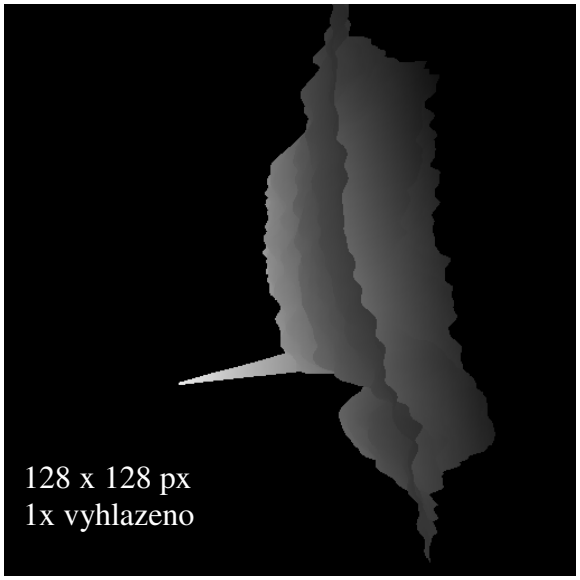
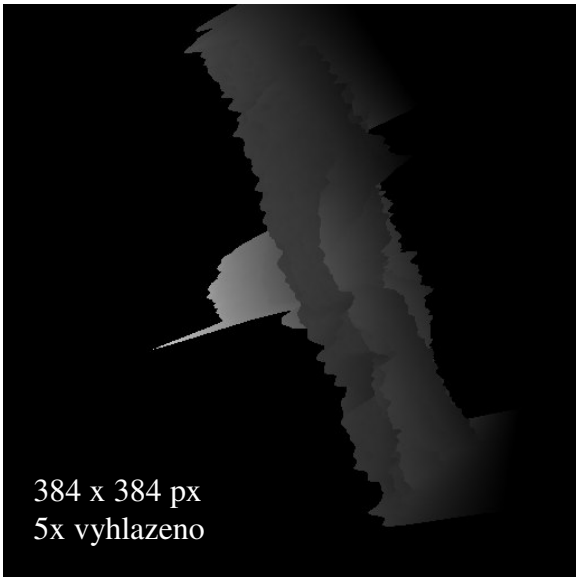
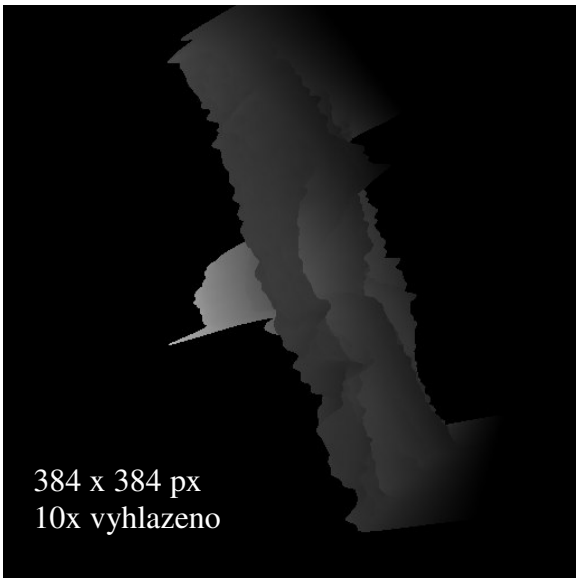
PŘÍLOHY

SEZNAM PŘÍLOH:

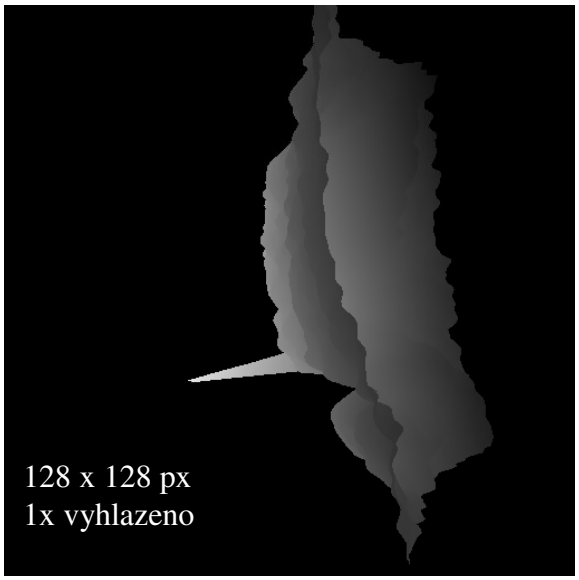
Vyhlazení povrchu pomocí zprůměrování čtyř okolních pixelů

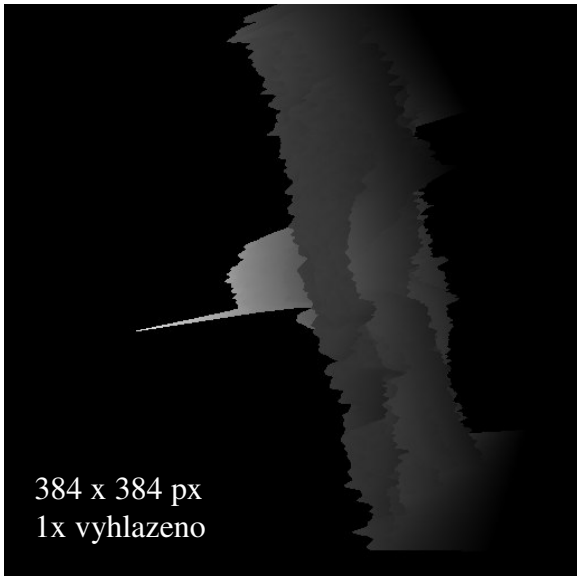


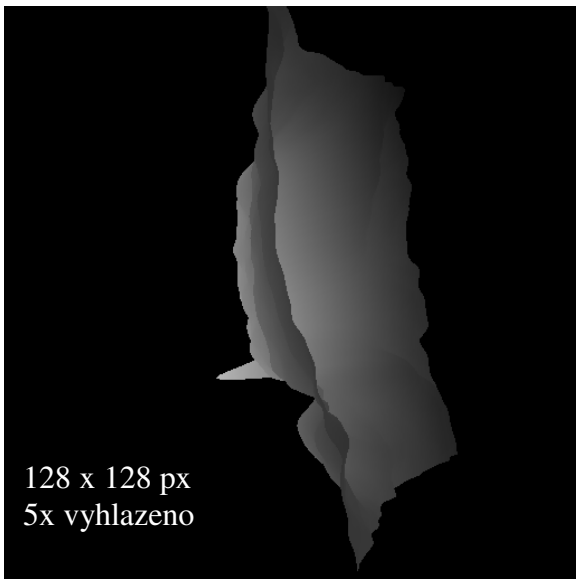


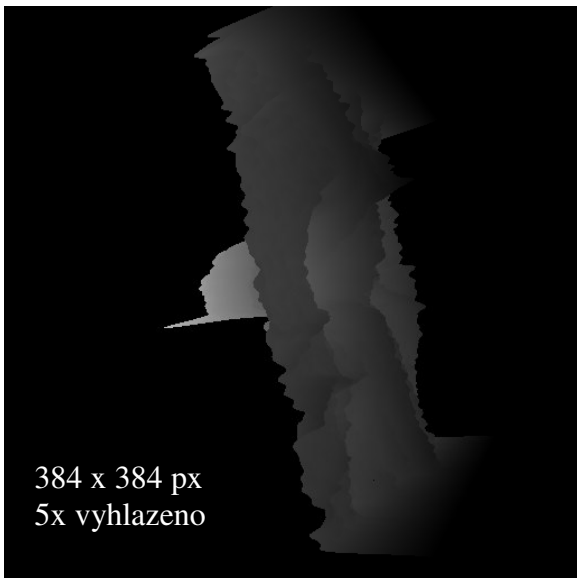


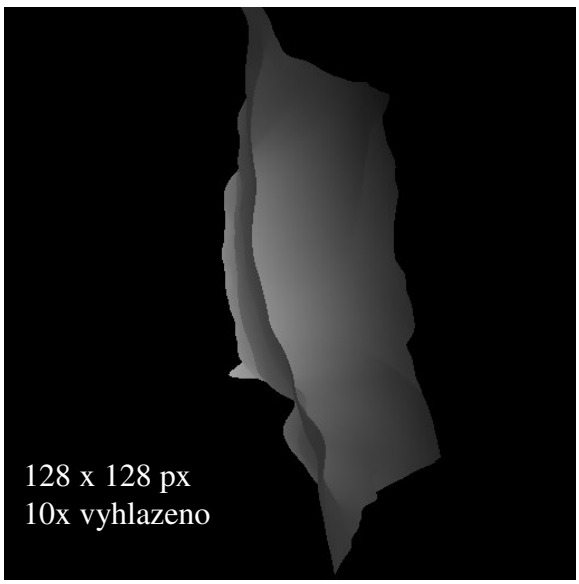
Vyhlazení povrchu pomocí zprůměrování osmi okolních pixelů

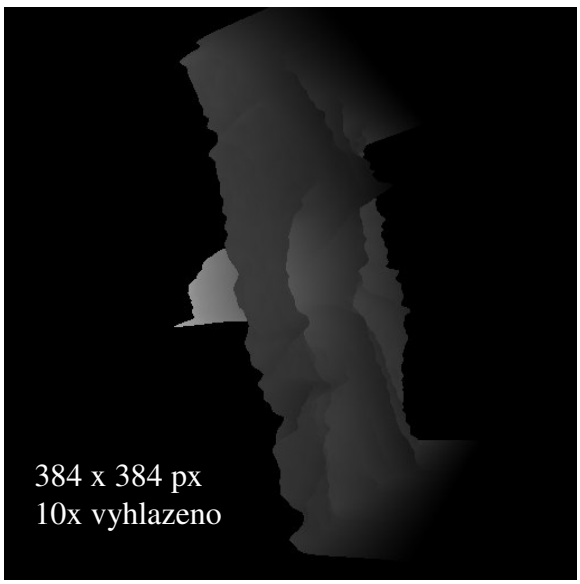




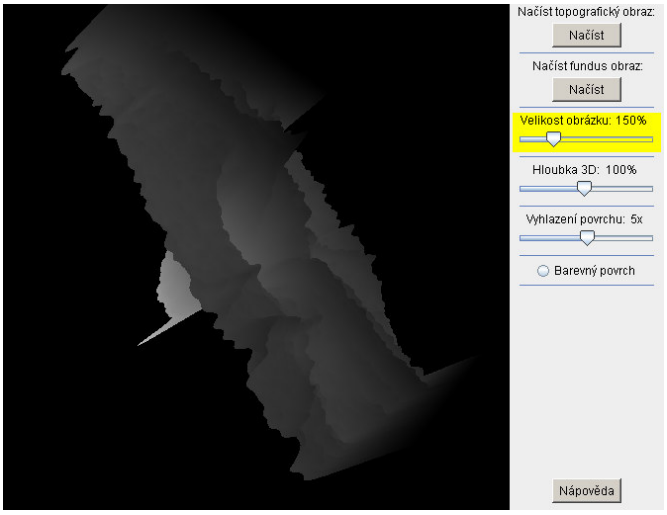


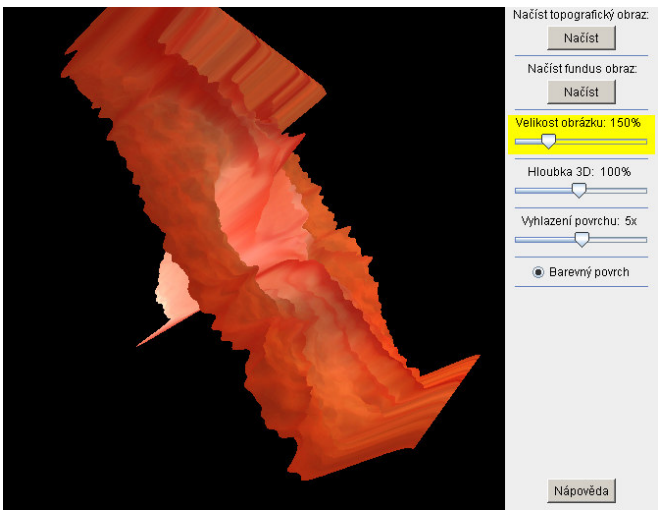


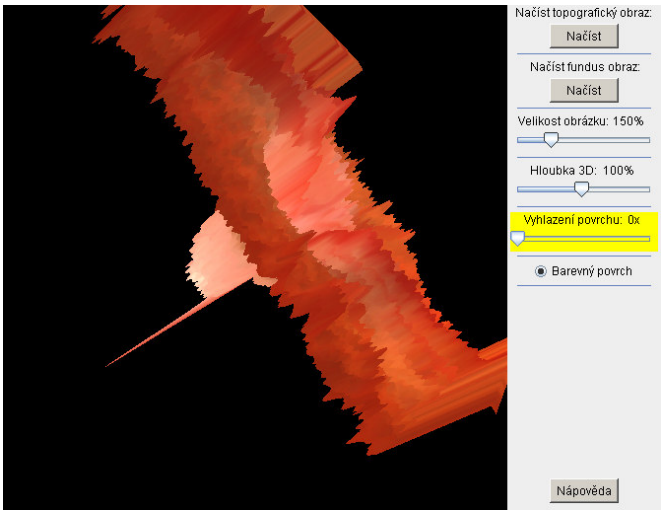


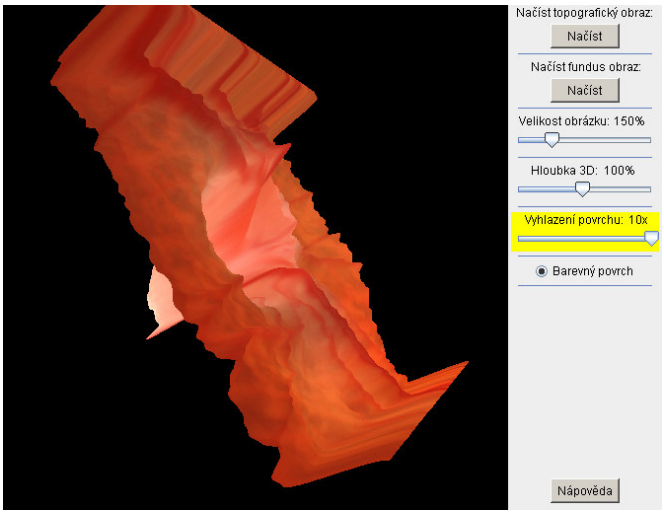


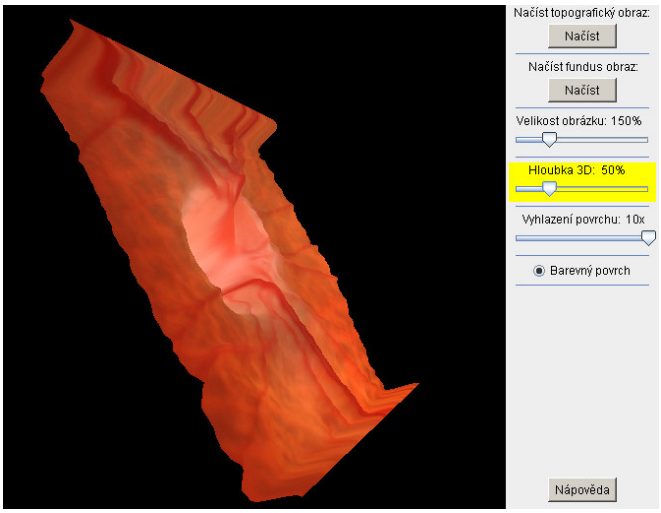
Ukázky výstupu z appletu OptickyDisk1

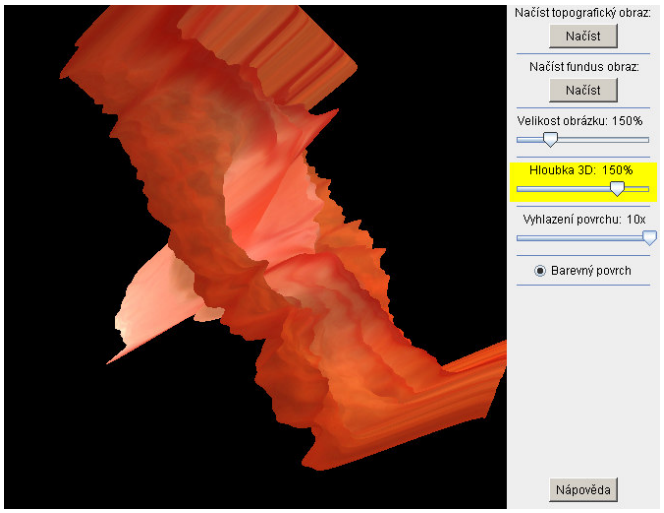




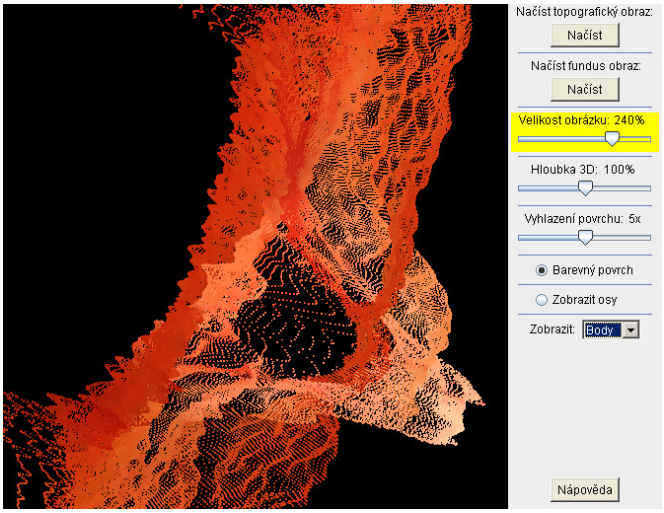


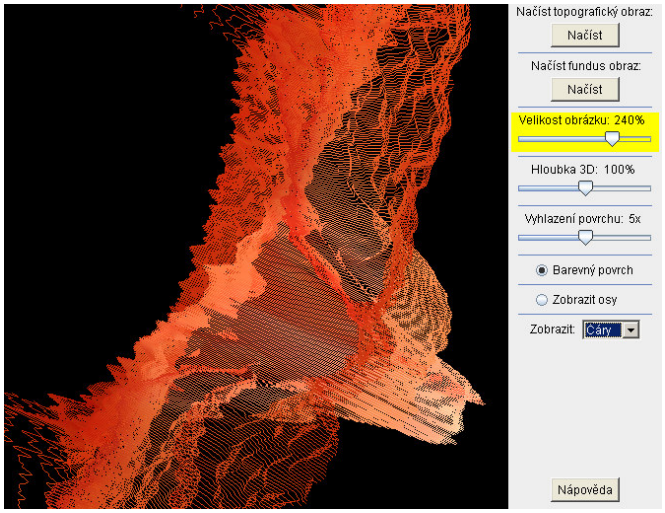


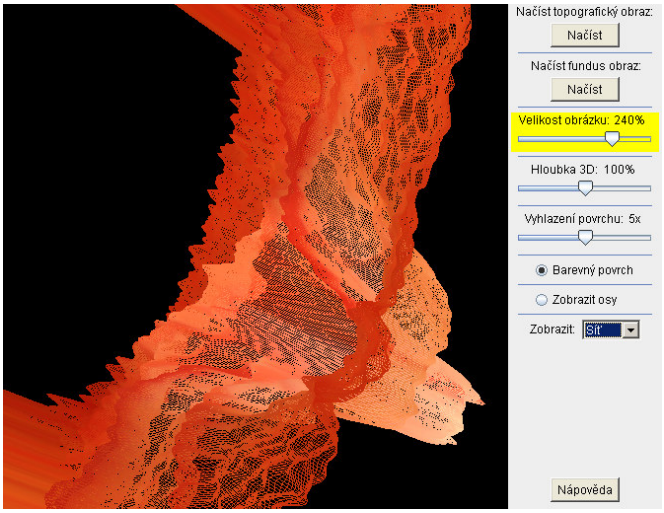


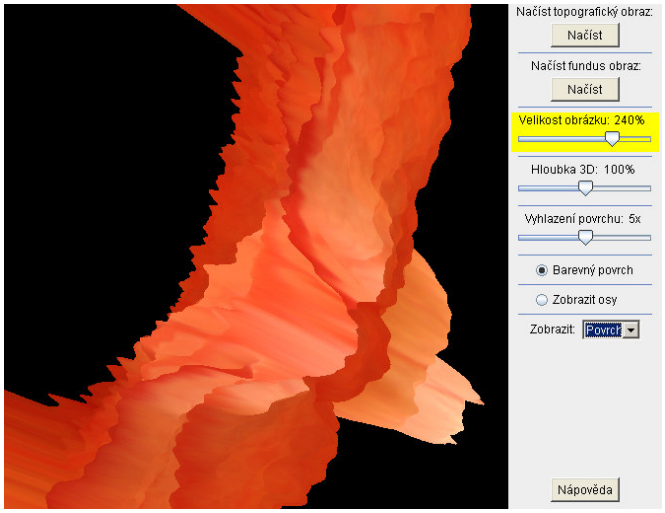


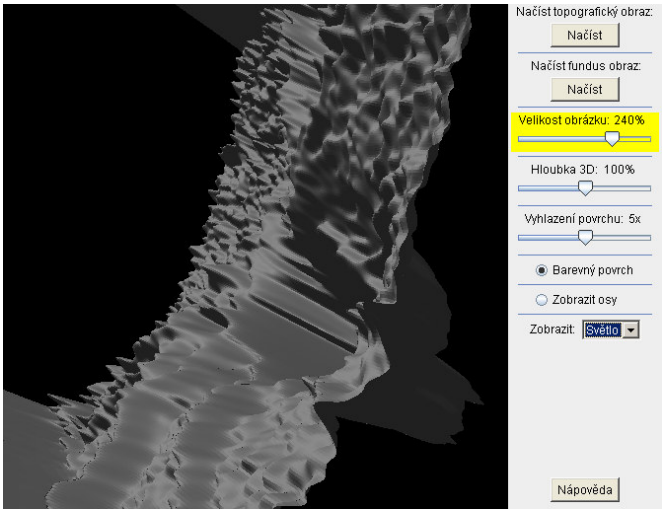
Ukázky výstupu z appletu OptickyDisk2

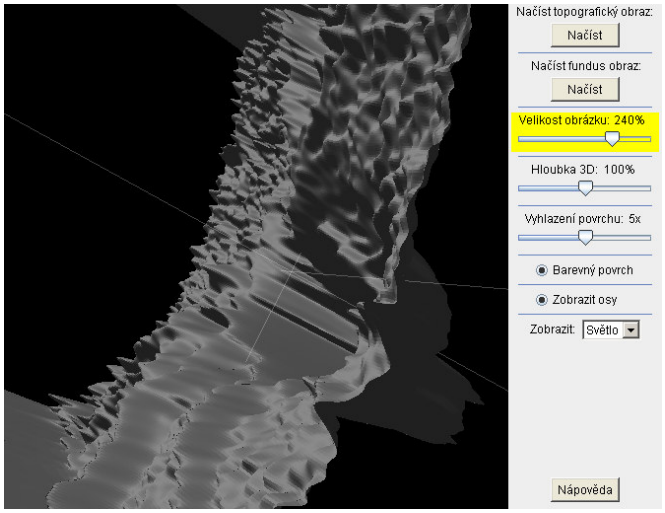










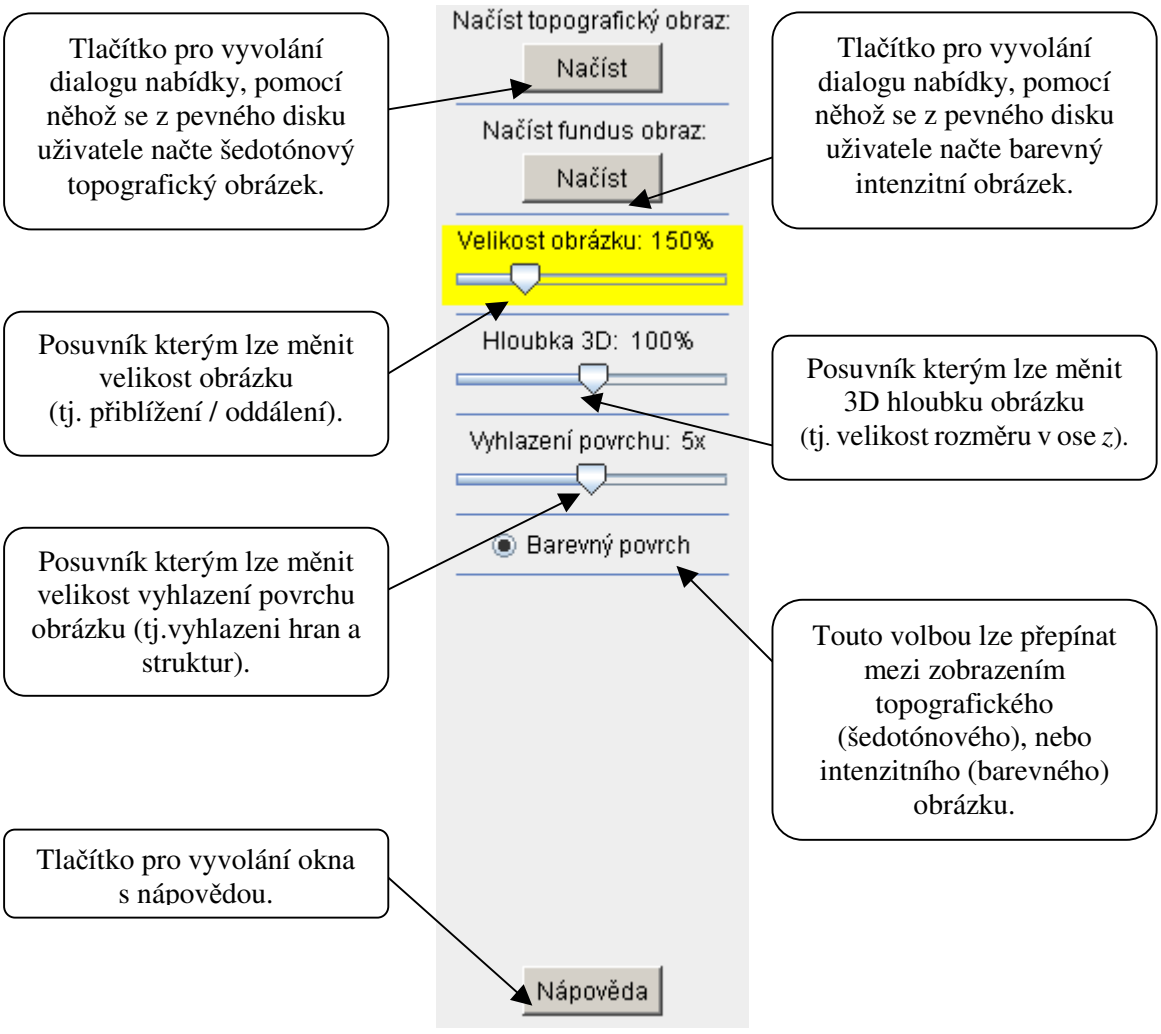


UŽIVATELSKÝ MANUÁL

Tento program byl vytvořen v jazyce Java, pomocí nadstavby Java3D. Před spuštěním je tedy nutné mít nainstalovaný program Java JRE (Java Runtime Environment), nebo Java JDK (Java Development Kit), jehož součástí je i JRE. Dále je nutné mít nainstalovanou nadstavbu Java3D.

Program je navržen jako applet, což znamená že je spouštěn v okně webového prohlížeče. Po spuštění souboru OptickyDisk1.html mohou některé webové prohlížeče z důvodu zabezpečení zakázat spuštění aktivního obsahu stránky. V tom případě je nutné ručně povolit a spustit zablokovaný aktivní obsah dle instrukcí ve webovém prohlížeči. Po povolení a spuštění aktivního obsahu se objeví okno s certifikátem zabezpečení, který je nutné potvrdit kliknutím na tlačítko „Run“. Pak by měl již být applet zobrazen v okně.

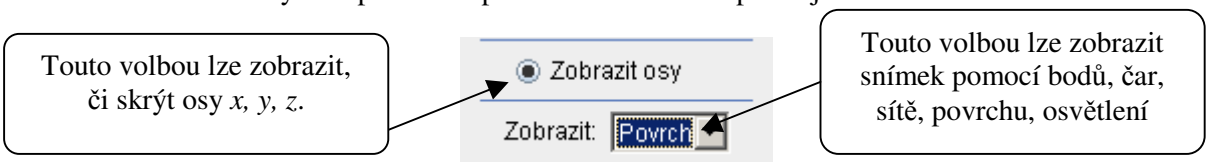
Applet je vytvořen tak, aby bylo možné ovládat jej především pomocí myši. Pokud by bylo ovládání pomocí myši z jakéhokoliv důvodu nedostupné, je možné ovládat některé funkce pomocí klávesnice. Popis funkcí a možnosti jejich ovládání jsou v tabulce *Seznam dostupných funkcí*. Applet je rozdělen na dvě části. První částí je 3D prostředí pro zobrazování, což je černá čtvercová plocha vlevo. Druhou částí je ovládací panel, který je umístěn vpravo vedle 3D pozadí. Na panelu jsou umístěny ovládací prvky, pomocí nichž je možné applet ovládat. Tlačítko „*Načíst topografický obraz*“ slouží pro načtení šedotónových topografických snímků. Tlačítko „*Načíst fundus obraz*“ slouží pro načtení barevných intenzitních snímků. Posuvníkem „*Velikost obrázku*“ lze měnit měřítko zobrazení v rozsahu 100 až 300%. Posuvníkem „*Hloubka 3D*“ lze měnit velikost rozměru v ose *z* v rozsahu 0 až 200%. Posuvníkem „*Vyhlazení povrchu*“ lze aplikovat algoritmus pro vyhlazení hran a struktur povrchu obrázku. Pro vyšší účinnost vyhlazení lze aplikovat algoritmus vícenásobně. Na displeji vedle popisu posuvníku je zobrazena hodnota, kolikrát byl algoritmus aplikován. Mezi posuvníky je možné přepínat pravým tlačítkem myši. Posuvník který pak lze ovládat kolečkem myši je obarven žlutě. Pomocí volby „*Barevný povrch*“ lze přepínat mezi zobrazením obrázku šedotónového a barevného. Popis těchto ovládacích prvků je uveden na obrázku:



Seznam dostupných funkcí (možnosti ovládání)

Funkce	Myš	Klávesnice	Panel
Načtení šedotónového topografického snímku	Kliknutí levým tlačítkem myši na tlačítko „*Načtení*“	Space (tlačítko musí být aktivní)	Kliknutí levým tlačítkem myši na tlačítko „*Načtení*“
Načtení barevného intenzitního snímku	Kliknutí levým tlačítkem myši na tlačítko „*Načtení*“	Space (tlačítko musí být aktivní)	Kliknutí levým tlačítkem myši na tlačítko „*Načtení*“
Otáčení obrázku	Stisk a držení levého tlačítka myši	←, →, ↑, ↓	nelze
Posun obrázku po ploše	Stisk a držení pravého tlačítka myši	PgDn, PgUp, Home, End	nelze
Přepnutí zobrazení šedotónový / barevný	Ovládat lze vždy levým tlačítkem myši	nelze	Ovládat lze vždy levým tlačítkem myši
Změna hodnoty na posuvníku	Kolečkem myši (posuvník musí být aktivní)	Q, W	Posun levým tlačítkem myši je možný vždy
Přepínání posuvníků mezi sebou	Kliknutí pravým tlačítkem myši na 3D plochu	Space (mezerník)	Ovládat lze vždy levým tlačítkem myši
Přepínání komponentů v okně mezi sebou	Ovládat lze vždy levým tlačítkem myši	Tab	Ovládat lze vždy levým tlačítkem myši

V rozšířené verzi, tedy v appletu OptickyDisk2, jsou navíc dostupné další funkce. Pomocí kliknutí na volbu „*Zobrazit osy*“, nebo na její popis, lze zobrazit nebo skrýt osy *x, y,* a *z* ve 3D prostředí. Pomocí předvolby „*Zobrazit*“ lze v prostoru vykreslit 3D model obrázku pomocí jednotlivých bodů, čar, sítě, povrchu, nebo osvětlení povrchu. Ovládání těchto komponentů je možné pouze pomocí kliknutí levého tlačítka myši na panelu. Popis těchto ovládacích prvků je uveden na obrázku:



<u>DŮLEŽITÉ UPOZORNĚNÍ</u> !

- Pokud je tlačítkem „*Načíst topografický obraz*“ načten barevný obrázek, nebudou barvy zobrazeny korektně a ani nebudou správně pracovat funkce pro zobrazování ve 3D.
- Tlačítkem „*Načíst fundus obraz*“ lze bezchybně načíst i šedotónový snímek, ale při přepnutí volby zobrazení pak nebude možné zobrazit povrch barevně.
- Pokud je tlačítkem „*Načíst fundus obraz*“ načten pouze barevný obrázek, nejsou dostupné funkce pro zobrazování ve 3D, protože informace o rozměru v ose *z* jsou zaneseny pouze v topografickém snímku.

Okno nápovědy pro applet OptickyDisk1

Okno nápovědy rozšířeného appletu OptickyDisk2 vychází z výše uvedeného obrázku. Protože však do appletu přibyly další funkce, je nápověda rozšířena o dvě položky v prostřední části „*Ovládací prvky na bočním panelu*“, které stručně tyto funkce popisují:

7. Zobrazit osy: je-li načten jakýkoliv obrázek, lze zobrazit 3D osy v prostoru.
8. Zobrazit: je-li načten jakýkoliv obrázek, lze jej vykreslit dle zvolené předvolby.

TECHNICKÁ DOKUMENTACE

OBSAH

1 Načtení 2D obrázku do 3D prostředí

1.1 Vytvoření třídy appletu

Protože program má pracovat jako applet, musí být potomkem rodičovské třídy **java.applet.Applet**. Toto odvození se provede uvedením rodičovské třídy v hlavičce odvozené třídy za jejím názvem a klíčovým slovem **extends**:

```
public class NacteniObrazku extends java.applet.Applet
```

První slovo **public** je tzv. modifikátor přístupu, který určuje jaké další třídy budou mít přístup k členské datové složce. Modifikátor **public** určuje, že složka je přístupná ze všech tříd a je tedy veřejná. Druhé slovo **class** deklaruje třídy (lze také deklarovat metody, proměnné atd.). Dále je třeba implementovat rozhraní třídy pro zpracování událostí **java.awt.event.ActionListener**.

```
public class NacteniObrazku extends java.applet.Applet implements ActionListener
```

Při použití třídy **ActionListener** je povinná implementace abstraktní metody **actionPerformed** z třídy **ActionEvent**, pomocí které se bude provádět obsluha událostí. Pokud by tato metoda nebyla implementována, v programu by byla hlášena chyba a nebylo by možné jej spustit.

```
public void actionPerformed(ActionEvent e)
```

Musí se také importovat balíčky, ve kterých jsou třídy obsaženy. Tento import se provede na úplném začátku programu zápisem za klíčové slovo **import**:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

Protože obě třídy jsou ve stejném balíčku **java.awt.event**, je možné zápis zjednodušit:

```
import java.awt.event.*;
```

Použitím zástupného znaku * se načtou všechny třídy, které jsou v balíčku obsaženy. Také třída **Applet** má povinné metody (viz. DP část. 3.1) a to **init()**, **start()**, **stop()** a **destroy()**. Po implementaci těchto metod bude tělo programu vypadat takto:

```
import java.awt.event.*;
public class NacteniObrazku extends java.applet.Applet implements ActionListener {
    public void init() { }
    public void start() { }
    public void stop() { }
    public void destroy() { }
    public void actionPerformed(ActionEvent e) { }
}
```

1.2 Volání metody init

Po spuštění appletu je nejdříve volána metoda **init()**. V té je vytvořena obsluha výjimky (exception handler), která je automaticky vyvolána v případě chyby při inicializaci GUI:

```
public void init() {
  try {
    java.awt.EventQueue.invokeAndWait(new Runnable() {
      public void run() {
        inicializaceKomponentu();
      }
    });
  } catch (Exception ex) {
  JOptionPane.showMessageDialog(null,"Chyba při vytváření GUI !",
                                "Pozor !", JOptionPane.WARNING_MESSAGE);
  }
}
```

V bloku **try{}** je umístěna část kódu, která může způsobit výjimku. Metodou **run()** se spouští podproces v objektu typu **Runnable()**. Spuštěný podproces zajistí inicializaci GUI voláním metody **inicializaceKomponentu()**. V bloku **catch{}** je obsluha výjimky. Ošetření výjimky je provedeno jako upozornění pomocí okna **JOptionPane**. V případě chyby při vytváření GUI se zobrazí varování (viz. DP obr.4.1). Okno je součást knihovny **swing** a je třeba načíst příslušnou knihovnu:

```
import javax.swing.JOptionPane;
```

1.3 Vytvoření grafického uživatelského rozhraní

V metodě **inicializaceKomponentu()** jsou inicializovány komponenty GUI. Nejdříve je prostředí pro 3D zobrazení pomocí třídy **SimpleUniverse**, což je podtřída třídy **VirtualUniverse**. Načtení balíčku s touto třídou se provede tímto zápisem:

```
import com.sun.j3d.utils.universe.SimpleUniverse;
```

Protože později bude objekt třídy **SimpleUniverse** s názvem **zobrazeni** využíván v dalších metodách programu, musí být deklarován na začátku programu, zápisem mimo metodu:

```
private SimpleUniverse zobrazeni = null;
```

Modifikátor přístupu **private** zde značí, že vytvořený objekt bude přístupný pouze v rámci své vlastní třídy. Za ním je vytvořen objekt třídy **SimpleUniverse** s názvem **zobrazeni** Klíčové slovo **null** ze operátorem přiřazení „**=**“ znamená, že objekt není prozatím inicializován. Pomocí tohoto objektu se pak bude zobrazovat obrázek na 3D pozadí **Canvas3D**:

```
Canvas3D pozadi = new Canvas3D (SimpleUniverse.getPreferredConfiguration());
zobrazeni = new SimpleUniverse(platno);
zobrazeni.getViewingPlatform().setNominalViewingTransform();
```

V prvním řádku se načítá trojrozměrné pozadí **Canvas3D** konfigurované pro použití spolu s objekty třídy **SimpleUniverse**. Klíčové slovo **new** je operátor jazyka Java pro vytvoření instance třídy (zde objektu **pozadi**). Za operátorem **new** následuje volání konstruktoru, který zajistí inicializaci nového objektu. Ve druhém řádku je načteno prostředí **SimpleUniverse** a ve třetím řádku je nastavena vzdálenost pozorovatele. Je třeba opět načíst balíček, který obsahuje třídu **Canvas3D**:

```
import javax.media.j3d.Canvas3D;
```

Nyní budou načteny ostatní grafické komponenty. Nejdříve **hlavniPanel**, na který se umístí nakonfigurované pozadí a **horniPanel** do kterého se umístí tlačítko pro otevření dialogu nabídky:

```
Panel hlavniPanel = new Panel();
Panel horniPanel = new Panel();
```

Na začátku programu se opět načte balíček, který obsahuje komponent třídy **awt.Panel**:

```
import java.awt.Panel;
```

Tlačítko a dialog nabídky musí být deklarovány na začátku programu:

```
private JFileChooser dialog = null;
private Button tlacitko = null;
```

I zde musí být načteny balíčky s příslušnými třídami:

```
import java.awt.Button;
import javax.swing.JFileChooser;
```

Ve vlastní metodě **inicializaceKomponentu()** jsou oba komponenty, tlačítko i dialog nabídky, inicializovány pomocí příkazů:

```
dialog = new JFileChooser();
tlacitko = new Button();
```

Aby byl na tlačítku napsaný nápis „Otevřít“ využije se konstruktor **setLabel()**:

```
tlacitko.setLabel("Otevřít");
```

Nakonec je třeba tlačítko i dialog nabídky přidat do metody obsluhy událostí:

```
dialog.addActionListener(this);
tlacitko.addActionListener(this);
```

Dále se pomocí konstruktoru **setBounds()** nastaví velikost a umístění grafických komponentů. První dvě číslice udávají umístění levého horního rohu a druhé dvě šířku a výšku komponentu:

```
horniPanel.setBounds(0, 0, 512, 30);
tlacitko.setBounds(4, 4, 56, 22);
pozadi.setBounds(0, 30, 512, 512);
```

Zarovnání komponentů v okně appletu se provede pomocí konstruktoru **setLayout()**:

```
setLayout(new BorderLayout());
hlavniPanel.setLayout(null);
horniPanel.setLayout(null);
add(hlavniPanel, BorderLayout.CENTER);
```

V prvním řádku zdrojového kódu je nastaveno roztažení komponentů na celé okno. Proto se používá **hlavniPanel**, kterým se ohraničí celé okno appletu. Zarovnání v komponentu **hlavniPanel** se ve druhém řádku nastaví jako „volný styl“ příkazem **null**. Až do takto upraveného panelu se vkládají další komponenty. Ve třetím řádku je také zarovnání v komponentu **horniPanel** nastaveno na „volný styl“. Kdyby se takto nenastavilo, ostatní komponenty které se do něho vloží (zde tlačítko) by se roztáhly přes celou plochu panelu. Ve čtvrtém řádku se příkazem **add** přidává **hlavniPanel** na střed okna. Musí být opět načten balíček s třídou pro zarovnání:

```
import java.awt.BorderLayout;
```

Posledním krokem je poskládání komponentů v okně. To se provede opět příkazem **add**:

```
horniPanel.add(tlacitko);
hlavniPanel.add(horniPanel);
hlavniPanel.add(pozadi);
```

V prvním řádku zdrojového kódu se do komponentu **horniPanel** přidává tlačítko, ve druhém se celý **horniPanel** i s tlačítkem přidává do komponentu **hlavniPanel**. Ve třetím řádku se do komponentu **hlavniPanel** přidá i 3D pozadí **Canvas3D**. Na jakých souřadnicích budou komponenty umístěny a jak budou velké, bylo již definováno konstruktorem **setBounds()**.

1.4 Obsluha událostí

Obsluha událostí vyvolaných grafickými komponenty je naprogramována a ošetřena v abstraktní metodě **public void actionPerformed**(ActionEvent e. Který komponent událost vyvolá se rozliší pomocí konstruktoru **getSource()**

```
if (e.getSource() == tlacitko)
    dialog.showOpenDialog(this);
```

V prvním řádku se standardním větvením programu zjišťuje, zda byla událost vyvolána grafickým tlačítkem. Pokud ano, tak je zobrazen dialog nabídky, k čemuž slouží kód na druhém řádku. Podobně se bude postupovat při ošetření událostí vyvolaných stiskem tlačítka na dialogu nabídky. V tomto případě je ovšem třeba ještě navíc rozlišit, které tlačítko na dialogu bylo stisknuto: zda tlačítko pro otevření souboru „Open“, nebo tlačítko pro zrušení akce „Cancel“:

```
else if (e.getSource() == dialog) {
    String prikaz = e.getActionCommand();
    if (prikaz.equals(„ApproveSelection“) {
        File soubor = dialog.getSelectedFile();
        nazevObrazku = soubor.toString();
        zobrazeni.addBranchGraph(vytvorScenu());
    }
}
```

V prvním řádku se opět standardním větvením programu zjišťuje, zda byla událost vyvolána dialogem nabídky. Pokud ano, pak se ve druhém řádku do lokální proměnné typu **String** s názvem **prikaz** načte řetězec odpovídající příkazu při stisku příslušného tlačítka. Ve třetím řádku se zjišťuje, zda je hodnota řetězce „ApproveSelection“, což znamená že bylo stisknuto tlačítko „Open“. Pokud ano, tak se ve čtvrtém řádku v dalším větvením programu, pomocí konstruktoru **getSelectedFile()**, načte cesta k vybranému obrázku do objektu **soubor**, který je typu **File**. Cesta k vybranému obrázku se pak na pátém řádku, pomocí konstruktoru **toString()**, převede na řetězec typu **String**. Ten je nazván **nazevObrazku** a je deklarován na začátku programu:

```
private String nazevObrazku = null;
```

V posledním, šestém řádku, se volá vlastní metoda typu **BranchGroup**, která je nazvána **vytvorScenu()** a ve které se již bude vykreslovat vlastní obrázek. Pomocí této metody pak bude celá scéna zobrazena v prostředí pro 3D zobrazení. Aby bylo celé volání této metody funkční je třeba nejdříve na začátku celého programu opět načíst příslušný balíček, ve kterém je obsažena tato třída:

```
import javax.media.j3d.BranchGroup;
```

Na rozdíl od předchozího případu se nepředpokládá, že by byla v programu použita ještě některá z dalších tříd, které jsou obsaženy v balíčku **java.io**. Takže je možné tento balíček načíst zápisem přímo do řádku programu:

```
java.io.File soubor = dialog.getSelectedFile();
```

1.5 Vytvoření scény

Vytvoření scény se provede metodou typu **BranchGroup**, která je nazvána **vytvorScenu()**.

```
private BranchGroup vytvorScenu() { }
```

V metodě je vytvořen objekt typu **BranchGroup**, který je nazván **scena**. Objekt je deklarován na začátku programu mimo metodu, protože pokud v něm bude již načten obrázek, tak bude třeba před načtením dalšího obrázku tento objekt rozložit příkazem **detach()**, jinak by se obrázky načítaly přes sebe stále do jedné scény. V programu totiž není možné použít název objektu před tím než je deklarován a to je právě důvod deklarace mimo metodu:

```
private BranchGroup scena = null;
```

Nejdříve je tedy nutno ošetřit rozkládání objektu **scena** pro případ načítání dalších obrázků. To se provede větvením programu:

```
if (scena != null)
    scena.detach();
```

Až pak se vytvoří nový objekt typu **BranchGroup** nazvaný **scena**, u kterého je třeba povolit rozklad. To se provádí konstruktorem **setCapability()** s hodnotou **ALLOW_DETACH**:

```
scena = new BranchGroup();
scena.setCapability(BranchGroup.ALLOW_DETACH);
```

Poté se vytvoří objekt typu **Shape3D** nazvaný **objekt**, pomocí něhož se vykreslí obrázek:

```
Shape3D objekt = new Shape3D();
objekt.setGeometry(vytvorGeometrii());
objekt.setAppearance(vytvorVzhled());
```

Ve druhém řádku se metodou **setGeometry()** volá vlastní metoda typu **Geometry**, nazvaná **vytvorGeometrii()**, kde bude definována geometrie objektu. Ve třetím řádku se metodou **setAppearance()** volá vlastní metoda typu **Appearance**, nazvaná **vytvorVzhled()**. V té je definován vzhled objektu. Opět je třeba načíst balíčky s příslušnými třídami:

```
import javax.media.j3d.Shape3D;
import javax.media.j3d.Appearance;
import javax.media.j3d.Geometry;
```

Objekt třídy **Shape3D** s názvem **objekt** se musí přidat od objektu třídy **BranchGroup** s názvem **scena** pomocí konstruktoru **addChild()**:

```
scena.addChild(objekt);
scena.compile();
return scena;
```

Ve druhém řádku se objekty ve scéně zkompilují a ve třetím řádku se nakonec návratovým příkazem **return** předává celá scéna do prostředí **SimpleUniverse** pro 3D zobrazení.

1.5.1 Vytvoření geometrie objektu

Zmíněnou vlastní metodu typu **Geometry** je třeba do těla programu opět dopsat. Metoda která je nazvána **vytvorGeometrii()** se zapíše tímto způsobem:

```
private Geometry vytvorGeometrii() { }
```

V metodě je vytvořen objekt třídy **GeometryArray** typu **QuadArray**, který je nazván **oblast**.

```
QuadArray oblast = new QuadArray(4, GeometryArray.COORDINATES |
                                    GeometryArray.TEXTURE_COORDINATE_2);
```

V konstruktoru **QuadArray** se v závorce zadá počet vrcholů objektu (zde 4), dále se povolí zadávání souřadnic objektu (**GeometryArray.COORDINATES**) a také zadávání souřadnic textury (příkaz **GeometryArray.TEXTURE_COORDINATE_2**). Číslo 2 u povolení souřadnic textury značí, že se bude jednat o 2D texturu (tj. obrázek v osách X a Y). Protože se povoluje více atributů, použije se pro jejich oddělení znak: | který zde zastupuje logický součin. Na začátek programu se opět musí načíst balíčky s příslušnými třídami:

```
import javax.media.j3d.GeometryArray;
import javax.media.j3d.QuadArray;
```

Pomocí konstruktoru **setCoordinate()** se zadají souřadnice vrcholů do objektu **oblast**:

```
oblast.setCoordinate(0, new Point3f(-0.5f,  0.5f, 0.0f));
oblast.setCoordinate(1, new Point3f(-0.5f, -0.5f, 0.0f));
oblast.setCoordinate(2, new Point3f( 0.5f, -0.5f, 0.0f));
oblast.setCoordinate(3, new Point3f( 0.5f,  0.5f, 0.0f));
```

V konstruktoru se musí nejdříve zadat index bodu a až pak jsou zadány souřadnice. Zde je pro zadání souřadnic použita třída **Point3f**, ve které se zadávají souřadnice pro osy X, Y a Z v číselném formátu **float** – což je nutné vyjádřit písmenem „f“ za číselnou hodnotou. Třídu **Point3f** je také nutné načíst na začátku programu pomocí příslušného balíčku:

```
import javax.vecmath.Point3f;
```

Obdobným způsobem se zadávají souřadnice pro texturu. Musí se použít konstruktor **setTextureCoordinate()** a souřadnice se zadávají pouze pro dva rozměry (osu X a osu Y) pomocí třídy **TexCoord2f**:

```
oblast.setTextureCoordinate(0, 0, new TexCoord2f(0.0f, 1.0f));
oblast.setTextureCoordinate(0, 1, new TexCoord2f(0.0f, 0.0f));
oblast.setTextureCoordinate(0, 2, new TexCoord2f(1.0f, 0.0f));
oblast.setTextureCoordinate(0, 3, new TexCoord2f(1.0f, 1.0f));
```

Třídu **TexCoord2f** je i zde opět nutné načíst na začátku programu pomocí příslušného balíčku:

```
import javax.vecmath.TexCoord2f;
```

Nakonec se objekt třídy **QuadArray** s názvem **oblast**, ve kterém je nastavena geometrie, odešle zpět pro vykreslení do objektu třídy Shape3D **objekt**, pomocí příkazu **return**:

```
return oblast;
```

1.5.2 Vytvoření vzhledu objektu

Zmíněnou vlastní metodu typu **Appearance** je třeba do těla programu opět dopsat. Metoda která je nazvána **vytvorVzhled()** se zapíše tímto způsobem:

```
private Appearance vytvorVzhled() { }
```

V metodě je vytvořen objekt třídy **Appearance**, který je nazván **vzhled**. Vytvoření se provede tímto zápisem do těla programu:

```
Appearance vzhled = new Appearance();
```

Samotné načtení obrázku jako textury, se pak provede pomocí objektu třídy **TextureLoader**:

```
TextureLoader zavadec = new TextureLoader(nazevObrazku, this);
ImageComponent2D obraz = zavadec.getImage();
Texture2D textura = new Texture2D(Texture2D.BASE_LEVEL,
                    Texture2D.RGBA, obraz.getWidth(), obraz.getHeight());
textura.setImage(0, obraz);
```

V prvním řádku se vytvoří nový objekt třídy **TextureLoader** s názvem **zavadec**, pomocí něhož se načte obrázek podle umístění na disku. Ve druhém řádku je obrázek pomocí metody **getImage()** převeden na objekt třídy **ImageComponent2D**, který je nazván **obraz**. Ve třetím řádku je vytvořen nový objekt třídy Texture2D, nazvaný **textura**. Tento objekt bude pracovat se standardní texturou v barevném formátu RGB, což je nastaveno pomocí příkazů **Texture2D.BASE_LEVEL** a **Texture2D.RGBA**. Dále má definované rozměry (tj. výšku a šířku) podle rozměrů načítaného obrázku, což je určeno pomocí metod **getWidth()** a **getHeight()**. V posledním řádku se pomocí metody **setImage()** objekt **obraz** načte do objektu **textura**. Je opět nutné načíst balíčky s příslušnými třídami:

```
import com.sun.j3d.utils.image.TextureLoader;
import javax.media.j3d.ImageComponent2D;
import javax.media.j3d.Texture2D;
```

Samotné mapování textury se provede pomocí metody **setTexture()**:

```
vzhled.setTexture(textura);
```

Nakonec se objekt třídy **Appearance** s názvem **vzhled**, ve kterém je určeno nastavení vzhledu, odešle zpět pro vykreslení do objektu třídy Shape3D **objekt**, pomocí příkazu **return**:

```
return vzhled;
```

2 Vykresleni jednotlivých pixelů

Metody **init()**, **inicializaceKomponentu()** a **vytvorScenu()** zůstanou beze změn. Musí se však upravit ostatní metody a kvůli výpočtům spojeným s použitím třídy **PixelGrabber** je třeba vytvořit navíc další metodu, kde budou prováděny výpočty.

2.1 Obsluha událostí

V metodě pro obsluhu událostí **actionPerformed()** nastane jediná změna. Do metody je přidán další řádek s příkazem, který volá vlastní metodu pro výpočet parametrů jednotlivých pixelů. Tato nová metoda je nazvána **vypocty()**:

```
public void actionPerformed(ActionEvent e)
    if (e.getSource() == tlacitko)
        dialog.showOpenDialog(this);
    else if (e.getSource() == dialog) {
        String prikaz = e.getActionCommand();
        if (prikaz.equals(„ApproveSelection“) {
            File soubor = dialog.getSelectedFile();
            nazevObrazku = soubor.toString();
            vypocty();
            zobrazeni.addBranchGraph(vytvorScenu());
        }
    }

}
```

2.2 Čtení a normalizace hodnot jednotlivých pixelů

V metodě **vypocty()** se provádí čtení pixelů obrázku a převod na hodnoty vhodné pro kreslení. Aby bylo možné použít konstruktor třídy **PixelGrabber()**, je třeba znát některé další parametry. Jedná se v první řadě o šířku a výšku obrázku, a také celkový počet pixelů v obrázku. Tyto parametry se zjistí pomocí již zmíněné třídy **TextureLoader**:

```
TextureLoader loader = new TextureLoader(nazevObrazku, this);
ImageComponent2D obraz = loader.getImage();
sirkaObrazku = obraz.getWidth();
vyskaObrazku = obraz.getHeight();
pocetPixelu = sirkaObrazku * vyskaObrazku;
```

Proměnné **sirkaObrazku**, **sirkaObrazku** a **pocetPixelu** jsou deklarovány na začátku programu:

```
private int sirkaObrazku = 0;
private int vyskaObrazku = 0;
private int pocetPixelu = 0;
```

Je třeba také vytvořit jednorozměrné číselné pole, do kterého budou uloženy informace o všech pixelech, které budou získány pomocí třídy **PixelGrabber**:

```
int[] pixely = new int[pocetPixelu];
```

Nakonec je třeba načíst z disku obrázek, který bude třídou **PixelGrabber** zpracováván. To lze povést pomocí třídy **URL**, kde se musí ošetřit případné vyvolání výjimky při načítání:

```
try {
  java.net.URL umisteni = new java.net.URL("file:/"+nazevObrazku);
  obrazek = this.getImage(umisteni);
}
catch (Exception ex) {
  JOptionPane.showMessageDialog(null, "Chyba při otvírání souboru !",
                                "Pozor !", JOptionPane.WARNING_MESSAGE);
}
```

Kvůli ošetření výjimky je nutné deklarovat jako datovou také proměnnou **obrazek**:

```
private Image obrazek = null;
```

Ošetření výjimky je opět provedeno pouze jako upozornění pomocí okna **JOptionPane**, ve kterém se v případě selhání při načtení zobrazí varování o chybě při otevírání souboru. Toto okno je součástí knihovny **swing**, která je již načtena pomocí zástupného znaku:

```
import javax.swing.*;
```

Nyní je již možné použít konstruktor třídy **PixelGrabber**:

```
PixelGrabber nacteni = new PixelGrabber(obrazek, 0, 0,
                       sirkaObrazku, vyskaObrazku, pixely, 0, sirkaObrazku);
```

Do závorky konstruktoru třídy **PixelGrabber** je nejdříve zapsán objekt typu **Image**, ze kterého je provedeno čtení pixelů. Dále jsou zapsány souřadnice oblasti ze které budou pixely čteny. Dalším parametrem je pole, kam budou informace zapsány. Poslední dva parametry určují délku jednoho čteného řádku. Systém čtení a indexování pixelů je v DP tab.4.1.

2.2.1 Normalizace odstínu pixelů

Zápis informací o pixelech do číselného pole se provede pomocí metody **grabPixels()**. Opět musí být ošetřeno případné vyvolání výjimky, což je řešeno jako v předchozím případě:

```
try {
   nacteni.grabPixels();
   }
catch (Exception ex){
JOptionPane.showMessageDialog(null, "Chyba při čtení dat obrázku !",
                              "Pozor !", JOptionPane.WARNING_MESSAGE);
   }
```

V poli **pixely** jsou zapsány informace o odstínu jednotlivých pixelů v obrázku. Nejdříve je tedy deklarováno jako datová proměnná jednorozměrné číselné pole typu **float**, do něhož budou přepočteny hodnoty z pole **pixely** které naplnil **PixelGrabber**:

```
private float[] normPixely = null;
```

Pole je nazváno **normPixely**, protože bude provedena normalizace hodnot, tak aby byly v rozmezí 0 až 1. Hodnota 0 bude odpovídat odstínu černé barvy a hodnota 1 odstínu bílé barvy. V metodě **vypocty()** je třeba nastavit velikost pole **normPixely**. Toto pole musí mít stejnou velikost jako pole **pixely**, aby do něho mohly být načteny normalizované hodnoty všech pixelů v obrázku:

```
normPixely = new float[pocetPixelu];
```

Vlastní normalizace hodnot odstínu jednotlivých pixelů se provede pomocí smyčky:

```
for (int i = 0; i < pocetPixelu; i++)
     normPixely[i] = ((float)pixely[i] / 16777216.0f) + 1.0f;
```

V prvním řádku je nastavena standardně funkce smyčky. Normalizace hodnot odstínu pixelů je naprogramována ve druhém řádku. Zde je každý prvek pole **pixely**, který je typu **integer**, převeden na typ **float**, což je provedeno zápisem **(float)** před tento prvek. Hodnota každého prvku je posléze vydělena maximem.

2.2.2 Vytvoření vzhledu objektu

Podobně jako byl normalizován odstín barvy, je třeba normalizovat také rozměry pro vykreslení. Aby bylo možné vykreslit pixely pomocí třídy **Point3f** od souřadnic **(0.0f, 0.0f, 0.0f)** do **(1.0f, 1.0f, 1.0f)**, je nutné normalizovat hodnoty šířky a výšky obrázku:

```
normSirka = 1.0f / sirkaObrazku;
normVyska = -1.0f / vyskaObrazku;
```

Proměnné budou deklarovány jako datové proměnné:

```
private float normSirka = 0;
private float normVyska = 0;
```

Normalizace spočívá v převrácení hodnoty šířky (resp. výšky) obrázku, což zajistí že obrázky o různých rozměrech jsou zobrazeny ve stejné velikosti na stejné ploše 3D oblasti. Záporná hodnota proměnné **normVyska** je zde z důvodu dodržení pravidla, že se posun děje po ose Y v záporném směru. Dále se provede v metodě **vypocty()** výpočet optimální velikosti zobrazovaného bodu.

Proměnná je nazvána **meritko** a vychází z předpokladu, že se bude zpracovávat čtvercový obrázek a velikost oblasti na které je zobrazován je 256 x 256 px:

```
meritko = 256.0f / sirkaObrazku;
```

Protože proměnná bude použita ve třídě **Appearance** je deklarována jako datová proměnná:

```
private float meritko = 0;
```

Kvůli posunutí počátku souřadného systému na střed oblasti pro zobrazování je třeba vypočítat hodnoty posunutí na osách X a Y, které zajistí že obrázek bude vykreslen na střed 3D oblasti. Toto posunutí je nazváno **vystredeniX** pro posun v ose X a **vystredeniY** pro posun v ose Y:

```
vystredeniX = ((sirkaObrazku / 2.0f) - 0.5f) / sirkaObrazku;
vystredeniY = ((vyskaObrazku / 2.0f) - 0.5f) / vyskaObrazku;
```

Tyto proměnné jsou deklarovány jako datové. Hodnotou **0.5f** je definován střed na osách X a Y:

```
private float vystredeniX = 0.5f;
private float vystredeniY = 0.5f;
```

Střed obrázku je určen polovinou jeho šířky a polovinou jeho výšky. Obrázek se bude vykreslovat pomocí třídy **Point3f** od souřadnic **(0.0f, 0.0f, 0.0f)** do **(1.0f, 1.0f, 0.0f)**. Střed zobrazení tedy bude v bodě **(0.5f, 0.5f, 0.0f)**. Proměnné je ještě nutné normalizovat, což je provedeno vydělením hodnot šířkou (resp.výškou) obrázku.

2.3 Vytvoření geometrie jednotlivých pixelů

Metoda **vytvorGeometrii()** se pro vykreslování jednotlivých pixelů změní naprosto celá. Musí se totiž použít třída **PointArray**, která je v Javě 3D k vykreslování bodů určena a která je opět potomkem třídy **GeometryArray**. Je zde vytvořen objekt který je nazván **oblast**:

```
PointArray oblast = new PointArray(pocetPixelu,
                GeometryArray.COORDINATES | GeometryArray.COLOR_3);
```

Do konstruktoru **PointArray()** se zadá počet bodů které mají být vykresleny, dále se musí povolit zadávání koordinátů a barevného odstínu. Vykreslení se provede pomocí dvou smyček, přičemž jedna vykresluje pixely v řádku zleva doprava po celé šířce obrázku a druhá posunuje tuto první smyčku shora dolů, tak aby byly vykresleny všechny řádky po celé výšce obrázku:

```
for (int i = 0; i < vyskaObrazku; i++) {
    for (int j = 0; j < sirkaObrazku; j++) { . . .}
}
```

V těle smyček je nejdříve třeba zajistit správné indexování jednotlivých pixelů, tak aby systém vykreslení odpovídal systému načtení (viz. DP tab.4.1). To se provede pomocí proměnných které slouží k řízení smyček. Protože vnitřní smyčka s proměnnou **j** slouží pro vykreslení bodů v řádcích, je možné využít její postupně nabyté hodnoty přímo pro indexování prvního řádku. Pro další řádek lze opět využít její postupně nabyté hodnoty, které se ovšem musí zvýšit o hodnotu šířky obrázku. To platí i pro další řádky, až do hodnoty celkového počtu řádků, kterou udává výška obrázku. Tento posun po řádcích je reprezentován proměnnou **i**, která řídí vnější smyčku. Z těchto faktů vyplývá následující algoritmus:

```
int indexBodu = j + (sirkaObrazku * i);
```

Odstín bodu je již normalizován v poli **normPixely**[] a tuto hodnotu je tedy možné použít přímo pro určení barvy pomocí již zmíněné třídy **Color3f**:

```
Color3f barva = new Color3f(normPixely[indexBodu],normPixely[indexBodu],
                            normPixely[indexBodu]);
```

Topografický snímek, který se bude zpracovávat, je šedotónový a u šedotónových obrázků platí pro barevné složky: RR = GG = BB. Proto zde může být v každé barevné složce použita stejná hodnota. Pro vykreslení na správné souřadnice je již obdobně připravena normalizovaná hodnota vzdálenosti mezi jednotlivými body **normSirka** (resp. **normVyska**) a aby byl obrázek umístěn do středu 3D plochy je již také připravena hodnota pro posunutí **vystredeniX** (resp. **vystredeniY**). Bod bude vykreslen pomocí již zmíněné třídy **Point3f**:

```
float x = (j * normSirka) - vystredeniX;
float y = (i * normVyska) + vystredeniY;
Point3f bod = new Point3f(x, y, 0);
```

Protože se vykresluje 2D obrázek v osách X a Y, je hodnota souřadnice Z = 0. Posledním krokem, který je nutný pro vykreslení, je nastavení vypočtených koordinátů (resp. barvy) v poli **oblast**. To se provede pomocí konstruktoru **setCoordinate** (resp. **setColor**).

```
oblast.setCoordinate(indexBodu, bod);
oblast.setColor(indexBodu, barva);
```

Nakonec se objekt třídy **PointArray oblast** s nastavenou geometrií odešle zpět pro vykreslení do objektu třídy Shape3D **objekt**, pomocí příkazu **return**:

```
return oblast;
```

2.4 Vytvoření vlastností jednotlivých pixelů

V metodě pro definování vlastností bodů **vytvorVzhled** je opět nejdříve vytvořen objekt třídy **Appearance**, který je nazván **vzhled**:

```
Appearance vzhled = new Appearance();
```

Pro nastavení vlastností jednotlivých bodů je zde využita třída **PointAttributes**:

```
PointAttributes vlastnostiBodu = new PointAttributes(meritko, true);
```

Je vytvořen objekt s názvem **vlastnostiBodu**, u kterého je možné nastavovat velikost zobrazovaných bodů a povolit či zakázat antialiasing. To je možné provádět v závorce konstruktoru, kde první proměnná **meritko**, určuje velikost bodů. Druhá proměnná je hodnota typu boolean **true**, která zde povoluje antialiasing. Samotné nastavení vlastností jednotlivých vykreslovaných bodů se provede pomocí konstruktoru **setPointAttributes()**, do jehož závorky se vloží objekt třídy **PointAttributes** s názvem **vlastnostiBodu**:

```
vzhled.setPointAttributes(vlastnostiBodu);
```

Nakonec se objekt třídy **Appearance** s názvem **vzhled**, odešle zpět pro vykreslení do objektu třídy Shape3D **objekt**, pomocí příkazu **return**:

```
return vzhled;
```

3 Vykreslení barevných pixelů

Do metody **inicializaceKomponentu** je třeba implementovat třídu **FileFilter**. Dále je třeba pozměnit metody **vypocty** a **vytvorGeometrii** tak, aby byly schopny pracovat s jednotlivými složkami barev v RGBA modelu. Ostatní metody zůstanou beze změny.

3.1 Filtr pro výběr souboru

Aby bylo možné pomocí dialogu nabídky načítat pouze soubory s koncovkou PNG, je třeba vytvořit tzv. „filtr souborů“. K tomuto účelu slouží třída **FileFilter**, která je v balíčku **javax.swing.filechooser**:

```
import javax.swing.filechooser.*;
```

Použití zástupného symbolu * je zde z důvodu využití ještě další třídy z tohoto balíčku. Jedná se o třídu s názvem **FileNameExtensionFilter**, jejíž konstruktor je zde použitý pro definování a popis souborů které se mají zobrazit:

```
dialog.setAcceptAllFileFilterUsed(false);
FileFilter filtr = new FileNameExtensionFilter("Obrázky PNG", "png");
dialog.setFileFilter(filtr);
```

V prvním řádku je použita metoda **setAcceptAllFileFilterUsed()** ve které se pomocí hodnoty **false** zajistí, že se v dialogu načtení zobrazí pouze uživatelsky definované filtry a ne defaultní filtr, který umožňuje načtení všech souborů. Ve druhém řádku je pomocí konstruktoru **FileNameExtensionFilter()** vytvořen objekt třídy **FileFilter** s názvem **filtr**. V konstruktoru je popis a definice souborů zobrazovaných pomocí tohoto filtru k výběru. Obsah prvního řetězce se zobrazí v řádku „Files of Type:“ v dialogu načtení, kde bude vypsána hodnota uvedená v uvozovkách: „Obrázky PNG“. Ve druhém řetězci je definována koncovka souborů, které budou zobrazeny k výběru. Zde jsou řetězcem „png“ definovány obrázky typu PNG. Ve třetím řádku je pomocí metody **setFileFilter()** tento objekt třídy **FileFilter** s názvem **filtr** aplikován na objekt třídy **JFileChooser** s názvem **dialog**.

3.2 Zobrazení obrázků obdélníkového tvaru

Pro zobrazení obdélníkových obrázků se musí provést změna v metodě **vypocty()**. Nejdříve se musí větvením určit zda je větší rozměr šířky, nebo výšky obrázku. Pak se z většího rozměru vypočítá vzdálenost bodů od sebe (převrácená hodnota rozměru). Tato proměnná se musí deklarovat jako datová, protože bude použita k vykreslení v metodě **vytvorGeometrii()**:

```
private float vzdalenostBodu = 0;
```

Dále se podle většího rozměru vypočítají proměnné **meritko**, **vystredeniX** a **vystredeniY**:

```
if (sirkaObrazku > vyskaObrazku)
  vzdalenostBodu = 1.0f / sirkaObrazku;
  meritko = 256.0f / sirkaObrazku;
  vystredeniX = ((sirkaObrazku / 2.0f) - 0.5f) / sirkaObrazku;
  vystredeniY = ((vyskaObrazku / 2.0f) - 0.5f) * vzdalenostBodu;
} else {
  vzdalenostBodu = 1.0f / vyskaObrazku;
  meritko = 256.0f / vyskaObrazku;
  vystredeniX = ((sirkaObrazku / 2.0f) - 0.5f) * vzdalenostBodu;
  vystredeniY = ((vyskaObrazku / 2.0f) - 0.5f) / vyskaObrazku;
}
```

3.3 Dekódování barev z RGBA modelu

Opět bude provedena změna v metodě **vypocty()**. Jedná se o úpravu smyčky pro určení odstínu pixelů tak, aby mohly být v metodě **vytvorGeometrii()** použity všechny 3 složky barvy pro správné zobrazení. Nejdříve se musí deklarovat jednorozměrná pole pro odstíny:

```
private float[] cervena = null;
private float[] zelena = null;
private float[] modra = null;
```

Před smyčkou pro naplnění polí, v těle metody **vypocty()**, je třeba nejdříve určit velikost těchto polí. Ta pro každou barevnou složku odpovídá celkovému počtu pixelů v obrázku:

```
cervena = new float[pocetPixelu];
zelena = new float[pocetPixelu];
modra = new float[pocetPixelu];
```

Výstupní hodnoty z objektu **PixelGrabber** jsou posunuté a nabývají hodnot -1 až -16777216. Nejdříve je tedy nutné získat pomocnou hodnotu v rozsahu 0 až 16777215, což se provede přičtením +1 k hodnotě odstínu pixelu z objektu **PixelGrabber** a zjištěním absolutní hodnoty tohoto čísla:

```
int pomocna = Math.abs(pixely[i] + 1);
```

Nejdříve se vypočítá barevný odstín červené RR a to tak, že řetězec se dělí hodnotou s váhovým faktorem, který je určen mocnitelem dle pořadí hodnoty zprava. Hodnota základu odpovídá hodnotě odstínu červené RR. Po dělení vznikl zbytek, v němž jsou stále zakódovány informace o odstínech GG a BB. Pro výpočet barevného odstínu zelené GG se musí zbytek po dělení nejdříve vynásobit hodnotou kterou byl vydělen, čímž se získá kódovaná hodnota řetězce s odstíny zelené a modré barvy, bez hodnoty barvy červené. Posléze se musí na tuto hodnotu aplikovat opět odpovídající dělení hodnotou s váhovým faktorem, který je určen mocnitelem dle pořadí hodnoty zprava. Hodnota základu pak odpovídá hodnotě odstínu zelené GG. Hodnota odstínu modré BB, se získá analogicky jako v předchozím případě: zbytek po dělení se násobí hodnotou kterou byl vydělen a výsledná hodnota se dělí hodnotou s váhovým faktorem, který je určen mocnitelem dle pořadí hodnoty zprava (zde by se měla dělit číslem 1). Hodnota výsledku odpovídá přesně hodnotě odstínu modré BB. Algoritmus který odpovídá výše popsanému způsobu je ještě nutné naprogramovat do metody **vypocty()**:

```
float cervenaF = (float)pomocna / 65536.0f;
int cervenaI = pomocna / 65536;
float zelenaF = (cervenaF - (float)cervenaI) * 256.0f;
int zelenaI = Math.round(zelenaF);
if ((float)zelenaI > zelenaF)
    zelenaI = zelenaI - 1;
float modraF = (zelenaF - (float)zelenaI) * 256.0f;
int modraI = Math.round(modraF);
```

(Pozn.: *Protože v příkladu výpočtu v DP je naznačeno, že se výsledek rovnice (2) nejdříve násobí hodnotou 65536 a pak dělí hodnotou 256, přičemž mezivýsledek z rovnice (3) není nikde používán, jsou v kódu algoritmu tyto operace provedeny najednou a v druhém řádku je tedy jednoduše naprogramováno násobení hodnotou 256, která je výsledkem operace 65536 / 256.*)

V programu Java neexistuje příkaz pro oddělení základu od zbytku. Je tedy použito zaokrouhlení, které však může základ zkreslit tím, že jej zaokrouhlí nahoru. Aby se toto nestalo, je na pátém řádku použito větvení, které v tomto případě sníží základ zpět na původní hodnotu.

Posledním krokem je normování vypočtených hodnot odstínů, aby bylo možné použít je v konstruktoru třídy **`Color3f`** v metodě **`vytvorGeometrii()`**. Po této operaci budou vypočtené odstíny nabývat hodnoty od 0 (pro světlý odstín), po 1 (pro tmavý odstín):

```
cervena[i] = (255.0f - (float)cervenaI) / 255.0f;
zelena[i] = (255.0f - (float)zelenaI) / 255.0f;
modra[i] = (255.0f - (float)modraI) / 255.0f;
```

3.4 Vytvoření geometrie barevných pixelů

V metodě **`vytvorGeometrii()`** nastanou dvě změny. První změna se týká vykreslení jednotlivých složek barvy pixelu. Tyto složky již byly vypočteny v metodě **`vypocty()`** a musí být nyní zadány do závorky konstruktoru objektu pro určení barvy, třídy **`Color3f`**:

```
Color3f barva = new Color3f(cervena[indexBodu], zelena[indexBodu], modra[indexBodu]);
```

Druhá změna se týká vykreslení obrázků, které nejsou čtvercového tvaru. V metodě **`vypocty()`** již byly upraveny výpočty příslušných hodnot, takže místo normovaných hodnot šířky a výšky stačí dosadit do smyčky pro vykreslení vzdálenost dvou bodů. U osy Y je nutné použít hodnotu zápornou, aby bylo dodrženo pravidlo směru vykreslování (vykresluje se směrem od nuly k záporným hodnotám). Hodnotu v ose Y je tedy třeba násobit hodnotou -1:

```
float x = ( j * vzdalenostBodu) - vystredeniX;
float y = (-i * vzdalenostBodu) + vystredeniY;
```

4 Vytvoření interaktivního ovládání

Applet s názvem **`Interaktivita`** vychází z appletu **`VykresleniBarev`**. Do tohoto appletu je třeba nejdříve implementovat příslušné třídy a jejich abstraktní metody pro práci s myší. Jedná se o třídy: **`MouseListener`**, **`MouseMotionListener`**, a **`MouseWheelListener`**, ke kterým musí být povinně implementovány do appletu tyto metody:

```
public void mouseClicked(MouseEvent e);
public void mousePressed(MouseEvent e);
public void mouseReleased(MouseEvent e);
public void mouseEntered(MouseEvent e);
public void mouseExited(MouseEvent e);
public void mouseDragged(MouseEvent e);
public void mouseMoved(MouseEvent e);
public void mouseWheelMoved(MouseWheelEvent e);
```

Všechny tři třídy jsou součástí balíčku pro obsluhu událostí, který musí být importován:

```
import java.awt.event.*;
```

Aby ovšem ovládání pomocí myši fungovalo, musí být příslušný grafický komponent přidán do metody obsluhy událostí. Protože se v appletu bude pomocí myši pracovat s obrázkem ve 3D prostoru, musí být v metodě **`inicializaceKomponentu()`** objekt třídy **`Canvas3D`** nazvaný **`pozadi`** přidán do metody obsluhy událostí:

```
pozadi.addMouseMotionListener(this);
pozadi.addMouseWheelListener(this);
pozadi.addMouseListener(this);
```

Použití klíčového slova **`this`** v rámci instančních metod představuje odkaz na aktuální objekt, jehož metoda či konstruktor jsou právě volány. Zde je to objekt třídy **`Canvas3D`** nazvaný **`pozadi`**.

4.1 Otáčení obrázku pomocí myši

Pro otáčení obrázku je použita metoda **mouseDragged**(MouseEvent e), protože otáčení se bude provádět pohybem myši v příslušném směru, při stisku a držení tlačítka. Nejdříve je třeba rozpoznat, které tlačítko na myši je stisknuto, protože otáčení se má provádět pouze levým tlačítkem. To ale není možné v této metodě, takže k rozpoznání bude sloužit příznak typu **boolean** s názvem **priznakOtaceni**:

```
private boolean priznakOtaceni = false;
```

Na začátku metody **mouseDragged**(MouseEvent e) bude nejdříve větvení, které zajistí, že otáčení obrázku pomocí myši bude možné pouze při použití levého tlačítka na myši:

```
if (priznakOtaceni == true) { . . .}
```

Dále je třeba pomocí metody **getX()** (resp. **getY()**) zjistit aktuální pozici kursoru myši na obrazovce a odečtením (resp. přičtením) poloviny šířky (resp. výšky) 3D prostoru **pozadi**, která je 256 pixelů, se posune počátek souřadného systému do středu tohoto 3D prostoru:

```
poziceX = e.getX() - 256;
poziceY = 256 - e.getY();
```

Proměnné **poziceX** a **poziceY**, jsou typu **integer**:

```
private int poziceX = 0;
private int poziceY = 0;
```

Následně je třeba zajistit, aby interakce probíhala pouze je-li kursor ve 3D prostoru appletu. To se provede pomocí větvení, kde se zjišťuje zda je kursor v oblasti ±256 px od středu 3D oblasti v osách X i Y. Pokud kursor 3D oblast opustí, jsou nastaveny proměnné **poziceX** a **poziceY** na hodnotu 0:

```
if (poziceX > 256 ^ poziceX < -256) {
    poziceX = 0;
    poziceY = 0;
}
if (poziceY > 256 ^ poziceY < -256) {
    poziceY = 0;
    poziceX = 0;
}
```

Pro získání hodnot, které budou vhodné pro ovládání rotace se použije dočasná hodnota s názvem **docasnaX** (resp. **docasnaY**), v níž bude uložena poslední známá pozice kursoru a která se bude zjišťovat při stisku tlačítka myši pomocí již zmíněné metody **getX()** (resp. **getY()**). Protože hodnotu je třeba načítat již při stisknutí tlačítka myši, musí být toto načtení provedeno v příslušné abstraktní metodě **public void mousePressed**(MouseEvent e). V té je nejdříve pomocí metody **getButton** provedeno rozpoznání stisknutého tlačítka. Pak se větvením rozpozná které tlačítko bylo stisknuto. Pouze při stisku levého tlačítka se nastaví příznak **priznakOtaceni** na hodnotu **true**, čímž se zajistí že otáčení bude možné provádět pouze levým tlačítkem:

```
int tlacitko = e.getButton();
if (tlacitko == 1)
    priznakOtaceni = true;
docasnaX = e.getX();
docasnaY = e.getY();
```

V metodě **public void mouseReleased**(MouseEvent e) se pak při uvolnění levého tlačítka provede nastavení příznaku **priznakOtaceni** zpět na hodnotu **false**:

```
int tlacitko = e.getButton();
if (tlacitko == 1)
    priznakOtaceni = false;
```

Hodnoty **docasnaX** a **docasnaY**, z metody **public void mousePressed**(MouseEvent e), se budou porovnávat s aktuální pozicí v metodě **public void mouseDragged**(MouseEvent e):

```
if (poziceX < docasnaX) {
    docasnaX = poziceX;
    rotaceX = rotaceX - (Math.PI / 90);
}
if (poziceX > docasnaX) {
    docasnaX = poziceX;
    rotaceX = rotaceX + (Math.PI / 90);
}
```

```
if (poziceY < docasnaY) {
    docasnaY = poziceY;
    rotaceY = rotaceY + (Math.PI / 90);
}
if (poziceY > docasnaY) {
    docasnaY = poziceY;
    rotaceY = rotaceY - (Math.PI / 90);
}
```

Porovnávání poslední známé pozice kursoru s pozicí aktuální je zajištěno větvením. Pokud jsou hodnoty **docasnaX** a **poziceX** (resp. **docasnaY** a **poziceY**) rozdílné, vyhodnotí se pohyb kursoru v ose X (resp. Y), následně je dočasná hodnota přepsána hodnotou nové pozice kursoru a povel k rotaci **rotaceX** v ose X (resp. **rotaceY** v Y) je inkrementován (resp. dekrementován) o zvolenou hodnotu. Tak jsou získány povely k rotaci v obou směrech na osách X a Y. V prvním větvení: pohyb v kladném směru po ose X, ve druhém: pohyb v záporném směru po ose X, ve třetím: pohyb v kladném směru po ose Y, ve čtvrtém: pohyb v záporném směru po ose Y. Použité proměnné je třeba deklarovat na začátku programu jako datové proměnné:

```
private int docasnaX = 0;
private int docasnaY = 0;
private double rotaceX = 0;
private double rotaceY = 0;
```

Nakonec je volána vlastní metoda s názvem **transformujObraz()**, která provádí rotaci i další transformace obrázku. Pro její spuštění nesmí být objekt **scena** prázdný (musí být načten obrázek).

```
if (scena != null)
    transformujObraz();
```

4.2 Transformační funkce

Funkce pro transformace obrázku budou naprogramovány ve vlastní metodě s názvem **transformujObraz**().Nejdříve se musí vytvořit transformační funkce pro pohyby, kterými jsou objekty třídy **Transform3D**. Pro pohyb v každé ose musí být vytvořen jeden objekt této třídy:

```
Transform3D otocX = new Transform3D();
Transform3D otocY = new Transform3D();
```

Pomocí metody **rotX()** (resp. **rotY()**) se do objektu **otocX** (resp. **otocY**) zapíše hodnota **rotaceX** (resp. **rotaceY**), která způsobí pootočení objektu o úhel, odpovídající této hodnotě:

```
otocX.rotY(rotaceX);
otocY.rotX(rotaceY);
```

Metodou **rotX()** se provádí rotace roviny XY okolo osy X, což způsobuje že celá tato rovina rotuje ve směru osy Y. Musí být tedy přiřazena k objektu **otocY** a její pohyb řízen hodnotou **rotaceY**, jinak by ovládání pomocí myši nebylo ergonomické. Totéž platí analogicky i pro metodu **rotY()**. Aby byla zajištěna funkce otáčení v obou osách (X i Y) zároveň, musí se použít metoda **mul()**, pomocí které se oba objekty sloučí:

```
otocX.mul(otocY);
```

Nakonec je pomocí metody **setTransform()** implementován sloučený objekt **otocX** do transformační skupiny, která bude přidána do metody **vytvorScenu()**.

```
pohyb.setTransform(otocX);
```

4.3 Implementace transformačních funkcí do skupiny

Aby mohly být transformační funkce implementovány do transformační skupiny, je nutné na začátku programu tuto transformační skupinu deklarovat, protože bude použita ve více metodách. Transformační skupina je deklarována pod názvem **pohyb**:

```
private TransformGroup pohyb = null;
```

Další úpravy programu se pak zapisují do metody **vytvorScenu()**. Nejdříve se vytvoří transformační skupina **pohyb**. Pak je v metodě **setCapability** pomocí hodnoty **ALLOW_TRANSFORM_WRITE** povolen zápis hodnot:

```
pohyb = new TransformGroup();
pohyb.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
```

Vlastní implementace transformačních funkcí je provedena voláním metody **transformujObraz** a přidáním vykresleného objektu (na který mají být transformace aplikovány). Zde je to objekt třídy **Shape3D** s názvem **objekt**, který je pomocí konstruktoru **addChild** přidán do výše vytvořené transformační skupiny **pohyb**:

```
transformujObraz();
pohyb.addChild(objekt);
```

Původně byl objekt třídy **Shape3D** s názvem **objekt** přidán přímo do objektu třídy **BranchGroup** s názvem **scena**, což bylo provedeno zápisem v těle metody **vytvorScenu()**:

```
scena.addChild(objekt);
```

V tomto appletu už ale byl objekt s názvem **objekt** přidán kvůli interaktivitě do transformační skupiny s názvem **pohyb**, takže aby se **objekt** zobrazil a bylo možné s ním otáčet, musí se do objektu třídy **BranchGroup** s názvem **scena**, přidat celá transformační skupina s názvem **pohyb** a tímto zápisem se musí nahradit řádek který je uvedený výše:

```
scena.addChild(pohyb);
```

4.4 Změna měřítka obrázku

Pro změnu měřítka obrázku je nutné přidat do metody **transformujObraz()** novou transformační funkci, pomocí které bude zvětšování (resp. zmenšování) obrázku prováděno:

```
Transform3D lupa = new Transform3D();
lupa.setScale(zmenaVelikosti);
```

Transformační funkce je nazvána **lupa** a pro samotnou změnu měřítka obrázku je zde využita metoda **setScale**, která je řízena proměnnou typu **float** s názvem **zmenaVelikosti**:

```
private float zmenaVelikosti = 1;
```

Počáteční hodnota proměnné **zmenaVelikosti** je 1. Tato hodnota zajistí, že obrázek se zobrazí ve 100% velikosti. Objekt **lupa** se musí sloučit s ostatními objekty transformačních funkcí pomocí metody **mul** objektem pro rotaci v ose X **otocY**:

```
otocY.mul(lupa);
```

Tímto je v metodě **transformujObraz()** připravena nová transformační funkce pro změnu měřítka obrázku, která je řízena pomocí proměnné **zmenaVelikosti**. Aby bylo možné měnit měřítko obrázku v rozsahu 100 až 300%, musí tato proměnná nabývat hodnot 1,0 až 3,0. Pro zvětšování (resp. zmenšování) obrázku se předpokládá využití kolečka myši. Ovládací funkce pro změnu měřítka se musí naprogramovat v abstraktní metodě **mouseWheelMoved**. Aby kolečko myši bylo funkční jen v případě že je kursor na 3D pozadí, lze využít abstraktní metody **mouseEntered** a **mouseExited**. V nich se nadefinuje hodnota příznaku pro práci s kolečkem s názvem **koleckoFunkcni**:

```
private boolean koleckoFunkcni = true;
public void mouseEntered(MouseEvent e) {
    koleckoFunkcni = true;
}
public void mouseExited(MouseEvent e) {
    koleckoFunkcni = false;
}
```

Tento příznak je pak možné použít v abstraktní metodě **public void mouseWheelMoved**`(MouseWheelEvent e)` a ihned na začátku této metody určit větvením, zda bude kolečko myši pracovat, nebo ne:

```
if (koleckoFunkcni == true) { . . .}
```

Pro samotné ovládání měřítka obrázku, je prvním krokem zachycení události při pohybu kolečka myši pomocí proměnné **e**. Ta je při pohybu kolečka myši vpřed (resp. vzad) inkrementována (resp. dekrementována) o hodnotu 1 pomocí metody **getWheelRotation**:

```
int rotace = e.getWheelRotation();
```

Je zde vytvořena pomocná hodnota typu **integer**, která má název **rotace**. Ta ovšem nemůže být použita přímo pro řízení změny měřítka obrázku, protože se jedná o proměnnou jiného číselného typu a tato proměnná má také nevyhovující rozsah hodnot. Musí být tedy v dalším kroku upravena:

```
zmenaVelikosti = zmenaVelikosti + (rotace * 0.05f);
```

Protože v metodě **transformujObraz**() je pro řízení změny měřítka připravena proměnná typu **float** s názvem **zmenaVelikosti**, pomocná proměnná **rotace** se nejdříve upraví na velikost vhodnou k ovládání změny měřítka a pak se již tato upravená hodnota přímo přičítá k proměnné **zmenaVelikosti**.. Aby bylo možné měnit měřítko obrázku pouze v daném rozsahu, musí být stanovena maximální a minimální hodnota proměnné **zmenaVelikosti**. To se provede větvením, ve kterém se při překročení zvoleného rozsahu přiřadí proměnné **zmenaVelikosti** velikost odpovídající minimu, resp. maximu:

```
if (zmenaVelikosti < 1.0f)
    zmenaVelikosti = 1.0f;
if (zmenaVelikosti > 3.0f)
    zmenaVelikosti = 3.0f;
```

Nakonec se volá metoda **transformujObraz**() pouze za předpokladu, je-li vytvořena scéna:

```
if (scena != null)
    transformujObraz();
```

4.5 Podmínky pro interaktivní ovládání

Po ukončení práce s jedním obrázkem je další obrázek zobrazen s hodnotami rotace a měřítka, které byly naposledy použity u předchozího obrázku. V appletu je tedy provedena ještě jedna změna. Ta zabraňuje aby si applet „pamatoval“ poslední nastavení těchto hodnot. V metodě **public void actionPerformed**(ActionEvent e) jsou ve větvení pro načtení nového obrázku nastaveny proměnné **otocX**, **otocY** a **zmenaVelikosti** na inicializační hodnoty, čímž se tato chyba odstraní:

```
rotaceX = 0;
rotaceY = 0;
zmenaVelikosti = 1;
```

5 Trojrozměrné zobrazování

5.1 Transformace bodů v prostoru

Pro transformaci bodů do prostoru se využije smyčka vykreslení v metodě **vytvorGeometrii**() a smyčka zpracování v metodě **vypocty**() z appletu **VykresleniPixelu**. V metodě **vytvorGeometrii** se změní konstruktor třídy **Point3f**:

```
Point3f bod = new Point3f(x, y, -normPixely[indexBodu]);
```

Protože proměnné z pole **normPixely[]** nabývají hodnot od 0,0 do 1,0 lze je použít přímo pro transformaci v závorce konstruktoru třídy **Point3f**. Záporná hodnota u proměnné z pole **normPixely[]** je z toho důvodu, aby se světlé body, které představují prohloubené oblasti, zobrazovaly na ose Z v záporném směru, čímž se budou jevit opravdu jako prohloubené. Pokud by se použila kladná hodnota, oblasti by se jevily jako vyvýšené. Toto zobrazení má ovšem ještě několik nedostatků. Prvním z nich je, že se 3D model neotáčí kolem počátku v ose Z. Pro odstranění tohoto problému se musí určit hodnoty nejsvětlejšího a nejtmavšího bodu v obrázku, což se provede v metodě **vypocty**. Nejdříve se v této metodě deklarují lokální proměnné typu **float**, s názvy **min** a **max**:

```
float min = 0.5f;
float max = 0.5f;
```

Hodnotou **0.5f** je definován střed na ose Z, kolem kterého se má 3D model otáčet. Ve smyčce pro normalizaci pixelů se budou porovnávat hodnoty světlosti (resp. tmavosti) pixelů s těmito proměnnými. Do proměnné **min** se uloží informace o hodnotě nejsvětlejšího bodu a do proměnné **max** o hodnotě nejtmavšího bodu. Celá smyčka pro normalizaci pixelů po úpravě vypadá takto:

```
for (int i = 0; i < pocetPixelu; i++) {
    normPixely[i] = ((float)pixely[i] / 16777216.0f) + 1.0f;
    min = Math.min(normPixely[i], min);
    max = Math.max(normPixely[i], max);
}
```

Výsledné hodnoty budou použity pro posunutí 3D modelu na střed osy Z, pomocí další proměnné typu **float** s názvem **vystredeniZ**:

```
vystredeniZ = (max + min) / 2;
```

Tuto proměnnou je třeba deklarovat na začátku programu jako datovou proměnnou:

```
private float vystredeniZ = 0.5f;
```

Posunutí 3D modelu na střed osy Z se provede v metodě **vytvorGeometrii**. Zde nastane změna v závorce konstruktoru třídy **Point3f**, kde je k souřadnici přičten posun **vystredeniZ**:

```
Point3f bod = new Point3f(x, y, vystredeniZ - normPixely[indexBodu]);
```

Nyní se střed 3D modelu otáčí kolem počátku souřadného systému XYZ.

5.2 Zobrazení povrchu 3D modelu

Dalším nedostatkem zobrazení z části 4.5.1 je absence povrchu 3D modelu. Pro generování povrchu 3D modelu se musí vytvořit nová metoda, která byla nazvána **vytvorPovrch**:

```
public Geometry vytvorPovrch() { . . .}
```

Touto metodou bude nahrazena dosavadní metoda pro vykreslování bodů s názvem **vytvorGeometrii**. Proto musí být nejdříve nahrazeno její volání v metodě **vytvorScenu**:

```
objekt.setGeometry(vytvorPovrch());
```

V nové metodě s názvem **vytvorPovrch**, bude pro zobrazení povrchu 3D modelu využita třída **TriangleStripArray**, pomocí níž se vykreslují pásy trojúhelníků. Objekt této třídy byl nazván **oblast**. V závorce konstruktoru je nutné zadat kolik vrcholů trojúhelníků se bude celkem pro vykreslení používat, dále je třeba povolit zadávání koordinátů a práci s barvou, a nakonec počet vrcholů trojúhelníků v jednom pásu, použije-li se vykreslení ve více pásech:

```
TriangleStripArray oblast = new TriangleStripArray(pocetVrcholu,
  GeometryArray.COORDINATES | GeometryArray.COLOR_3, pocetVPasu);
```

První proměnná s názvem **pocetVrcholu** je typu integer a je třeba ji deklarovat jako datovou proměnnou na začátku programu, protože její hodnota se bude zjišťovat v metodě **vypocty**:

```
private int pocetVrcholu = 0;
```

Pro povolení zadávání koordinátů a práci s odstínem barvy jsou použity hodnoty **GeometryArray.COORDINATES** a **GeometryArray.COLOR_3**, které jsou typu **boolean**. Poslední proměnná je nazvána **pocetVPasu** a musí být zadána jako jednorozměrné pole typu **integer**. Protože použití více pásů pro vykreslení by bylo v tomto případě nepřehledné a také obtížnější, byla zvolena varianta kdy je vykreslován jediný pás a proto také pole **pocetVPasu** bude mít stejný počet prvků jako je hodnota proměnné **pocetVrcholu**:

```
int[] pocetVPasu = {pocetVrcholu};
```

Před vytvořením vlastního algoritmu vykreslování, je však ještě nutné vypočítat hodnotu proměnné **pocetVrcholu**. Jak již bylo napsáno, tento výpočet bude proveden v metodě **vypocty**, protože postačuje zjistit tuto hodnotu pouze jednou před vykreslováním 3D modelu. Výpočet počtu vrcholů trojúhelníků vychází z DP obr.4.5, kde je naznačen postup vykreslování jednotlivých trojúhelníků. Jak je z DP obr.4.5 patrné, vrcholy trojúhelníků v horním a dolním řádku jsou pro vykreslení použity 2x, a ostatní vrcholy uprostřed jsou pro vykreslení použity 4x. Na této skutečnosti je založen i výpočet celkového počtu vrcholů trojúhelníků v metodě **vypocty**:

```
pocetVrcholu = ((4 * sirkaObrazku) * (vyskaObrazku - 1));
```

Vlastní algoritmus vykreslování povrchu je opět proveden pomocí vnořených smyček, kdy dvě vnořené smyčky vykreslují souřadnice v celém řádku a vnější smyčka posunuje vykreslování na další řádky po celé výšce. Dvě vnořené smyčky jsou použity proto, že jedna řídí vykreslování na vnější straně 3D modelu směrem zleva doprava pomocí inkrementace a druhá řídí vykreslování na vnitřní straně 3D modelu směrem zprava doleva pomocí dekrementace:

```
for (int i = 0; i < vyskaObrazku - 1; i++) {
    for (int j = 0; j < sirkaObrazku; j++) {. . .}
    for (int k = sirkaObrazku - 1; k >= 0 ; k-=1) {. . .}
}
```

Určení indexu bodu v 3D modelu je provedeno jako v části 2.3. Vnitřní smyčka s proměnnou **j** slouží k vykreslení bodů v řádcích. Posun po řádcích je reprezentován proměnnou **i**, která řídí vnější smyčku:

```
int indexBodu = j + (sirkaObrazku * i);
```

Bod s vypočteným indexem bude opět vykreslen pomocí třídy **Point3f**:

```
float x = ( j * vzdalenostBodu) - vystredeniX;
float y = (-i * vzdalenostBodu) + vystredeniY;
float z = vystredeniZ - normPixely[indexBodu];
Point3f bod = new Point3f(x, y, z);
```

Proměnné **x**, **y** a **z** jsou souřadnice bodu ve 3D prostoru a jejich hodnoty jsou získány stejně jako v částech 3.4 (hodnoty **x** a **y**) a 5.1 (hodnota **z**). Barva vykresleného bodu se určí pomocí třídy **Color3f**:

```
Color3f barva = new Color3f(normPixely[indexBodu], normPixely[indexBodu],
                            normPixely[indexBodu]);
```

Na začátku metody se také musí nejdříve lokálně deklarovat pomocná proměnná typu **integer** s názvem **poradi**, a která bude sloužit k určení pořadí v jakém budou jednotlivé body vykreslovány:

```
int poradi = -1;
```

Určení pořadí, v jakém budou body vykreslovány, je nutné pro vykreslení a nastavení odstínu povrchu pomocí objektu třídy **TriangleStripArray** s názvem **oblast**:

```
poradi = poradi + 1;
oblast.setCoordinate(poradi, bod);
oblast.setColor(poradi, barva);
```

Vykreslování se provádí tzv. metodou „cik-cak“, což znamená že každý druhý bod který má být vykreslen, leží o řádek níže. Proto je do první smyčky k výše uvedenému kódu, který čte a vykresluje body v řádku, přidána analogicky druhá část, která čte a vykresluje body položené o řádek níže:

```
indexBodu = indexBodu + sirkaObrazku;
x = ( j * vzdalenostBodu) - vystredeniX;
y = (-i * vzdalenostBodu) + vystredeniY - vzdalenostBodu;
z = vystredeniZ - normPixely[indexBodu];
bod = new Point3f(x, y, z);
Color3f barva = new Color3f(normPixely[indexBodu], normPixely[indexBodu],
                            normPixely[indexBodu]);
poradi = poradi + 1;
oblast.setCoordinate(poradi, bod);
oblast.setColor(poradi, barva);
```

V prvním řádku je určen index bodu který je o řádek níže a to tak, že je k původnímu indexu přičtena hodnota šířky řádku. V dalších třech řádcích se určují souřadnice bodu a protože je vykreslován bod který je nyní položen o řádek níže (tedy směrem k záporným hodnotám na ose Y), je poloha tohoto bodu na ose Y posunuta o jeden řádek níže pomocí proměnné **vzdalenostBodu**. Zbytek kódu je totožný s kódem uvedeným v první části smyčky. Tak je vykreslen povrch na vnější straně. Pro vykreslení povrchu na vnitřní straně je analogicky použita druhá smyčka s dekrementací:

```
for (int k = sirkaObrazku - 1; k >= 0 ; k-=1) {
  int indexBodu = k + (i * sirkaObrazku);
  poradi = poradi + 1;
  float x = ( k * vzdalenostBodu) - vystredeniX;
  float y = (-i * vzdalenostBodu) + vystredeniY;
  float z = vystredeniZ - normPixely[indexBodu];
  Point3f bod = new Point3f(x, y, z);
  Color3f barva = new Color3f(normPixely[indexBodu], normPixely[indexBodu],
                              normPixely[indexBodu]);
  oblast.setCoordinate(poradi, bod);
  oblast.setColor(poradi, barva);
  indexBodu = indexBodu + sirkaObrazku;
  poradi = poradi + 1;
  x = ( k * vzdalenostBodu) - vystredeniX;
  y = (-i * vzdalenostBodu) + vystredeniY - vzdalenostBodu;
  z = vystredeniZ - normPixely[indexBodu];
  bod = new Point3f(x, y, z);
  barva = new Color3f(normPixely[indexBodu], normPixely[indexBodu],
                      normPixely[indexBodu]);
  oblast.setCoordinate(poradi, bod);
  oblast.setColor(poradi, barva);
  }
```

Nakonec se objekt třídy **TriangleStripArray** s názvem **oblast** odešle zpět pro vykreslení do objektu třídy Shape3D **objekt**, pomocí příkazu **return**:

```
return oblast;
```

5.3 Vyhlazení hran ve 3D modelu

Pro vyhlazení hran byla vytvořena nová metoda s názvem **vyhlazeniPovrchu**. V té budou nejdříve zprůměrovány hodnoty pixelů, které jsou v rozích obrázku. Algoritmus vychází z DP obr.4.8a:

```
    pomPixely[0] =
  (normPixely[0]
+ normPixely[1]
+ normPixely[sirkaObrazku]
+ normPixely[sirkaObrazku + 1]) / 4;
```

Takto je provedeno zprůměrování pixelu v levém horním rohu. V prvním řádku se do pomocného pole s názvem **pomPole** bude zapisovat hodnota zprůměrovaného pixelu s indexem 0, která se vypočte. Ve druhém řádku se načte původní hodnota pixelu s indexem 0 a pak se k ní přičtou hodnoty okolních pixelu. Ve třetím řádku se přičte hodnota vedlejšího pixelu, ve čtvrtém hodnota pixelu níže a v pátém hodnota pixelu ležícího diagonálně. Na konci pátého řádku se provede dělení počtem všech sčítanců, tedy číslem 4. Tím je zprůměrována hodnota pixelu s indexem 0 v levém horním rohu obrázku. Obdobně jsou vytvořeny algoritmy, pro zprůměrování hodnot pixelů v ostatních rozích:

```
    pomPixely[ sirkaObrazku - 1] =
  (normPixely[ sirkaObrazku - 1]
+ normPixely[ sirkaObrazku - 2]
+ normPixely[(sirkaObrazku * 2) - 1]
+ normPixely[(sirkaObrazku * 2) - 2]) / 4;
    pomPixely[pocetPixelu -  sirkaObrazku] =
  (normPixely[pocetPixelu -  sirkaObrazku]
+ normPixely[pocetPixelu -  sirkaObrazku + 1]
+ normPixely[pocetPixelu - (sirkaObrazku * 2)]
+ normPixely[pocetPixelu - (sirkaObrazku * 2) + 1]) / 4;
    pomPixely[pocetPixelu - 1] =
  (normPixely[pocetPixelu - 1]
+ normPixely[pocetPixelu - 2]
+ normPixely[pocetPixelu - 1 - sirkaObrazku]
+ normPixely[pocetPixelu - 2 - sirkaObrazku]) / 4;
```

Dalším krokem je zprůměrování hodnot pixelů, které jsou umístěny na vnějších stranách obrázku. K tomu byl vytvořen algoritmus, který vychází z DP obr.4.8b: Pro pixely umístěné na horní straně:

```
for (int j = 0; j < sirkaObrazku - 2; j++) {
    pomPixely[j + 1] =
  (normPixely[j]
+ normPixely[j + 1]
+ normPixely[j + 2]
+ normPixely[j + sirkaObrazku]
+ normPixely[j + sirkaObrazku + 1]
+ normPixely[j + sirkaObrazku + 2]) / 6;
}
```

V prvním řádku je smyčka, zajišťující horizontální posun vymezené oblasti. V druhém řádku se do pomocného pole s indexem **[j + 1]** zapíše vypočtená hodnota zprůměrovaného pixelu. Ve třetím řádku se načte hodnota pixelu vlevo, k té se přičte ve čtvrtém řádku původní hodnota vyhlazovaného pixelu a v pátém hodnota pixelu vpravo. Dále se v šestém řádku přičte hodnota pixelu diagonálně vlevo, v sedmém hodnota pixelu níže a v osmém hodnota pixelu diagonálně vpravo. Na konci pátého řádku je opět provedeno dělení tohoto součtu počtem všech sčítanců, zde číslem 6. Pomocí smyčky se algoritmus opakuje po celé délce řádku. Tím jsou zprůměrovány hodnoty pixelů na horní straně obrázku (krom rohů zprůměrovaných v předchozím algoritmu). Obdobným způsobem jsou vytvořeny algoritmy pro zprůměrování hodnot pixelů na ostatních stranách (dolní, levé a pravé):

```
for (int j = 0; j < sirkaObrazku - 2; j++) {
   pomPixely[pocetPixelu - 2 - j] =
 (normPixely[pocetPixelu - 3 - j - sirkaObrazku]
+ normPixely[pocetPixelu - 2 - j - sirkaObrazku]
+ normPixely[pocetPixelu - 1 - j - sirkaObrazku]
+ normPixely[pocetPixelu - 3 - j]
+ normPixely[pocetPixelu - 2 - j]
+ normPixely[pocetPixelu - 1 - j]) / 6;
}
```

```
for (int j = 0; j < vyskaObrazku - 2; j++) {
   pomPixely[sirkaObrazku + (j * sirkaObrazku)] =
 (normPixely[(j * sirkaObrazku)]
+ normPixely[(j * sirkaObrazku) + 1]
+ normPixely[(j * sirkaObrazku) +  sirkaObrazku]
+ normPixely[(j * sirkaObrazku) +  sirkaObrazku + 1]
+ normPixely[(j * sirkaObrazku) + (sirkaObrazku * 2)]
+ normPixely[(j * sirkaObrazku) + (sirkaObrazku * 2) + 1]) / 6;
}
for (int j = 0; j < vyskaObrazku - 2; j++) {
   pomPixely[pocetPixelu - 1 - (j * sirkaObrazku) - sirkaObrazku] =
 (normPixely[pocetPixelu - 2 - (j * sirkaObrazku) - (sirkaObrazku * 2)]
+ normPixely[pocetPixelu - 1 - (j * sirkaObrazku) - (sirkaObrazku * 2)]
+ normPixely[pocetPixelu - 2 - (j * sirkaObrazku) - sirkaObrazku]
+ normPixely[pocetPixelu - 1 - (j * sirkaObrazku) - sirkaObrazku]
+ normPixely[pocetPixelu - 2 - (j * sirkaObrazku)]
+ normPixely[pocetPixelu - 1 - (j * sirkaObrazku)]) / 6;
}
```

Posledním krokem, který je nutný pro vyhlazení povrchu 3D modelu, je zprůměrování hodnot pixelů, které jsou uvnitř obrázku. K tomu účelu byl vytvořen algoritmus, který vychází z DP obr.4.7b:

```
for (int i = 0; i < vyskaObrazku - 2; i++) {
   for (int j = 0; j < sirkaObrazku - 2; j++) {
      pomPixely[(i*vyskaObrazku) + j + sirkaObrazku + 1] =
    (normPixely[(i*vyskaObrazku) + j]
   + normPixely[(i*vyskaObrazku) + j + 1]
   + normPixely[(i*vyskaObrazku) + j + 2]
   + normPixely[(i*vyskaObrazku) + j +  sirkaObrazku]
   + normPixely[(i*vyskaObrazku) + j +  sirkaObrazku + 1]
   + normPixely[(i*vyskaObrazku) + j +  sirkaObrazku + 2]
   + normPixely[(i*vyskaObrazku) + j + (sirkaObrazku * 2)]
   + normPixely[(i*vyskaObrazku) + j + (sirkaObrazku * 2) + 1]
   + normPixely[(i*vyskaObrazku) + j + (sirkaObrazku * 2) + 2]) / 9;
   }
}
```

Vnější smyčka v prvním řádku zajišťuje vertikální posun vymezené oblasti. Pomocí vnitřní smyčky ve druhém řádku se posunuje vymezená oblast v horizontálním směru. Ve třetím řádku se zapisuje do pomocného pole vypočtená hodnota zprůměrovaného pixelu. Na dalších třech řádcích se sčítají hodnoty tří pixelů ležících výše, na osmém řádku se načte původní hodnota pixelu který se průměruje a na sedmém a devátém řádku se přičtou hodnoty pixelů ležících vedle. Na posledních třech řádcích se přičtou hodnoty tří pixelů ležících níže. Zprůměrování je dokončeno na konci, kde se součet dělí počtem sčítanců, kterých je 9. Pomocné pole s názvem **pomPole** je deklarováno jako dvojrozměrné. Hodnoty se do něho po výpočtech uloží jako do paměti a budou ihned připraveny k vykreslení:

```
private float[][] pomPixely = null;
```

Protože proměnná **normPixely** je u 3D modelu s nevyhlazeným povrchem využívána přímo pro vykreslování bodů a hodnoty z pole **pomPixely** ji budou přepisovat, je vhodné vytvořit ještě pole se zálohou původních hodnot pixelů v obrázku. Za tímto účelem je tedy na začátku programu deklarováno jako datová proměnná jednorozměrné pole s názvem **zalozniPixely**:

```
private float[] zalozniPixely = null;
```

Toto záložní pole bude mít stejný počet prvků jako pole **normPixely**:

```
zalozniPixely = new float[pocetPixelu];
```

a naplní se ihned po normování hodnot pixelů ve smyčce v metodě **vypocty**:

```
zalozniPixely[i] = normPixely[i];
```

„Vyhlazené“ hodnoty z pole **pomPixely** se však budou přepisovat do pole **normPixely** pomocí jiné smyčky, která je na konci metody **vyhlazeniPovrchu**:

```
for (int m = 0; m < pocetPixelu; m++) {
  normPixely[m] = pomPixely[m];
  min = Math.min(pomPixely[m], min);
  max = Math.max(pomPixely[m], max);
}
```

Ve třetím (resp. čtvrtém) řádku smyčky je určována hodnota nejtmavšího (resp. nejsvětlejšího) bodu po vyhlazení. Tyto hodnoty jsou zjišťovány, protože s postupným vyhlazováním povrchu se bude posunovat také střed 3D modelu. Hodnoty **min** a **max**, pak budou použity pro výpočet nového posunutí 3D modelu na střed souřadné soustavy. Nové posunutí se bude počítat na konci metody **vyhlazeniPovrchu**, takže proměnné mohou být deklarovány lokálně na začátku této metody:

```
float min = 0.5f;
float max = 0.5f;
```

Z důvodu posunu středu 3D modelu je také vytvořena záloha proměnné **vystredeniZ**, která řídí posun středu 3D modelu do počátku souřadného systému u 3D modelu s nevyhlazeným povrchem. Tato záložní proměnná je nazvána **memoZ** a je deklarována jako datová proměnná na začátku programu:

```
private float memoZ = 0;
```

Tato záloha se opět vytvoří v metodě **vypocty**, ihned po výpočtu hodnoty **vystredeníZ**:

```
memoZ = vystredeniZ;
```

Hodnota proměnné **vystredeniZ** se musí s každým krokem vyhlazení opravit, aby mohl 3D model rotovat kolem počátku souřadného systému i při změně svého tvaru příp. velikosti, kterou může vyhlazení způsobit. Tyto opravné hodnoty se budou ukládat do pole, které je nazváno **korekceZ** a je deklarováno na začátku programu:

```
private float[] korekceZ = null;
```

Počet paměťových polí, do nichž se budou ukládat „vyhlazené“ hodnoty a hodnoty posunu, je stanoven na 10:

```
pomPixely = new float[pocetPixelu][10];
korekceZ = new float[10];
```

Také paměťová smyčka, pomocí níž se do polí budou hodnoty ukládat bude mít 10 cyklů. Navíc se do **pomPole** uloží hodnoty v každém cyklu, takže proměnné tohoto pole budou dvojrozměrné :

```
for (int k = 0; k < 10; k++) {
  pomPixely[0][k] = ...
  pomPixely[ sirkaObrazku - 1][k] = ...
  pomPixely[pocetPixelu - sirkaObrazku][k] = ...
  pomPixely[pocetPixelu - 1][k] = ...
  pomPixely[j + 1][k] = ...
  pomPixely[pocetPixelu - 2 - j][k] = ...
  pomPixely[sirkaObrazku + (j * sirkaObrazku)][k] = ...
  pomPixely[pocetPixelu - 1 - (j * sirkaObrazku) - sirkaObrazku][k] = ...
  pomPixely[(i*vyskaObrazku) + j + sirkaObrazku + 1][k] = ...
}
```

Opravné hodnoty pro proměnou **vystredeniZ**, které se budou ukládat do pole **korekceZ** budou vypočteny a uloženy ihned po uložení „vyhlazených“ hodnot, na konci paměťové smyčky:

```
korekceZ[k] = (max + min) / 2.0f;
```

Na konci celé metody **vyhlazeniPovrchu**, po skončení paměťové smyčky, se ještě musí do pole **normPixely** vrátit ze zálohy původní „nevyhlazené“ hodnoty, aby ostatní části appletu mohly tyto hodnoty používat.

```
for (int n = 0; n < pocetPixelu; n++) {
  normPixely[n] = zalozniPixely[n];
}
```

Posledním krokem je volání celé metody **vyhlazeniPovrchu**:

```
private void vypocty() {. . .
vyhlazeniPovrchu();
. . .}
```

5.4 Interaktivní ovládání vyhlazení povrchu

Aby bylo možné vyhlazování povrchu interaktivně ovládat, je do GUI přidán nový ovládací prvek: posuvník. Ten je součástí knihovny **swing** a je třeba jej deklarovat na začátku programu:

```
private JSlider ovlVyhlazeni = null;
```

Posuvník je nazván **ovlVyhlazeni** a jeho načtení a základní nastavení musí být provedeno v metodě **inicializaceKomponentu**():

```
ovlVyhlazeni = new JSlider();
ovlVyhlazeni.setValue(5);
ovlVyhlazeni.setMaximum(10);
ovlVyhlazeni.addChangeListener(this);
ovlVyhlazeni.setBounds(64, 4, 100, 22);
horniPanel.add(ovlVyhlazeni);
```

V prvním řádku je provedeno načtení posuvníku. Ve druhém řádku je nastavena inicializační hodnota, která je v tomto případě 5. Ve třetím řádku je určen maximální rozsah posuvníku, který je zde 10. Ve třetím řádku je posuvník přidán do tzv. **ChangeListeneru**, pomocí kterého lze zjišťovat polohu jezdce posuvníku a který tak zajistí funkčnost tohoto prvku. Ve čtvrtém řádku jsou zadány souřadnice pro umístění posuvníku do GUI, které určují že bude umístěn vedle tlačítka „Otevřít“ pro otvírání dialogu načtení a že jeho velikost bude 100 x 22 pixelů (viz. obr.4.9). V pátém řádku je posuvník přidán do komponentu **horniPanel**.

Aby byla zajištěna funkčnost posuvníku, musí být ještě implementována třída **ChangeListener**. To se provede zápisem za klíčové slovo **implements** k ostatním interaktivním metodám v deklaraci veřejné třídy, která pak bude vypadat takto:

```
public class Zobrazeni3D extends java.applet.Applet implements MouseListener,
MouseWheelListener, MouseMotionListener, ActionListener, ChangeListener {. . .}
```

Je také vyžadována povinná implementace abstraktní metody **stateChanged a** příslušné balíčky:

```
public void stateChanged(ChangeEvent e) { }
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
```

Jejich načtení lze opět zjednodušit použitím zástupného symbolu *:

```
import javax.swing.event.*;
```

Nyní je vše připraveno pro programování metody **stateChanged**. Nejdříve se větvením programu a pomocí metody **getSource** zjistí zda byl původcem akce posuvník s názvem **ovlVyhlazeni**. Pak se do proměnné s názvem **hodnota**, která je typu **integer**, zapíše aktuální hodnota polohy jezdce posuvníku a to pomocí metody **getValue**. Nakonec se hodnota této proměnné odešle do metody s názvem **vyhlad**:

```
if (e.getSource() == ovlVyhlazeni) {
  int hodnota = ovlVyhlazeni.getValue();
  vyhlad(hodnota);
}
```

Metoda s názvem **vyhlad** je vlastní a je vytvořena z toho důvodu, aby mohla být volána jak z metody **stateChanged**, tak i po načtení obrázku, kvůli zobrazení 3D modelu s povrchem vyhlazeným dle inicializační hodnoty. Toto druhé volání metody musí být provedeno po ukončení všech výpočtů a uložení hodnot vyhlazeného povrchu do paměťových smyček. Metoda **vyhlad** je tedy podruhé volána z abstraktní metody **actionPerformed**, kde v posledním větvení nahradí volání pro vytvoření scény: `zobrazeni.addBranchGraph(vytvorScenu());` které se musí přesunout právě na konec nově vytvořené metody **vyhlad**:

```
vyhlad(ovlVyhlazeni.getValue());
```

Celá metoda s názvem **vyhlad**, která slouží k ovládání vyhlazení povrchu bude vypadat takto:

```
private void vyhlad(int v) {
  if (nazevObrazku != null) {
    if (v > 0){
      vystredeniZ = korekceZ[v - 1];
      for (int i = 0; i < pocetPixelu; i++)
        normPixely[i] = pomPixely[i][v - 1];
    }
    else if (v == 0) {
      vystredeniZ = memoZ;
      for (int i = 0; i < pocetPixelu; i++)
        normPixely[i] = zalozniPixely[i];
    }
    zobrazeni.addBranchGraph(vytvorScenu());
  }
```

V prvním řádku v těle metody se větvením zjistí, zda je načítán nějaký obrázek. Ve druhém řádku se větvením zjišťuje, zda hodnota proměnné **hodnota**, která byla přečtena z posuvníku nebo odeslána po novém načtení obrázku je větší než nula. Pokud ano, pak se ve třetím řádku načte hodnota pro posun 3D modelu na střed souřadného systému, která odpovídá pozici jezdce na posuvníku *n* a tím pádem také *n*-krát provedenému vyhlazení povrchu. Ve čtvrtém řádku je smyčka, která v pátém řádku naplní pole **normPixely** hodnotami *n*-krát provedeného vyhlazení povrchu. Na sedmém řádku pokračuje větvení v případě že je zjištěno, že hodnota z posuvníku je rovna nule. V tom případě je na osmém řádku načten posun v ose Z, který byl v záložní proměnné a na devátém řádku je smyčka, která na desátém řádku naplní pole **normPixely** původními hodnotami, které jsou v záložním poli. Na konci metody je volání metody pro vytvoření scény, které sem bylo přepsáno z metody **actionPerformed**.

5.5 Interaktivní ovládání hloubky 3D zobrazení

Podobně jako v části 4.5.4 je pro ovládání hloubky 3D zobrazení použitý posuvník. Ten je opět třeba nejdříve deklarovat na začátku programu:

```
private JSlider ovlHloubky = null;
```

Posuvník je nazván **ovlHloubky** a jeho načtení a základní nastavení je opět provedeno v metodě **inicializaceKomponentu**():

```
ovlHloubky = new JSlider();
ovlHloubky.setValue(100);
ovlHloubky.setMaximum(200);
ovlHloubky.addChangeListener(this);
ovlHloubky.setBounds(168, 4, 100, 22);
horniPanel.add(ovlHloubky);
```

Oproti posuvníku **ovlVyhlazeni** z části 4.5.4, zde nastaly změny pouze ve druhém řádku, kde je inicializační hodnota nastavena na 100 (což zde znamená 100%), dále ve třetím řádku je maximální rozsah posuvníku nastaven na 200 a v pátém řádku jsou nastaveny souřadnice tak, aby byl posuvník umístěn do GUI tak jak je to zachyceno na obr.4.10.

Odezva na akci se opět musí naprogramovat v metodě **stateChanged**. Větvením programu a pomocí metody **getSource** se zjistí zda byl původcem akce posuvník s názvem **ovlHloubky**. Do proměnné typu **integer** s názvem **hodnota**, se zapíše aktuální hodnota polohy jezdce posuvníku pomocí metody **getValue**. Nakonec se hodnota této proměnné odešle do metody **modeluj**:

```
if (e.getSource() == ovlHloubky) {
  int hodnota = ovlHloubky.getValue();
  modeluj(hodnota);
}
```

Metoda s názvem **modeluj** je vlastní a je vytvořena z důvodu, aby mohla být volána jak z metody **stateChanged**, tak i po načtení obrázku z metody **vyhlad**, kde nahradí volání pro vytvoření scény: `zobrazeni.addBranchGraph(vytvorScenu());` které se přesune na konec metody **modeluj**:

```
modeluj(ovlHloubky.getValue());
```

Celá metoda s názvem **modeluj**, která bude sloužit pro ovládání vyhlazování povrchu 3D modelu pak bude vypadat takto:

```
private void modeluj(int m) {
    if (nazevObrazku != null) {
        hloubka = m / 100.0f;
        zobrazeni.addBranchGraph(vytvorScenu());
    }
}
```

Proměnná **hloubka**, jejíž hodnota bude po zpracování v metodě nabývat hodnot 0,0 až 2,0 s krokem 0,1, může být nyní použita přímo v metodě **vytvorPovrch** pro řízení hodnoty zobrazení bodu Z ve třídě **Point3f**. Nejdříve je však třeba deklarovat ji jako datovou proměnnou:

```
private float hloubka = 0;
```

Nyní již zbývá pouze upravit příslušné řádky v metodě **vytvorPovrch**, tak aby do výpočtu souřadnice **z** byla zahrnuta i změna 3D hloubky, řízená proměnnou **hloubka**:

```
float z = (vystredeniZ - normPixely[indexBodu]) * hloubka;
```

6 Realizace funkčního appletu OptickyDisk1

6.1 Návrh GUI

Budou použity dva dialogy nabídky. První s názvem **dialog** je již naprogramován v appletu **Zobrazeni3D** a bude sloužit k načtení topografických obrázků, druhý s názvem **dialogF** bude nově naprogramován a bude sloužit k načtení obrázků z fundus kamery:

```
private JFileChooser dialogF = null;
```

Podobně je tomu u atributů šířky a výšky obrázků, počtu pixelů v obrázcích, tlačítek pro vyvolání nabídky, a řetězců pro názvy obrázků. Písmeno F za názvem atributu či komponentu znamená, že ten slouží pro práci s obrázkem z fundus kamery:

```
private int sirkaObrazkuF = 0;
private int vyskaObrazkuF = 0;
private int pocetPixeluF = 0;
private Button tlacitkoF = null;
private String nazevObrazkuF = null;
```

Musí se také deklarovat nové komponenty a proměnné, které v appletu **Zobrazeni3D** nebyly:

```
private JRadioButton prepinac = null;
private Button napoveda = null;
private JSlider ovlVelikosti = null;
private Color obarveni = null;
private Label stavVyhlazeni = null;
private Label stavHloubky = null;
private Label stavVelikosti = null;
private Label popisVyhlazeni = null;
private Label popisHloubky = null;
private Label popisVelikosti = null;
private Label popisPrepinace = null;
private float[] cervena = null;
private float[] zelena = null;
private float[] modra = null;
private boolean priznakPrepinace = false;
private int pravaMys = 0;
private int posun1 = 0;
private int posun2 = 0;
```

Komponent třídy **JRadioButton** s názvem **prepinac** se bude využívat k přepínání zobrazení povrchu 3D modelu (šedotónový/barevný). Tlačítko třídy **Button** s názvem **napoveda** bude sloužit k zobrazení okna s nápovědou. Posuvník třídy **JSlider** s názvem **ovlVelikosti** se bude používat paralelně s kolečkem myši k nastavování změny velikosti (tj. přiblížení/oddálení) obrázku. Proměnná třídy **Color** s názvem **obarveni** bude využita k obarvování komponentů, aby měly stejnou barvou. Komponenty třídy **Label**, které mají v názvu předponu **stav-** budou použity jako displeje k zobrazení hodnot nastavení (tj. velikosti vyhlazení povrchu, 3D rozměru v ose Z, přiblížení/oddálení) a ty které mají v názvu předponu **popis-** jsou použity pro popis komponentů. Proměnné typu **float** s názvy **cervena**, **zelena**, **modra** již byly použity a budou v nich opět uloženy složky barvy pro správné zobrazení barevného povrchu. Proměnná typu **boolean** s názvem **priznakPrepinace** bude sloužit k ovládání a zjišťování stavu komponentu **prepinac**. Poslední proměnné typu **integer** budou sloužit jako pomocné hodnoty pro ovládání appletu pomocí myši. Dále dojde také k několika změnám v metodě **inicializaceKomponentu**. Nejdříve musí být načteny nové komponenty:

```
dialogF = new JFileChooser();
tlacitkoF = new Button();
napoveda = new Button();
prepinac = new JRadioButton();
ovlVelikosti = new JSlider();
Panel bocniPanel = new Panel();
```

Pomocí metody **setLabel** jsou do komponentů vepsány nápisy, které vyjádří jejich funkci:

```
tlacitko.setLabel("Načíst");
tlacitkoF.setLabel("Načíst");
napoveda.setLabel("Nápověda");
```

Pro upřesnění funkce ostatních komponentů budou vytvořeny pomocné nápisy pomocí třídy **Label**:

```
stavVyhlazeni = new Label("5x");
stavHloubky = new Label("100%");
stavVelikosti = new Label("100%");
popisVyhlazeni = new Label("Vyhlazení povrchu: ");
popisHloubky = new Label("Hloubka 3D: ");
popisVelikosti = new Label("Velikost obrázku: ");
popisPrepinace = new Label("Barevný povrch");
Label popisT = new Label("Načíst topografický obraz:");
Label popisF = new Label("Načíst fundus obraz:");
```

Posuvníkům se metodou **`setValue`** resp. **`setMaximum`** nastaví inicializační hodnoty a rozsahy:

```
ovlVyhlazeni.setValue(5);
ovlVyhlazeni.setMaximum(10);
ovlVelikosti.setValue(0);
ovlVelikosti.setMaximum(40);
ovlHloubky.setValue(10);
ovlHloubky.setMaximum(20);
```

Pro oddělení komponentů v okně se použijí oddělovače třídy **`JSeparator`** (bude jich zde 6):

```
JSeparator oddelovac1 = new JSeparator(); . . .
. . .JSeparator oddelovac6 = new JSeparator();
```

Musí být také nastaven filtr souborů u dialogu nabídky pro načítání obrázku z fundus kamery, aby bylo možné načítat jen obrázky typu PNG. Také musí být nové tlačítko, dialog nabídky, nápověda, ovladač velikosti zobrazení, komponent **`prepinac`** a jeho popis, přidány do metod pro obsluhu událostí:

```
dialogF.setAcceptAllFileFilterUsed(false);
dialogF.setFileFilter(filtr);
dialogF.addActionListener(this);
tlacitkoF.addActionListener(this);
napoveda.addActionListener(this);
prepinac.addActionListener(this);
popisPrepinace.addMouseListener(this);
ovlVelikosti.addChangeListener(this);
```

Dalším krokem je zadávání souřadnic, které určí kde v okně mají být tyto komponenty umístěny:

```
bocniPanel.setBounds(512, 0, 156, 512);
popisT.setBounds(6, 0, 141, 15);
tlacitko.setBounds(43, 19, 70, 25);
oddelovac1.setBounds (10, 49, 136, 1);
popisF.setBounds(21, 55, 114, 15);
tlacitkoF.setBounds(43, 74, 70, 25);
oddelovac2.setBounds(10, 104, 136, 1);
popisVelikosti.setBounds(3, 110, 105, 15);
stavVelikosti.setBounds(105, 110, 48, 15 );
ovlVelikosti.setBounds(3, 125, 150, 25);
oddelovac3.setBounds(10, 154, 136, 1);
popisHloubky.setBounds(3, 160, 95, 15);
stavHloubky.setBounds(95, 160, 58, 15);
ovlHloubky.setBounds(3, 175, 150, 25);
oddelovac4.setBounds(10, 204, 136, 1);
popisVyhlazeni.setBounds(3, 210, 117, 15);
stavVyhlazeni.setBounds(120, 210, 33, 15);
ovlVyhlazeni.setBounds(3, 225, 150, 25);
oddelovac5.setBounds(10, 254, 136, 1);
prepinac.setBounds(24, 260, 16, 20);
popisPrepinace.setBounds(40, 260, 90, 20);
oddelovac6.setBounds(10, 284, 136, 1);
napoveda.setBounds(43, 480, 70, 25);
pozadi.setBounds(0, 0, 512, 512);
```

Po zadání souřadnic je nutné každý komponent vložit do panelu, ve kterém má být zobrazován:

```
bocniPanel.setLayout(null);
bocniPanel.add(popisT);
bocniPanel.add(tlacitko);
bocniPanel.add(oddelovac1);
bocniPanel.add(popisF);
bocniPanel.add(tlacitkoF);
```

```
bocniPanel.add(oddelovac2);
bocniPanel.add(popisVelikosti);
bocniPanel.add(stavVelikosti);
bocniPanel.add(ovlVelikosti);
bocniPanel.add(oddelovac3);
bocniPanel.add(popisHloubky);
bocniPanel.add(stavHloubky);
bocniPanel.add(ovlHloubky);
bocniPanel.add(oddelovac4);
bocniPanel.add(popisVyhlazeni);
bocniPanel.add(stavVyhlazeni);
bocniPanel.add(ovlVyhlazeni);
bocniPanel.add(oddelovac5);
bocniPanel.add(prepinac);
bocniPanel.add(popisPrepinace);
bocniPanel.add(oddelovac6);
bocniPanel.add(napoveda);
hlavniPanel.add(bocniPanel);
```

Posledním krokem je obarvení komponentů tak, aby měly všechny stejnou, světle šedou barvu:

```
obarveni = Color.getHSBColor(0f, 0f, 0.935f);
bocniPanel.setBackground(obarveni);
popisT.setBackground(obarveni);
popisF.setBackground(obarveni);
popisVyhlazeni.setBackground(obarveni);
stavVyhlazeni.setBackground(obarveni);
popisHloubky.setBackground(obarveni);
stavHloubky.setBackground(obarveni);
popisPrepinace.setBackground(obarveni);
popisVelikosti.setBackground(Color.YELLOW);
stavVelikosti.setBackground(Color.YELLOW);
ovlVelikosti.setBackground(Color.YELLOW);
```

6.2 Návrh interaktivity

K načtení topografického obrázku je využita metoda **actionPerformed** a tlačítko s názvem **tlacitko** z předchozího appletu **Zobrazeni3D**. Jedinou změnou v algoritmu je odstranění řádku: **zmenaVelikosti = 1;** protože změna velikosti obrázku bude nyní řízena i posuvníkem a její hodnota bude zobrazena na příslušném displeji. Pro načtení obrázku z fundus kamery bude použito další tlačítko a do metody **actionPerformed** musí být dopsána obsluha tohoto tlačítka a dialogu:

```
else if (e.getSource() == tlacitkoF)
    dialogF.showOpenDialog(this);
else if (e.getSource() == dialogF) {
    String prikaz = e.getActionCommand();
    if (prikaz.equals(„ApproveSelection“) {
        java.io.File soubor = dialogF.getSelectedFile();
        nazevObrazkuF = soubor.toString();
        rotaceX = 0;
        rotaceY = 0;
        posunX = 0;
        posunY = 0;
        cteniBarvy();
    }
}
```

Na konci obsluhy dialogu pro načtení obrázku z fundus kamery jsou nastaveny nulové hodnoty u dvou nových proměnných s názvy **posunX** a **posunY**. Tyto proměnné budou použity pro posun 3D modelu po osách X a Y pomocí pravého tlačítka myši a budou popsány na konci kapitoly, při programování tohoto posunu. Zde se jejich hodnoty nastaví na hodnotu 0, aby při načítání dalších obrázků nebyly tyto obrázky posunuté v osách X a Y do stran, ale zobrazily se na středu 3D oblasti. Tyto dva řádky jsou z téhož důvodu připsány i do obsluhy dialogu pro načtení topografického obrázku. Dále je volána nová vlastní metoda s názvem **cteniBarvy**, kde se získají informace o barvě povrchu obrázku. Tato metoda slouží k načtení obrázku z fundus kamery. V metodě **actionPerformed** musí být dále provedena také obsluha komponentu **JRadioButton** s názvem **prepinac**:

```
else if (e.getSource() == prepinac) {
  if (priznakPrepinace == true)
    priznakPrepinace = false;
  else if (priznakPrepinace == false)
    priznakPrepinace = true;
  if (sirkaObrazku == sirkaObrazkuF && vyskaObrazku == vyskaObrazkuF){
    if (nazevObrazku != null && nazevObrazkuF != null)
      zobrazeni.addBranchGraph(vytvorScenu());
  }
  else {
  JOptionPane.showMessageDialog(null,"Rozměry obrázků:
  "+vyskaObrazku+"x"+sirkaObrazku+"px a " +vyskaObrazkuF+"x"+sirkaObrazkuF+"
  px nejsou shodné !","Špatný rozměr obrázku !", JOptionPane.WARNING_MESSAGE);
  }
}
```

V prvním řádku se zjišťuje zda je událost způsobena tímto komponentem. Pokud ano, pak se ve druhém řádku zjišťuje hodnota příznaku tohoto komponentu, která pak bude přímo obsluhovat vykreslování povrchu obrázku. Ve třetím řádku je tedy tato hodnota převrácena, jako reakce na změnu stavu komponentu. Totéž platí analogicky pro řádek čtvrtý a pátý. V šestém řádku je větvením testováno, zda jsou stejné rozměry obrázků (topografického a z fundus kamery). Pokud ano, tak se v sedmém řádku zjistí zda jsou vůbec nějaké obrázky načteny a v případě že ano, tak se na osmém řádku volá metoda pro zobrazení 3D modelu. Pokud by z nějakého důvodu byly rozdílné rozměry obrázků, je na desátém řádku voláno zobrazení výstražného okna, ale žádná jiná akce není provedena.

Poslední úpravou metody je obsluha tlačítka nápovědy. Řetězec se znakem **\n** značí zalamování řádku. Kvůli množství textu zde není uveden celý kód, ale je pouze naznačena struktura psaní jednotlivých vět se zalamováním řádků:

```
else if (e.getSource() == napoveda){JOptionPane.showMessageDialog(null,
  "Ovládání obrázku lze provádět pomocí myši, nebo pomocí ovládacích prvků na
     bočním panelu."+"\n"+ . . .
   . . . "Nápověda", JOptionPane.QUESTION_MESSAGE);
}
```

Další metodou zajišťující interaktivitu je metoda **stateChanged**. Ta byla opět již naprogramována v appletu **Zobrazeni3D** v části. Do algoritmu pro nastavení vyhlazení resp. 3D hloubky se pouze přidá řádek pro zobrazení aktuální hodnoty na displeji:

```
stavVyhlazeni.setText(hodnota+"x");
```

resp. `stavHloubky.setText((hodnota * 10)+"%");`

Musí být ale ještě přidána metoda pro ovládání změny velikosti obrázku:

```
if (e.getSource() == ovlVelikosti) {
        int hodnota = ovlVelikosti.getValue();
        stavVelikosti.setText(((hodnota * 5) + 100)+"%");
        zmenaVelikosti = ((float)hodnota / 20.0f) + 1.0f;
        if (scena != null)
            transformujObraz();
}
```

První tři řádky pro získání hodnoty z posuvníku a zobrazení na displeji jsou analogické jako u ostatních algoritmů. Na čtvrtém řádku však není volána další metoda jako u ostatních algoritmů, ale je zde vypočítána hodnota proměnné **zmenaVelikosti**, která přímo řídí změnu velikosti obrázku v metodě **transformujObraz**. Tato metoda je volána až na posledním řádku, a to pouze v případě, je-li větvením programu zjištěno, že je vytvořena nějaká 3D scéna.

Aby bylo možné ovládat applet pouze myší, bude použita doposud nevyužitá abstraktní metoda **mouseClicked**. V té bude naprogramováno ovládání posuvníků kolečkem myši, ovládání komponentu **prepinac** pomocí kliknutí na jeho upřesňující popis s názvem **popisPrepinace**, a také přepínání mezi jednotlivými posuvníky, k čemuž bude využito pravé tlačítko myši. Nejdříve tedy bude metodou **getButton** zjištěno na které tlačítko myši bylo kliknuto. Podle navrácené hodnoty se zjistí zda se jedná o levé tlačítko (hodnota 1), nebo o pravé tlačítko (hodnota 3). Levým tlačítkem se bude ovládat pouze přepínání komponentu **prepinac**, takže pokud se v dalším větvení zjistí že bylo kliknuto na popis tohoto komponentu, změní se příkazem **doClick** hodnota (**true**, **false**) vracená komponentem **prepinac**:

```
int stiskTlacitka = e.getButton();
  if (stiskTlacitka == 1) {
    if (e.getSource() == popisPrepinace)
        prepinac.doClick();
  }
```

Pokud ovšem bude hodnota vrácená metodou **getButton** rovna číslu 3, bude se jednat o kliknutí na pravé tlačítko myši. Tím bude realizováno přepínání mezi jednotlivými posuvníky. Protože je do obsluhy událostí myši **MouseListener** přidán i komponent **popisPrepinace**, je třeba jej eliminovat, protože přepínání mezi posuvníky by jinak fungovalo i při kliknutí na tento komponent. Eliminace se provede v prvním větvení zároveň s rozpoznáním tlačítka. Samotné rozpoznání, přepínání a obarvení posuvníků bude provádět v nové metodě s názvem **prepni**:

```
if (stiskTlacitka == 3 && e.getSource() != popisPrepinace) {
    prepni();
}
```

V nové metodě s názvem **prepni** je pro rozpoznání posuvníků využito větvení a pomocná proměnná s názvem **pravaMys**:

```
if (pravaMys == 0) {
    popisVyhlazeni.setBackground(obarveni);
    stavVyhlazeni.setBackground(obarveni);
    ovlVyhlazeni.setBackground(obarveni);
    popisHloubky.setBackground(Color.YELLOW);
    stavHloubky.setBackground(Color.YELLOW);
    ovlHloubky.setBackground(Color.YELLOW);
    popisVelikosti.setBackground(obarveni);
    stavVelikosti.setBackground(obarveni);
    ovlVelikosti.setBackground(obarveni);
    pravaMys = 1;
}
else if (pravaMys == 1) { . . .
    . . .pravaMys = 2;
}
else if (pravaMys == 2) { . . .
    . . .pravaMys = 0;
}
```

Posuvník který lze ovládat pomocí kolečka myši, je obarven žlutou barvou, pomocí hodnoty **Color.YELLOW**. Pokud je rozpoznáno kliknutí na pravé tlačítko, je podle hodnoty proměnné **pravaMys** provedeno obarvení následujícího komponentu a hodnota proměnné se změní na hodnotu příslušející následujícímu komponentu. Defaultně je obarven posuvník pro změnu velikosti měřítka a hodnota proměnné **pravaMys** je 0. Při prvním kliknutí je obarven posuvník pro změnu hloubky 3D a hodnota proměnné **pravaMys** je 1. Při třetím kliknutí je obarven posuvník pro vyhlazení povrchu a hodnota proměnné **pravaMys** je 2. Při dalším kliknutí je hodnota proměnné **pravaMys** změněna zpět na původní hodnotu 0 a obarven je opět posuvník pro změnu velikosti měřítka. Takto lze mezi posuvníky přepínat pravým tlačítkem, je-li kursor v oblasti pro 3D zobrazováni s názvem **pozadi**. Ve výše uvedeném výpisu programu je celý kód obarvení komponentů zobrazen pouze v prvním větvení, z důvodu délky kódu. V dalších dvou větveních je samozřejmě také vždy provedeno obarvení příslušných komponentů.

Poslední úprava nastane v metodě pro ovládání kolečkem myši s názvem **mouseWheelMoved**. Opět se vychází ze stejné metody naprogramované v předchozím appletu **Zobrazeni3D**. Ovšem ihned po získání hodnoty s názvem **rotace**, navrácené při pohybu kolečka, bude provedeno volání nové vlastní metody s názvem **kolecko**, která jako argument bude posílat proměnnou **rotace**.

```
kolecko(rotace);
```

Do metody **kolecko** se přesune algoritmus pro změnu velikosti obrázku, který byl původně v metodě **mouseWheelMoved**. V tomto algoritmu ovšem bude provedeno několik úprav. První spočívá ve větvení programu, aby se zjistilo který komponent je aktivní pro ovládání kolečkem myši. Je zde opět použita proměnná **pravaMys**, pomocí níž se tato skutečnost zjistí. Je-li její hodnota 0, pak je aktivní funkce pro změnu velikosti obrázku. Z posuvníku se tedy nejdříve pomocí metody **getValue** přečte aktuální hodnota, která je přepočítána tak aby mohla být použita dále pro zpracování (tj. aby

nabývala hodnot v rozsahu **1.0f** až **3.0f**. Dále je použitý původní kód v němž se připočítává hodnota při pootočení kolečkem myši a jsou stanoveny hranice pro minimum a maximum. Z proměnné **zmenaVelikosti** je pak vypočtena pomocná proměnná s názvem **hodnota**, která je použita pro zobrazení aktuální hodnoty na displeji metodou **setText**, a pro nastavení jezdce posuvníku metodou **setValue**. Hodnota proměnné **zmenaVelikosti** je nejdříve vynásobena hodnotou 100, čímž je převedena na procentuelní vyjádření. Proměnná je dále zaokrouhlena příkazem **Math.round**, aby nebyla zobrazována desetinná místa. Nakonec je k aktuálně zobrazované hodnotě přidán znak: %. Je-li vytvořena scéna, volá se na závěr metoda **transformujObraz**, tak jako v původním algoritmu:

```
if (pravaMys == 0) {
    zmenaVelikosti = (ovlVelikosti.getValue() * 0.05) + 1; . . .
    . . .int hodnota = (int)Math.round(zmenaVelikosti * 100.0f);
         stavVelikosti.setText(hodnota + "%");
         ovlVelikosti.setValue((hodnota - 100) / 5); . . .
. . . }
```

Velmi podobně jsou vytvořeny také algoritmy, které ovládají hloubku 3D a vyhlazení povrchu:

```
else if (pravaMys == 1) {
    posun1 = ovlHloubky.getValue();
    posun1 = posun1 + rotace;
    if (posun1 < 0)
        posun1 = 0;
    if (posun1 > 20)
        posun1 = 20;
    stavHloubky.setText((posun1 * 10)+"%");
    ovlHloubky.setValue(posun1);
    modeluj(ovlHloubky.getValue());
}
```

V prvním řádku je větvení, kde se pomocí proměnné **pravaMys** zjistí, který komponent je aktivní pro ovládání kolečkem myši (1 = hloubka 3D, resp. 2 = vyhlazení povrchu). Ve druhém řádku se do proměnné (**posun1**, resp. **posun2**) metodou **getValue** načte aktuální hodnota z posuvníku. Ta se ve třetím řádku přepočítá, aby mohla být použita pro zpracování. V dalších čtyřech řádcích jsou pomocí větvení stanoveny hranice minima a maxima (0 až 20, resp. 0 až 10). Pak se pomocí metody **setText** zobrazí aktuální hodnota na displeji a metodou **setValue** se nastaví jezdec posuvníku. Nakonec je volána příslušná metoda pro zpracování (**modeluj**, resp. **vyhlad**).

Pro posun 3D modelu po ploše v ose X a Y bude využito stisknutí a držení pravého tlačítka myši. Pro tuto funkci se musí upravit kód v metodě **mouseDragged**. Původní kód zůstane zachován, s výjimkou posledního větvení a volání metody **transformujObraz**. Větvení se musí přesunout na začátek celé metody, aby metoda fungovala pouze je-li vytvořena scéna. Volání metody **transformujObraz** se naopak přesune na konec celého kódu, protože bude využíváno i nově přidaným kódem, který bude ovládat posun 3D modelu po ploše v ose X a Y. Před toto volání a za původní kód, který slouží pro ovládání rotace, bude přidán následující kód:

```
if (priznakPosunu == true) {
  if (poziceX < docasnaX) {
      docasnaX = poziceX;
      posunX = posunX - 0.01f;
  }
  if (poziceX > docasnaX) {
      docasnaX = poziceX;
      posunX = posunX + 0.01f;
  }
  if (poziceY < docasnaY) {
      docasnaY = poziceY;
      posunY = posunY - 0.01f;;
  }
  if (poziceY > docasnaY) {
      docasnaY = poziceY;
      posunY = posunY + 0.01f;
  }
  vektorPosunu.set(posunX, posunY, 0);
}
```

V prvním řádku se pomocí proměnné typu **boolean** s názvem **priznakPosunu** zjišťuje zda je posun 3D modelu po ploše povolen. Tato proměnná musí být deklarována jako datová proměnná:

```
private boolean priznakPosunu = false;
```

V dalších řádcích se větvením a pomocí dočasných hodnot určuje směr posunu. Následně jsou o hodnotu **0.01f** inkrementovány resp. dekrementovány proměnné typu **float** s názvem **posunX** a **posunY**, které řídí posun. Tyto proměnné musí být deklarovány jako datové proměnné:

```
private float posunX = 0;
private float posunY = 0;
```

Hodnota **0.01f** upravuje rychlost posunování tak, aby odpovídala rychlosti pohybu kurzoru po ploše. Hodnoty proměnných **posunX** a **posunY** jsou nakonec pomocí metody **set** zapsány do vektoru třídy **Vector3f** s názvem **vektorPosunu**, který bude v metodě **transformujObraz** řídit posun a který musí být také deklarován na začátku programu:

```
private Vector3f vektorPosunu = null;
```

Do metody **transformujObraz** je přidána transformační funkce s názvem **posun**, pro posun 3D modelu po ploše pomocí metody **setTranslation** a musí být přidána do transformační skupiny:

```
Transform3D posun = new Transform3D();
posun.setTranslation(vektorPosunu);
posun.mul(lupa);
```

Také do metody **vytvorScenu** musí být přidán řádek, který zajistí nastavení hodnot vektoru s názvem **vektorPosunu**, aby i při načtení obrázku byl vektor definován:

```
vektorPosunu = new Vector3f(posunX, posunY, 0);
```

Posledním krokem je úprava metod **mousePressed** a **mouseReleased**. V nich se musí zajistit, aby posun 3D modelu po ploše byl možný pouze při stisku pravého tlačítka myši. Do metody **mousePressed** bude přidáno větvení, které při stisku pravého tlačítka povolí možnost posunování:

```
if (stisk == 3)
    priznakPosunu = true;
```

Analogicky se do metody **mouseReleased** přidá větvení pro zákaz posunu při uvolnění tlačítka:

```
if (stisk == 3)
    priznakPosunu = false;
```

Pokud by však bylo nutné ovládat applet bez pomoci myši, musí se implementovat ovládání pomocí klávesnice, což zajišťují metody třídy **KeyListener**.:

```
public class Zobrazeni3D extends java.applet.Applet implements MouseListener,
MouseWheelListener,  MouseMotionListener,  ActionListener,  ChangeListener,
KeyListener { . . .}
```

Povinné metody pro obsluhu událostí ve třídě **KeyListener** jsou:

```
public void keyTyped(KeyEvent e) { }
public void keyPressed(KeyEvent e) { }
public void keyReleased(KeyEvent e) { }
```

Aby ovládání pomocí kláves bylo vždy funkční, musí se komponenty přidat do obsluhy událostí:

```
pozadi.addKeyListener(this);
tlacitko.addKeyListener(this);
tlacitkoF.addKeyListener(this);
napoveda.addKeyListener(this);
```

Metody **keyTyped** a **keyReleased** zůstanou nevyužité a programovat se bude pouze metoda **keyPressed**. V té bude nejdříve větvením zajištěno, aby klávesy pro ovládání fungovaly pouze pokud je vytvořena scéna. Pak bude pomocí metody **getKeyCode** zjištěn kód stisknuté klávesy a pomocí větvení budou klávesy rozděleny do skupin které volají stejnou metodu pro zpracování:

```
if (scena != null) {:
    int kod = e.getKeyCode();
    if (kod > 32 && kod < 41) { . . .
    . . .}
    if (kod == 81 ^ kod == 87) { . . .
    . . .}
    if (kod == 32) {
        prepni();
    }
}
```

V posledním větvení je volána metoda **prepni**, protože hodnota 32 je kód pro klávesu Space (mezerník). Touto klávesou tedy bude možné přepínat mezi posuvníky, stejně jako kliknutím na pravé tlačítko myši. Druhé větvení bude aktivní, pokud bude stisknuta klávesa s kódem 81, nebo s kódem 82. Hodnota 81 je kód pro klávesu Q a hodnota 82 je kód pro klávesu W:

```
int hodnota = 0;
if (kod == 81)
    hodnota--;
if (kod == 87)
    hodnota++;
kolecko(hodnota);
```

Na konci tohoto větvení je volána metoda **kolecko** a je do ní předávána jako argument proměnná **hodnota**, která je nejdříve deklarována jako lokální proměnná typu **integer** s hodnotou 0 a následně inkrementována (resp. dekrementována) při stisku klávesy Q (resp.W). Tento algoritmus slouží jako emulace kolečka myši pomocí dvou kláves. První větvení bude aktivní, pokud bude stisknuta některá z kláves, které přísluší kódy v rozsahu 33 až 40. Skupina kláves v tomto prvním větvení je určena pro ovládání rotace a posunování 3D modelu ve 3D prostředí. Rotace a posunování jsou ovládány pomocí příslušných transformačních funkcí, které jsou v metodě **transformujObraz,** takže na konci větvení je tato metoda volána. Protože posunování 3D modelu po ploše je řízeno pomocí vektoru s názvem **vektorPosunu**, jsou ještě před voláním metody **transformujObraz** nastaveny pomocí metody **set** aktuální souřadnice tohoto vektoru:

```
if (kod == 37)
    rotaceX = rotaceX - (Math.PI / 90);
if (kod == 38)
    rotaceY = rotaceY - (Math.PI / 90);
if (kod == 39)
    rotaceX = rotaceX + (Math.PI / 90);
if (kod == 40)
    rotaceY = rotaceY + (Math.PI / 90);
if (kod == 33)
    posunY = posunY + 0.01f;
if (kod == 34)
    posunY = posunY - 0.01f;
if (kod == 35)
    posunX = posunX + 0.01f;
if (kod == 36)
    posunX = posunX - 0.01f;
vektorPosunu.set(posunX, posunY, 0);
transformujObraz();
```

6.3 Další úpravy appletu

Aby bylo možné používat komponent třídy **JRadioButton** s názvem **prepinac**, který bude využíván k přepínání zobrazení povrchu 3D modelu (šedotónový/barevný), musí se v provést úprava v metodě pro vykreslování povrchu obrázku s názvem **vytvorPovrch**. V předchozím appletu s názvem **Zobrazeni3D**, byly vykreslovány pixely pouze v odstínech šedé, a to pomocí příkazu:

```
Color3f barva = new Color3f(normPixely[indexBodu], normPixely[indexBodu],
                            normPixely[indexBodu]);
```

Tento řádek kódu musí být nahrazen tak, aby bylo možné volit mezi vykreslením pixelů v odstínech šedé, nebo pomocí tří barevných složek ve formátu RGB. Řádek je tedy nahrazen ve všech čtyřech částech metody **vytvorPovrch** tímto zápisem:

```
float r = normPixely[indexBodu];
float g = normPixely[indexBodu];
float b = normPixely[indexBodu];
if (priznakPrepinace == true && sirkaObrazku == sirkaObrazkuF &&
    vyskaObrazku == vyskaObrazkuF) {
    r = cervena[indexBodu];
    g = zelena[indexBodu];
    b = modra[indexBodu];
}
Color3f barva = new Color3f(r , g , b);
```

Na prvních třech řádcích jsou nejdříve deklarovány proměnné typu **float** s názvy **r**, **g** a **b**, které jsou defaultně nastaveny pro zápis hodnot odstínů šedé. Na čtvrtém řádku je větvením programu zajištěno, aby v případě nastavení komponentu **prepinac** na hodnotu **true** a v případě shodných rozměrů obrázků, bylo možné použít tyto proměnné pro zápis hodnot s odstíny barvy, což je provedeno na následujících třech řádcích. Na posledním řádku je pomocí třídy **Color3f** určen odstín pixelu.

Další změna je v metodě **vypocty**, kde je na konec celé metody přidáno větvení. Tímto větvením je zajištěno, že při načítání topografického obrázku bude tento obrázek vždy zobrazen. To poslouží uživateli jako kontrola, že byl obrázek opravdu načten.

```
if (priznakPrepinace == true)
    prepinac.doClick();
```

Nakonec je vytvořena nová vlastní metoda s názvem **cteniBarvy**, která je volána z abstraktní metody **actionPerformed** Zde budou získány informace o barvě povrchu obrázku. Tato metoda je založena na metodě **vypocty** z appletu **VykresleniBarvy**. Rozdíl je v tom, že proměnné mají na konci názvu připsáno písmeno F, aby bylo jasné, že se jedná o práci s obrázkem z fundus kamery. Dalším rozdílem je, že po naplnění polí informacemi o barvě je přidáno větvení, které má za úkol v případě nenačtení topografického obrázku naplnit celé pole **normPixely** hodnotou **0.5f**, která odpovídá odstínu středně šedé barvy. Je to z toho důvodu, že bez hodnot v poli **normPixely** by nemohl být obrázek zobrazen a uživatel by neměl kontrolu že obrázek byl opravdu načten:

```
if (nazevObrazku == null) {
    normPixely = new float[pocetPixeluF];
    normPixely[i] = 0.5f;
}
```

Následující část metody **cteniBarvy**, která je modifikací metody **vypocty** z appletu **VykresleniBarvy**, je také pozměněna. Opět je nejdříve aplikováno větvení, které zjišťuje zda byl načten topografický obrázek. Pokud ne, pak se musí ještě dopočítat hodnoty posunutí obrázku na střed 3D oblasti zobrazení v rovině XY. Protože není načten topografický obrázek a nejsou tedy známé ani atributy šířky a výšky topografického obrázku, použijí se hodnoty obrázku z fundus kamery: Ze stejných hodnot se bude počítat i počet trojúhelníků v obrázku. Algoritmus pro výpočet hodnot je naprosto shodný s původním algoritmem v metodě **vypocty**. Poslední změna je na konci metody. V původní metodě **vypocty** bylo na konci provedeno stanovení minimální velikosti pixelu, pokud by jeho hodnota vycházela menší než 0. To zde není nutné, takže tato část je vypuštěna a je zde použito větvení, kterým je zajištěno že při načítání obrázku z fundus kamery bude tento obrázek vždy zobrazen, což poslouží uživateli jako kontrola, že byl obrázek opravdu načten. Nakonec je volána metoda pro zobrazení **addBranchGraph**(**vytvorScenu**()):

```
if (nazevObrazku == null) {
  sirkaObrazku = sirkaObrazkuF;
  vyskaObrazku = vyskaObrazkuF; . . .
. . .}
if (priznakPrepinace == false)
    prepinac.doClick();
zobrazeni.addBranchGraph(vytvorScenu());
```

7 Realizace funkčního appletu OptickyDisk2

7.1 Rozšíření GUI

Zobrazování os je ovládáno pomocí komponentu třídy **JRadioButton**, který je nazván **osy** a je popsán pomocí komponentu třídy **Label** s názvem **popisOs**. Ovládání voleb vykreslení je ovládáno pomocí komponentu třídy **Choice**, který je nazván **volba**. Komponenty musí být deklarovány:

```
private JRadioButton osy = null;
private Label popisOs = null;
private Choice volba = null;
```

Pro zajištění funkčnosti komponentu **volba**, musí být implementována třída **ItemListener**.

```
public class Zobrazeni3D extends java.applet.Applet implements MouseListener,
MouseWheelListener,  MouseMotionListener,  ActionListener,  ChangeListener,
KeyListener, ItemListener { . . .}
```

Je také vyžadována povinná implementace abstraktní metody **itemStateChanged**:

```
public void itemStateChanged(ItemEvent e) { }
```

V metodě **inicializaceKomponentu** je nyní třeba komponenty načíst a popsat. Navíc bude třeba načíst také jeden objekt třídy **JSeparator** pro oddělení obou komponentů a ještě jeden lokální objekt třídy **Label** pro popis komponentu **volba**. Do komponentu **volba** je navíc metodou **insert** vepsán text popisu volby a pak index, pod kterým bude příslušná volba volána:

```
osy = new JRadioButton();
volba = new Choice();
popisOs = new Label(" Zobrazit osy");
JSeparator oddelovac7 = new JSeparator();
Label popisVolby = new Label("Zobrazit: ");
volba.insert ("Body", 0);
volba.insert ("Čáry", 1);
volba.insert ("Síť", 2);
volba.insert ("Povrch", 3);
volba.insert ("Světlo", 4);
volba.select(0);
```

V posledním řádku byla také nastavena inicializační hodnota komponentu **volba**. To znamená že po načtení appletu bude na komponentu **volba** zvolena volba s indexem 0 tj. vykreslení bodů. Je také nutné nastavit koordináty, které určí umístění komponentů a komponenty je nutno přidat do ovládacího panelu. Komponenty **osy**, **popisVolby** a **popisOs** mají bílé pozadí, takže musí být obarveny:

```
osy.setBounds(165, 4, 35, 22);
popisOs.setBounds(61, 4, 38, 22);
oddelovac7.setBounds(10, 284, 136, 1);
popisVolby.setBounds(61, 4, 38, 22);
volba.setBounds(100, 4, 60, 22);
bocniPanel.add(osy);
bocniPanel.add(popisOs);
bocniPanel.add(oddelovac7);
bocniPanel.add(popisVolby);
bocniPanel.add(volba);
popisVolby.setBackground(obarveni);
popisOs.setBackground(obarveni);
osy.setBackground(obarveni);
```

Komponenty **osy**, **popisOs** a **volba** musí být ještě přidány do příslušné metody obsluhy událostí:

```
osy.addActionListener(this);
popisOs.addMouseListener(this);
volba.addItemListener(this);
```

7.2 Rozšíření interaktivity

Nejdříve je ošetřena možnost přepnutí komponentu **JRadioButton** s názvem **osy** pomocí kliknutí levým tlačítkem myši na popis s názvem **popisOs** u tohoto komponentu. To je provedeno v metodě **mouseClicked** pomocí větvení a metody **doClick**:

```
if (e.getSource() == popisOs)
    osy.doClick();
```

Do následujícího větvení musí být analogicky také přidána podmínka, která zajistí že při kliknutí pravým tlačítkem myši na komponent **popisOs** se nebudou přepínat posuvníky:

```
if (stiskTlacitka == 3 && e.getSource() != popisPrepinace &&
    e.getSource() != popisOs)
```

Dále je provedena obsluha událostí při změně stavu komponentu **osy** pomocí metody **actionPerformed**. Nejdříve se do stávajícího větvení vloží další řádek, který bude analogicky jako u ostatních větvení pomocí metody **getSource** zjišťovat zda byla událost vyvolána příslušným komponentem, zde komponentem **osy**. Pokud tedy bude na komponent kliknuto, v dalším řádku se pomocí metody **isSelected** načte do proměnné typu **boolean** s názvem **priznakOs** hodnota příslušející aktuálnímu stavu. Následně se volá metoda pro zobrazení scény, ve které bude dle hodnoty proměnné **priznakOs** větvením zjištěno, zda se mají osy přidat do scény.

```
else if (e.getSource() == osy) {
    priznakOs = osy.isSelected();
    if (scena != null)
       zobrazeni.addBranchGraph(vytvorScenu());
}
```

Proměnná **priznakOs** musí být deklarována na začátku programu jako datová proměnná:

```
private boolean priznakOs = false;
```

Nakonec musí být naprogramována metoda pro obsluhu událostí komponentu **volba**. Tou je metoda **itemStateChanged** a je do ní zapsán tento kód:

```
if (e.getSource() == volba) {
    if (scena != null)
       zobrazeni.addBranchGraph(vytvorScenu());
}
```

V prvním řádku je větvením zjištěno, zda byla událost vyvolána komponentem **volba**. Pokud byla, je dalším větvením zjištěno zda je vytvořena scéna. Pokud ano, volá se metoda pro zobrazeni **vytvorScenu**. V metodě **vytvorScenu** se ovšem musí upravit kód tak, aby metoda dokázala reagovat na změnu výběru v komponentu **volba**. Do metody je tedy přidán následující kód:

```
int vyber = volba.getSelectedIndex();
    if (vyber == 0)
        objekt.setGeometry(vytvorBody());
    else if (vyber == 1)
        objekt.setGeometry(vytvorCary());
    else if (vyber == 2)
        objekt.setGeometry(vytvorSit());
    else if (vyber == 3)
        objekt.setGeometry(vytvorPovrch());
    else if (vyber == 4)
        objekt.setGeometry(Osvetleni());
```

V prvním řádku se do proměnné s názvem **vyber** uloží pomocí metody **getSelectedIndex** hodnota indexu volby, která byla vybrána. V dalších pěti větveních je podle hodnoty této proměnné volána příslušná metoda, která zajistí požadované zobrazení.

7.3 Další úpravy programu

Nejdříve se vytvoří metoda **vytvorOsy.** Aby osy nepůsobily rušivě, je v objektu třídy **Appearance** nastavena průhlednost:

```
Appearance vzhled = new Appearance();
TransparencyAttributes pruhlednost =
            new TransparencyAttributes(TransparencyAttributes.FASTEST, 0.75f);
vzhled.setTransparencyAttributes(pruhlednost);
Color3f barva = new Color3f(1.0f, 1.0f, 1.0f);
```

Hodnota **0.75f** v závorce konstruktoru třídy **TransparencyAttributes** znamená, že osy budou průhledné ze 75%. Pomocí třídy **Color3f** je také definována barva os, která je zde bílá. Samotné osy budou vykresleny pomocí třídy **LineArray**:

```
LineArray X = new LineArray(2, LineArray.COORDINATES | GeometryArray.COLOR_3);
X.setCoordinate(0, new Point3f(-1.0f, 0.0f, 0.0f));
X.setCoordinate(1, new Point3f( 1.0f, 0.0f, 0.0f));
X.setColor(0, barva);
X.setColor(1, barva);
osaX = new Shape3D(X);
osaX.setAppearance(vzhled);
```

V prvním řádku je definován objekt třídy **LineArray** s názvem **X**, u kterého je v závorce konstruktoru zadán počet bodů pro kreslení čáry, je povoleno zadávání koordinátů a nastavení barvy čáry. V dalších dvou řádcích je zadán počáteční a koncový bod pro vykreslení a v následujících dvou řádcích je definována barva obou bodů a tím pádem i čáry. Nakonec je vytvořen objekt třídy **Shape3D** s názvem **osaX**, který bude sloužit k vykreslení a na který budou také aplikovány vlastnosti definované v objektu **vzhled**. Osy Y a Z jsou vytvořeny analogicky, pouze se změnou souřadnic v objektu třídy **Point3f**. Objekty třídy **Shape3D** musí být deklarovány na začátku programu:

```
private Shape3D osaX = null;
private Shape3D osaY = null;
private Shape3D osaZ = null;
```

V metodě pro zobrazení scény s názvem **vytvorScenu** bude dle hodnoty proměnné **priznakOs** větvením zjištěno, zda se mají osy přidat do scény:

```
if (priznakOs == true) {
    vytvorOsy();
    pohyb.addChild(osaX);
    pohyb.addChild(osaY);
    pohyb.addChild(osaZ);
}
```

Aby bylo možné vykreslovat povrch obrázku jako jednotlivé body pomocí události vyvolané komponentem **volba**, jsou použity metody **vytvorGeometrii** a **vytvorVzhled** z appletu Interaktivita. Metoda **vytvorGeometrii** se přejmenuje na **vytvorBody** a aby bylo možné vykreslit body barevně pomocí obrázku z fundus kamery, použijí se stejná větvení i stejný algoritmus pro deklaraci bodů a barev jako v metodě **vytvorPovrch**. Metoda **vytvorVzhled** zůstane beze změny. Je také nutné přidat do metody **vytvorScenu** řádek s voláním metody **vytvorVzhled**:

```
objekt.setAppearance(vytvorVzhled());
```

Dále je upraven konec metody **vypocty** tak jako v appletu **Interaktivita**. Tzn. opět je deklarována a vypočtena proměnná **meritko**, která určuje vzdálenost bodů od sebe. Stejně tak je upraven konec metody **cteniBarvy**, ve které je nový řádek, zajišťující výpočet proměnné **pocetPixelu**, nutné pro vykreslení bodů i v případě, že nebude načten topografický obrázek:

```
pocetPixelu = pocetPixeluF;
```

7.3.1 Vykreslení povrchu pomocí čar

Pro vykreslení povrchu pomocí čar, je vytvořena nová vlastní metoda **vytvorCary**. Základem je třída **LineArray**. Do jejího konstruktoru je nutné zadat kolik bodů je třeba pro vykreslení všech čar:

```
LineArray oblast = new LineArray(pocetCar, GeometryArray.COORDINATES |
                                           GeometryArray.COLOR_3);
```

Pro tento účel je na začátku programu deklarována proměnná typu **integer** s názvem **pocetCar**:

```
private int pocetCar = 0;
```

Počet bodů nutných pro vykreslení čar bude vypočítán v metodách **vypocty** a **cteniBarvy**. Výpočet vychází z DP obr.5.1, kde jsou použity všechny pixely 1x (**pocetPixelu**), ovšem navíc jsou ještě pro zadávání použity 2x pixely ve vnitřních sloupcích (**(sirkaObrazku – 2) * vyskaObrazku)**:

```
pocetCar = pocetPixelu + ((sirkaObrazku - 2) * vyskaObrazku);
```

Z DP obr.5.1 a systému vykreslování na něm zobrazeném vychází i algoritmus pro vykreslení obrázku pomocí čar. Proměnná **poradi** odpovídá pořadí vykreslování bodů a tedy červeným číslům:

```
if (j == 0) {
    poradi = poradi + 1;
    Point3f bod = new Point3f(x, y, z);
    Color3f barva = new Color3f(r , g , b);
    oblast.setCoordinate(poradi, bod);
    oblast.setColor(poradi, barva);
}
else if (j > 0 && j <= sirkaObrazku - 1) {
    poradi = poradi + 1;
    Point3f bod = new Point3f(x, y, z);
    Color3f barva = new Color3f(r , g , b);
    oblast.setCoordinate(poradi, bod);
    oblast.setColor(poradi, barva);
    if (j < sirkaObrazku - 1) {
        poradi = poradi + 1;
        bod = new Point3f(x, y, z);
        barva = new Color3f(r , g , b);
        oblast.setCoordinate(poradi, bod);
        oblast.setColor(poradi, barva);
    }
}
```

Metoda **vytvorCary** je koncipována stejně jako metoda **vytvorBody**. Na začátku je deklarována proměnná která bude určovat pořadí vykreslovaných bodů, dále je vytvořen objekt třídy **GeometryArray**, který zajistí vykreslení požadovaného tvary. Dále jsou použity dvě vykreslovací smyčky pro vykreslení řádků a pro vertikální posun. Následuje určení indexu bodu s nímž se bude pracovat, výpočet souřadnic a odstínů barvy. Je aplikováno také větvení, které umožní volbu mezi barevným a šedotónovým obrázkem. Pak již následuje výše uvedený algoritmus, který zajistí zadávání bodů ve správném pořadí a tím také správné vykreslení obrázku.

7.3.2 Vykreslení povrchu pomocí sítě

Podobně je vytvořena také metoda **vytvorSit** pro vykreslení obrázku pomocí sítě. I zde je použita třída **LineArray**. Vykreslování je zde prováděno stejně jako v metodě **vytvorCary** v horizontálním směru, ale navíc ještě ve vertikálním směru. Počet čar pro vykreslení je tedy dvojnásobný. Pro zadání počtu bodů do konstruktoru, slouží proměnná **caryVSiti**, která je deklarována jako datová proměnná a výpočet je opět proveden v metodách **vypocty** a **cteniBarvy**:

```
private int pocetCar = 0;
caryVSiti = pocetCar * 2;
```

Metoda **vytvorSit** je opět koncipována stejně jako předchozí metody **vytvorBody** a **vytvorCary**. Rozdíl je pouze v systému vykreslování a tedy i v navrženém algoritmu. První část algoritmu je totožná s metodou **vytvorCary**, protože se vykreslují čáry v horizontálním směru. Pro vykreslování ve směru vertikálním je použitý stejný algoritmus, ale je změněno indexování bodů dle DP obr.5.2. Hodnota proměnné **poradi** odpovídá v první polovině algoritmu pořadí vykreslování bodů tak jako u metody **vytvorCary**. Ve druhé polovině algoritmu odpovídá hodnota proměnné **poradi** vykreslení vertikální části sítě. Ve algoritmu který vykresluje čáry vertikálně, tedy dojde k tomu že se budou pomocí vnitřní smyčky vykreslovat nejdříve sloupce a toto vykreslování bude posunováno pomocí vnější smyčky po celé šířce obrázku. Tomu odpovídá tento algoritmus:

```
for (int i = 0; i < sirkaObrazku; i++) {
    for (int j = 0; j < vyskaObrazku; j++) {
        int indexBodu = i + (sirkaObrazku * j);
        float x = ( i * vzdalenostBodu) - vystredeniX;
        float y = (-j * vzdalenostBodu) + vystredeniY; . . .
  . . . }
}
```

7.3.3 Osvětlení povrchu

Pro osvětlení povrchu modelu je navržena nová vlastní metoda s názvem **Osvetleni**. K osvětlení povrchu se zde bude využívat podobný algoritmus jako v metodě **vytvorPovrch**, ale aby byly zobrazeny a osvětleny vykreslené trojúhelníky, budou se muset vygenerovat jejich normálové vektory. V metodě **vytvorVzhled** se ale nejdříve musí nastavit vlastnosti povrchu, jako je odstín světla odraženého do okolí (ambinent), odstín vlastního vyzařování povrchu (emissive), odstín rozptýlené složky světla (diffuse), odstín odlesku materiálu povrchu (specular). Tyto vlastnosti jsou sdruženy do objektu třídy **Material** s názvem **material** a přidány do objektu **vzhled**:

```
Color3f ambinent = new Color3f(0.5f, 0.5f, 0.5f);
Color3f emissive = new Color3f(0,0,0);
Color3f diffuse = new Color3f(0.75f, 0.75f, 0.75f);
Color3f specular = new Color3f(0.25f, 0.25f, 0.25f);
Material material = new Material(ambinent, emissive, diffuse, specular, 128);
vzhled.setMaterial(material);
```

Musí být také určeny vlastnosti samotného světelného zdroje, který povrch osvětluje To se provede v nově vytvořené vlastní metodě s názvem **vytvorSvetlo**, která využívá metody třídy **Light**:

```
public Light vytvorSvetlo(){
  Vector3f smerSvetla = new Vector3f(0, -1f, -0.5f);
  Color3f barvaSvetla = new Color3f(1, 1, 1);
  DirectionalLight svetlo = new DirectionalLight(barvaSvetla, smerSvetla);
  BoundingSphere oblastPusobeni = new BoundingSphere(new Point3d(0, 0, 0), 4);
  svetlo.setInfluencingBounds(oblastPusobeni);
  return svetlo;
}
```

V prvním řádku v kódu metody se určuje vektor směru odkud světlo přichází. V druhém řádku je určena barva světla (zde bílá). Ve třetím řádku je vytvořeno přímé světlo, pro které se ve čtvrtém řádku vymezí oblast působení. Nastavení oblasti působení světla je provedeno v pátém řádku.

Může být také nastaveno světlo (resp. osvětlení) v okolí 3D modelu. To se provede v nově vytvořené vlastní metodě s názvem **vytvorOkolniSvetlo**, která využívá metody třídy **Light**:

```
public Light vytvorOkolniSvetlo(){
  AmbientLight okolniSvetlo = new AmbientLight(new Color3f(0.25f, 0.25f, 0.25f));
  BoundingSphere oblastPusobeni = new BoundingSphere(new Point3d(0,0,0), 4);
  okolniSvetlo.setInfluencingBounds(oblastPusobeni);
  return okolniSvetlo;
}
```

V prvním řádku v kódu metody je vytvořeno světlo v okolí (zde barva šedá). V druhém řádku je vymezena oblast působení a ve třetím řádku je nastavena oblast působení světla. Stejně jako je z metody **vytvorScenu** volána metoda **vytvorVzhled**, musí být volány také tyto dvě metody:

```
scena.addChild(vytvorSvetlo());
scena.addChild(vytvorOkolniSvetlo());
```

Jak již bylo zmíněno, k osvětlení povrchu se bude využívat podobný algoritmus jako v metodě **vytvorPovrch**. Bude tedy použitý objekt třídy **TriangleStripArray**. Algoritmus ale bude nutné upravit tak, aby dokázal vypočítat normálové vektory vykreslovaných trojúhelníků:

```
float x0 = ( j * vzdalenostBodu) - vystredeniX;
float y0 = (-i * vzdalenostBodu) + vystredeniY;
float z0 = (vystredeniZ - normPixely[indexBodu]) * hloubka;
float x1 = ( j * vzdalenostBodu) - vystredeniX;
float y1 = (-i * vzdalenostBodu) + vystredeniY - vzdalenostBodu;
float z1 = (vystredeniZ - normPixely[indexBodu]) * hloubka;
float x2 = ( j * vzdalenostBodu) - vystredeniX + vzdalenostBodu;
float y2 = (-i * vzdalenostBodu) + vystredeniY;
float z2 = (vystredeniZ - normPixely[indexBodu]) * hloubka;
Vector3f v1 = new Vector3f(x1-x0, y1-y0, z1-z0);
Vector3f v2 = new Vector3f(x2-x0, y2-y0, z2-z0);
Vector3f v3 = new Vector3f();
v3.cross(v1, v2);
v3.normalize();
for (int m = -1; m < 2; m++) {
    int indx = m + poradi;
    oblast.setNormal(indx, v3);
}
```

V algoritmu se nejdříve určí souřadnice bodů v prostoru, které tvoří vrcholy trojúhelníku. Ty se zapíší do proměnných typu **float**. Vektory odvěsen trojúhelníku **v1** a **v2** jsou získány odečtením počáteční souřadnice od koncové a to v každém rozměru. Normálový vektor **v3** se získá tak, že vypočítané vektory odvěsen **v1** a **v2** se vektorově násobí pomocí metody **cross**. Názorně je tento postup ilustrován na obr.5.3. Normálový vektor **v3** je třeba ještě znormalizovat metodou **normalize**. Nakonec se pomocí smyčky nastaví vždy těmto třem bodům v jednom trojúhelníku jejich normálový vektor. Totéž se opakuje pro všechny body tvořící trojúhelníky v obrázku.