu. Pro výpočet úhlu ω je nutné vypočítat vektory: $\underset{u}{\rightarrow} = (u_x, u_y)$ a $\underset{v}{\rightarrow} = (v_x, v_y)$ jako:

$$\underset{u}{\rightarrow} = p_i - q,$$

$$\underset{v}{\rightarrow} = p_{i+1} - q.$$

Velikost úhlu se pak spočítá podle vztahu:

$$\cos(\omega) = \frac{\underset{u}{\rightarrow} * \underset{v}{\rightarrow}}{|| \underset{u}{\rightarrow} || * || \underset{v}{\rightarrow} ||}$$

Pro následující kroky je nutné pracovat s absolutní hodnotu velikosti úhlu, jelikož se bude v dalším kroku rozhodovat o jeho přičtení, nebo odečtení.

$$|\omega| = \arccos\left(\frac{\underset{u}{\rightarrow} * \underset{v}{\rightarrow}}{|| \underset{u}{\rightarrow} || * || \underset{v}{\rightarrow} ||}\right)$$

Bude-li *det < 0*, pak *-= ω*, bude-li *det > 0*, pak *+= ω.*

Následně se všechny úhly sečtou podle vztahu $\Omega = 2\pi$, $q \in O$, nebo $\Omega < 2\pi$, $q \notin O$.

Speciální případy algoritmu

Algoritmus popsaný výše je schopný najít bod uvnitř nebo vně polygonu. Pro případ, kdy se bod nachází na hraně polygonu, tedy *det = 0*, je třeba algoritmus modifikovat. K modifikaci ale nestačí pouze podmínka nulového determinantu, a proto se dále dá detekovat bod na hraně právě tehdy, když $\omega = \pi$.

Výše uvedené dva vztahy ovšem nezohledňují situaci, kdy se bod nachází v jednom z vrcholů množiny *P*. Pro ošetření této možné situace se využije podmínky:

$$if\ q\ =\ p_{i.}$$

Bod se pak nachází na vrcholu, neboli na hraně polygonu.

Pseudokód

V metodě windingNumberAlgorithm v souboru algorithms.py se nachází vlastní implementace Winding Number Algorithm. Níže je shrnuta v pseudokódu.

n = len(pol) → délka polygonu
total_angle = 0 → inicializace velikosti úhlu
eps = 1.0e-10 → velmi malá kladná prahová hodnota blízká 0

for i in range(n): → pro všechny vrcholy v polygonu v rozsahu *n* udělej:
    if (q == pol[i]) or (q == pol[(i + 1) % n]): → pokud bod leží na vrcholu
        return -1 → vrať hodnotu -1 (bod leží na hraně)

    ux = pol[(i+1) % n].x() - q.x() → spočítej vektor p: p_{i+1} - i pro x
    uy = pol[(i+1) % n].y() - q.y() → spočítej vektor p: p_{i+1} - i pro y

    vx = q.x() - pol[i].x() → spočítej vektor s: q - p_i pro x
    vy = q.y() - pol[i].y() → spočítej vektor s: q - p_i pro y

    *det = (ux * vy) - (vx * uy)* → spočítej determinant z vektorů *p* a *s*

    ux = pol[i].x() - q.x() → spočítej vektor u: p_i - q
    uy = pol[i].y() - q.y() → spočítej vektor u: p_i - q

    vx = pol[(i+1) % n].x() - q.x() → spočítej vektor v: p_{i+1} - q
    vy = pol[(i+1) % n].y() - q.y() → spočítej vektor v: p_{i+1} - q

    angle = computeAngle(ux, uy, vx, vy) → spočítej úhel ω z vektorů u a v

    if det > 0: → pokud je determinant větší než 0, pak
        total_angle += angle → přičti daný úhel do sumy úhlu

    elif det < 0: → pokud je determinant menší než 0, pak
        total_angle -= angle → odečti daný úhel od sumy úhlu

    if det == 0 and abs(angle - pi) < eps: → pokud je determinant 0 a zároveň je abs. hodnota z úhlu (u a v) - π menší než eps, pak:
        return -1 → vrať hodnotu -1 (bod leží na hraně)

*if abs(abs(total_angle) - 2*pi) < eps:* → pokud abs.hodnota (z (abs. hodnoty total_angle)) - 2π je menší než eps, pak:
    return 1 → vrať hodnotu 1 (bod leží uvnitř polygonu)

else: → jinak
    return 0 → vrať hodnotu 0 (bod leží vně polygonu)

Vlastní vypracování úkolu

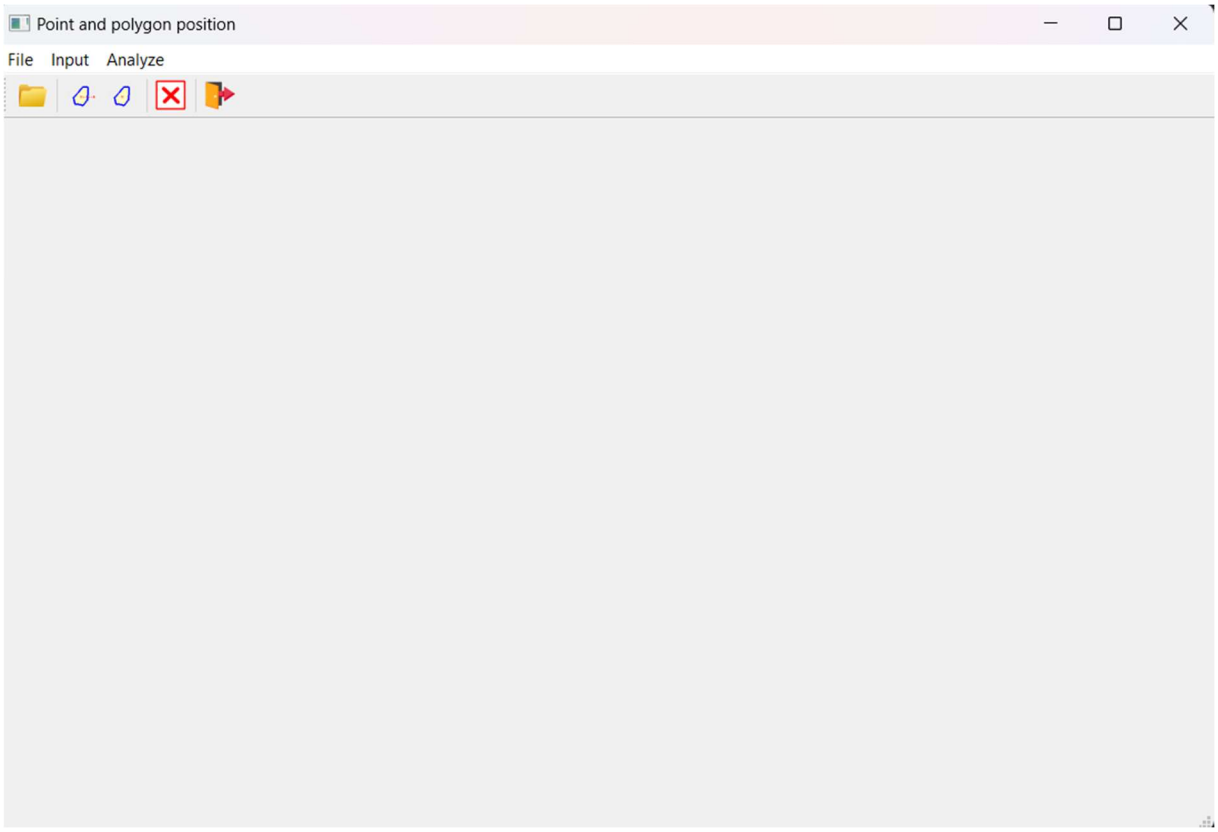
Pro zpracování bylo pomocí frameworku Qt vytvořeno grafické uživatelské rozhraní, ve kterém lze oba algoritmy vyzkoušet na nahrané polygonové vrstvě.

Vstupní data

Jakožto vstupní data pro demonstraci byla použita polygonová vrstva krajů ČR bez Středočeského kraje v kartografickém zobrazení EPSG: 5514 – S-JTSK / Krovak East North. Do aplikace lze nahrát jakýkoli soubor typu Esri Shapefile (.shp). Testovací data jsou přiložena ve složce input_files.

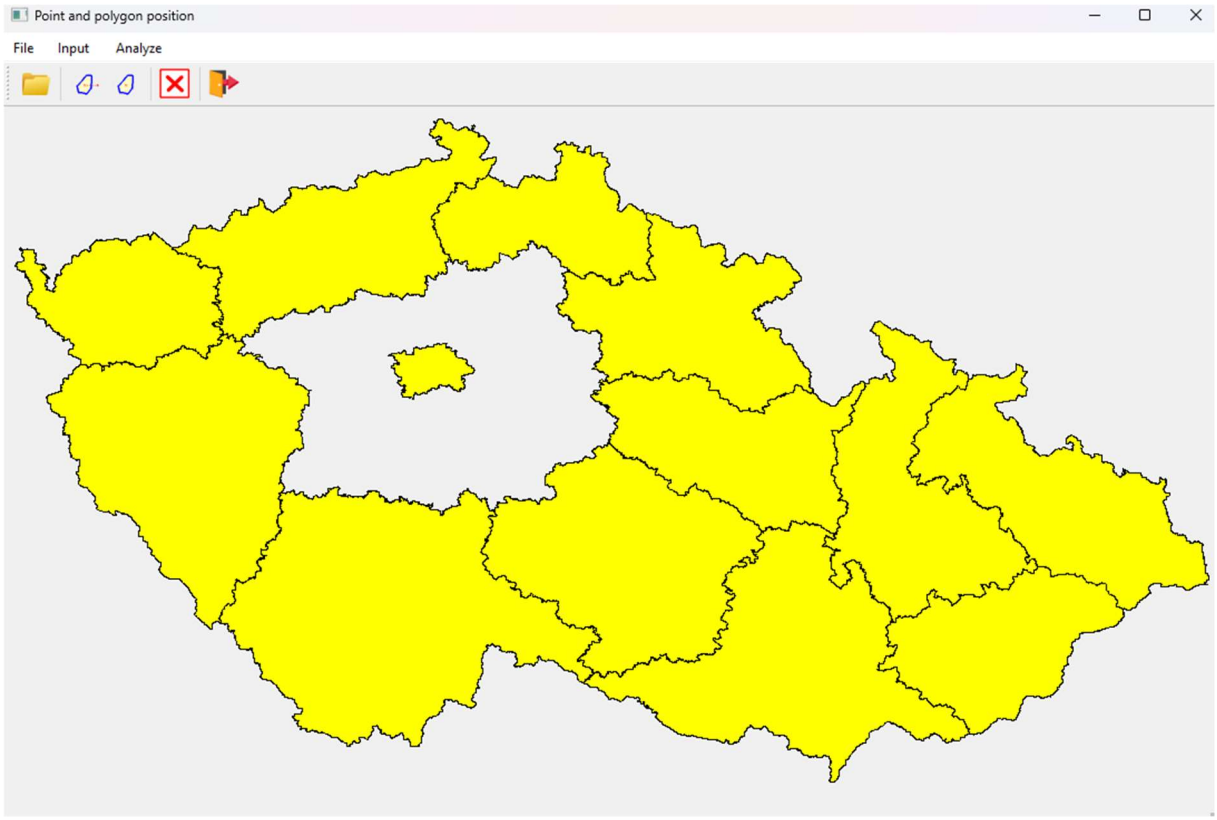
Grafické uživatelské rozhraní

Grafické rozhraní aplikace (Obrázek 1) bylo vytvořeno pomocí Qt Designeru a následně dále upravováno pomocí jazyka Python.



Obrázek 1 - uživatelské rozhraní aplikace

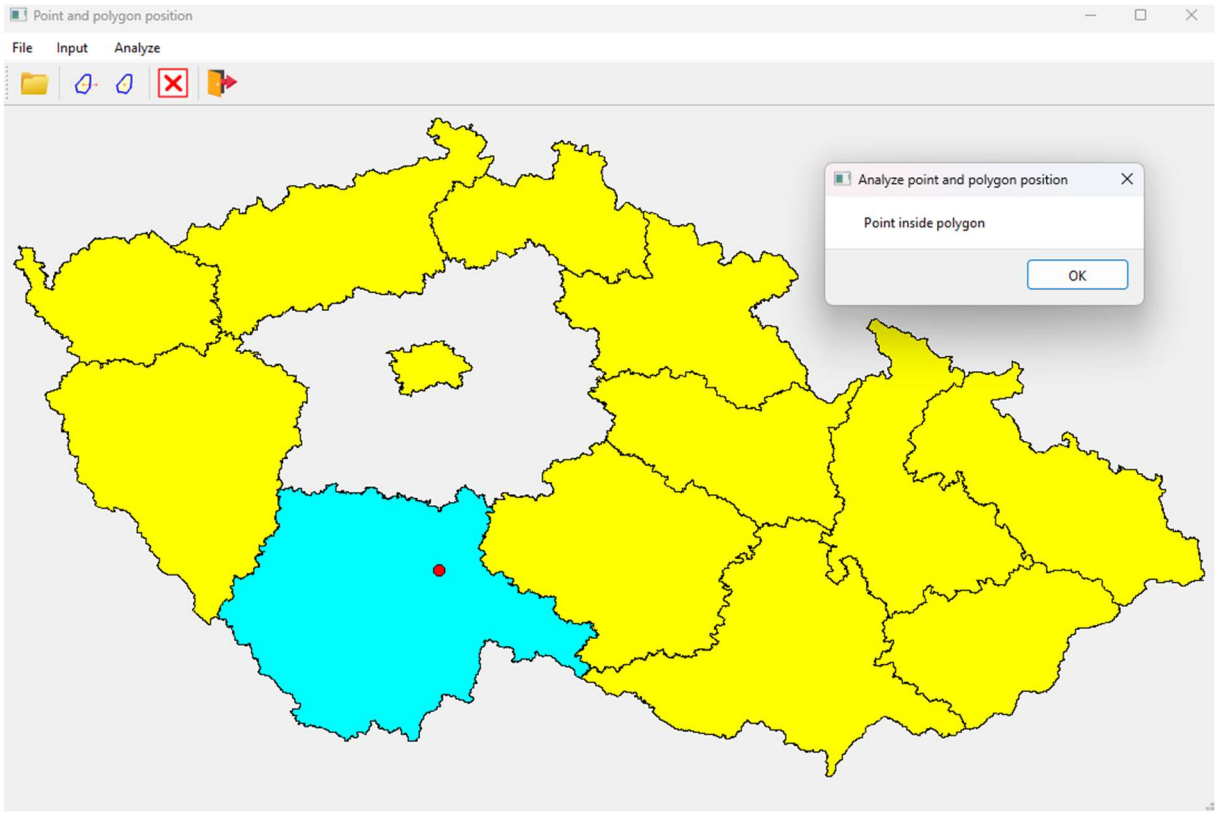
V horní části aplikace se nachází panel nástrojů o celkem pěti ikonách. První ikona po stisknutí otevře prohlížeč souborů v právě spuštěné složce a vyzve uživatele k nahrání polygonové vrstvy ve formátu Shapefile. Při nahrání modelových dat krajů ČR bez Středočeského kraje vypadá aplikace následovně (Obrázek 2):



Obrázek 2 - aplikace s nahranou vrstvou

Následným krokem pro správné použití aplikace je kliknutí do libovolného místa v okně aplikace. V tomto místě se zobrazí bod *q*, pro který bude následně zjišťováno, jestli se nachází v nějakém polygonu.

V panelu nástrojů se dále zleva vyskytují dvě ikony – Ray Crossing Algorithm a Winding Number Algorithm. Při najetí myší na tyto ikony se zobrazí nápověda, o který algoritmus se jedná. Při stisknutí jednoho z těchto tlačítek se provede odpovídající algoritmus. Ten zjistí, v jakém polygonu z nahrané vrstvy se předem kliknutý bod *q* nachází. Pokud se bod nachází v nějakém ze zadaných polygonů, daný polygon změní svoji barvu ze žluté na světle modrou. Zároveň se vždy zobrazí dialogové okno popisující vzniklou situaci (Obrázek 3).



Obrázek 3 - aplikace po zvolení bodu a algoritmu na analýzu vzájemné polohy

To řekne, jestli je bod v nějakém polygonu a zároveň zjistí, jestli se bod nenachází na hraně mezi dvěma polygony. V takovém případě změní bravu všechny polygony, kterých se to týká. Tlačítko s červeným křížkem umožňuje vymazat nahraná data pro případné nahrání jiné polygonové vrstvy. Toto lze udělat i rychleji – pokud uživatel nahraje novou vrstvu, stará se automaticky smaže a není tedy třeba klikat na tlačítko mazající vrstvu.
Poslední ikona aplikaci ukončí.

Třídy a metody

Pro spuštění aplikace jsou třeba tři soubory typu skriptu v jazyce Python (.py), ve kterých se nachází potřebné třídy a metody k běhu aplikace - *MainForm.py*, *algorithms.py* a *draw.py*.

Třída MainWindow

Třída MainWindow v souboru MainForm.py obsahuje příkazy potřebné k vytvoření samotného okna aplikace a k provádění uživatelských akcí. Hlavní část této třídy byla vygenerována v Qt Designeru a následně přeložena do jazyka Python (metody *SetupUi()* a *retranslateUi()*). Zbytek metod byl implementován ručně, zde je jejich přehled:

openClick() - Umožňuje načíst libovolný soubor s příponou .shp. Nejprve vyčistí případné dříve načtené soubory. Pokud je soubor nečitelný, je uživatel upozorněn vyskakovacím oknem.
openFile() - Samotné načtení souboru pomocí knihovny Geopandas. Soubor je uložen do proměnné *data*.

clearClick() - Zavolá metodu *clearData()* třídy *Draw*, která odstraní načtenou vrstvu.

rayCrossingClick() - Spustí analýzu polohy bodu pomocí algoritmu Ray Crossing pomocí metody *rayCrossingAlgorithm()* třídy *Algorithms*.

windingNumberClick() - Spustí analýzu polohy bodu pomocí algoritmu Winding Number pomocí metody *windingNumberAlgorithm()* třídy *Algorithms*.

Třída Algorithms

Tato třída obsahuje samotné algoritmy využívané aplikací pro vyhledání vzájemné polohy bodu a polygonu.

rayCrossingAlgorithm(q, pol) - pro bod *q* a polygon *pol* analyzuje, jestli se bod nachází uvnitř polygonu pomocí algoritmu Ray Crossing.

windingNumberAlgorithm(q, pol) - pro bod *q* a polygon *pol* analyzuje, jestli se bod nachází uvnitř polygonu pomocí algoritmu Winding Number.

computeAngle(ax, ay, bx, by) - pomocná metoda pro algoritmus Winding Number, která vypočítá úhel mezi zadanými vektory *a* a *b*

Třída Draw

Třída Draw se stará o všechny funkcionality, které se týkají získávání, upravování a zobrazování nahraných dat. Hned při spuštění se inicializuje několik potřebných proměnných:

self.q - vytvoří bod *q*, který je dále používán pro analýzu a nastaví jeho souřadnice na -100, -100, tedy mimo viditelnou oblast.

self.list_of_pols - vytvoří seznam, do kterého budou postupně přidávány všechny polygony z nahraných dat.

self.polyg_status - vytvoří seznam, do kterého bude při načtení každého polygonu ukládán jeho status (viz níže).

self.minmaxbox_list - vytvoří seznam, do kterého se budou přidávat min-max boxy nahraných polygonů

Tato třída obsahuje následující metody:

mousePressEvent() - zjistí polohu kurzoru myši při kliknutí, v daném místě vykreslí bod a zároveň nastaví status všech polygonů na 0.

paintEvent() - vykresluje všechny objekty (polygony a bod), obsahuje podmínku, která mění barvu polygonu, pokud se bod nachází uvnitř.

getQ() - vrací bod *q.*

getPol() - vrací seznam polygonů.

getMmb() - vrací seznam min-max boxů.

ClearData() - vyčistí seznamy polygonů i min-max boxů a nastaví polohu bodu mimo viditelnou plochu.

findBoundingPoints(p, xmin, ymin, xmax, ymax) - porovná souřadnice bodu *p* a momentální minimální a maximální souřadnice datasetu a změní je, pokud by bod byl mimo tyto extrémy.

resizePolygons(pol_list, xmin, ymin, xmax, ymax) - změní souřadnice polygonů ze seznamu tak, aby byly viditelné na momentální velikosti okna aplikace.

minMaxBox(pol) - vytvoří a vrátí min-max box polygonu *pol.*

loadData(data) - hlavní metoda, která postupně projde celý dataset, vytvoří seznamy polygonů i jejich min-max boxů a následně vykreslí všechny polygony.

Závěr

Pro splnění úkolu byla vytvořena aplikace prostřednictvím frameworku Qt, která umí řešit Point in Polygon Problem pro nekonvexní mnohoúhelníky. Problém je řešen dvěma algoritmy – Ray Crossing Algorithm a Winding Number Algorithm. Aplikace graficky zobrazuje výsledky algoritmů a lze do ní nahrát jakoukoli polygonovou vrstvu ve formátu Esri Shapefile (přípona .shp). Hlavní funkcionalita aplikace byla vytvořena v programovacím jazyce Python.

Jakožto hlavním úskalím aplikace se ukázaly Multi Polygony. S těmito typy polygonů si aplikace neumí efektivně poradit, problém byl vyřešen tím, že pokud je prvek Multi Polygon, nejprve se vytvoří konvexní obálka a analýza (včetně zobrazení) se dále provádí nad takto upravenými daty, což podává nepravdivé výsledky. Dalším vylepšením aplikace by bylo umožnit uživateli nahrát data i v jiném formátu než Shapefile (např. JSON).

Zdroje

BAYER, T. (2024): Point Location Problem. Přednáška pro předmět Algoritmy počítačové kartografie. Katedra aplikované geoinformatiky a kartografie. Přírodovědecká fakulta UK (cit. 22. 3. 2024).

ROURKE, O. J. (2005): Computational Geometry in C. Cambridge University Press, Cambridge.