ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE FAKULTA STAVEBNÍ, OBOR GEODÉZIE A KARTOGRAFIE KATEDRA GEOMATIKY					
název předmětu ALGORITMY V DIGITÁLNÍ KARTOGRAFII					
číslo úlohy 1	název úlohy Geometrické vyhledávání bodu				
školní rok 2020/2021	studijní skup. 60	číslo zadání U1	Zpracoval Usik Svetlana, Vaňková Zuzana	datum 21. 10. 2020	klasifikace

Zadání:

Úloha č. 1: Geometrické vyhledávání bodu

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

Výstup: P_i, $q \in P_i$.

Nad polygonovou mapou implementujete následující algoritmy pro geometrické vyhledávání:

- Ray Crossing Algorithm (varianta s posunem těžiště polygonu).
- Winding Number Algorithm.

Nalezený polygon obsahující zadaný bod q 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 na hranici polygonu.	10b
Ošetření singulárního případu u Winding Number Algorithm: bod leží na hraně polygonu.	*+2b*
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.	*+2b*
Algoritmus pro automatické generování nekonvexních polygonů.	*+5b*
Max celkem:	**21b**

Čas zpracování: 2 týdny.

Popis a rozbor problému

Často řešenou problematikou geoinformatiky je otázka: „Ve kterém polygonu leží bod?“. Nalezení polygonu, který obsahuje bod je běžným a často frekventovaným úkonem v GIS softwaru.

Předpokládejme, že v rovině jsou dány mnohoúhelníky (označme je *P1, P2, ..., Pn*). Každý polygon je definován příslušnou množinou bodů-vrcholů (*V1, V2, ..., Vn*) a spojnice všech vrcholů v množině tvoří množinu hran polygonu (*E1, E2, ..., En*). Rovněž předpokládejme, že v rovině existuje bod, nazvěme jej *q*. Cílem je poté určit polohu bodu q vzhledem k polygonům, tedy vyhledat polygony, které bod *q* obsahují. Pro bod *q* ve vztahu k určitému polygonu (*Pi*) mohou nastat tyto možnosti:

1. bod $q \notin Pi$ (bod leží vně polygonu)
2. bod $q \in Pi$, přičemž tuto variantu můžeme rozdělit ještě na:
    a. $q \in \{Pi \setminus (Ei \cup Vi)\}$ (bod q leží uvnitř polygonu)
    b. $q \in Ei$ (bod leží na hraně polygonu)
    c. $q \in Vi$ (bod leží v jednom z vrcholů polygonu).

Pro možnost 2.b. a c. může nastat, že jde o společný vrchol nebo hranu více polygonům. V tom případě bod *q* leží ve všech takových polygonech zároveň.

Řešení je hned několik, avšak některá jsou závislá na tvaru polygonu, který je buď konvexní, nebo nekonvexní. Konvexní polygon je takový mnohoúhelník, jehož všechny vnitřní úhly jsou menší nebo rovny 180°. Nekonvexní polygon je naopak takový mnohoúhelník, jehož alespoň jeden vnitřní úhel je větší než 180°. Protože se pohybujeme v rovině (2D prostor) je zřejmé, že nekonvexní polygon musí mít alespoň 4 vrcholy. Pro součet úhlů n-úhelníku, který má *n* vrcholů platí: $soucet\_uhlu = 180°(n-2)$.
Pro trojúhelník je tato hodnota 180° z čehož vyplývá, že žádný jeho vnitřní úhel nemůže být větší než 180°. Pro čtyřúhelník už je součet 360°, tedy hodnota vnitřního úhlu větší než 180° je již přípustná.

U konvexních polygonů je nejčastěji využito těchto dvou způsobů:

- Test polohy bodu vůči každé hraně mnohoúhelníku (opakovaný HalfPlane test),
- Paprskový algoritmus (Ray Crossing Algorithm).

V úloze jde ovšem o nekonvexní polygony, pro ni se používají tyto algoritmy:

- Paprskový (Ray Crossing Algorithm),
- Metoda ovíjení (Winding Number Algorithm).

Tyto algoritmy byly v rámci úlohy naprogramovány.

[3]

Údaje o bonusových úlohách

Byly ošetřeny singulární případy u Winding Number Algorithm (bod leží na hraně polygonu a bod leží ve vrcholu polygonu). Také byl ošetřen singulární případ u Ray Crossing Algorithm (bod leží ve vrcholu polygonu). Jak již bylo uvedeno výše, díky těmto případům může nastat situace, že bod leží na společné hraně dvou polygonů nebo ve společném vrcholu více polygonů. V těchto situacích bod obsahuje více polygonů. Bylo nastaveno, aby se zvýraznily všechny takové polygony.

Dále byl vytvořen algoritmus pro generování polygonů, k němu jsou uvedeny poznámky v závěru úlohy.

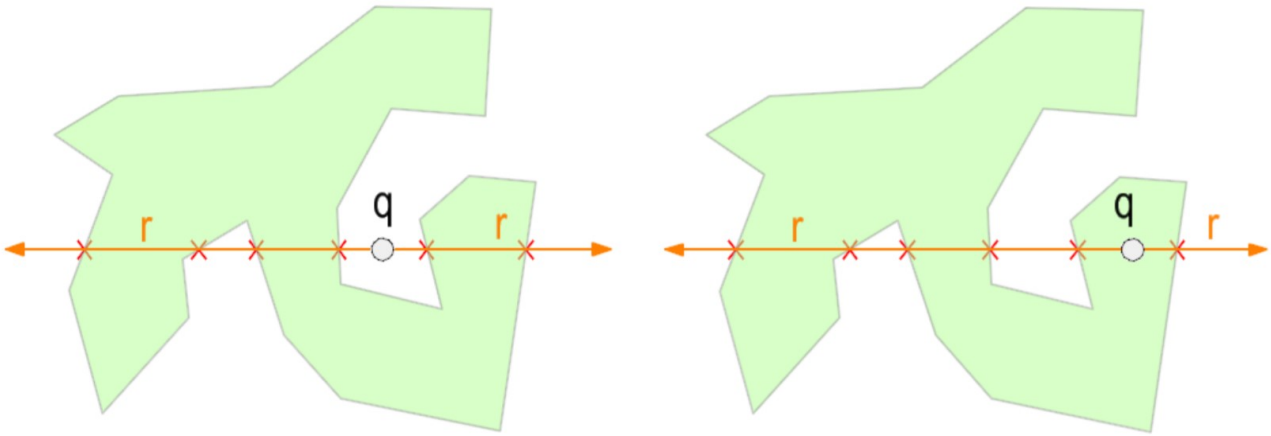
Popisy algoritmů

Ray Crossing Algorithm

Princip algoritmu spočívá ve vedení polopřímky *r* (paprsek) bodem *q*. Poté jednoduše spočítáme počet průsečíků s hranami polygonu. V případě konvexního polygonu, pro který je úloha primárně určená, dostaneme 1 průsečík, když bod leží v polygonu, a 2 průsečíky pokud leží vně. Úprava vyhodnocení pro nekonvexní polygony spočívá v posouzení sudého/lichého počtu průsečíků. Když je sudý resp. lichý, bod leží vně resp. uvnitř. Sudý a lichý počet odlišíme jako zbytek po dělení 2. Zbytek po dělení průsečíků 2 nabývá hodnot:

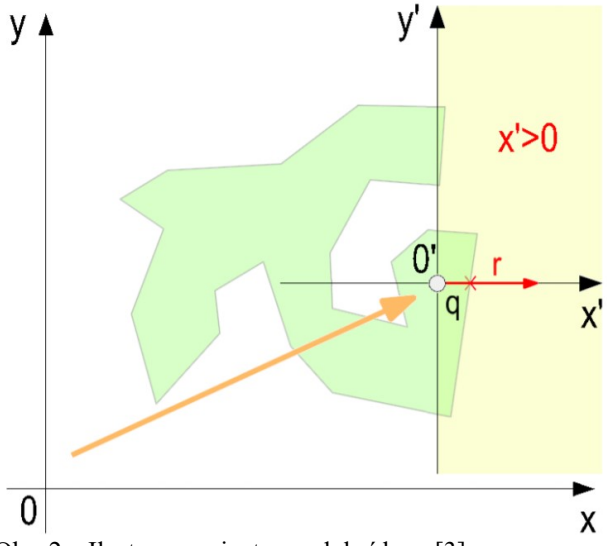
- $0 \ldots q \notin P$,
- $1 \ldots q \in P$.

Tento algoritmus o řád rychlejší než Winding Number Algorithm.



Obr. 1 Princip Paprskového algoritmu [3]

Problém představují singularity, které musíme vyřešit. Pokud totiž bude polopřímka *r* procházet vrcholem, můžeme dostat nesprávný počet průsečíků, taktéž případ kdy polopřímka prochází hranou polygonu je problematický. Proto upravíme algoritmus tak, že souřadnice bodů polygonu redukujeme do lokální kartézské souřadnicové soustavy s nulou v bodě *q*.



Obr. 2 Ilustrace varianty s redukcí ke q [3]

Slovně bychom algoritmus zapsali následujícím způsobem:

1. Inicializace počtu průsečíků: $k = 0$.
2. Opakuj pro všechny body polygonu pi ∈ P:
3. redukce souřadnic k bodu q: $x_i' = x_i - x_q$, $y_i' = y_i - y_q$.
4. Testujeme podmínku: *if*$(y_i' > 0)\ \&\&(y_{i-1}' <= 0)\ )||(y_{i-1}' > 0)\&\&(y_i' <= 0)$ *//Vhodný segment.*
5. Při splnění: $x_m' = (x_i' y_{i-1}' - x_{i-1}' y_i')/(y_i' - y_{i-1}')$ *//Vhodný průsečík.*
6. Pokud x_m' je větší než 0n zvýšíme poč. průsečíků: *if*$(x_m' > 0,)$ *pak* $k = k + 1$.
7. Testujeme sudý/lichý počet průsečíků: *if (k%2) != 0, pak* $q \in P$.
8. Else $q \notin P$.

[3]

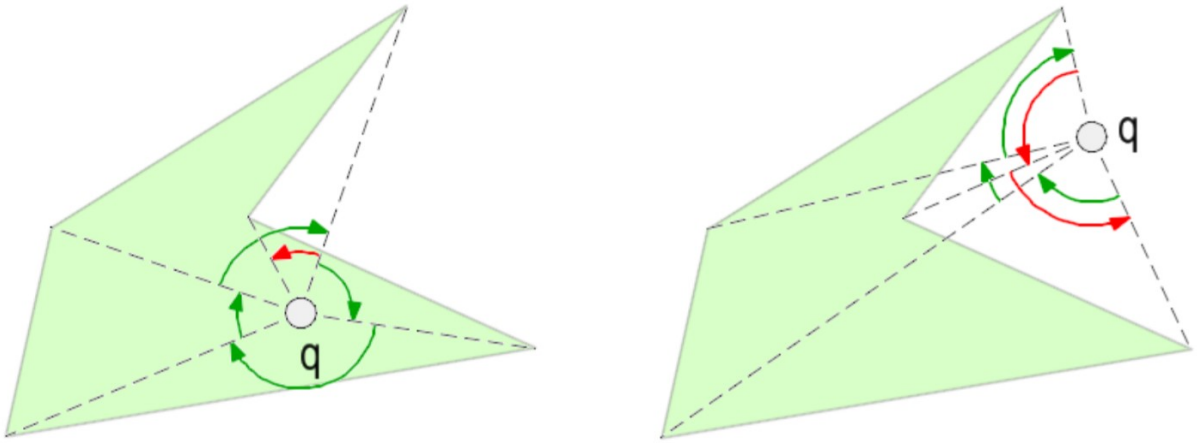
Winding Number Algorithm

Winding Number Algorithm nebo též Metoda ovíjení je algoritmus pracující s úhly. Představme si, že stojíme v bodě q. Postupně se otáčíme tak, abychom viděli v postupném pořadí ($v1, v2, ..., vn$) všechny vrcholy posuzovaného polygonu. Pokud stojíme uvnitř polygonu, nakonec se otočíme o celý kruh (2π). Pokud stojíme vně, tak se kolem své osy neotočíme.

Pro výpočet bereme vždy úhel daný bodem polygonu v_i, bodem q a bodem v_{i+1}. Je-li úhel po směru hodinových ručiček pak jej přičítáme, je-li proti směru, pak jej odečítáme. Výpočet provedeme pro všechny takové úhly v polygonu. Σ takových úhlů je buď 2π (bod leží uvnitř polygonu) nebo 0 (bod leží vně). Tento součet vydělíme 2π nazveme výsledek Ω. Mohou nastat 2 možnosti:

- $\Omega = 1$, pak $q \in P$,
- $\Omega = 0$, pak $q \notin P$.

Pro posouzení velikosti úhlu byla uvažována malá tolerance, protože z různých důvodů (ukládání čísel – periodický rozvoj, omezení počtu des. míst, …) nemusí být výsledné číslo přesně nulové.



Obr. 3 Princip Winding Number [3]

Zápis algoritmu by byl následující:

1. Inicializace: $\Omega = 0$, *tolerance* ε *//blízká 0, např. 10e-6.*
2. Opakuj pro všechny úhly (v_i, q, v_{i+1}):
3. Urči polohu q vzhledem k $v = (vi , vi+1)$.
4. Urči úhel $\omega i = \angle vi,q, vi+1$.
5. *If* $q \in$ *ol , pak* $\Omega = \Omega + \omega i$ *//Bod v levé polorovině,*
6. *else* $\Omega = \Omega - \omega i$. *//Bod v pravé polorovině.*
7. Testujeme : *if* $||\Omega| - 2\pi| < \varepsilon$, *pak* $q \in P$ *//Test na odchylku od* 2π.
8. Else $q \in/ P$.

[3]

Problematické situace a jejich rozbor

Ray Crossing Algorithm

U tohoto algoritmu nastávala problematická singularita hned v několika bodech. Singularity týkající se polopřímky *r* byly vyřešeny úpravou algoritmu (již bylo popsáno výše v kapitole „Popisy algoritmů“).

Winding Number Algorithm+ Ray Crossing Algorithm

Dále jsou problematické situace, kdy bod leží na hraně/vrcholu polygonu. To je společné pro oba algoritmy. Tyto singularity byly ošetřeny v jedné metodě, kterou využijí oba algoritmy. Jedná se o metodu *getPointLinePosition.*

Metoda v případě, že bod je na hraně/vrcholu polygonu, vrací hodnotu 2. V úseku kde je toto ošetřeno, je nejprve nastavena určitá tolerance (problém s nulovými hodnotami – nastavení čísla blízkého 0). Dále byla vytvořena nová metoda *distancePoints* třídy A*lgorithms*, která počítá vzdálenost mezi dvěma body.

Pro posouzení, zda bod leží ve vrcholu polygonu, je poté pomocí metody *distancePoints* vypočtena vzdálenost mezi bodem *q* a vrcholem polygonu a ta je porovnána s hodnotou tolerance. Pokud je menší než tolerance, metoda *getPointLinePosition* vrátí hodnotu 2.

Pro posouzení, zda bod leží na hraně, bylo vycházeno ze skutečnosti, že tvoří-li body *p1, p2, q* trojúhelník, bod neleží na hraně polygonu (a součet dvou libovolných hran je větší než strana třetí). V takovém případě by platilo $|p1\ p2| < |p1\ q| + |p2\ q|$. Pokud však bod na hraně leží, bude platit rovnost, protože bod *q* pouze dělí úsečku na dvě části. Po úpravě můžeme zapsat $||p1\ p2| - |p1\ q| + |p2\ q|| = 0$. Tedy opět porovnáme s výše zmíněnou tolerancí levou stranu rovnice. Pokud je absolutní rozdíl menší než tolerance, vracíme opět hodnotu 2 (mezi případy hrana/vrchol nebylo třeba rozlišovat). Pro výpočet jednotlivých vzdáleností byla opět použita metoda *distancePoints.*

Následně v obou algoritmech, v místě, kde se nachází for cyklus procházející všechny vrcholy, byla zařazena metoda *getPointLinePosition* a posouzení, zda vrácená hodnota je rovna 2. V případě, že ano, dojde k vrácení 1 (bod leží v polygonu).

Vstupní data

Bod

Bod je vytvořen na základě kliknutí uživatele myší do kreslící plochy.

Polygon

Načítání dat ze souboru – data jsou načítána z .TXT souboru. Po volbě možnosti *Import Polyons* se otevře prohlížeč souborů, kde je možné vybrat daný soubor z paměti počítače. Po vybrání souboru se objeví hlášení, zda byl soubor správně importován, nebo ne. Data v souboru jsou ve špagetovém modelu. To znamená, že body polygonu jsou zapsány postupně podle pořadí v tomto formátu: [CB X Y]. Nový polygon začíná opět od CB = 1. Tento model je velice jednoduchý. Jeho nevýhoda spočívá v duplicitní kresbě společných hran polygonů.

Vstupní cvičný soubor byl vytvořen jako polygony ve tvaru kostek původní verze hry TETRIS [1].

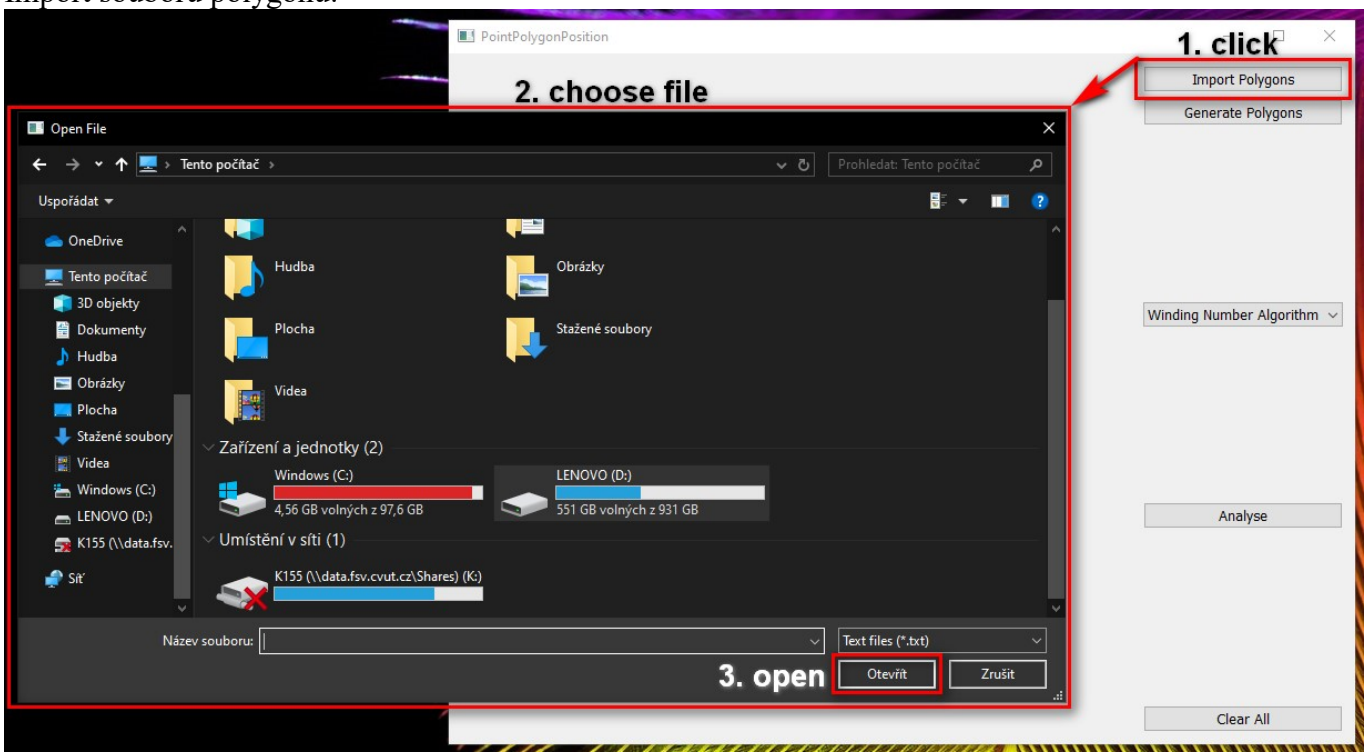
Generování polygonů – polygony jsou generovány po stisknutí tlačítka Generate Polygons. Komentář k samotnému generování je uveden v závěru.

Výstupní data

Výstupem celé úlohy je grafická aplikace, která umožňuje zobrazit vstupní data, dále provést zvolený výpočet a graficky zobrazit jeho výsledek. Formát výsledku je pouze grafický – polygony v nichž se nachází bod q se vyplní barvou.

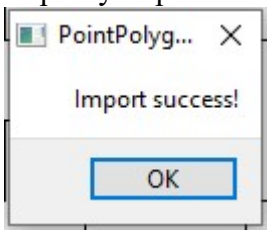
Printscreen vytvořené aplikace

Import souboru polygonů:



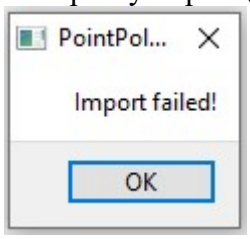
Obr. 4 Import

Úspěšný import:



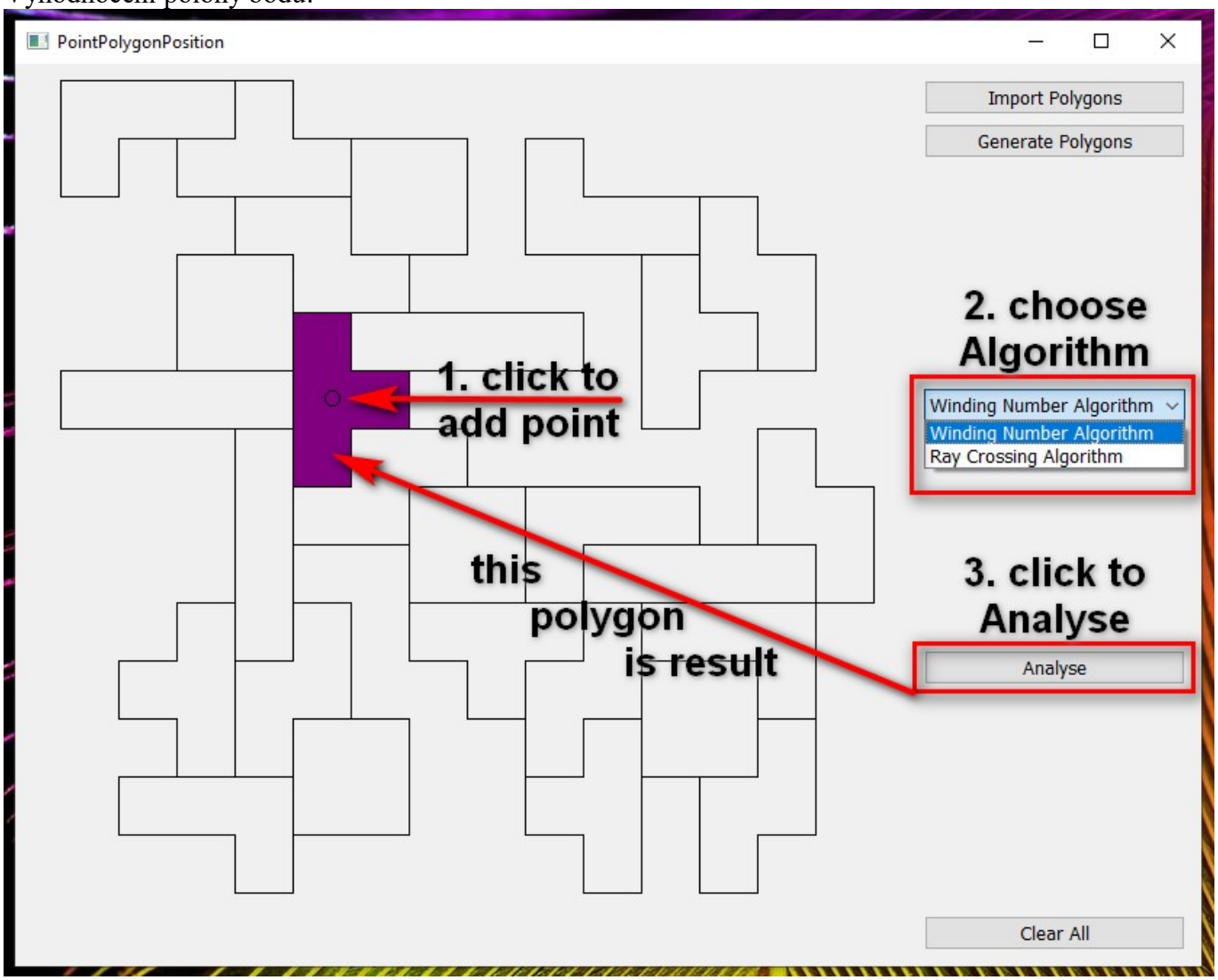
Obr. 5 Import success!

Neúspěšný import (například při špatném formátu dat v .TXT):



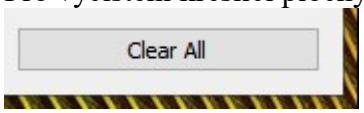
Obr. 6 Import failed!

Vyhodnocení polohy bodu:



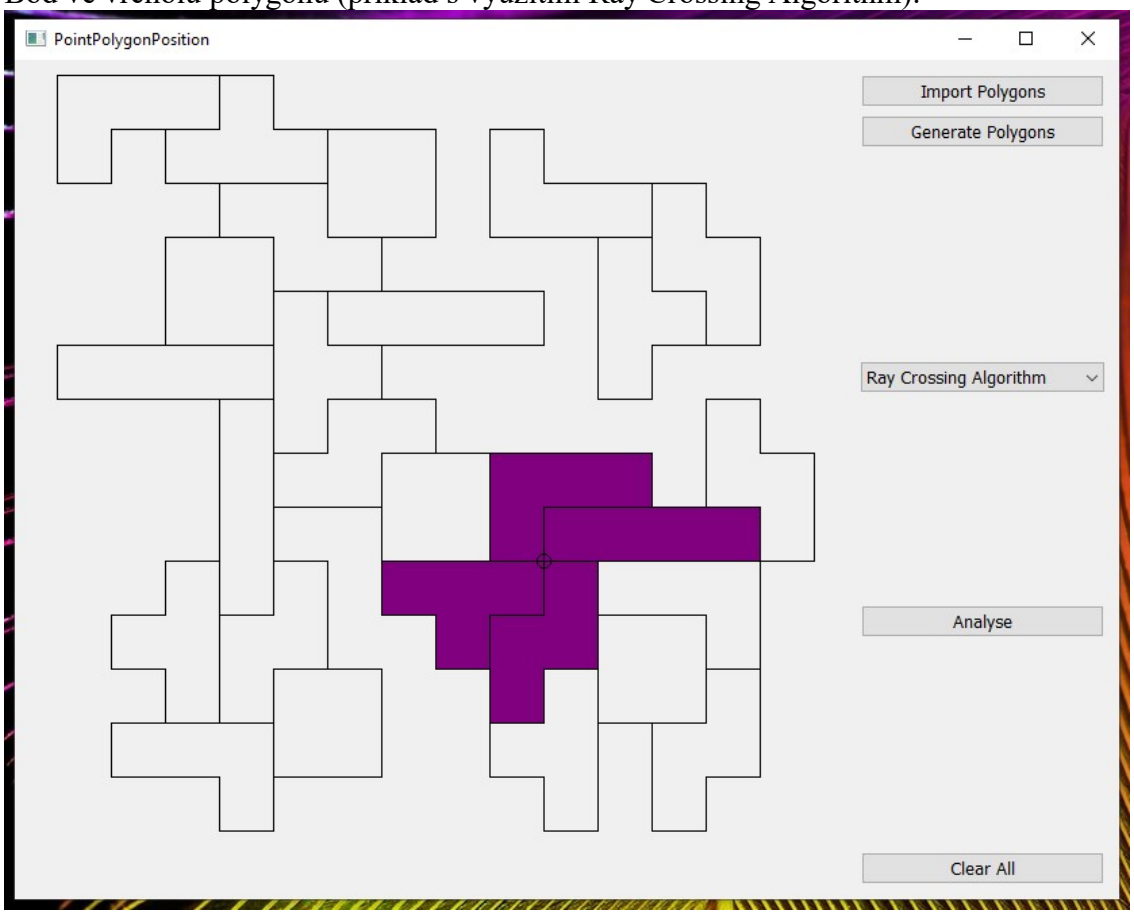
Obr. 7 Analyse

Pro vyčištění kreslící plochy je obsaženo tlačítko:



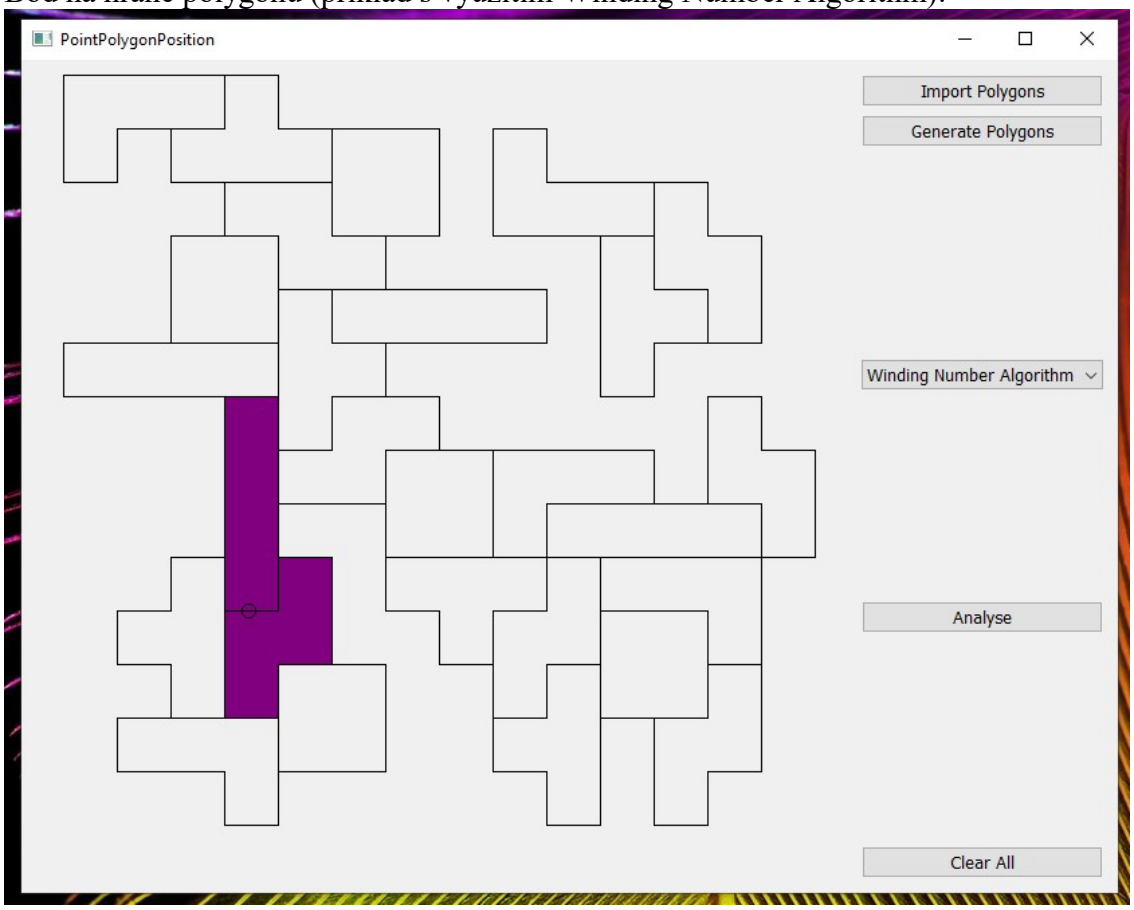
Obr. 8 Clear All

Bod ve vrcholu polygonu (příklad s využitím Ray Crossing Algorithm):



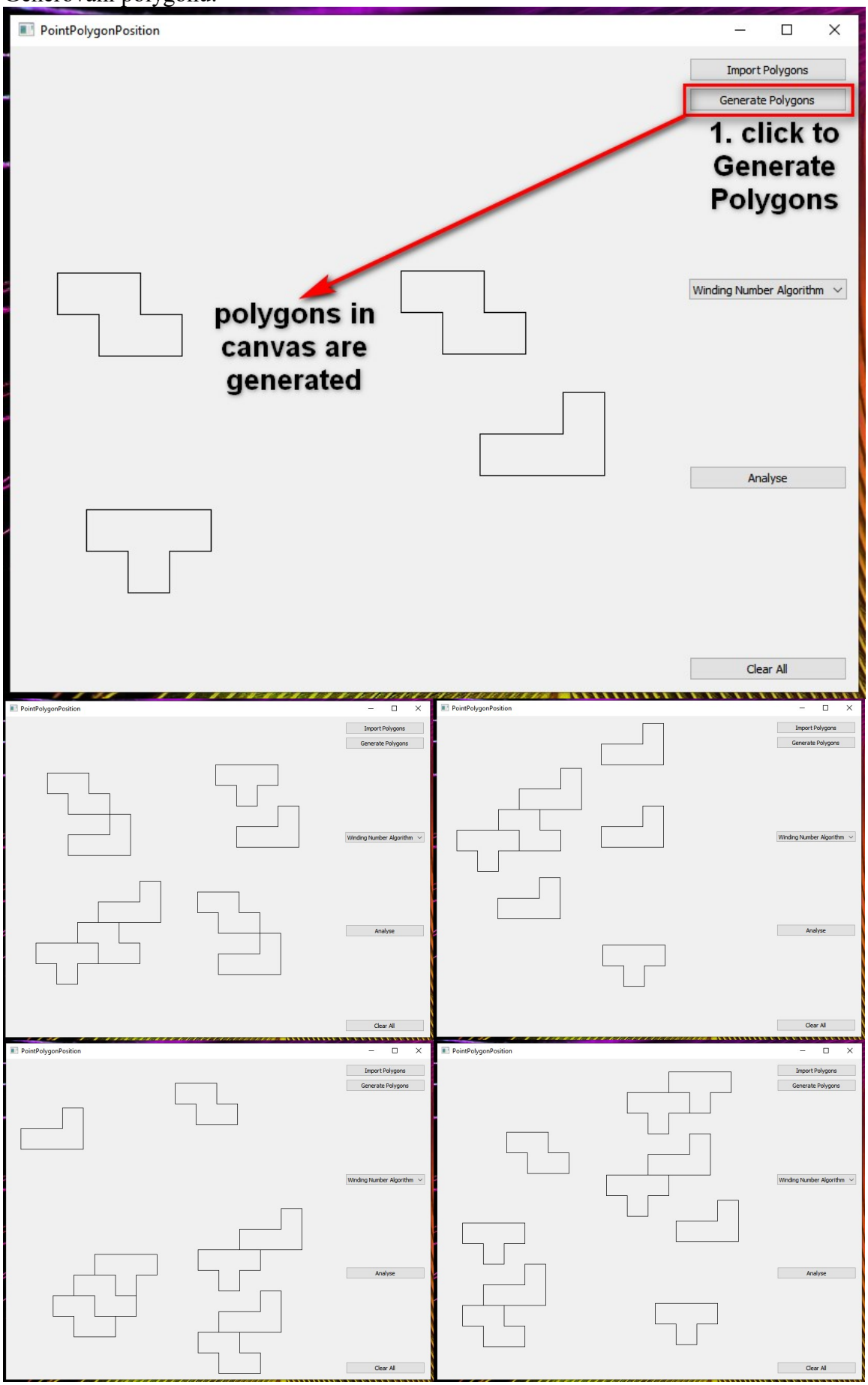
Obr. 9 Bod společný 4 polygonům

Bod na hraně polygonu (příklad s využitím Winding Number Algorithm):



Obr. 10 Bod společný 2 polygonům

Generování polygonů:



Obr. 11 Generate polygons – postup generování a příklady vygenerovaných polygonů

Dokumentace

Aplikace obsahuje 4 třídy, přičemž všechny kromě GenerateRandomPolygon mají svůj hlavičkový soubor a k němu příslušný zdrojový soubor. Níže jsou uvedeny jednotlivé třídy a metody, které obsahují.

Třída Algorithms

Třída Algorithms obsahuje konstruktor a metody používané pro řešení lokálních problémů konkrétní úlohy. Dále jsou popsané jednotlivé metody.

int getPointLinePosition(QPointF &q, QPointF &p1, QPointF &p2)

Tato funkce určuje polohu bodu *q* vůči přímce zadané dvěma body *p1* a *p2*. Přímka reprezentuje jednu hranu polygonu.

Ve funkci jsou ošetřené singulární případy, kdy bod leží na hraně a na vrcholu polygonu. Pomocná funkce *distancePoints* slouží k zjištění vzdálenosti mezi trojicí bodů *q*, *p1* a *p2*. Pokud absolutní hodnota rozdílu vzdálenosti mezi body *p1*, *p2* a součtu vzdáleností mezi body *p1*, *q* a *p2*, *q* je menší než stanovená hodnota tolerance, pak se bod nachází blízko hrany a je vrácena hodnota 2. Podobně se kontroluje totožnost bodu *q* a jednoho z bodů *p1* nebo *p2*. V případě, že jsou body totožné v rámci hodnoty tolerance, se vrací hodnota 2. Tolerance byla nastavena na hodnotu 0.4, aby se uživatel dokázal jednoduše dostat do singulárních situací.

Dále se z vektorů vypočte determinant. Jestliže je hodnota determinantu větší než 0, bod se nachází v levé polorovině vzhledem ke zkoumané přímce a funkce vrací hodnotu 1. Pokud je hodnota determinantu menší než 0, pak dojde k opačnému případu a bod *q* leží v pravé polorovině. Pokud nenastane žádný z výše uvedených případů, funkce vrátí hodnotu 1, což bude znamenat, že se někde stala chyba.

double getAngle(QPointF &q1, QPointF &q2, QPointF &q3, QPointF &q4)

Tato funkce vrací hodnotu úhlu, který svírají dvě hany polygonu. Každá hrana je reprezentovaná dvěma body. Pro výpočet bylo využito skalárního součinu.

int getPositionWinding(QPointF &q, QPolygonF &pol)

Tato metoda určuje polohu bodu vůči polygonu metodou Winding Number Algorithm. Pro účely rozhodování o poloze bodu je nastavena hodnota tolerance. Nejdříve je v cyklu vypočten úhel mezi sousedními hranami polygonu. K výpočtu je použita výše zmíněná funkce *getAngle*. Pokud součet všech uhlů rovná se hodnotě 2π v rámci zadané tolerance, pak funkce vrací hodnotu 1 a bod q je uvnitř polygonu. Jestli hodnota tolerance bude překročena, pak funkce vrátí 0 a znamená to, že bod leží mimo polygon. Singulární případy jsou ošetřeny ve funkci *getPointLinePosition*, kterou metoda *getPositionWinding* volá. Jestli funkce getPointLinePosition vrátí 2, pak bod *q* leží buď na hraně nebo na vrcholu polygonu a v *getPositionWinding* je vrácena hodnota 1 (toto je ošetřeno v kódu před jinými možnostmi vrácených hodnot).

int getPositionRay(QPointF &q, QPolygonF &pol)

Tato funkce určuje polohu bodu vůči polygonu metodou Ray Crossing Algorithm. Pro ošetření singulárních případů byla zavolána funkce *getPointLinePosition*. Jestli bod leží na hraně nebo je totožný s vrcholem polygonu, *getPointLinePosition* vrací hodnotu 2 a na základě toho je vrácena hodnota 1. Vrácená hodnota metody je pak zbytek po dělení 2. V případě, že počet protnutí polopřímky *r* s polygonem je liché číslo, pak se vrací hodnota 1 a bod leží uvnitř polygonu. Jestli čítač protnutí nabude sudé hodnoty, pak bod leží mimo polygon a vrátí se hodnota 0.

double distancePoints(QPointF &p, QPointF &q)

Toto je pomocná funkce, která počítá vzdálenost mezi dvěma body pomocí Pythagorovy věty.

Třída Draw

Třída Draw obsahuje konstruktor, který nastaví kurzor mimo kreslicí okno a níže popsané metody.

void mousePressEvent(QMouseEvent *e)

Tato metoda nastavuje pozici kurzoru přiřazením souřadnic *x* a *y*.

void paintEvent(QPaintEvent *e)

Tato funkce slouží k vykreslení bodu *q* a polygonu z textového souboru. Dále umí vybraný polygon zabarvit.

QPointF & getPoint (){return q;}

Metoda, která vrací privátní bod *q* třídy Draw.

QPolygonF & getPolygon (unsigned int index){return map_polygons[index];}

Metoda, která vrací i-tý polygon třídy Draw.

std::vector<QPolygonF> & getMapPolygon (){return map_polygons;}

Metoda, která vrací mapu polygonů třídy Draw.

void importPolygonsFromFile (QVector<QPolygonF> polygons)

Tato funkce slouží k importu polygonu z textového souboru. Do proměnných *n*, *x* a *y* jsou postupně ukládány hodnoty ze souboru. Začátek polygonu se pozná podle hodnoty $n = 1$.

bool importPolygons(std::string &fail_path)

Tato funkce načítá polygony, které jsou generované metodou *generateRandomPolygons.*

void saveResult(std::vector<int> res){results = res;}

Metoda, která do privátní proměnné *results* ukládá výsledky analýz.

void clearCanvas();

Tato funkce vyčistí kreslící plátno a proměnnou s výsledky rozhodovaní, zdali bod leží uvnitř nebo mimo polygon. Dále funkce nastaví kurzor mimo kreslicí okno. Dojde k překreslení.

Třída Widget

Pomocí této třídy komunikujeme s uživatelským rozhraním. Nejdříve se uživatelské rozhraní vyčistí.

void on_pushButton_2_clicked()

Tato metoda na základě vybraného algoritmu a zmačknutí tlačítka *Analyse* vizuálně zobrazí, zda bod je v polygonu (včetně hran a vrcholů) nebo mimo něj. Když jde o singulární případ, kdy bod leží ve společném vrcholu/hraně pro více polygonů zvýrazní se všechny takové polygony.

void on_pushButton_3_clicked()

Tato funkce umožňuje vyčistění okna aplikace. Spustí se po zmáčknutí tlačítka *Clear*.

void on_pushButton_4_clicked()

Funkce se zavolá po kliknutí na tlačítko *Import Polygons*. V ní je uveden způsob, kterým se načtou polygony z textového souboru.

void on_pushButton_clicked()

Stisknutím tlačítka *Generate Polygons* se zavolá metoda generující náhodně rozmístěné polygony.

Třída GenerateRandomPolygon

Třída obsahuje metody s jediným účelem – vygenerovat polygony. Jako jediná je vytvořena bez hlavičkového souboru.

QVector<QPolygonF> generateRandomPolygons()

Je to veřejná metoda, která vrací vektor vygenerovaných polygonů. Popis generování polygonů je popsán v komentáři v závěru. Pro generování konkrétních tipů polygonů metoda volá privátní metodu *generateRandomPolygon.*

QPolygonF generateRandomPolygon(int &m, int &n, int &o)

Jde o privátní metodu, která je volána metodou *generateRandomPolygons.* Do metody vstupují parametry *m* a *n*, které určují pozici daného typu polygonu a *o*, což značí typ objektu (polygonu). Metoda vrací vygenerovaný polygon, který je přidán k ostatním polygonům (ty jsou poté vráceny metodou *generateRandomPolygons*).

Závěr

Shrnutí

Byla vytvořena aplikace dle zadání, dávající požadované výstupy. Aplikace umožňuje uživateli nahrát polygony z textového souboru, ve kterém jsou uvedena data ve špagetovém modelu. Poté může uživatel graficky zvolit bod a zvolit algoritmus, který chce pro zjištění výsledku použít. Aplikace dává výsledek formou obarvení všech polygonů, ve kterých leží zvolený bod. Poté je možné vyčistit plochu a nahrát další soubor, či automaticky generovat polygony.

Komentář k automatickému generování polygonů

Automatické generování polygonů není pravděpodobně řešeno tak, jak bylo myšleno v zadání, ačkoli zadání bylo doslovně splněno (ale není cílem slovíčkařit). Rozhodně není zcela náhodné a generuje polygony předdefinovaných tvarů proto, aby byly nekonvexní a polygonů mohlo být více.

Původně byla uvažována varianta s algoritmem pro generování náhodného nekonvexního polygonu. Pro generování dalších polygonů by bylo náhodně rozhodnuto, zda bude mít s předchozími polygony společné body, náhodně vybrány takové polygony a následně i body společné s předchozími polygony. Pro to, aby se polygony nepřekrývaly, by mohla být využita jedna z metod pro polohu bodu. Ale takový postup byl shledán jako příliš časově náročný na tvorbu, proto od něj bylo upuštěno.

Byl vytvořen takový algoritmus, který dělí kreslící plochu (600x600) na 4 pole (kvadranty, 300x300). Na kvadranty je pole rozděleno proto, aby výsledek byl rozmanitější, nemá to jiný důvod. V těchto kvadrantech algoritmus generuje náhodný počet polygonů v rozsahu jeden až šest polygonů (pozn.: zde by bylo vhodné spočítat, jaký maximální počet polygonů lze při nejideálnější konfiguraci do pole zobrazit a podle toho nastavit maximální počet polygonů - číslo 6 bylo nastaveno na základě odhadu). V rámci každého kvadrantu je volen posun n ve směru osy x a posun m ve směru osy y. Dále byly vytvořeny tři různé tvary konvexních polygonů (opět převzaty tvary ze hry TETRIS [1]). Z nich se pro každý kvadrant a každý jeho polygon náhodně zvolí tvar. První polygon v kvadrantu je umístěn na souřadnice bodů tvaru, k jejichž složce x je přičten posun n v a taktéž ke složce y je přičten posun m.

Posuny jsou voleny tak, aby žádný z tvarů nepřekročil po přičtení k souřadnicím hranici kvadrantu.

Pro všechny další polygony kvadrantu je opět volen tvar a na základě tvaru vybrán pevně daný posun posunů m a n (to zní trochu překombinovaně, ale zkrátka dochází k tomu, že se tvar posune tak, aby sousedil s předchozím tvarem a aby se tvary nepřekrývaly). Nakonec je vždy vyhodnoceno, zda posuny i po změně leží ve stanovených mezích a pokud ano, vytvoří se polygon. Pokud ne, tak se polygon nevytvoří a není-li vyčerpán počet polygonů, opakuje se znovu. Je totiž možné, že se dalším změnou posuny m a n dostanou do správných mezí a další polygon bude vykreslen.

Pokus o slovní zjednodušený zápis algoritmu:

1. Pro všechny kvadranty (od i = 0; dokud i < 4; i++)
    a. Volba náhodného počtu polygonů od 1 do 6
    b. Když i je 0 (I. kvadrant)
        i. Zvol náhodně m a n
            (n ∈ [min(X) kvadrantu; max(X) kvadrantu - max(X) tvarů];
            m ∈ [min(Y) kvadrantu; max(Y) kvadrantu - max(Y) tvarů])
        ii. Pro všechny polygony (od j = 0; dokud j<počet polygonů; j++)
            1. Náhodná volba tipu polygonu (tip 1, 2, 3)
            2. Pokud je j == 0 vytvoř první polygon daného tipu s posunutými souř. o m, n
            3. Jinak
                a. Pokud je daný tip 1, posuň m, n o danou hodnotu
                b. Pokud je daný tip 2, posuň m, n o danou hodnotu
                c. Pokud je daný tip 3, posuň m, n o danou hodnotu
            4. Pokud leží nové m, n v mezích (viz. b.i.) vytvoř polygon
    c. Když i je 1 … opakujeme pro všechny kvadranty totéž
2. Vrať vytvořené polygony

Poznámka k toleranci – vybarvení více polygonů

V případě, že bod leží na hraně nebo ve vrcholu polygonu, vybarví se všechny takové polygony. Aby to bylo možno testovat, byla nastavena vysoká tolerance pro odchylku tak, aby bylo možno takový bod běžně myší vybrat. V praxi by byla odchylka nastavena menší.

Seznam literatury

[1] Tetris v číslech: 7 údajů o legendární hře. *RedBull* [online]. RED BULL, c2020, 06.06.2018 · 16:00 CEST [cit. 2020-10-23]. Dostupné z: https://www.redbull.com/cz-cs/hry-gaming-konzole-pc-mobilni-tetris-v-cislech

[2] Qt Documentation. *Qt* [online]. The Qt Company, c2020 [cit. 2020-10-23]. Dostupné z: https://doc.qt.io/

[3] BAYER, Tomáš. *Geometrické vyhledání: Ray algoritmus. Winding algoritmus. Lichobežníkové (trapezoidální) mapy.* [online]. In: . s. 1-19 [cit. 2020-10-23]. Dostupné z: https://web.natur.cuni.cz/~bayertom/images/courses/Adk/adk3.pdf

V Praze dne 23. 10. 2020 Usik S., Vaňková Z.