

České vysoké učení technické v Praze

Fakulta stavební



155ADKI Algoritmy digitální kartografie a GIS

Digitální model terénu a jeho analýzy

Bc. Adriana Brezničanová, Bc. Martin Kouba

19. 12. 2020

### Úloha č. 3: Digitální model terénu

*Vstup:* množina  $P = \{p_1, \dots, p_n\}$ ,  $p_i = \{x_i, y_i, z_i\}$ .

*Výstup:* polyedrický DMT nad množinou  $P$  představovaný vrstevnicemi doplněný vizualizací sklonu trojúhelníků a jejich expozicí.

Metodou inkrementální konstrukce vytvořte nad množinou  $P$  vstupních bodů 2D Delaunay triangulaci. Jako vstupní data použijte existující geodetická data (alespoň 300 bodů) popř. navrhněte algoritmus pro generování syntetických vstupních dat představujících významné terénní tvary (kupa, údolí, spočinek, hřbet, ...).

Vstupní množiny bodů včetně níže uvedených výstupů vhodně vizualizujte. Grafické rozhraní realizujte s využitím frameworku QT. Dynamické datové struktury implementujte s využitím STL.

Nad takto vzniklou triangulací vygenerujte polyedrický digitální model terénu. Dále proveďte tyto analýzy:

- S využitím lineární interpolace vygenerujte vrstevnice se *zadaným krokem* a v *zadaném intervalu*, proveďte jejich vizualizaci s rozlišením zvýrazněných vrstevnic.
- Analyzujte sklon digitálního modelu terénu, jednotlivé trojúhelníky vizualizujte v závislosti na jejich sklonu.
- Analyzujte expozici digitálního modelu terénu, jednotlivé trojúhelníky vizualizujte v závislosti na jejich expozici ke světové straně.

Zhodnot'te výsledný digitální model terénu z kartografického hlediska, zamyslete se nad slabinami algoritmu založeného na 2D Delaunay triangulaci. Ve kterých situacích (různé terénní tvary) nebude dávat vhodné výsledky? Tyto situace graficky znázorněte.

Zhodnocení činnosti algoritmu včetně ukázek proveďte alespoň na **3 strany** formátu A4.

#### Hodnocení:

Krok	Hodnocení
Delaunay triangulace, polyedrický model terénu.	10b
Konstrukce vrstevnic, analýza sklonu a expozice.	10b
Triangulace nekonvexní oblasti zadané polygonem.	+5b
Výběr barevných stupnic při vizualizaci sklonu a expozice.	+3b
Automatický popis vrstevnic.	+3b
Automatický popis vrstevnic respektující kartografické zásady (orientace, vhodné rozložení).	+10b
Algoritmus pro automatické generování terénních tvarů (kupa, údolí, spočinek, hřbet, ...).	+10b
3D vizualizace terénu s využitím promítání.	+10b
Barevná hypsometrie.	+5b
<b>Max celkem:</b>	<b>65b</b>

Čas zpracování: 4 týdny

## Obsah

1. Popis a rozbor problému .....	4
1.1 Údaje o bonusových úlohách .....	4
2. Popis algoritmů.....	4
2.1 Delauneyova triangulace.....	4
2.1.1 Vlastnosti Delauneyovy triangulace .....	4
2.1.2 Implementace metody.....	4
2.2 Interpolace vrstevnic .....	5
2.3 Sklon terénu.....	5
2.4 Expozice terénu .....	6
3. Bonusové úlohy .....	7
3.1 Automatický popis vrstevnic .....	7
3.2 Algoritmus pro automatické generování terénních tvarů.....	7
3.3 Barevná hypsometrie .....	7
4. Vstupní data .....	8
5. Výstupní data.....	8
6. Printscreeny z vytvořené aplikace.....	9
7. Dokumentace .....	15
7.1 Třída algorithms.h .....	15
7.2 Třída draw.h .....	15
7.3 Třída edge.h.....	16
7.4 Třída qpoint3d.h .....	16
7.5 Třída sortbyx.h.....	16
7.6 Třída triangle.h .....	16
7.7 Třída widget.h.....	17
7.8 algorithms.cpp.....	17
7.9 draw.cpp.....	25
7.10 widget.cpp .....	29
8. Zhodnocení dosažených výsledků a závěr.....	33
8.1 Možné vylepšení.....	33
9. Seznam obrázků .....	34

## 1. Popis a rozbor problému

Problematika úlohy se týká tvorby digitálního modelu terénu prostředím QT Creator. Nad množinou bodů  $P = \{p_1, \dots, p_n\}$ , kde  $p_i = \{x_i, y_i, z_i\}$  byla vytvořena síť pomocí Delaney triangulace. Po vytvoření této sítě jsou vygenerovány vrstevnice a je možné vizualizovat terén pomocí sklonu a orientace terénu.

### 1.1 Údaje o bonusových úlohách

Kromě výše uvedených funkcí byly zpracovány tyto bonusové úlohy:

- Automatický popis vrstevnic
- Algoritmus pro automatické generování terénních tvarů
- Barevná hypsometrie

## 2. Popis algoritmů

### 2.1 Delauneyova triangulace

Delauneyova triangulace  $n$  bodů v rovině je taková triangulace, kde kružnice opsaná každému trojúhelníku neobsahuje vevnitř žádné body z množiny. Platí, že libovolný trojúhelník je součástí Delauneyovy triangulace množiny bodů právě tehdy, když jeho opsaná kružnice neobsahuje uvnitř žádné body. Je to nejčastěji používaná triangulace jak pro 2D, tak 3D množiny.

#### 2.1.1 Vlastnosti Delauneyovy triangulace

- Uvnitř kružnice k opsané libovolnému trojúhelníku  $t_i \in \mathcal{T}$  neleží žádný jiný bod množiny  $P$ .
- $\mathcal{T}$  maximalizuje minimální úhel v  $\forall t$ , avšak  $\mathcal{T}$  neminimalizuje maximální úhel v  $t$ .
- $\mathcal{T}$  je lokálně i globálně optimální vůči kritériu minimálního úhlu.
- $\mathcal{T}$  je jednoznačná, pokud žádné čtyři body neleží na kružnici.
- Výsledné trojúhelníky se nejvíce blíží rovnostranným trojúhelníkům.

#### 2.1.2 Implementace metody

1. Seřazení vstupní množiny bodů podle souřadnice  $X$ .
2. Nalezení pívotu  $q$  s minimální souřadnicí  $X$ ,  $q = \min_{\forall p_i \in P} (x_i)$ .
3. Nalezení nejbližšího bodu  $p_{nearest}$  k pívotu  $\|q - p_1\| = \min$ .
4. Vytvoření hrany  $e = (q, p_{nearest})$ .
5. Nalezení Delauneyovho bodu  $-p_{min} = \argmin_{\forall p_i \in \sigma_L(e)} r(k_i)$ , kde  $k_i = (a, b, p_i)$  a  $e = (a, b)$ .
6. Pokud takový bod není nalezen, prohození orientace hrany  $e$  a opakování kroku 5.
7. Vytvoření zbývajících hran trojúhelníku  $e_2 = (p_2, p_{min})$ ,  $e_3 = (p_{min}, p_1)$ .
8. Přidání hran do  $\mathcal{T}$  -  $\mathcal{T} \leftarrow e$ ,  $\mathcal{T} \leftarrow e_2$ ,  $\mathcal{T} \leftarrow e_3$ .
9. Přidání hran do listu hran  $AEL$  -  $AEL \leftarrow e$ ,  $AEL \leftarrow e_2$ ,  $AEL \leftarrow e_3$ .
10. Pokud je  $AEL$  není prázdné:
11. Vezmi první hranu z  $AEL$  -  $AEL \rightarrow e$ ,  $e = (p_1 p_2)$
12. Prohoď její orientaci  $e = (p_2 p_1)$ .
13. Nalezení Delauneyovho bodu -  $p_{min} = \argmin_{\forall p_i \in \sigma_L(e)} r(k_i)$ , kde  $k_i = (a, b, p_i)$  a  $e = (a, b)$ .
14. Jestli existuje takový bod - *if*  $\exists p_{min}$
15. Vytvoř zbývajících hran trojúhelníku  $e_2 = (p_2, p_{min})$ ,  $e_3 = (p_{min}, p_1)$ .
16. Přidej hranu do  $\mathcal{T}$ , ale ne do  $AEL$  -  $\mathcal{T} \leftarrow e$
17. Update  $AEL$ .

### 2.1.2 Problematické situace

Problém může nastat při bodech na linii – kolineárních bodech. Problémem je i časová náročnost našeho naprogramovaného algoritmu při vložení většího počtu bodů. Také je problematické zpracování okrajů plochy – nevíme okolí bodu.

### 2.2 Interpolace vrstevnic

Pro naši aplikaci jsme využili lineární interpolace – spád mezi dvěma body, mezi kterými provádíme interpolaci, je konstantní. Rozestup vrstevnic mezi dvěma body je také konstantní. Tento způsob je výpočetně jednoduchý, ale nevystihuje realitu.

Úkolem je najít průsečnici roviny určené trojúhelníkem Delauneyovy triangulace a vodorovné roviny s výškou  $h$ . Průsečnice je dána body  $A$  a  $B$ :

$$x_A = \frac{x_3 - x_1}{z_3 - z_1}(z - z_1) + x_1$$

$$y_A = \frac{y_3 - y_1}{z_3 - z_1}(z - z_1) + y_1$$

$$x_B = \frac{x_2 - x_1}{z_2 - z_1}(z - z_1) + x_1$$

$$y_B = \frac{y_2 - y_1}{z_2 - z_1}(z - z_1) + y_1$$

Spojením těchto bodů vznikne hrana tvořící vrstevnici v daném trojúhelníku.

### 2.3 Sklon terénu

Sklon je definován jako úhel mezi normálovým vektorem vodorovné roviny a normálovým vektorem trojúhelníka.

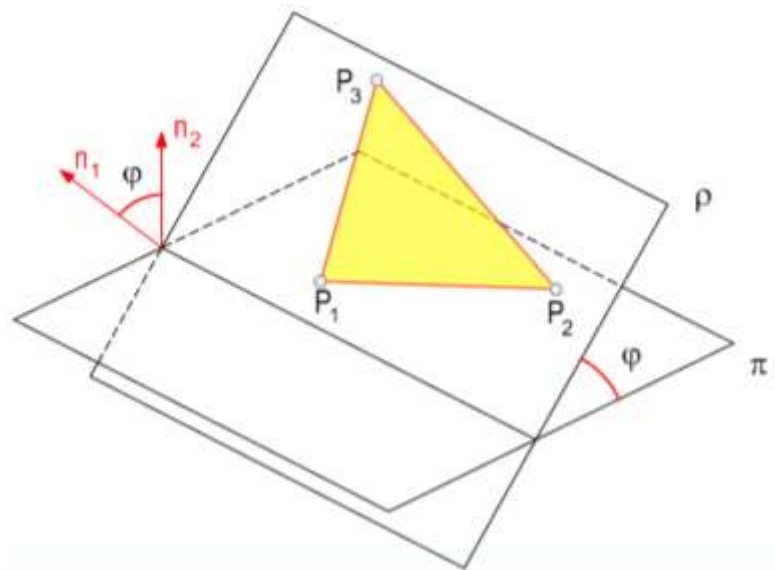
$$n_x = u_y \cdot v_z - u_z \cdot v_y$$

$$n_y = -(u_x \cdot v_z - u_z \cdot v_x)$$

$$n_z = u_x \cdot v_y - u_y \cdot v_x$$

$$n_t = \sqrt{n_x^2 + n_y^2 + n_z^2}$$

$$\varphi = \arccos\left(\frac{n_z}{n_t}\right)$$



Obrázek 1 Sklon terénu

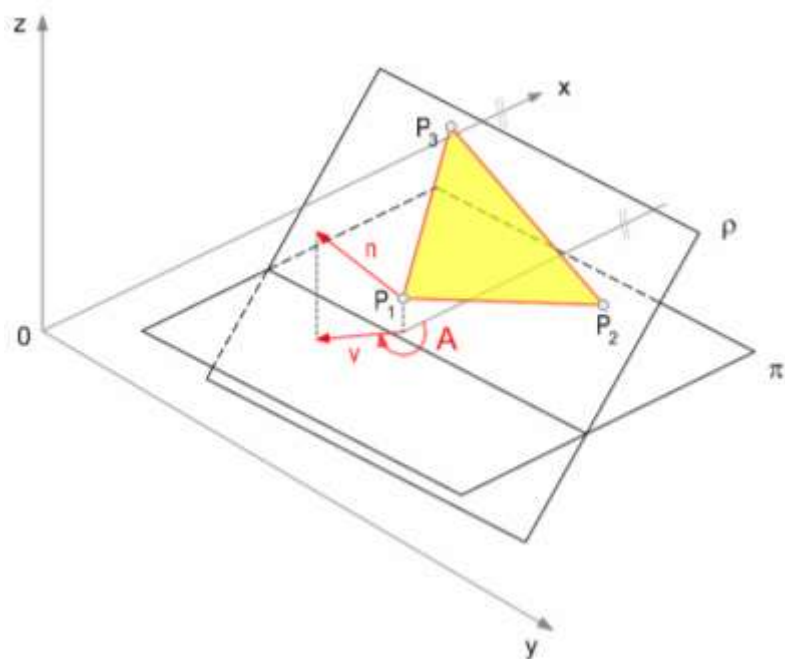
## 2.4 Expozice terénu

Expozice se rozumí natočení terénu (trojúhelníků) k světovým stranám.

$$n_x = u_y \cdot v_z - u_z \cdot v_y$$

$$n_y = -(u_x \cdot v_z - u_z \cdot v_x)$$

$$\alpha = \text{atan}\left(\frac{n_x}{n_y}\right)$$



Obrázek 2 Orientace terénu

### 3. Bonusové úlohy

#### 3.1 Automatický popis vrstevnic








Nejdříve pro popis vrstevnic byl přepracován stávající setter pro získání výšek a jejich rozdílů. Poté pro každou vrstevnici bylo rozhodnuto, zda je hlavní či ne a u hlavních vrstevnic byl vložen text do půlky vrstevnice podle jejich vzdálenosti. Také byla zvětšena šířka vykreslené vrstevnice.

#### 3.2 Algoritmus pro automatické generování terénních tvarů

Pro generování terénních tvarů byla vybrána kupa, údolí, spočinek a hřbet. Nejdříve byl přidán ComboBox pro vybírání terénních tvarů a tlačítko pro vyvolání metody. Pro kupu a spočinek bylo zvoleno náhodné generování bodů v elipsách s náhodnou výškou. U kupy byl vytvořen centrální bod a kolem něho byly vytvořeny další body vždy s menší výškou. Pro spočinek byla vytvořena tři „patra“ bodů s rozdílnou výškou. Pro údolí a hřbet byly vytvořeny dvě paralelních linky bodů o stejné výšce a další body byly vždy přidány s daným rozestupem. Pro hřbet byly body přidávány s menší výškou a pro údolí s větší výškou.

#### 3.3 Barevná hypsometrie

Barevnou hypsometrii si může uživatel vybrat z ComboBoxu společně se *Slope* a *Aspect*. Nejprve se getterem získají všechny tři výšky pro každý bod v trojúhelníku. Tyto výšky se vloží do nového vektoru, seřadí a vypočítá se jejich vážený průměr. Podle váženého průměru se poté určí, která barva se do trojúhelníku vykreslí (viz Legenda). Dále se tyto trojúhelníky už jenom vykreslí.

Výška [m]	Barva
<=100	
<=200	
<=300	
<=400	
<=500	
<=600	
>600	

Tab. 1 Legenda

## 4. Vstupní data

Vstupní data jsou generována podle výběru uživatelem – kupa, údolí, spočinek a hřbet výběrem z ComboBoxu nebo je možné importovat vlastní textový formát. Body je také možné ručně vložit kliknutím myši. Po kliknutí na tlačítko *Generate terrain/Load points* se body vygenerují a zobrazí v Canvasu.

X	Y	Z
572191.82	1086095.45	335.77
572194.31	1086093.67	335.98
572199.90	1086090.80	336.84
572206.35	1086086.06	336.73
572215.52	1086079.62	337.05
571914.92	1086215.79	313.86
571894.17	1086378.18	309.67
571893.63	1086378.38	310.05
571892.13	1086379.28	309.98
571899.17	1086375.68	309.64
571899.21	1086375.74	309.76
571899.39	1086377.62	309.82
571893.01	1086376.53	309.55
571893.11	1086376.66	309.66
571891.30	1086376.81	309.51

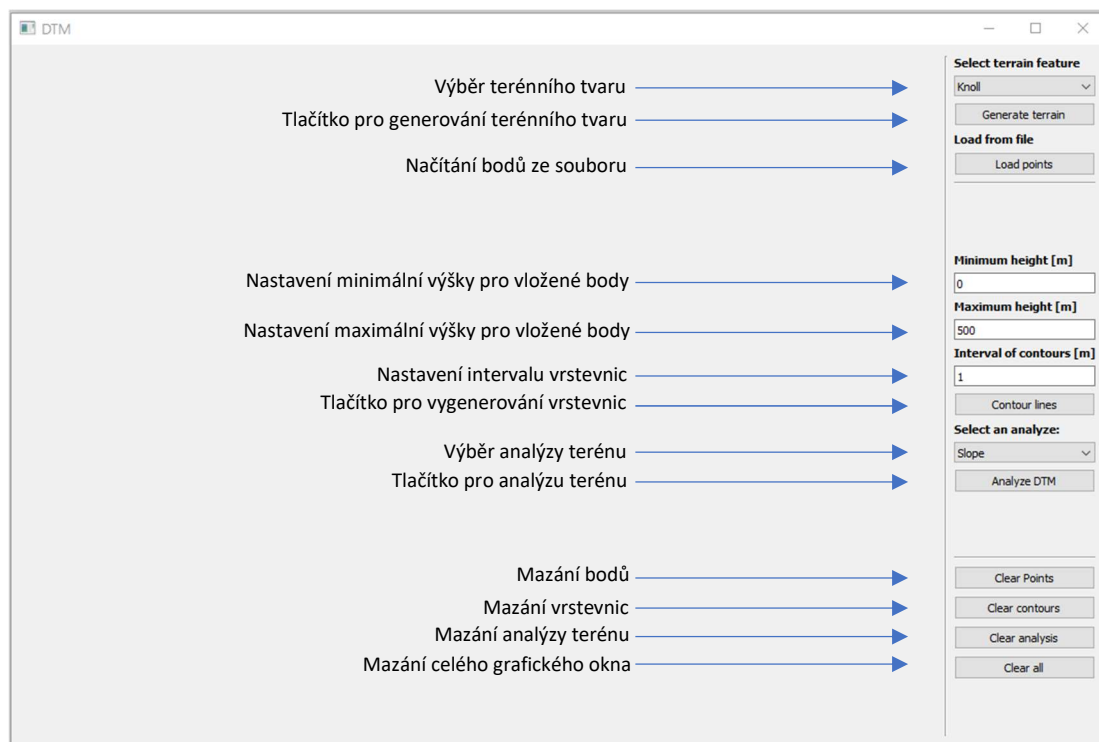
Obrázek 3 Formát vstupních dat

## 5. Výstupní data

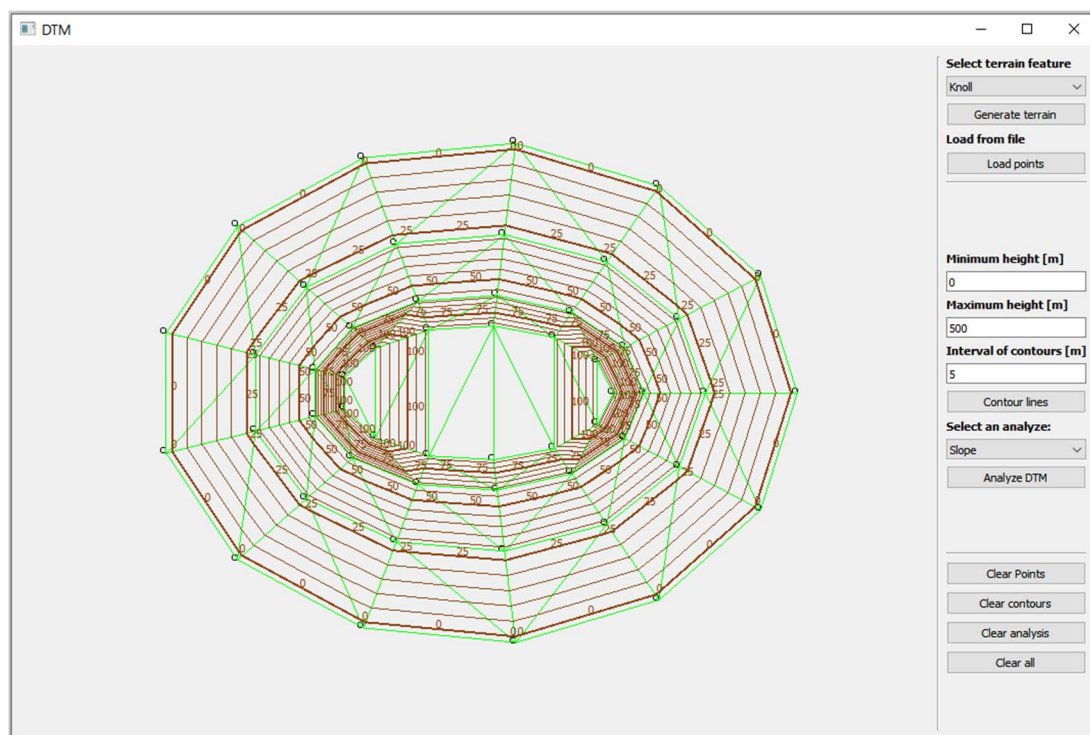
Výstupem rozumíme vykreslení Delauneyových trojúhelníků s vrstevnicemi po kliknutí na tlačítko *Contour lines* zobrazení sklonu/expozice/barevné hypsometrie po kliknutí na tlačítko *Analyze DTM*. Tento výstup si může uživatel vymazat *Clear Points/Clear contours/Clear analysis/Clear all* a uskutečnit další výpočet.



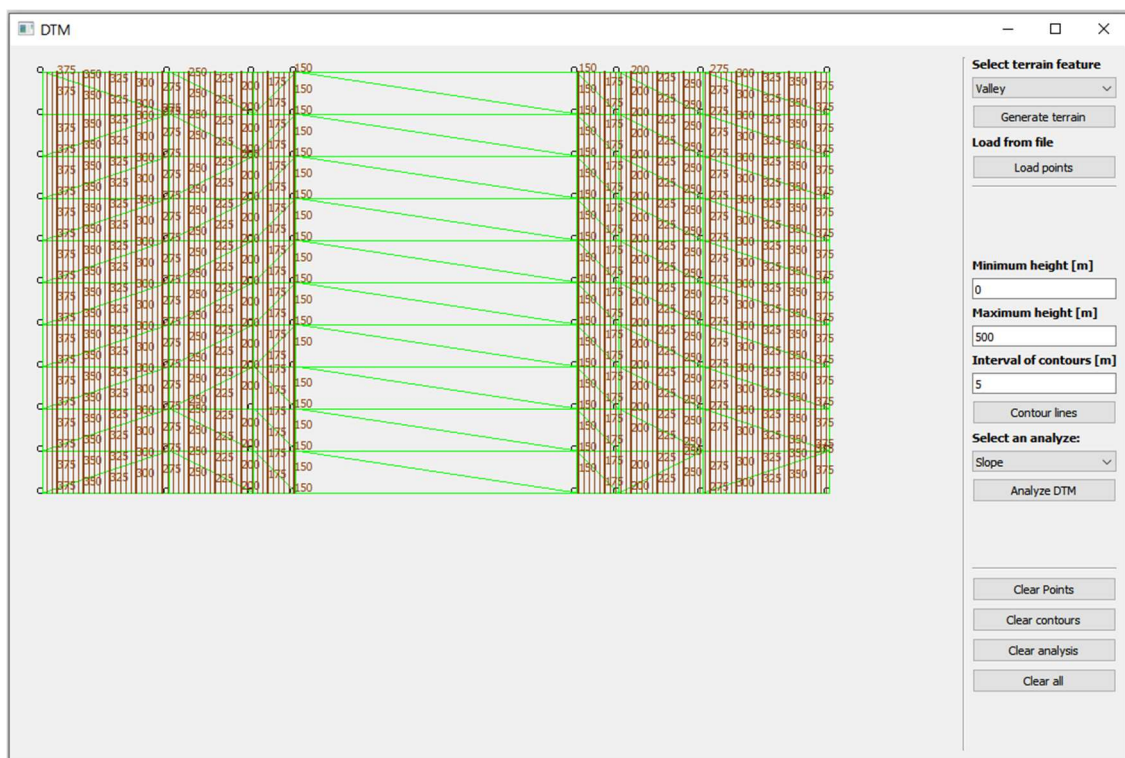
## 6. Printscreensy z vytvořené aplikace



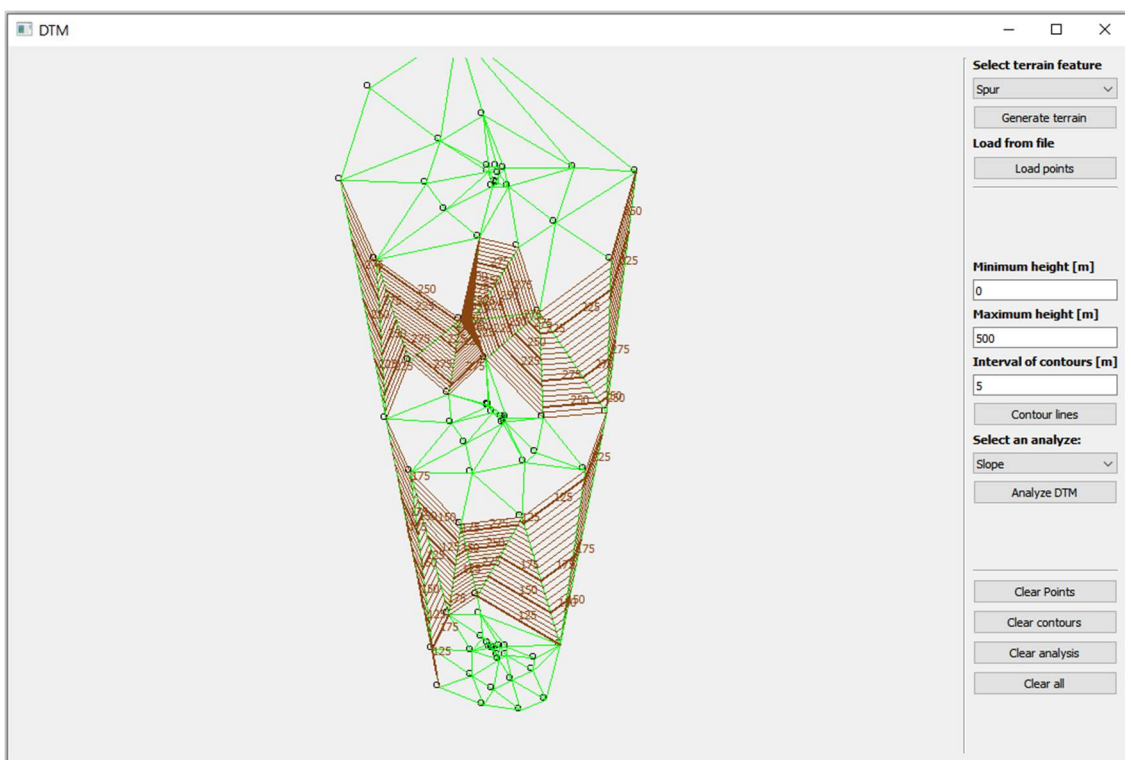
Obrázek 4 Grafické okno aplikace



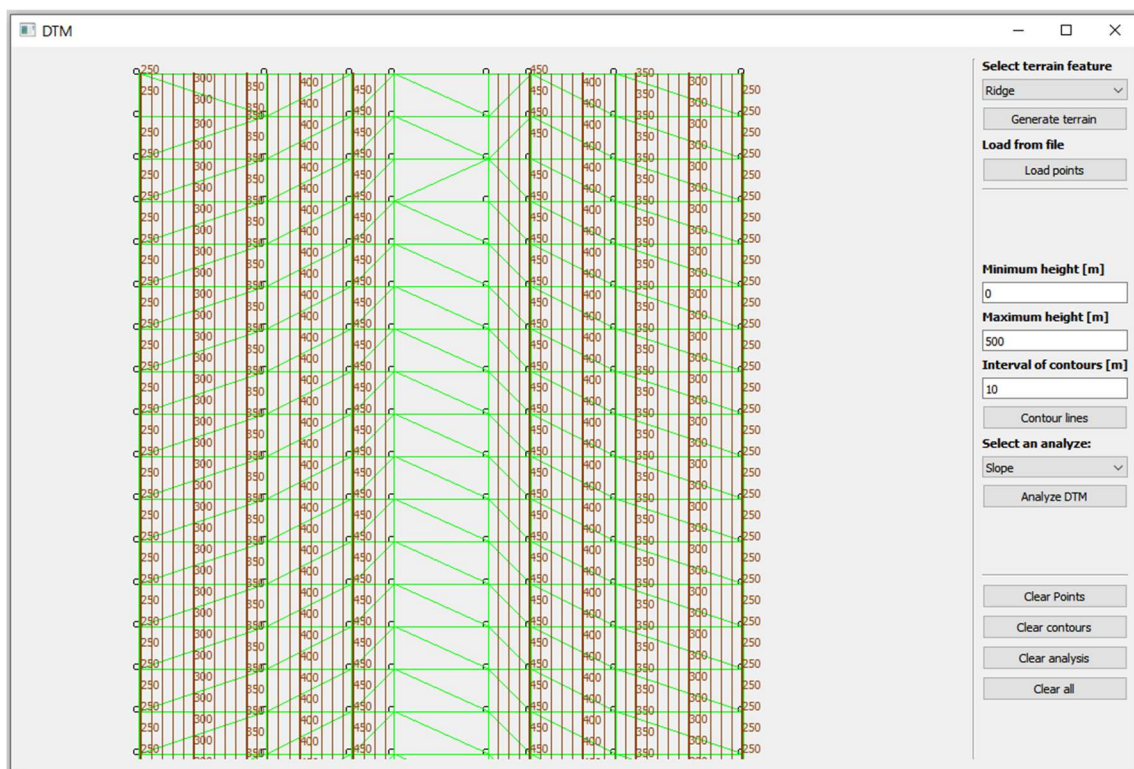
Obrázek 5 Generování kupy



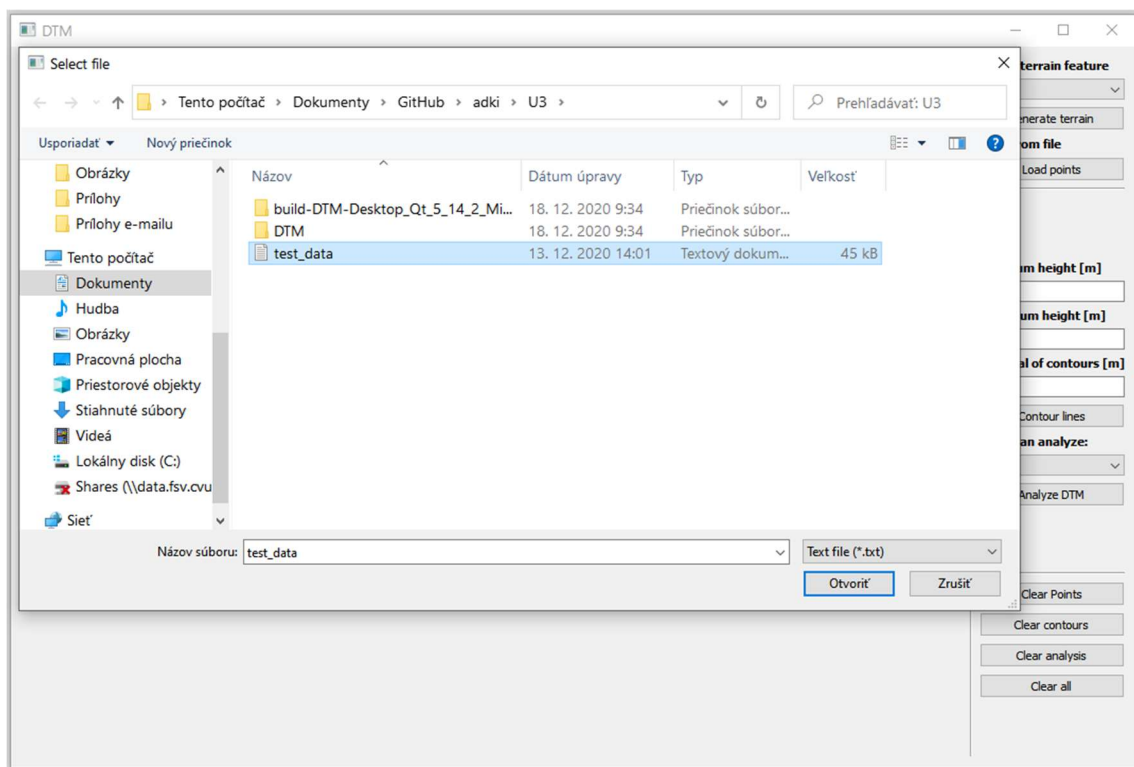
Obrázek 6 Generování údolí



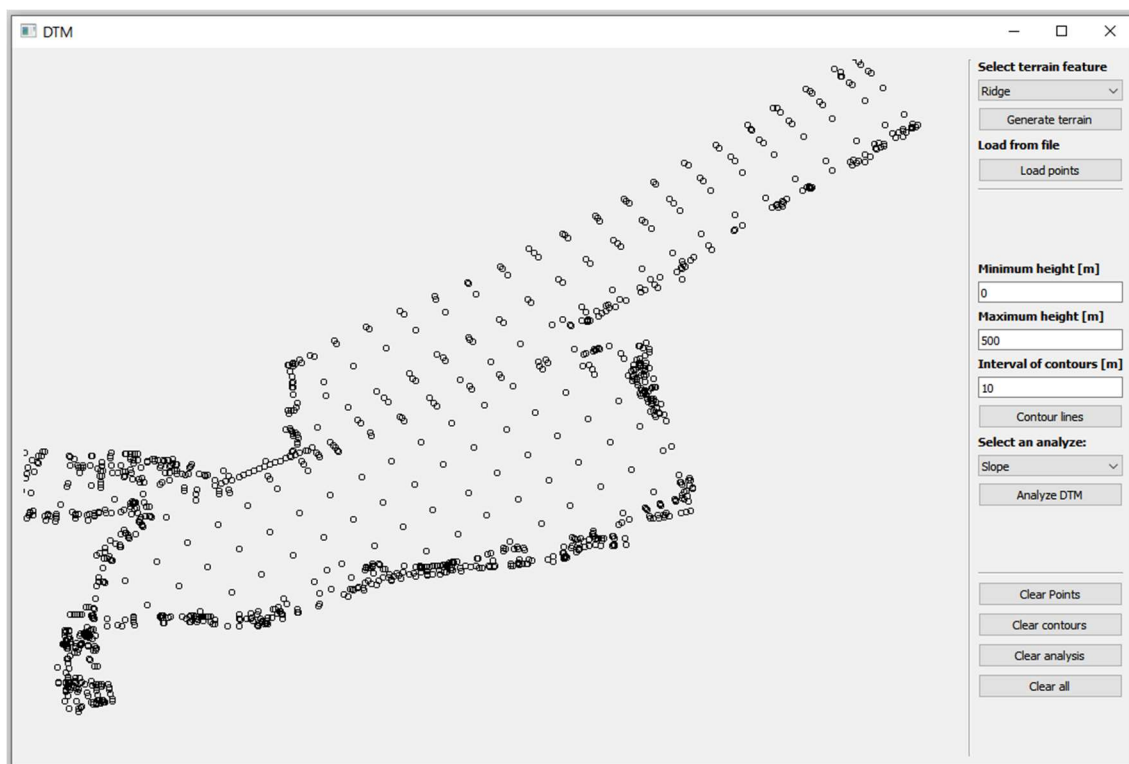
Obrázek 7 Generování spočinku



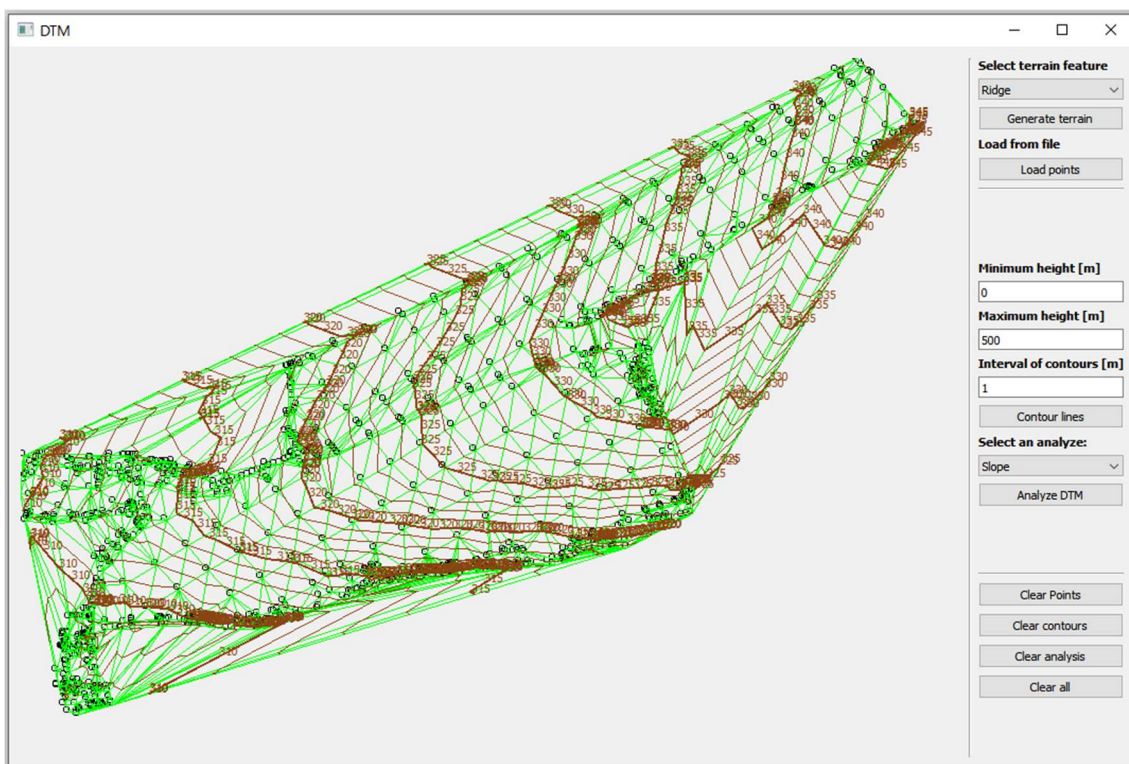
Obrázek 8 Generování hřbetu



Obrázek 9 Import textového souboru se souřadnicemi bodů

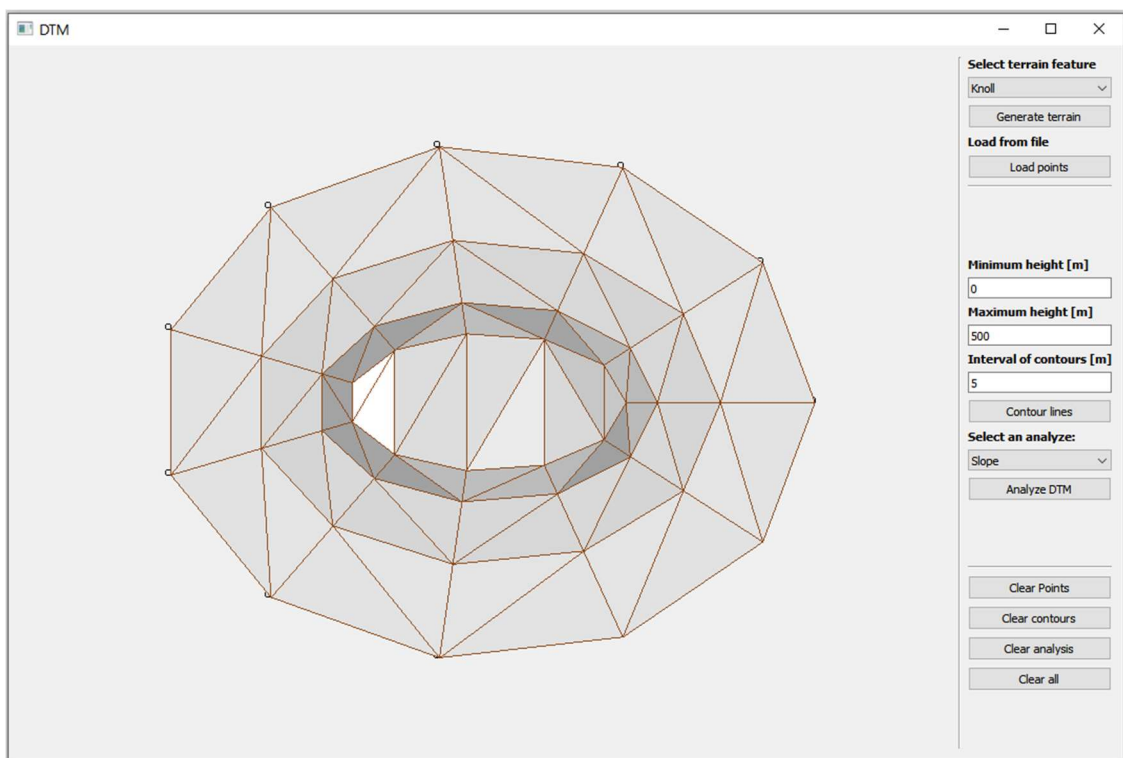


Obrázek 10 Importované body z měření

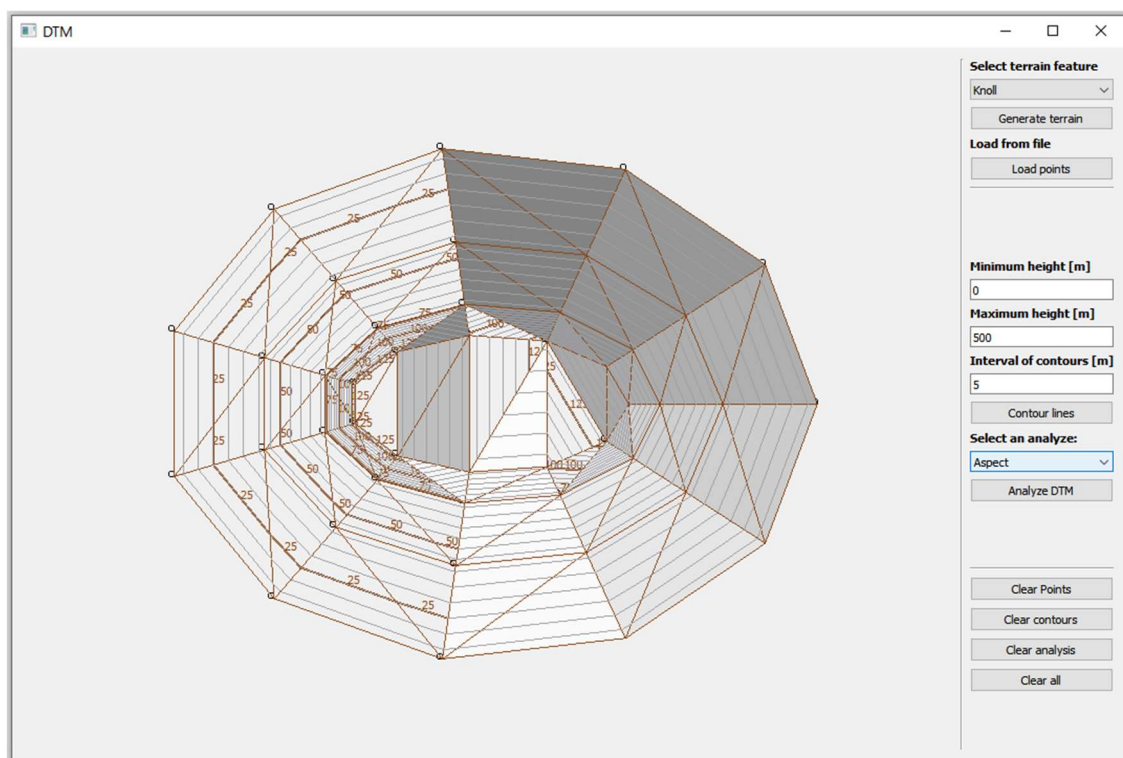


Obrázek 11 Vygenerování vrstevnic na importovaných bodech

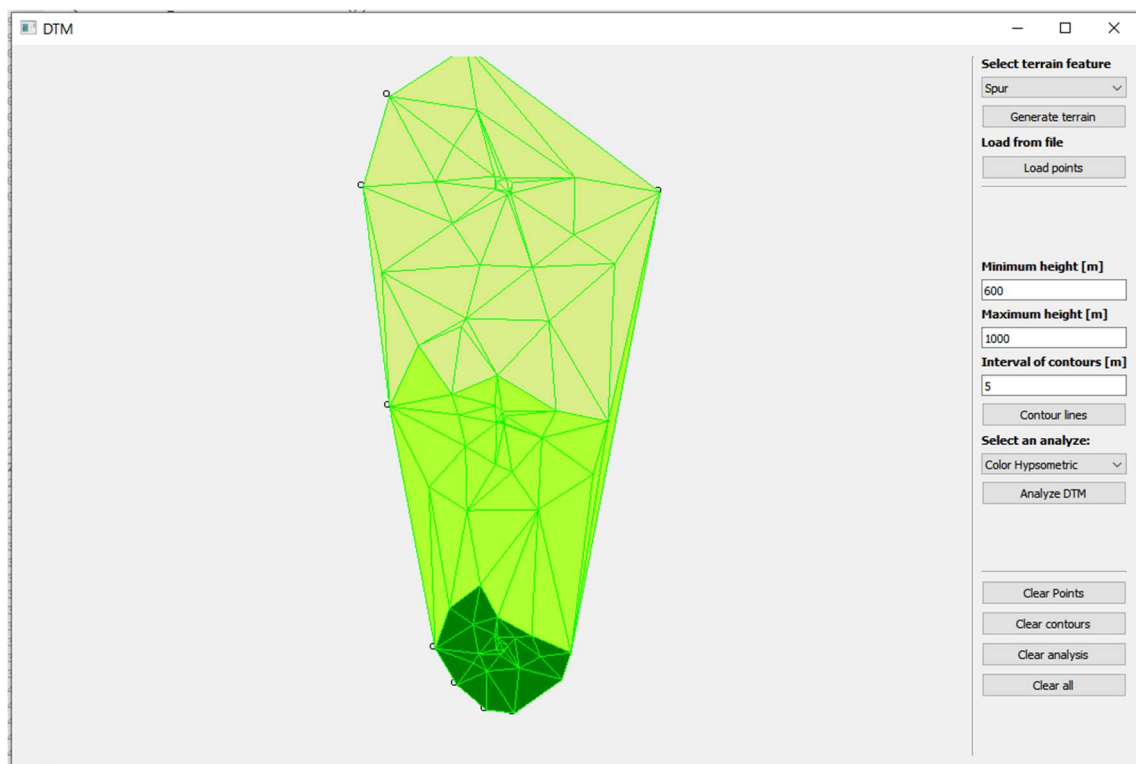




Obrázek 12 Sklon terénu



Obrázek 13 Orientace terénu



Obrázek 14 Barevná hypsometrie

## 7. Dokumentace

### 7.1 Třída algorithms.h

```
public:
    Algorithms();
    // Funkce na určení polohy bodu q vůči přímce zadané dvěma body p1 a p2
    int getPointLinePosition(QPoint3D &q, QPoint3D &p1, QPoint3D &p2);
    // Funkce na výpočet velikosti poloměru kružnice tvořené třemi vstupními body
    void circleCenterAndRadius(QPoint3D &p1, QPoint3D &p2, QPoint3D &p3, double &r, QPoint3D
&s);
    // Funkce na nalezení indexu bodu, který splňuje Delaunayove vlastnosti
    int findDelaunayPoint(QPoint3D &pi, QPoint3D &pj, std::vector<QPoint3D> &points);
    // Funkce na výpočet vzdálenosti dvou bodů
    double dist(QPoint3D &p1, QPoint3D &p2);
    // Funkce na nalezení indexu nejbližšího bodu z množiny bodů k bodu p
    int getNearestpoint(QPoint3D &p, std::vector<QPoint3D> &points);
    // Funkce, která vytvoří Delauneyovu triangulaci a je ukádána do vektoru hran
    std::vector<Edge> DT(std::vector<QPoint3D> &points);
    // Funkce na změnu orientace hrany, pokud hrana není v AEL
    void updateAEL(Edge &e, std::list<Edge> &ael);
    // Funkce na výpočet průsečníku hrany tvořené body p1 a p2 s rovinou danou souřadnicí Z
    QPoint3D getContourPoint(QPoint3D &p1, QPoint3D &p2, double z);
    // Funkce na tvoření kran vrstevnic
    std::vector<Edge> contourLines(std::vector<Edge> &dt, double z_min, double z_max,
double dz, std::vector<double> &contour_heights);
    // Funkce na výpočet sklonu trojúhelníku definovaném 3D body p1, p2 a p3
    double getSlope(QPoint3D &p1, QPoint3D &p2, QPoint3D &p3);
    // Funkce na výpočet expozice trojúhelníku definovaném 3D body p1, p2 a p3
    double getAspect(QPoint3D &p1, QPoint3D &p2, QPoint3D &p3);
    // Funkce, která počítá analýzu sklonu a expozice pro jednotlivé trojúhelníky
    std::vector<Triangle> analyzeDTM(std::vector<Edge> &dt);
    // Funkce na generování terénních tvarů
    std::vector<QPoint3D> generateTF(int width, int height, int tf);
```

### 7.2 Třída draw.h

```
private:
    // Deklarace proměnné pro množinu bodů
    std::vector<QPoint3D> points;
    // Deklarace vektoru hran Delauneyovho trojúhelníka
    std::vector<Edge> dt;
    // Deklarace vektoru hran vrstevnic
    std::vector<Edge> contours;
    // Deklarace vektoru trojúhelníků digitálního modelu terénu
    std::vector<Triangle> dtm;
    // Deklarace vektoru výšek vrstevnic
    std::vector<double> contour_heights;
    // Deklarace převýšení
    int dz;
    // Deklarace proměnných sklon, expozice a hypsometrie
    bool slope, aspect, colHyps;
public:
    // Okno kreslení
    explicit Draw(QWidget *parent = nullptr);
    // Funkce pro získání souřadnic bodů z Canvasu (co se stane po kliknutí myši do Canvasu)
    void mousePressEvent(QMouseEvent *e);
    // Funkci na kreslení bodů a konvexní obálky
    void paintEvent(QPaintEvent *e);
    // Funkce pro získání souřadnic bodů
    std::vector<QPoint3D> & getPoints(){return points;}
    // Funkce pro nastavení privátní položky points, aby byla vidět v třídě draw
    void setPoints(std::vector<QPoint3D> &points_){points=points_;}
    // Funkce pro nastavení privátní položky dt
    void setDT(std::vector<Edge> &dt_){dt = dt_;}
    // Funkce pro získání privátní složky dt
    std::vector<Edge> & getDT(){return dt;}
    // Funkce pro nastavení privátní složky vrstevnic
    void setContours(std::vector<Edge> &contours_, std::vector<double> &contour_heights_, int
dz_);
    // Funkce pro získání privátní složky vrstevnic
    std::vector<Edge> & getContours(){return contours;}
    // Funkce pro nastavení privátní složky dtm
    void setDTM(std::vector<Triangle> &dtm_){dtm = dtm_;}
    // Funkce pro získání trojúhelníků z privátní složky dtm
    std::vector<Triangle> & getDTM(){return dtm;}
```

```
// Funkce pro nastavení privátní složky slope, aspect
void setSlope(bool &slope_){slope = slope_;}
// Funkce pro nastavení privátní složky hypsometrie
void setColHyps(bool &colHyps_){colHyps = colHyps_;}
// Funkce pro načítání souřadnic ze souboru
void loadFile(std::string &path, std::vector<QPoint3D> &points, QSizeF &canvas_size,
double &min_z, double &max_z);
```

### 7.3 Třída edge.h

Třída je vytvořena ze dvou 3D bodů – počátku *s* a konce *e* hrany. Slouží ke uložení hrany triangulace i vrstevnic.

```
class Edge
{
    private:
        QPoint3D s, e;
    public:
        Edge(): s(0, 0, 0), e(0, 0, 0){}
        Edge(QPoint3D &s_, QPoint3D &e_):s(s_), e(e_){}
        void setStart(QPoint3D &s_){s = s_;}
        void setEnd(QPoint3D &e_){e = e_;}
        QPoint3D &getStart(){return s;}
        QPoint3D &getEnd(){return e;}
        void changeOrientation()
        {
            QPoint3D a = s;
            s = e;
            e = a;
        }
        bool operator == (const Edge &h) const
        {
            return (h.s == s) && (h.e == e);
        }
};
```

### 7.4 Třída qpoint3d.h

Třída odvozena z třídy *QPointF* sloužící na uložení 3D souřadnic bodů.

```
class QPoint3D:public QPointF
{
    private:
        double z;
    public:
        QPoint3D():QPointF(0, 0),z(0){}
        QPoint3D(double x, double y, double z_):QPointF (x, y), z(z_){}
        double getZ(){return z;}
        void setZ(double z_) {z = z_;}
};
```

### 7.5 Třída sortbyx.h

Třída na setřídění souřadnic bodů podle osy *X*.

```
class sortByX
{
    public:
        bool operator() (QPoint3D &p1, QPoint3D &p2)
        {
            return p1.x() < p2.x();
        }
};
```

### 7.6 Třída triangle.h

Třída sloužící k uložení trojúhelníku definovaného body *p1*, *p2* a *p3* a jeho informací o sklonu a expozici.

```
class Triangle
{
    private:
        QPoint3D p1, p2, p3;
        double slope, aspect;
```



```

public:
    Triangle(QPoint3D &p1_, QPoint3D &p2_, QPoint3D &p3_, double slope_, double aspect_):
        p1(p1_), p2(p2_), p3(p3_), slope(slope_), aspect(aspect_){};

    QPoint3D getP1(){return p1;}
    QPoint3D getP2(){return p2;}
    QPoint3D getP3(){return p3;}
    double getSlope(){return slope;}
    double getAspect(){return aspect;}

    void setP1(QPoint3D &P1){p1 = P1;}
    void setP2(QPoint3D &P2){p2 = P2;}
    void setP3(QPoint3D &P3){p3 = P3;}
    void setSlope(double &slope_){slope = slope_;}
    void setAspect(double &aspect_){aspect = aspect_;}
};

```

## 7.7 Třída widget.h

```

class Widget : public QWidget
{
    Q_OBJECT
private:
    double z_min, z_max, dz;
public:
    Widget(QWidget *parent = nullptr);
    ~Widget();
private slots:
    // Po kliknutí se vygenerují vrstevnice zadaných bodů
    void on_pushButton_7_clicked();
    // Mazání bodů
    void on_pushButton_11_clicked();
    // Mazání vrstevnic
    void on_pushButton_12_clicked();
    // Po vložení hodnoty se nastaví minimální výška ručně vloženému bodu
    void on_lineEdit_editingFinished();
    // Po vložení hodnoty se nastaví maximální výška ručně vloženému bodu
    void on_lineEdit_2_editingFinished();
    // Po vložení hodnoty se nastaví interval souřadnic
    void on_lineEdit_3_editingFinished();
    // Po kliknutí se vykoná analýza terénu - sklon, expozice, barevná hypsometrie
    void on_pushButton_2_clicked();
    // Mazání analýzy
    void on_pushButton_3_clicked();
    // Vymazání celého grafického okna
    void on_pushButton_4_clicked();
    // Po stisknutí se načítají body ze souboru
    void on_pushButton_clicked();
    // Po stisknutí se vygeneruje vybraný terénní tvar
    void on_pushButton_5_clicked();
private:
    Ui::Widget *ui;
};

```

## 7.8 algorithms.cpp

```

int Algorithms::getPointLinePosition(QPoint3D &q, QPoint3D &p1, QPoint3D &p2)
{
    //Analyze point and line position
    //1 point in the left half plane
    //0 point in the right half plane
    //-1 point on the line
    double ux = p2.x() - p1.x();
    double uy = p2.y() - p1.y();

    double vx = q.x() - p1.x();
    double vy = q.y() - p1.y();

    //Test criterion
    double t = ux * vy - uy * vx;

    //Point in the left half plane
    if (t>0)
        return 1;
}

```

```

        //Point in the right halfplane
        if (t<0)
            return 0;

        //Collinear point
        return -1;
    }

void Algorithms::circleCenterAndRadius(QPoint3D &p1, QPoint3D &p2, QPoint3D &p3, double &r,
QPoint3D &s)
{
    //Center and radius of circle given by 3 points p1, p2, p3
    double k1 = p1.x() * p1.x() + p1.y() * p1.y();
    double k2 = p2.x() * p2.x() + p2.y() * p2.y();
    double k3 = p3.x() * p3.x() + p3.y() * p3.y();
    double k4 = p1.y() - p2.y();
    double k5 = p1.y() - p3.y();
    double k6 = p2.y() - p3.y();
    double k7 = p1.x() - p2.x();
    double k8 = p1.x() - p3.x();
    double k9 = p2.x() - p3.x();
    double k10 = p1.x() * p1.x();
    double k11 = p2.x() * p2.x();
    double k12 = p3.x() * p3.x();

    //Compute m: x-coordinate of circle center
    double m_numerator = -k12 * k4 + k11 * k5 - (k10 + k4 * k5) * k6;
    double m_denominator = -p3.x() * k4 + p2.x() * k5 - p1.x() * k6;
    double m = 0.5 * m_numerator / m_denominator;

    //Compute n: y-coordinate of circle center
    double n_numerator = -k1 * k9 + k2 * k8 - k3 * k7;
    double n_denominator = -p1.y() * k9 + p2.y() * k8 - p3.y() * k7;
    double n = 0.5 * n_numerator / n_denominator;

    //Compute circle radius
    r = sqrt((p1.x() - m) * (p1.x() - m) + (p1.y() - n) * (p1.y() - n));

    //Set m,n to s
    s.setX(m);
    s.setY(n);
}

int Algorithms::findDelaunayPoint(QPoint3D &pi, QPoint3D &pj, std::vector<QPoint3D> &points)
{
    //Find optimal Delaunay point (third vertex of of triangle)
    int i_min = -1;
    double r_min = INFINITY;

    //Browse all input points
    for (int i = 0; i < points.size(); i++)
    {
        //Check, whether points[i] different from pi, pj
        if((points[i]!=pi) && (points[i]!=pj))
        {
            //Point in the left halfplane
            if(getPointLinePosition(points[i], pi, pj)==1)
            {
                //Compute circle center and radius
                QPoint3D s;
                double r;
                circleCenterAndRadius(pi, pj, points[i], r, s);

                //Circle center in the right halplane
                if(getPointLinePosition(s, pi, pj)==0)
                    r = -r;

                //Actualize minimum
                if(r<r_min)
                {
                    r_min = r;
                    i_min = i;
                }
            }
        }
    }
}

```

```

    }

    return i_min;
}

double Algorithms::dist(QPoint3D &p1, QPoint3D &p2)
{
    //Get Euclidean distance between two points
    double dx = (p1.x() - p2.x());
    double dy = (p1.y() - p2.y());

    return sqrt(dx*dx + dy*dy);
}

int Algorithms::getNearestpoint(QPoint3D &p, std::vector<QPoint3D> &points)
{
    //Find nearest point to p
    int i_min = 1;
    double d_min = dist(p, points[1]);

    //Browses all points
    for (unsigned int i = 2; i < points.size(); i++)
    {
        //Compute distance
        double d = dist(p, points[i]);

        //Actualize minimum i and distance
        if (d < d_min)
        {
            d_min=d;
            i_min=i;
        }
    }

    return i_min;
}

std::vector<Edge> Algorithms:: DT(std::vector<QPoint3D> &points)
{
    //Delaunay triangulation
    std::vector<Edge> dt;
    std::list<Edge> ael;

    //Sort by X
    sort(points.begin(), points.end(), sortByX());

    //Pivot
    QPoint3D q = points[0];

    //Nearest point
    int index = getNearestpoint(q, points);
    QPoint3D p_nearest = points[index];

    //Find optimal Delaunay point
    int i_o = findDelaunayPoint(q, p_nearest, points);

    //Create new edge
    Edge e(q, p_nearest);

    //No suitable point found in the left halfplane
    if (i_o == -1)
    {
        //Change orientation of the edge
        e.changeOrientation();

        //Find optimal Delaunay points once more
        i_o = findDelaunayPoint(e.getStart(), e.getEnd(), points);
    }

    //3rd vertex of triangle
    QPoint3D p_3 = points[i_o];

    //Create inicial triangle
    Edge e2(e.getEnd(), p_3);
    Edge e3(p_3, e.getStart());

    //Add triangle to DT

```

```

dt.push_back(e);
dt.push_back(e2);
dt.push_back(e3);

//Add edges to AEL
ael.push_back(e);
ael.push_back(e2);
ael.push_back(e3);

//Until ael is empty process candidate edges
while (!ael.empty())
{
    //Get last edge
    Edge edge1 = ael.back();
    ael.pop_back();

    //Change orientation
    edge1.changeOrientation();

    //Find optimal Delaunay point
    i_o = findDelaunayPoint(edge1.getStart(), edge1.getEnd(), points);

    //Optimal point found
    if (i_o != -1)
    {
        //3rd vertex of triangle
        QPoint3D p3 = points[i_o];

        //Create inicial triangle
        Edge edge2(edge1.getEnd(), p3);
        Edge edge3(p3, edge1.getStart());

        //Add triangle to DT
        dt.push_back(edge1);
        dt.push_back(edge2);
        dt.push_back(edge3);

        //Change orientation of edges
        edge2.changeOrientation();
        edge3.changeOrientation();

        //Update AEL
        updateAEL(edge2, ael);
        updateAEL(edge3, ael);
    }
}
return dt;
}

void Algorithms::updateAEL(Edge &e, std::list<Edge> &ael)
{
    //Update AEL
    std::list<Edge>::iterator ie = find(ael.begin(), ael.end(), e);

    //Edge is not in AEL
    if(ie == ael.end())
    {
        //Change orientation
        e.changeOrientation();

        //Add edge to AEL
        ael.push_back(e);
    }

    //Edge is already in list, erase
    else ael.erase(ie);
}

QPoint3D Algorithms::getContourPoint(QPoint3D &p1, QPoint3D &p2, double z)
{
    //Create point on contour
    double xa = (p2.x()-p1.x()) / (p2.getZ()-p1.getZ()) * (z-p1.getZ())+p1.x();
    double ya = (p2.y()-p1.y()) / (p2.getZ()-p1.getZ()) * (z-p1.getZ())+p1.y();

    QPoint3D A(xa, ya, z);
    return A;
}

```

```

std::vector<Edge> Algorithms::contourLines(std::vector<Edge> &dt, double z_min, double z_max,
double dz, std::vector<double> &contour_heights)
{
    //Create contour lines using linear interpolation
    std::vector<Edge> contours;

    //Browse all triangles one by one
    for(int i=0; i<dt.size(); i+=3)
    {
        //Get triangle vertices
        QPoint3D p1 = dt[i].getStart();
        QPoint3D p2 = dt[i].getEnd();
        QPoint3D p3 = dt[i+1].getEnd();

        //Compute all contour lines for a triangle
        for (double z = z_min; z <= z_max; z += dz)
        {
            //Get height differences
            double dz1 = z - p1.getZ();
            double dz2 = z - p2.getZ();
            double dz3 = z - p3.getZ();

            //Compute test criteria
            double t12 = dz1*dz2;
            double t23 = dz2*dz3;
            double t31 = dz3*dz1;

            //Triangle in the plane
            if ((dz1 == 0) && (dz2 == 0) && (dz3 == 0))
                continue;

            //Edge 1,2 of triangle in plane
            else if ((dz1 == 0) && (dz2 == 0))
                contours.push_back(dt[i]);

            //Edge 2,3 of triangle in plane
            else if ((dz2 == 0) && (dz3 == 0))
                contours.push_back(dt[i+1]);

            //Edge 3,1 of triangle in plane
            else if ((dz3 == 0) && (dz1 == 0))
                contours.push_back(dt[i+2]);

            //Edges 1,2 and 2,3 are intersected by plane
            else if ((t12 < 0) && (t23 <= 0) || (t12 <= 0) && (t23 < 0))
            {
                //Compute points of intersection
                QPoint3D A = getContourPoint(p1, p2, z);
                QPoint3D B = getContourPoint(p2, p3, z);

                //Create contour line and add to the list
                Edge e(A, B);
                contours.push_back(e);
                contour_heights.push_back(z);
            }

            //Edges 2,3 and 3,1 are intersected by plane
            else if ((t23 < 0) && (t31 <= 0) || (t23 <= 0) && (t31 < 0))
            {
                //Compute points of intersection
                QPoint3D A = getContourPoint(p2, p3, z);
                QPoint3D B = getContourPoint(p3, p1, z);

                //Create contour line and add to the list
                Edge e(A, B);
                contours.push_back(e);
                contour_heights.push_back(z);
            }

            //Edges 1,2 and 3,1 are intersected by plane
            else if ((t12 < 0) && (t31 <= 0) || (t12 <= 0) && (t31 < 0))
            {
                //Compute points of intersection
                QPoint3D A = getContourPoint(p1, p2, z);
                QPoint3D B = getContourPoint(p3, p1, z);
            }
        }
    }
}

```

```

        //Create contour line and add to the list
        Edge e(A, B);
        contours.push_back(e);
        contour_heights.push_back(z);
    }
}

return contours;
}

double Algorithms::getSlope(QPoint3D &p1, QPoint3D &p2, QPoint3D &p3)
{
    //Compute slope of triangle in DT
    double ux = p2.x() - p1.x();
    double uy = p2.y() - p1.y();
    double uz = p2.getZ() - p1.getZ();

    double vx = p3.x() - p1.x();
    double vy = p3.y() - p1.y();
    double vz = p3.getZ() - p1.getZ();

    //Normal vector
    double nx = uy*vz - uz*vy;
    double ny = -(ux*vz - uz*vx);
    double nz = ux*vy - uy*vx;

    //Norm
    double nt = sqrt(nx*nx + ny*ny + nz*nz);

    return acos(nz/nt)*180/M_PI;
}

double Algorithms::getAspect(QPoint3D &p1, QPoint3D &p2, QPoint3D &p3)
{
    //Compute aspect of triangle in DT
    double ux = p2.x() - p1.x();
    double uy = p2.y() - p1.y();
    double uz = p2.getZ() - p1.getZ();

    double vx = p3.x() - p1.x();
    double vy = p3.y() - p1.y();
    double vz = p3.getZ() - p1.getZ();

    //Normal vector and its norm
    double nx = uy * vz - vy * uz;
    double ny = -(ux * vz - vx * uz);

    return atan2(nx, ny) / M_PI * 180;
}

std::vector<Triangle> Algorithms::analyzeDTM(std::vector<Edge> &dt)
{
    //Analyze slope and aspect of DTM
    std::vector<Triangle> triangles;

    //Process each triangle
    for (int i = 0; i<dt.size() ;i += 3)
    {
        //Get vertices
        QPoint3D p1 = dt[i].getStart();
        QPoint3D p2 = dt[i].getEnd();
        QPoint3D p3 = dt[i+1].getEnd();

        //Compute slope and aspect
        double slope = getSlope(p1, p2, p3);
        double aspect = getAspect(p1, p2, p3);

        //Create triangle
        Triangle triangle (p1, p2, p3, slope, aspect);

        //Add triangle to the list
        triangles.push_back(triangle);
    }

    return triangles;
}

```

```

std::vector<QPoint3D> Algorithms::generateTF(int width, int height, int tf)
{
    std::vector<QPoint3D> points;

    //Knoll
    if (tf==0)
    {
        QPoint3D r_points;
        QPoint3D top;

        //Ellipse parameters
        double a = width * 0.15;
        double b = height * 0.1;
        int n = rand()%5+10; //Number of point in particular ellipse

        //Divide the ellipse to angles according to n value
        double fi = (2*M_PI)/(n);

        //Set coordinates to the peak
        top.setX(width/2);
        top.setY(height/2);
        top.setZ(rand()%200 + 100);

        //Create more ellipses with different radius and with different heights (bigger
radius, smaller height)
        for(int j = 0;j<5;j++)
        {
            //Generate ellipse with n points
            for(int i = 0;i<n;i++)
            {
                r_points.setX(top.x() + a*cos(i*fi));
                r_points.setY(top.y() + b*sin(i*fi));
                r_points.setZ(top.getZ()-j*30);
                points.push_back(r_points);
            }
            a+=j*30;
            b+=j*30;
        }
    }

    //Valley
    else if (tf==1)
    {
        std::vector<QPoint3D> r_points;
        QPoint3D center;
        QPoint3D right;
        QPoint3D left;

        int n = rand()%20 + 10; //number of points in one line
        double a = width * 0.15; //length of lines from center point

        //Center lowest point
        center.setX(width/2-50);
        center.setY(height/4-150);
        center.setZ(rand()%50);

        for(int j = 0;j<5;j++)
        {
            //Create two parallel lines of points with the same height
            for(int i = 0;i<n;i++)
            {
                right.setX(center.x()+a);
                right.setY(center.y()+i*40);
                right.setZ(center.getZ()+a);

                left.setX(center.x()-a);
                left.setY(center.y()+i*40);
                left.setZ(center.getZ()+a);

                r_points.push_back(right);
                r_points.push_back(left);
            }

            a+=j*40;
        }
    }
}

```

```

        // Adding elements one by one to the vector
        for(int i=0; i < r_points.size(); i++)
        {
            QPoint3D a = r_points[i];
            points.push_back(a);
        }
    }

    //Spur
    else if (tf==2)
    {
        std::vector<QPoint3D> r_points;
        QPoint3D bot, mid, top; //bottom, middle and top parts of spur
        QPoint3D random;
        QPoint3D center;

        //Ellipse parameters
        double a = width * 0.15;
        double b = height * 0.1;
        int n = rand()%10 + 3; //Number of point in particular ellipse
        int m = rand()%100 + 150; //Size of terrain shape

        center.setX(width/2);
        center.setY(height/2);

        // Center points of parts
        bot.setX(center.x());
        bot.setY(center.y()+m);
        bot.setZ(100);

        mid.setX(center.x());
        mid.setY(center.y());
        mid.setZ(200);

        top.setX(center.x());
        top.setY(center.y()-m);
        top.setZ(300);

        r_points.push_back(bot);
        r_points.push_back(mid);
        r_points.push_back(top);

        double fi = 2*M_PI/10;
        for(int i = 0; i<3; i++)
        {
            //Create the bottom part
            for (int j = 0; j < n; j++)
            {
                random.setX(bot.x() + i*30*cos(j*fi) + rand()%20);
                random.setY(bot.y() + i*30*sin(j*fi) + rand()%20);
                random.setZ(bot.getZ());
                r_points.push_back(random);
            }

            //Create the middle part
            for (int j = 0; j < n; j++)
            {
                random.setX(mid.x() + i*50*cos(j*fi) + rand()%20);
                random.setY(mid.y() + i*50*sin(j*fi) + rand()%20);
                random.setZ(mid.getZ());
                r_points.push_back(random);
            }

            //Create the top part
            for (int j = 0; j < n; j++)
            {
                random.setX(top.x() + i*70*cos(j*fi) + rand()%20);
                random.setY(top.y() + i*70*sin(j*fi) + rand()%20);
                random.setZ(top.getZ());
                r_points.push_back(random);
            }

            a+=i*50;
            b+=i*50;
        }
    }
}

```



```

        // Adding elements one by one to the vector
        for(int i=0; i < r_points.size(); i++)
        {
            QPoint3D a = r_points[i];
            points.push_back(a);
        }
    }

    //Ridge
    else if (tf==3)
    {
        std::vector<QPoint3D> r_points;
        QPoint3D center;
        QPoint3D right;
        QPoint3D left;

        int n = rand()%20 + 10; //number of points in one line
        double a = width * 0.05; //length of lines from center point

        //Center lowest point
        center.setX(width/2-50);
        center.setY(height/4-150);
        center.setZ(rand()%50+500);

        for(int j = 0; j<5; j++)
        {
            //Create two parallel lines of points with the same height
            for(int i = 0; i<n; i++)
            {
                right.setX(center.x()+a);
                right.setY(center.y()+i*40);
                right.setZ(center.getZ()-a);

                left.setX(center.x()-a);
                left.setY(center.y()+i*40);
                left.setZ(center.getZ()-a);

                r_points.push_back(right);
                r_points.push_back(left);
            }

            a+=j*40;
        }

        // Adding elements one by one to the vector
        for(int i=0; i < r_points.size(); i++)
        {
            QPoint3D a = r_points[i];
            points.push_back(a);
        }
    }
    return points;
}

```

## 7.9 draw.cpp

```

Draw::Draw(QWidget *parent) : QWidget(parent)
{
}

void Draw::setContours(std::vector<Edge> &contours_, std::vector<double> &contour_heights_, int
dz_) {
    contours = contours_;
    contour_heights = contour_heights_;
    dz = dz_;
}

void Draw::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.begin(this);

    //Draw points
    for (int i = 0; i < points.size(); i++)
    {
        painter.drawEllipse(points[i].x() - 5, points[i].y() - 5, 5, 5);
    }
}

```

```

}

//Draw Delaunay edges
QPen p(Qt::green, 1);
painter.setPen(p);

for (int i = 0; i < dt.size(); i++)
{
    painter.drawLine(dt[i].getStart(), dt[i].getEnd());
}

//Draw contour lines
int main_contour = dz * 5;

for (int i = 0; i < contours.size(); i++)
{
    double text_x = (contours[i].getStart().x() + contours[i].getEnd().x())/2;
    double text_y = (contours[i].getStart().y() + contours[i].getEnd().y())/2;
    int height = contour_heights[i];
    if(height%(main_contour))
    {
        QPen q(QColor(139,69,19), 1);
        painter.setPen(q);
        painter.drawLine(contours[i].getStart(), contours[i].getEnd());
    }
    else
    {
        QPen q(QColor(139,69,19), 2);
        painter.setPen(q);
        painter.drawLine(contours[i].getStart(), contours[i].getEnd());
        painter.drawText(text_x, text_y, QString::number(height));
    }
}

//Draw slope
if(slope == TRUE)
{
    for (Triangle t : dtm)
    {
        double k = 255/180.0;
        //Get triangle vertices
        QPoint3D p1 = t.getP1();
        QPoint3D p2 = t.getP2();
        QPoint3D p3 = t.getP3();

        //Get slope
        int colorSlope = 255 - t.getSlope() * k;

        //Create color and set brush
        QColor c(colorSlope,colorSlope,colorSlope);
        painter.setBrush(c);

        //Create triangle, add vertices
        QPolygonF triangle;
        triangle.append(QPointF(p1.x(), p1.y()));
        triangle.append(QPointF(p2.x(), p2.y()));
        triangle.append(QPointF(p3.x(), p3.y()));

        //Draw triangle
        painter.drawPolygon(triangle);
    }
    painter.end();
}

//Draw aspect
if(aspect == TRUE)
{
    for (Triangle t : dtm)
    {
        double k = 255.0 / 360;
        //Get triangle vertices
        QPoint3D p1 = t.getP1();
        QPoint3D p2 = t.getP2();
        QPoint3D p3 = t.getP3();

        //Get aspect
        int colorAspect = 255 - t.getAspect() * k;
    }
}

```

```

        //Create color and set brush
        QColor c(colorAspect, colorAspect, colorAspect);
        painter.setBrush(c);

        //Create triangle, add vertices
        QPolygonF triangle;
        triangle.append(QPointF(p1.x(), p1.y()));
        triangle.append(QPointF(p2.x(), p2.y()));
        triangle.append(QPointF(p3.x(), p3.y()));

        //Draw triangle
        painter.drawPolygon(triangle);
    }

    //Draw contour lines
    QPen q(Qt::gray, 1);
    painter.setPen(q);

    for (int i = 0; i < contours.size(); i++)
    {
        painter.drawLine(contours[i].getStart(), contours[i].getEnd());
    }

    painter.end();
}

//Draw color hypsometric
if(colHyps == TRUE)
{
    for (Triangle t : dtm)
    {
        //Get triangle vertices
        QPoint3D p1 = t.getP1();
        QPoint3D p2 = t.getP2();
        QPoint3D p3 = t.getP3();

        double z1 = p1.getZ();
        double z2 = p2.getZ();
        double z3 = p3.getZ();
        double z1_;
        double z2_;
        double z3_;

        std::vector<double> zs;

        zs.push_back(z1);
        zs.push_back(z2);
        zs.push_back(z3);

        std::sort (zs.begin(),zs.end());

        if ((zs[0]-zs[2])>20)
        {
            z1_ = zs[0]*20;
            z2_ = zs[1]*40;
            z3_ = zs[2]*40;
        }
        else
        {
            z1_ = zs[0]*100/3;
            z2_ = zs[1]*100/3;
            z3_ = zs[2]*100/3;
        }

        double average = (z1_ + z2_ + z3_)/100;

        //double average = (z1 + z2 + z3)/3;

        if (average <= 100)
        {
            painter.setBrush(QColor(0,128,0)); // green
        }
        else if (average <= 200)
        {
            painter.setBrush(QColor(173,255,47)); // greenyellow
        }
    }
}

```

```

        else if (average <= 300)
        {
            painter.setBrush(QColor(217,238,136)); // lightgreen
        }

        else if (average <= 400)
        {
            painter.setBrush(QColor(255,255,0)); // yellow
        }

        else if (average <= 500)
        {
            painter.setBrush(QColor(255,165,0)); // orange
        }

        else if (average <= 600)
        {
            painter.setBrush(QColor(192,128,0)); // lightbrown
        }

        else if (average > 600)
        {
            painter.setBrush(QColor(139,69,19)); // saddlebrown
        }

        //Create triangle, add vertices
        QPolygonF triangle;
        triangle.append(QPointF(p1.x(), p1.y()));
        triangle.append(QPointF(p2.x(), p2.y()));
        triangle.append(QPointF(p3.x(), p3.y()));

        //Draw triangle
        painter.drawPolygon(triangle);
    }
    painter.end();
}

}

void Draw::mousePressEvent(QMouseEvent *event)
{
    //Get coordinates of cursor
    QPoint3D p;
    p.setX(event->x());
    p.setY(event->y());
    double random = std::rand() * 800.0 / RAND_MAX;
    p.setZ(random);

    //Add point to the list
    points.push_back(p);

    repaint();
};

//void Draw::loadFile(std::string &string_path)
void Draw::loadFile(std::string &path, std::vector<QPoint3D> &points, QSizeF &canvas_size,
double &min_z, double &max_z)
{
    double x, y, z;
    QPoint3D p;

    //Storing all polygons
    double min_x = std::numeric_limits<double>::max();
    double min_y = std::numeric_limits<double>::max();
    double max_x = std::numeric_limits<double>::min();
    double max_y = std::numeric_limits<double>::min();
    min_z = std::numeric_limits<double>::max();
    max_z = std::numeric_limits<double>::min();

    std::ifstream myfile(path);
    if(myfile.is_open())
    {
        while(myfile >> x >> y >> z)
        {
            p.setX(x);
            p.setY(y);

```

```

        p.setZ(z);

        points.push_back(p);

        if(x < min_x) min_x = x;
        if(x > max_x) max_x = x;
        if(y < min_y) min_y = y;
        if(y > max_y) max_y = y;
        if(z < min_z) min_z = z;
        if(z > max_z) max_z = z;
    }

    myfile.close();
}

//Scale points to canvas size
double h = canvas_size.height() - 40;
double w = canvas_size.width() - 40;

double x_coef = w/(max_x-min_x);
double y_coef = h/(max_y-min_y);

for(unsigned int i = 0; i < points.size(); i++)
{
    points[i].setX((points[i].x()-min_x)*x_coef);
    points[i].setY((points[i].y()-min_y)*y_coef);
}
}

```

## 7.10 widget.cpp

```

Widget::Widget(QWidget *parent)
: QWidget(parent)
, ui(new Ui::Widget)
{
    ui->setupUi(this);
    z_min = 0;
    z_max = 500;
    dz = 1;
}

Widget::~Widget()
{
    delete ui;
}

void Widget::on_pushButton_7_clicked()
{
    //Create contour lines
    Algorithms a;
    std::vector<Edge> dt;

    //DT needs to be created
    if(dt.size() == 0)
    {
        //Get points
        std::vector<QPoint3D> points = ui->Canvas->getPoints();

        //Create DT
        dt = a.DT(points);

        //Set DT
        ui->Canvas->setDT(dt);
    }

    //Create contour lines
    int dz = ui -> lineEdit_3 -> text().toInt();
    std::vector<double> contour_heights;
    std::vector<Edge> contours = a.contourLines(dt, z_min, z_max, dz, contour_heights);

    //Set contours
    ui -> Canvas -> setContours(contours, contour_heights, dz);

    //Repaint
    repaint();
}

```

```

void Widget::on_pushButton_11_clicked()
{
    //Get points
    std::vector<QPoint3D> &points = ui->Canvas->getPoints();

    //Clear points
    points.clear();

    //Repaint
    repaint();
}

void Widget::on_pushButton_12_clicked()
{
    //Get DT and contours
    std::vector<Edge> &dt = ui->Canvas->getDT();
    std::vector<Edge> &contours = ui->Canvas->getContours();

    //Clear DT and contour lines
    dt.clear();
    contours.clear();

    //Repaint
    repaint();
}

void Widget::on_lineEdit_editingFinished()
{
    //Set z_min
    z_min = ui -> lineEdit -> text().toDouble();
}

void Widget::on_lineEdit_2_editingFinished()
{
    //Set z_max
    z_max = ui -> lineEdit_2 -> text().toDouble();
}

void Widget::on_lineEdit_3_editingFinished()
{
    //Set dz
    dz = ui -> lineEdit_3 -> text().toDouble();
}

void Widget::on_pushButton_2_clicked()
{
    Algorithms a;
    std::vector<Edge> dt;
    bool slope = FALSE;
    bool aspect = FALSE;
    bool colHyps = FALSE;

    //DT needs to be created
    if(dt.size() == 0)
    {
        //Get points
        std::vector<QPoint3D> points = ui->Canvas->getPoints();

        //Create DT
        dt = a.DT(points);

        //Set DT
        ui->Canvas->setDT(dt);
    }

    //Analyze DTM
    std::vector<Triangle> dtm = a.analyzeDTM(dt);

    //Set analysis
    ui->Canvas->setDTM(dtm);

    //Set method
    if (ui->comboBox->currentIndex()==0) {
        slope = TRUE;
        aspect = FALSE;
        colHyps = FALSE;
    }
    else if (ui->comboBox->currentIndex()==1) {

```

```

        slope = FALSE;
        aspect = TRUE;
        colHyps = FALSE;
    }
    else if (ui->comboBox->currentIndex()==2) {
        slope = FALSE;
        aspect = FALSE;
        colHyps = TRUE;
    }

    ui->Canvas->setAspect(aspect);
    ui->Canvas->setSlope(slope);
    ui->Canvas->setColHyps(colHyps);

    //Repaint
    repaint();
}

void Widget::on_pushButton_3_clicked()
{
    //Clear DTM analysis
    std::vector<Edge> &dt = ui->Canvas->getDT();
    std::vector<Triangle> &dtm = ui->Canvas->getDTM();

    //Clear DTM
    dt.clear();
    dtm.clear();

    //Repaint
    repaint();
}

void Widget::on_pushButton_4_clicked()
{
    //Get all
    std::vector<QPoint3D> &points = ui->Canvas->getPoints();
    std::vector<Edge> &dt = ui->Canvas->getDT();
    std::vector<Edge> &contours = ui->Canvas->getContours();
    std::vector<Triangle> &dtm = ui->Canvas->getDTM();

    //Clear all
    points.clear();
    dt.clear();
    contours.clear();
    dtm.clear();

    //Repaint
    repaint();
}

void Widget::on_pushButton_clicked()
{
    //ui->Canvas->clearDT();

    std::vector<QPoint3D> points;

    QSizeF canvas_size = ui->Canvas->size();

    QString path = QFileDialog::getOpenFileName(
        this,
        tr("Select file"),
        "../",
        "Text file (*.txt);;All files (*.*)");

    std::string path_utf8 = path.toUtf8().constData();

    QString msg;

    Draw drw;
    drw.loadFile(path_utf8, points, canvas_size, z_min, z_max);

    ui->Canvas->setPoints(points);

    ui->Canvas->repaint();
}

void Widget::on_pushButton_5_clicked()

```

```

{
    Algorithms alg;

    std::vector<QPoint3D> &points = ui->Canvas->getPoints();
    std::vector<Edge> &dt = ui->Canvas->getDT();
    std::vector<Edge> &contours = ui->Canvas->getContours();
    std::vector<Triangle> &dtm = ui->Canvas->getDTM();

    points.clear();
    dt.clear();
    contours.clear();
    dtm.clear();

    //Get window size
    int width = ui -> Canvas -> size().width();
    int height = ui -> Canvas -> size().height();

    //Get terrain feature
    int tf = ui->comboBox_2->currentIndex();

    std::vector<QPoint3D> tf_points = alg.generateTF(width,height,tf);

    ui -> Canvas ->setPoints(tf_points);

    repaint();
}

```



## 8. Zhodnocení dosažených výsledků a závěr

Byla vytvořena funkční aplikace, která umí zadávat body jak ručně, tak importovat z textového souboru, či generovat podle zvoleného terénního tvaru – hřbet, údolí, spočinek, kupa. Nad těmito body umí vytvořit Delauneyovu triangulaci – trojúhelníky, nad kterými můžeme provádět analýzu sklonu, orientace terénu a zobrazit barevnou hypsometrii. Aplikace také vykreslí vrstevnice spolu s vyznačením hlavní vrstevnice a popisem vrstevnic. V rámci bonusových úloh aplikace generuje terénní tvary, vytváří barevnou hypsometrii a popisuje vrstevnice.

### 8.1 Možné vylepšení

Do aplikace by bylo vhodné implementovat algoritmus, který by popisoval vrstevnice v souladu s kartografickými zásadami. V aktuální aplikaci je popis náhodný a často je digitální model nečitelný – když je nastavený malý rozestup vrstevnic. Taktéž by bylo vhodné do aplikace naprogramovat algoritmus na vybírání oblasti, kterou chceme analyzovat – polygonem nebo obdélníkem. Aplikace také neumí pracovat s nekonvexními oblastmi.

## 9. Seznam obrázků

Obrázek 1 Sklon terénu .....	6
Obrázek 2 Orientace terénu .....	6
Obrázek 3 Formát vstupních dat .....	8
Obrázek 4 Grafické okno aplikace .....	9
Obrázek 5 Generování kupy .....	9
Obrázek 6 Generování údolí .....	10
Obrázek 7 Generování spočinku .....	10
Obrázek 8 Generování hřbetu .....	11
Obrázek 9 Import textového souboru se souřadnicemi bodů .....	11
Obrázek 10 Importované body z měření .....	12
Obrázek 11 Vygenerování vrstevnic na importovaných bodech .....	12
Obrázek 12 Sklon terénu .....	13
Obrázek 13 Orientace terénu .....	13
Obrázek 14 Barevná hypsometrie .....	14