Sveučilište u Zagrebu
Prirodoslovno-matematički fakultet
Matematički odjel

Ivan Rogošić

Primjena biblioteke VTK u vizualizaciji znanstvenih podataka

Diplomski rad

Zagreb, studeni 2009.

Sveučilište u Zagrebu
Prirodoslovno-matematički fakultet
Matematički odjel

Ivan Rogošić

Primjena biblioteke VTK u vizualizaciji znanstvenih podataka

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, studeni 2009.

Ovaj diplomski rad obranjen je dana _____ pred nastavničkim povjerenstvom u sastavu:

1. _____ , predsjednik
2. _____ , član
3. _____ , član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Vizualizacija je proces kreiranja grafičke reprezentacije podataka. Podatke u slikovnom obliku lakše percipiramo i interpretiramo. Vizualizacija nam, dakle, omogućava da u trenutku sagledamo i protumačimo veliku količinu podataka.

U današnjem svijetu računala i računalno generiranih podataka, vizualizacija je koristan, ali i neophodan alat za tumačenje informacija.

Podaci koji se vizualiziraju mogu biti razni, a u ovom radu posebno se bavimo znanstvenim podacima, odnosno njihovom vizualizacijom.

Tipičan primjer izvora takvih podataka su razne numeričke metode i simulacije. Metode konačnih elemenata, volumena i diferencija generiraju podatke na različitim, više ili manje pravilnim mrežama. Kreiranje slike, odnosno grafičkog modela, olakšava nam interpretaciju tako dobivenih podataka.

Računala i računalni programi imaju ključnu ulogu u vizualizaciji podataka. Pisanje programa za računalnu vizualizaciju podataka može biti jako složeno i zahtijevati detaljno poznavanje računalne grafike. To možemo bitno pojednostavniti koristeći neku gotovu biblioteku kao što je VTK.

VTK (Visualization Toolkit) je C++ biblioteka otvorenog koda (*open source*) za računalnu 3D grafiku. VTK nam omogućava da brzo i jednostavno izrađujemo programe sa mogućnošću vizualizacije i interakcije sa trodimenzionalnim modelima.

Biblioteka definira različite strukture podataka i algoritme koji na njima djeluju, što nam olakšava spremanje i transformiranje podataka. Definiran je i apstraktni grafički model za komunikaciju sa grafičkim podsustavom, što pojednostavljuje pisanje programa i čini program prenosivim na razne platforme.

Osim kao C++ biblioteka, VTK se može koristiti u kompajliranom obliku, kao jezgra sustava sa automatski generiranim "omotačem" za interpretaciju koda pisanog u jezicima Tcl, Java ili Python.

U radu koji slijedi opisat ćemo osnovne koncepte i metode vizualizacije kroz konkretne primjere korištenja biblioteke VTK u pisanju C++ programa.

Poglavlje 1

Osnovni koncepti vizualizacije u VTK

Vizualizacija je složen proces transformacije podataka u slike. U ovom poglavlju dat ćemo kratak pregled tog procesa te definirati objektne modele koje VTK koristi u njegovoj implementaciji.

1.1 Objektni modeli

Vizualizacijski proces možemo podijeliti na dva dijela. Prvi dio je proces transformacije numeričkih podataka u grafičke objekte. Drugi dio procesa je kreiranje dvodimenzionalne slike tih grafičkih objekata.

Ilustrirajmo to jednostavnim primjerom. Pretpostavimo da želimo vizualizirati graf funkcije dvije varijable $f(x, y)$ na nekom segmentu ravnine. Interpretacija niza uređenih trojki brojeva $(x, y, f(x, y))$ kao prostornih koordinata točaka i povezivanje tih točaka u plohu u trodimenzionalnom prostoru odgovara prvom dijelu vizualizacijskog procesa. Pozicioniranje plohe u prostoru, određivanje boje, kuta gledanja, odnosno načina projekcije trodimenzionalnog objekta na dvodimenzionalnom ekranu, te samo iscrtavanje pomoću grafičkog sustava računala, drugi je dio vizualizacijskog procesa.

Te procese unutar VTK implementira vizualizacijski, odnosno grafički cjevovod. Svakom od njih odgovara jedan objektni model.

Vizualizacijski model

VTK koristi model toka podataka (*data flow*) za transformaciju podataka u grafički oblik. Pri tom se u modelu pojavljuju dva temeljna tipa objekata.

- vtkDataObject
- vtkAlgorithm

Razni tipovi podataka spremaju se u instance različitih potklasa od `vtkDataObject`. Najčešće se pri tom koriste podatkovni skupovi (*datasets*), tj. potklase od `vtkDataSet`.

`vtkAlgorithm` je natklasa za filtere koji se koriste za transformaciju podataka, odnosno podatkovnih objekata. Filteri ne mijenjaju direktno podatkovne objekte, već generiraju nove objekte sa izmijenjenim podacima.

Spajanjem filtera sa podatkovnim objektima kreiramo vizualizacijski cjevovod pri čemu krajnji podatkovni objekt predstavlja grafički objekt kojeg želimo iscrtati.

Dijelovi vizualizacijskog cjevovoda detaljnije su opisani u poglavljima 2 i 3.

Grafički model

Za iscrtavanje podatkovnih objekata na ekranu koristimo grafički model. On formira apstraktni sloj iznad grafičkog jezika.[1] Apstrakcijom grafičkih koncepata postignuta je neovisnost o platformi na kojoj se program izvršava.

Spomenut ćemo neke ključne objekte koji čine grafički model.

- vtkActor, vtkVolume (potklase od vtkProp)
- vtkLight
- vtkCamera
- vtkMapper, vtkVolumeMapper (potklase od vtkAbstractMapper)
- vtkRenderer
- vtkRenderWindow
- vtkRenderWindowInteractor

Imena klasa unutar grafičkog modela preuzeta su iz filmske industrije. Tako kreiranjem "glumaca" i "rekvizita", "svjetala" i "kamera", kreiramo scenu. Slika koja se iscrtava na ekranu ovisi o svojstvima i položaju glumaca i svjetala u odnosu na kameru.

Potklase klase `vtkAbstractMapper` predstavljaju poveznicu između vizualizacijskog i grafičkog cjevovoda, tj. između "glumca" i podatkovnog objekta.

Instanca klase `vtkRenderer` povezuje elemente scene i iscrtava sliku unutar prozora, kojeg u grafičkom modelu predstavlja `vtkRenderWindow`.

Interakciju sa scenom omogućava nam klasa `vtkRenderWindowInteractor`.

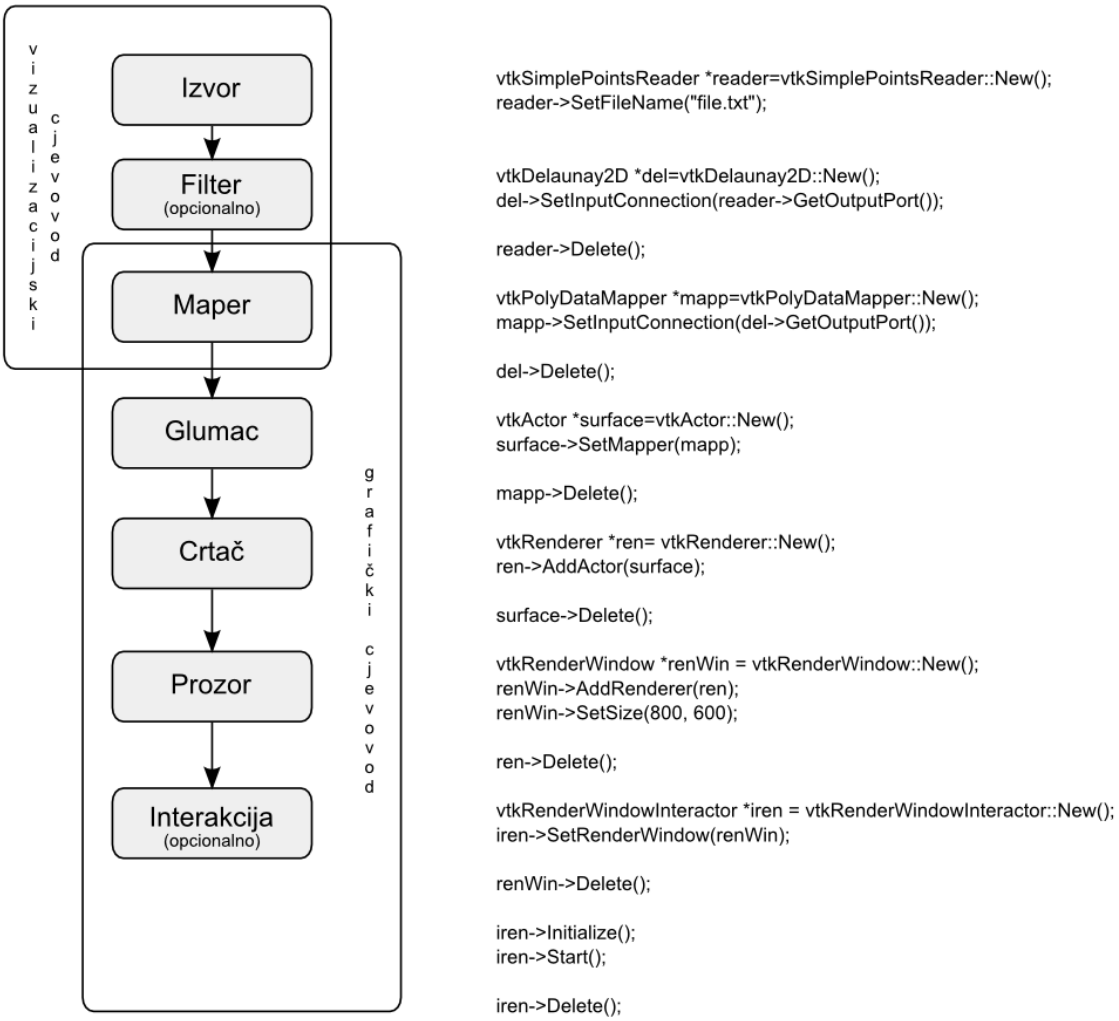
O grafičkom modelu detaljnije govori 4. poglavlje.

1.2 Struktura VTK programa

Na Slici 1.1 prikazana je tipična struktura jednostavnog VTK programa.

Na tom primjeru lako je uočiti ranije spomenute dijelove vizualizacijskog procesa, povezivanje objekata u cjevovod, kao i prepoznati standardne objekte iz vizualizacijskog i grafičkog objektnog modela. Međutim, važno je napomenuti još nekoliko stvari karakterističnih za svaki VTK program.

[1] Trenutno je jedini podržani grafički jezik OpenGL.



Slika 1.1: Struktura VTK programa

Kreiranje objekata

Bazna klasa unutar VTK je `vtkObjectBase`. Ta klasa definira metode za brojanje referenci (*reference counting*). To znači da svaki objekt u VTK sadrži brojač referenci koje pokazuju na njega. Kada nestanu sve reference na kreirani objekt, on se automatski briše. Kako bi se to postiglo sve klase unutar VTK imaju zaštićeni (*protected*) konstruktor i destruktor, te ih je potrebno kreirati korištenjem statičke metode `New()`.

```
vtkExampleClass *object=vtkExampleClass::New();
```

Brojač referenci povećava se prilikom kreiranja objekta, ali i prilikom povezivanja objekta sa drugim objektom, npr. pozivom neke `Set___()` metode.

```
otherObject->SetExample(object);
```

Kada nam pristup kreiranom objektu preko pokazivača više nije potreban, poziva se metoda `Delete()`. Ona neće obrisati objekt, već samo smanjiti brojač referenci. Ukoliko je to bila posljednja referenca, objekt će biti obrisan.

Izvršavanje cjevovoda

Na primjeru na slici 1.1 vidjeli smo kako se objekti međusobno povezuju i formiraju cjevovod. Kod izvršavanja cjevovoda VTK koristi odgođenu evaluaciju (*lazy evaluation*). To znači da se evaluacija podataka događa tek kada se pojavi zahtjev za iscrtavanje.

Ukoliko želimo prije pristupiti vrijednostima podataka u nekom konkretnom objektu, potrebno je eksplicitno pozvati metodu `Update()`.

```
vtkPolyDataReader *reader=vtkPolyDataReader::New();
reader->SetFileName("file.vtk");
reader->GetOutput()->GetNumberOfPoints(); //vraca 0
reader->Update();
reader->GetOutput()->GetNumberOfPoints(); //vraca tocan broj tocaka
```

Poglavlje 2

Strukture podataka

Svi podaci koje želimo vizualizirati spremaju se u strukture podataka VTK-a. Iako nije uvijek potrebno direktno rukovati sa instancama tih struktura, njihovo poznavanje može bitno pridonijeti smanjenju memorijskih zahtjeva i brzini izvođenja programa.

2.1 Skupovi podataka

Najčešće korištene strukture u VTK su skupovi podataka (*datasets*). Skup podataka predstavlja zajedno složenu geometrijsku strukturu sastavljenu od jedne ili više građevnih ćelija i toj strukturi pridružene podatke. O pridruženim podacima govorit ćemo u odjeljku 2.2.

Ćelije u VTK predstavljaju standardne jednodimenzionalne, dvodimenzionalne i trodimenzionalne geometrijske strukture kao što su: točka, linija, izlomljena linija, trokut, četverokut, pravokutnik, mnogokut, tetraedar, heksaedar, kvadar...

Sve ćelije u VTK interno se prikazuju u obliku uređene n-torke točaka. Tip ćelije i redoslijed točaka jednoznačno određuju ćeliju. Svakom tipu ćelije u VTK pridružena je odgovarajuća klasa.

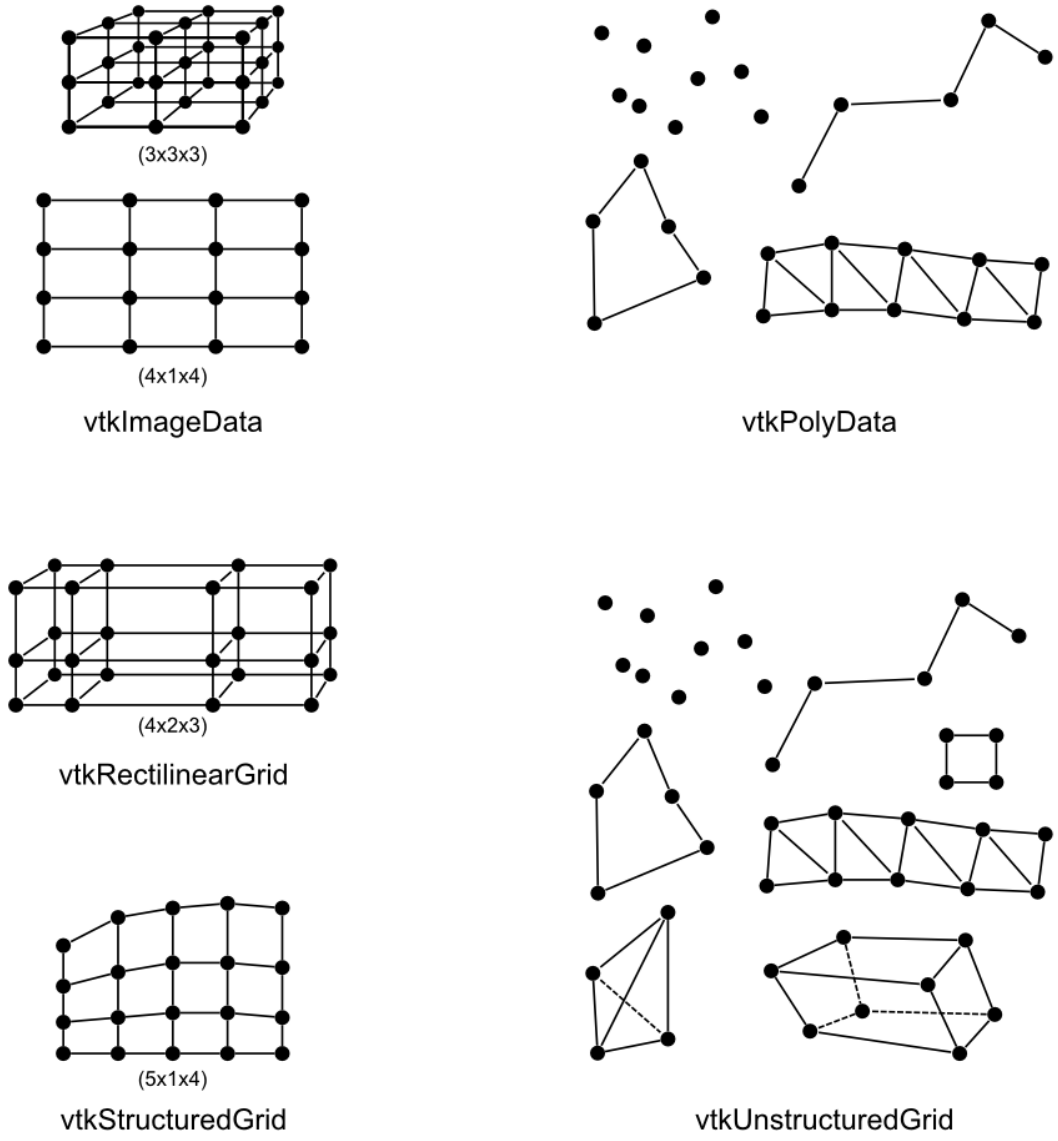
VTK implementira pet različitih podatkovnih skupova kroz sljedeće klase: `vtkImageData`, `vtkRectilinearGrid`, `vtkStructuredGrid`, `vtkPolyData`, `vtkUnstructuredGrid`. Strukture koje one predstavljaju međusobno se razlikuju po topologiji i geometriji.

vtkImageData

Najpravilniju topološku i geometrijsku strukturu predstavlja klasa `vtkImageData`. Dimenzija te strukture je varijabilna i zadaje se u obliku uređene trojke (n_x, n_y, n_z) čije vrijednosti redom predstavljaju broj točaka duž x-y-z koordinatnih osi. Ovisno o dimenziji, građevne ćelije su kvadri (*voxel*), pravokutnici (*pixel*), odnosno linije.

`vtkImageData` predstavlja, dakle, mrežu pravilno razmaknutih točaka poravnatih sa globalnim koordinatnim osima. Redoslijed točaka i ćelija implicitno je određen tako da se niz slaže redom po veličini x, zatim y i konačno z koordinate.

Geometrijska struktura definirana je dvjema vrijednostima – ishodištem i razmakom. Ishodište određuje položaj točke sa najmanjim x-y-z koordinatama. Zadavanjem razmaka među susjednim točkama duž svake koordinatne osi, definiran je položaj svih ostalih točaka mreže.



Slika 2.1: Skupovi podataka u VTK

vtkRectilinearGrid

`vtkRectilinearGrid` definira istu topološku strukturu kao `vtkImageData`. Razlika je u geometrijskim ograničenjima. Dok su kod `vtkImageData` svi razmaci među točkama duž jedne osi jednaki, kod `vtkRectilinearGrid`-a razmaci među točkama duž svake od koordinatnih osi mogu varirati. Točke su i dalje poravnate sa koordinatnim osima, ali sa varijabilnim razmacima među redovima.

Podaci o položaju zapisuju se u tri liste. Svaka lista određuje položaje točaka na jednoj koordinatnoj osi. Liste su reprezentirane pomoću instance klase `vtkDataArray`.

vtkStructuredGrid

I kod `vtkStructuredGrid`-a topološka struktura zadana je kao kod prethodne dvije klase. Međutim, geometrija je nešto slobodnija. Razmještaj točaka u prostoru definira se pojedinačno za svaku točku, pri čemu jedino nije dopušteno preklapanje ćelija.

Ćelije su, ovisno o dimenziji, heksaedri ili četverokuti.

Koordinate točaka spremaju se u instanci klase `vtkPoints`, koja pak interno koristi instancu klase `vtkDataArray`.

Ako se polje točaka eksplicitno zadaje, treba paziti da broj točaka odgovara dimenzijama mreže.

vtkPolyData

`vtkPolyData` ima nepravilnu topološku i geometrijsku strukturu. Može sadržavati sljedeće ćelije: točke, skupove točaka, linije, izlomljene linije, mnogokute i nizove trokutova. Primijetimo da dimenzija ćelija varira od 0 do 2.

Dopušteni tipovi ćelija u `vtkPolyData` predstavljaju geometrijske strukture sa kojima standardno rukuju razne grafičke biblioteke. Stoga `vtkPolyData` predstavlja efikasno sučelje između podataka i grafičkog sustava.

Podaci o ćelijama spremaju se u četiri liste. Tri liste grupiraju ćelije iste dimenzije, s tim da se u četvrtoj posebno spremaju nizovi trokutova.

vtkUnstructuredGrid

Najopćenitija implementacija skupa podataka je `vtkUnstructuredGrid`. Nepravilne je topološke i geometrijske strukture i može sadržavati sve ćelije podržane unutar VTK-a.

Zbog svoje općenitosti ima najsloženiji način zapisivanja podataka. Stoga je korištenje te strukture preporučljivo samo ako je nužno.

2.2 Pridruženi podaci

Podatke možemo asocirati sa točkama i ćelijama podatkovnih struktura. Takve podatke nazivamo pridruženi podaci. Pridruživanjem podataka geometrijskim strukturama definiramo položaj podataka u prostoru.

Pridružene podatke razlikujemo po tipu. Sljedeći popis ukratko opisuje najčešće tipove pridruženih podataka.

Skalari su najjednostavniji i najčešći tip pridruženih podataka. Predstavljaju po jednu numeričku vrijednost pridruženu točkama ili ćelijama neke strukture.

Vektori se najčešće pojavljuju u obliku trodimenzionalnih prostornih vektora, tj. kao uređene trojke brojeva (u, v, w). Međutim, pridruženi podatak može biti i bilo koja uređena n-torka brojeva, tj. n-dimenzionalni vektor.

Normale su vektori norme $|n| = 1$. Normale standardno koristi grafički sustav kod iscrtavanja sjena na objektima.

Koordinate tekstura koriste se za definiranje načina preslikavanja teksture na 3D objekt.

Tenzori su općenito generalizacija vektora i matrica. VTK podržava samo simetrične, 3×3 tenzore sa realnim vrijednostima.

Podatke je moguće pridružiti samo točkama i ćelijama strukture, ali ne i dijelovima ćelija kao što su bridovi ili stranice.

Klase `vtkPointData` i `vtkCellData` predstavljaju odgovarajuće podatke pridružene točkama, odnosno ćelijama. Za pridruživanje i čitanje pridruženih podataka, obje klase sadrže funkcije specifične tipu pridruženih podataka, kao što su npr. `SetScalars()` ili `SetVectors()`.

Sljedeći primjer demonstrira kreiranje `vtkImageData` objekta i dodjeljivanje skalarnih vrijednosti njegovim točkama.

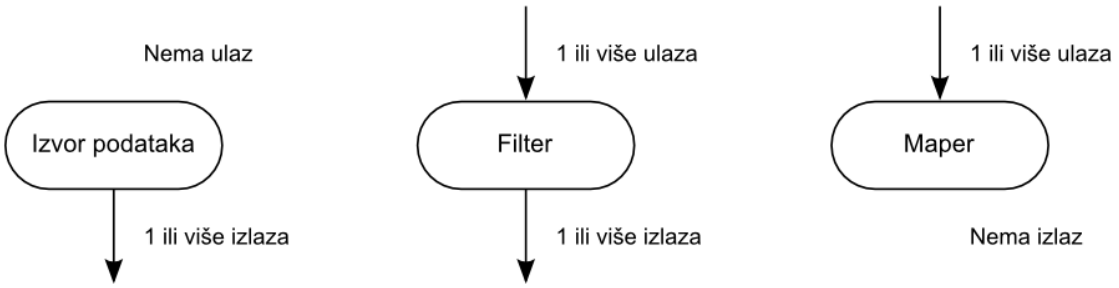
```
//kreiramo novi objekt
vtkImageData *image = vtkImageData::New();

//zadajemo dimenzije, polozaj ishodista i razmak
image->SetDimensions(5,10,10);
image->SetOrigin(0.0,0.0,0.0);
image->SetSpacing(0.2,0.1,0.1);

//tockama strukture dodjeljujemo skalarne vrijednosti
image->GetPointData()->SetScalars(scalars);

/* pri tom je "scalars" instanca neke podklase od vtkDataArray
koja sadrzi tocno 500 elemenata (5x10x10) */
```

Poglavlje 3

Vizualizacijski cjevovod

Proces vizualizacije započinje vizualizacijskim cjevovodom. Ključnu ulogu pri tom imaju *algoritmi* – instance potklasa klase `vtkAlgorithm`. Algoritmi djeluju na podatkovnim objektima kako bi kreirali nove podatkovne objekte. Cjevovod završava povezivanjem krajnjeg podatkovnog objekta sa maperom.

Na slici 3.1 vidimo podjelu elemenata vizualizacijskog cjevovoda s obzirom na broj ulaznih i izlaznih podataka.



Slika 3.1: Elementi vizualizacijskog cjevovoda

Elementi cjevovoda međusobno se povezuju tako da se izlazni podaci jednog elementa proslijede kao ulazni podaci sljedećem. Za to koristimo metode `GetOutputPort()` i `SetInputConnection()`.

```
aFilter->SetInputConnection(anotherFilter->GetOutputPort());
```

3.1 Izvori podataka

Podatke na početku vizualizacijskog cjevovoda možemo kreirati na dva načina. Možemo ih pročitati iz datoteke ili proceduralno generirati. S obzirom na to razlikujemo proceduralne izvore podataka (*procedural source*) i čitače podataka (*reader source*).

Proceduralni izvori podataka

Proceduralni izvor podataka je algoritam koji na temelju zadanih parametara na svom izlazu kreira podatkovni objekt.

Jednostavan primjer je klasa `vtkCylinderSource`. Kako bismo kreirali poligonalnu reprezentaciju valjka dovoljno je kreirati instancu te klase i zadati potrebne parametre.

```
vtkCylinderSource *cylinder=vtkCylinderSource::New();
//visina i polumjer valjka
cylinder->SetHeight(5.0);
cylinder->SetRadius(3.2);
//broj stranica poliedra koji aproksimira valjak
cylinder->SetResolution(8);
```

Na kraju izlaz algoritma jednostavno proslijedimo dalje nekom filteru ili maperu.

Čitači podataka

Za čitanje iz datoteka koristimo čitače podataka. VTK sadrži velik broj čitača za različite tipove datoteka. Razlikuju se po podatkovnoj strukturi koju čitaju, ali i po načinu na koji je ta struktura zapisana u datoteku.

Sljedeći odsječak koda demonstrira kreiranje čitača za čitanje `vtkPolyData` podataka iz XML datoteke.

```
vtkXMLPolyDataReader *reader=vtkXMLPolyDataReader::New();
reader->SetFileName("poly.vtk");
```

Više primjera korištenja raznih čitača može se naći u dodacima A i B.

3.2 Filteri

Filteri su jezgra vizualizacijskog procesa. Pomoću njih podatke transformiramo kako bismo ih prikazali u najpogodnijem obliku.

VTK nudi stotine različitih filtera. Jedna metoda njihove klasifikacije je prema tipu pridruženih podataka na kojem djeluju. U tom smislu razlikujemo skalarne, vektorske i tenzorske filtere, te filtere za modeliranje. U posljednju kategoriju spadaju filteri koji mijenjaju topološku ili geometrijsku strukturu podataka, generiraju normale ploha ili teksturne podatke.

Često se podjela radi i s obzirom na podatkovne strukture nad kojima pojedini algoritam djeluje. Mnogi filteri djeluju samo na jednom tipu podatkovnih objekata. Međutim, neki filteri su prilagođeni za više tipova podataka, a neki primaju kao ulaz jedan tip, a kao izlaz vraćaju drugi tip podatka.

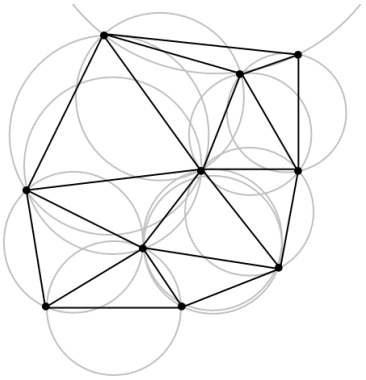
Izazov učenja korištenja biblioteke VTK uključuje upoznavanje što više različitih filtera, te njihovo kombiniranje kako bismo dobili korisne i interesantne vizualizacije.

Ovdje nećemo dati neku detaljnu klasifikaciju filtera koji se mogu naći u VTK, već ćemo na nekoliko primjera proučiti način funkcioniranja filtera i njihovu ulogu u vizualizaciji podataka.

Delaunayeva triangulacija (`vtkDelaunay2D`)

`vtkDelaunay2D` primjer je filtera koji transformira topološku strukturu podataka. Filter kao ulazni podatak prima skup točaka (`vtkPoinSet` ili neku od potklasa), a na izlazu daje mrežu trokuta (`vtkPolyData`).

Delaunayeva triangulacija za zadani skup točaka ravnine je takva triangulacija za koju vrijedi da nijedna točka iz skupa ne leži unutar kružnice opisane bilo kojem trokutu triangulacije.



Slika 3.2: Delaunayeva triangulacija u ravnini

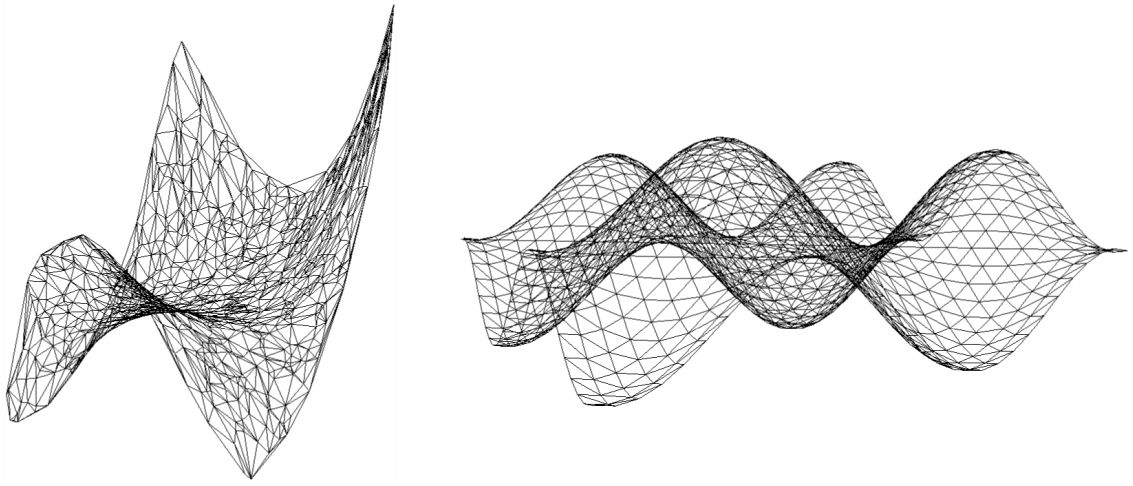
Skup točaka koji proslijedimo filteru ne mora ležati u ravnini. `vtkDelaunay2D` ignorira z koordinate točaka, te vrši triangulaciju u xy ravnini. Izlazni podatak zadržava sve tri koordinate točaka, pa je rezultat mrežasta ploha u prostoru.

Sljedeći odsječak koda demonstrira upotrebu filtera `vtkDelaunay2D` za triangulaciju skupa točaka učitanih iz tekstualne datoteke.

```
//txt datoteku citamo pomocu vtkSimplePointsReader
vtkSimplePointsReader *simpleReader=vtkSimplePointsReader::New();
simpleReader->SetFileName("tocke.txt");

//kreiramo filter
vtkDelaunay2D *del=vtkDelaunay2D::New();
del->SetInputConnection(simpleReader->GetOutputPort());

//izlazne podatke prosljedjujemo maperu
vtkPolyDataMapper *mapp=vtkPolyDataMapper::New();
mapp->SetInputConnection(del->GetOutputPort());
```

Na slici 3.3 vidimo primjere rezultata dobivenih korištenjem filtera `vtkDelaunay2D`.

Napomenimo da VTK sadrži i filter `vtkDelaunay3D` za triangulaciju u prostoru. Ulazni podatak je isti kao kod `vtkDelaunay2D`, a izlazni podatak je mreža tetraedara (`vtkUnstructuredGrid`).



Slika 3.3: Triangulacije dobivene pomoću filtera `vtkDelaunay2D`

Generiranje skalara (`vtkElevationFilter`)

Filter `vtkElevationFilter` generira skalare na temelju geometrijske strukture podatkovnog objekta i kao izlaz vraća objekt sa pridruženim skalarnim vrijednostima.

Za razliku od prethodno opisanog `vtkDelaunay2D` filtera, `vtkElevationFilter` ne mijenja topološku ni geometrijsku strukturu ulaznog objekta, već utječe samo na pridružene podatke.

Skalari se generiraju tako da se točke ulaznog podatkovnog skupa projiciraju na dužinu zadanu dvjema točkama u prostoru. Vrijednost skalara na pojedinoj točki dužine dobije se linearnom interpolacijom, pri čemu je potrebno zadati raspon skalara, tj. vrijednosti u rubnim točkama dužine.

Ukoliko navedene vrijednosti nisu eksplicitno zadane filter koristi točke $(0, 0, 0)$ i $(0, 0, 1)$, te segment $[0, 1]$.

Sljedeći primjer prikazuje kako pomoću filtera `vtkElevationFilter` možemo točkama podatkovnog objekta pridružiti skalarne vrijednosti tako da svakoj točki pridružimo vrijednost njene z koordinate.

```
/* raspon vrijednosti z koordinata dobijemo iz
prostornih granica podatkovnog objekta */
double bounds[6];
simpleReader->Update();
simpleReader->GetOutput()->GetBounds(bounds);

//tockama se dodjeljuju skalarne vrijednosti pomocu vtkElevationFilter-a
vtkElevationFilter *elf=vtkElevationFilter::New();
elf->SetInputConnection(simpleReader->GetOutputPort());
elf->SetLowPoint(0,0,bounds[4]);
elf->SetHighPoint(0,0,bounds[5]);
elf->SetScalarRange(bounds[4], bounds[5]);
```

Izoplohe (`vtkContourFilter`)

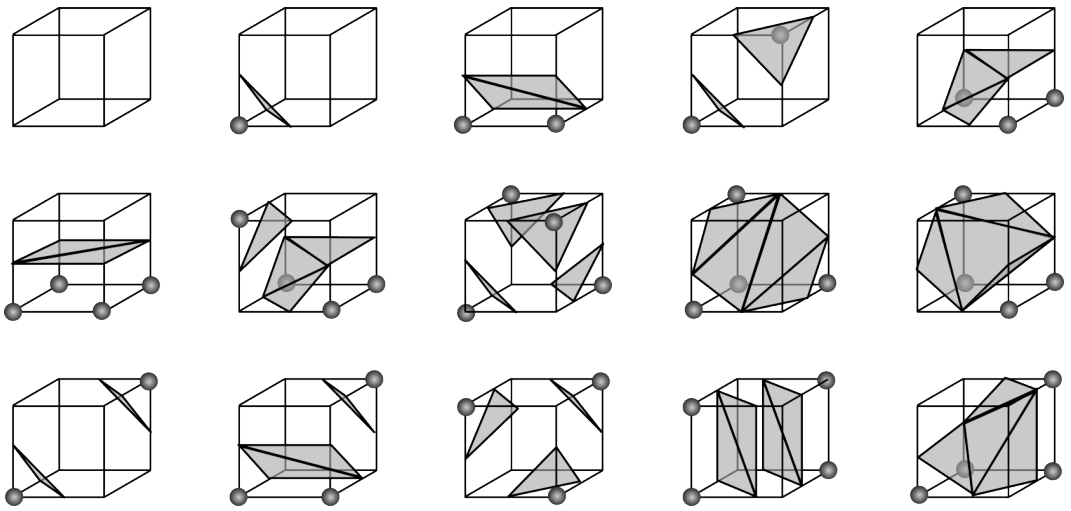
Dvije najčešće tehnike za vizualizaciju skalarnih podataka su bojanje objekata prema skalarnim vrijednostima (*color mapping*) i generiranje izolinija, odnosno izoploha (*contouring*).

Bojanje je u VTK usko vezano uz mapere, pa ćemo se njim baviti u odjeljku 3.3.

Izolinije i izoplohe, ovisno o dimenziji podatkovnog objekta, su krivulje, odnosno plohe konstantne skalarne vrijednosti. Unutar VTK za njihovo kreiranje možemo koristiti `vtkContourFilter`.

Za kreiranje izolinija vtkContourFilter koristi *marching squares* algoritam, odnosno *marching cubes* za izoplohe.

Ideja algoritma je da se izoploha generira po dijelovima, za svaku ćeliju posebno, te se na kraju ti dijelovi spoje. Za svaki vrh ćelije odredi se ima li skalarnu vrijednost veću ili manju od zadane vrijednosti izoplohe. S obzirom na to, ćelija spada u jedan od 15 slučajeva prikazanih na slici 3.4. Ovisno o slučaju, definirano je na kojim bridovima se interpolacijom traži zadana skalarna vrijednost, te se generiraju dijelovi izoplohe. Njihovim spajanjem dobije se tražena ploha.



Slika 3.4: *Marching cubes* – 256 mogućih slučajeva svedeno na 15 s obzirom na simetriju. Istaknuti su vrhovi sa vrijednostima većim od zadane izoplohe.

Kako bismo koristili `vtkContourFilter`, dovoljno je proslijediti mu podatkovni objekt i zadati jednu ili više skalarnih vrijednosti. `vtkContourFilter` kao ulazni podatak prima bilo koji podatkovni skup. Dimenzija izlaznih podataka za 1 je manja od dimenzije ulaznog skupa.

Skalarne vrijednosti ploha moguće je zadati direktno pomoću metode `SetValue()`

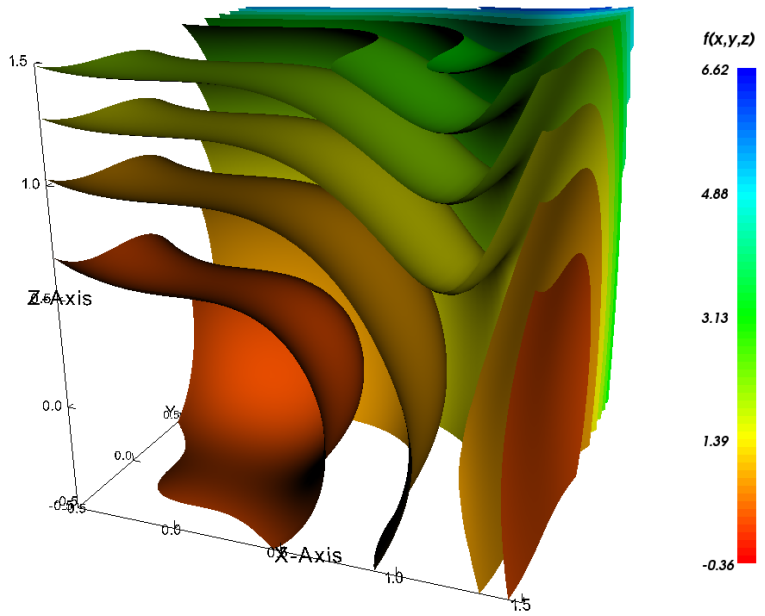
```
vtkContourFilter *contours=vtkContourFilter::New();
contours->SetInputConnection(sample->GetOutputPort());
//zelimo dobiti 2 izoplohe sa vrijednostima 0.5 i 3.8
contours->SetValue(0, 0.5);
contours->SetValue(1, 3.8);
```

gdje je prvo potrebno zadati oznaku, a zatim skalarnu vrijednost izoplohe.

Umjesto toga, metoda `GenerateValues()` nam omogućava da zadamo raspon i broj izoploha u tom rasponu.

```
contours->GenerateValues(5, 0.5, 7.0);
```

U dodatku B vidi se primjer korištenja `vtkContourFilter`-a za generiranje izoploha na podatkovnom skupu `vtkImageData`.



Slika 3.5: Funkcija $sin(x^3) + y^3 + z^2$ iscrtana pomoću 13 izoploha koristeći vtkContourFilter

Presjek (vtkCutter)

Korisna metoda za vizualizaciju podataka je kreiranje presjeka kroz podatkovni skup. Na taj način možemo dobiti detaljan pogled u unutrašnjost trodimenzionalnog objekta.

Filter `vtkCutter` omogućava nam da bilo koji podatkovni skup presiječemo zadanom implicitnom funkcijom (`vtkImplicitFunction`). Izlazni podatak je uvijek `vtkPolyData`.

Implicitna funkcija je funkcija oblika $F(x, y, z) = c$. Korisno svojstvo takve funkcije je da pomoću nje lako možemo svakoj točki u prostoru (x_i, y_i, z_i) pridružiti skalarnu vrijednost $c_i = F(x_i, y_i, z_i)$.

To svojstvo iskorišteno je za kreiranje presječne plohe implicitne funkcije i podatkovnog skupa. Naime, ako točkama skupa pridružimo po tom principu skalarne vrijednosti, tada je ploha presjeka zapravo izoploha sa vrijednošću c. Metodu generiranja izoploha opisali smo u prethodnom odjeljku.

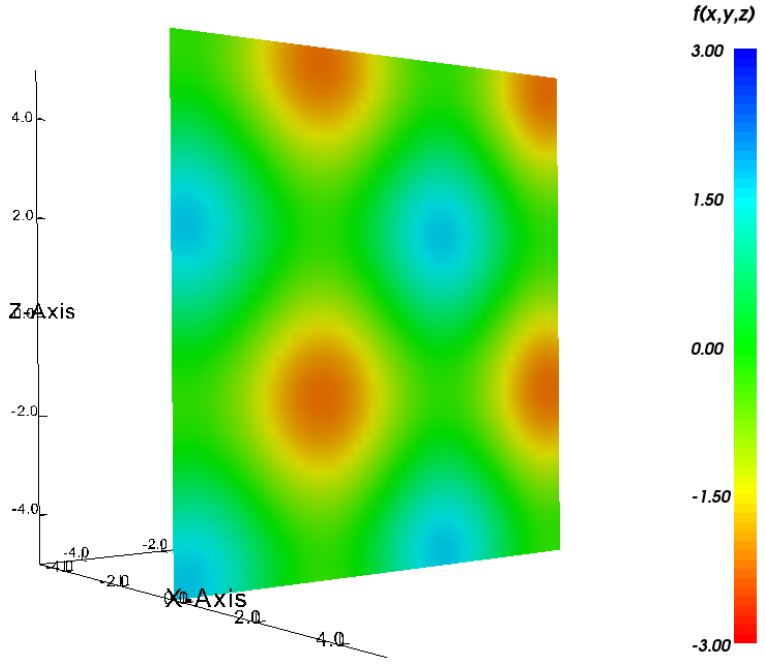
Naravno, na kraju se točkama plohe dodjeljuju vrijednosti nastale interpolacijom iz originalnih skalarnih vrijednosti pridruženih podatkovnom skupu.

Za korištenje filtera dovoljno je zadati podatkovni skup i implicitnu funkciju. U ovom primjeru funkcija predstavlja ravninu.

```
/* vtkPlane je potklasa od vtkImplicitFunction
pomocu koje lako definiramo ravninu */
```

```
vtkPlane *plane=vtkPlane::New();
//ravninu mozemo zadati jednom tockom i vektorom normale
plane->SetOrigin(1.0,1.0,1.0);
plane->SetNormal(1.0,0.0,0.0);

vtkCutter *planeCut=vtkCutter::New();
//zadamo podatkovni skup
planeCut->SetInputConnection(sample->GetOutputPort());
//i funkciju presjeka
planeCut->SetCutFunction(plane);
```

Pretpostavljena vrijednost funkcije je 0, tj. ploha je definirana sa $F(x, y, z) = 0$. Međutim, moguće je i eksplicitno zadati jednu ili više vrijednosti c_i, pri čemu će filter generirati sve plohe $F(x, y, z) = c_i$. Zadavanje vrijednosti radi se pomoću metoda `SetValue()` ili `GenerateValues()` kao i kod `vtkContourFilter`-a.



Slika 3.6: Funkcija $sin(x) + sin(y) + sin(z)$ presječena ravninom $x = 0.3$

3.3 Maperi

Vizualizacijski cjevovod završava maperom. On prosljeđuje podatkovni objekt grafičkoj biblioteci (*graphics mapper*) ili ga zapisuje u datoteku (*writer*).

Maperi koji zapisuju podatke u datoteku slični su podatkovnim čitačima. Za njihovo korištenje dovoljno je maperu proslijediti objekt koji želimo zapisati, ime datoteke, te pozvati metodu `Write()`.

```
vtkXMLImageDataWriter *writer=vtkXMLImageDataWriter::New();
writer->SetInputConnection(sample->GetOutputPort());
```

```
writer->SetFileName("file.vti");
writer->Write();
```

Još jedan primjer korištenja mapera za zapisivanje u datoteku nalazi se u dodatku A u datoteci `vtkMySaveCallback.cxx`.

Ako podatak želimo iscrtati na ekranu, potrebno je koristiti jedan od grafičkih mapera. Odabir mapera ovisi o podatkovnoj strukturi koju iscrtavamo. Npr. za iscrtavanje `vtkPolyData` objekta možemo koristiti `vtkPolyDataMapper`.

```
vtkXMLPolyDataReader *reader=vtkXMLPolyDataReader::New();
reader->SetFileName("poly.vtk");
vtkPolyDataMapper *mapper=vtkPolyDataMapper::New();
mapper->SetInputConnection(reader->GetOutputPort());
```

Grafički maper na kraju vizualizacijskog cjevovoda predstavlja poveznicu sa grafičkim cjevovodom. Naime, maper se povezuje sa "glumcem" i tako postaje dio scene koja se iscrtava na ekranu.

```
vtkActor *actor=vtkActor::New();
actor->SetMapper(mapper);
```

Bojanje grafičkih objekata

Vjerojatno najčešća metoda vizualizacije skalarnih podataka je bojanje različitih vrijednosti različitim bojama.

Bojanje je povezano sa grafičkim maperom. Ukoliko su podatkovnom skupu pridruženi skalarni podaci, maper će automatski podatke obojati. Ovisno želimo li bojanje po skalarnim vrijednostima ili ne, to je moguće zabraniti ili pak kontrolirati zadavanjem dodatnih parametara.

Pozivom metode `ScalarVisibilityOff()` definiranje boje prepuštamo glumcu s kojim je maper povezan.

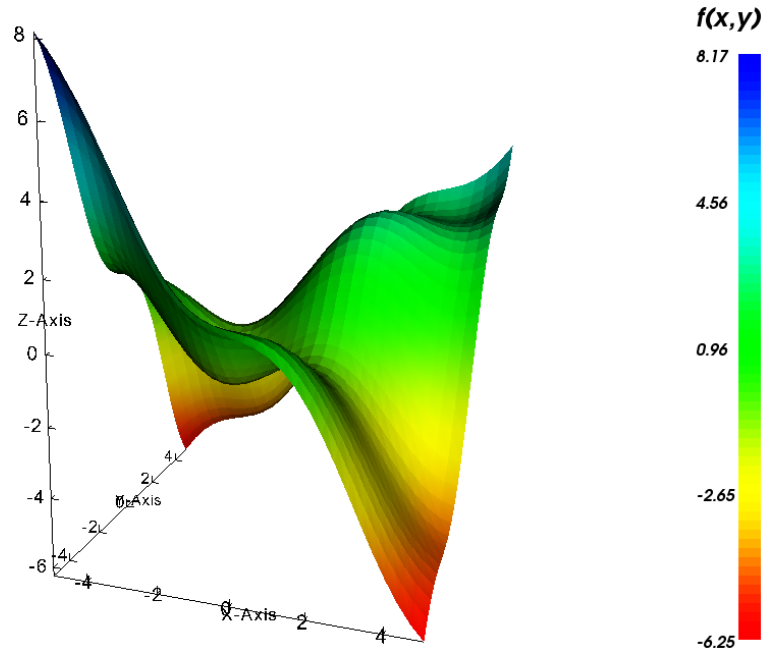
```
mapper->ScalarVisibilityOff();
```

Kontrolu nad bojanjem objekta postižemo definiranjem raspona skalara i načina na koji se skalarne vrijednosti preslikavaju u boje. Način preslikavanja zadaje se instancom klase `vtkLookupTable` ili `vtkColorTransferFunction`.

Raspon skalara zadaje se metodom `SetScalarRange()`. Skalari unutar zadanog raspona bojaju se na zadani način, dok se skalarima koji nisu u rasponu pridjeljuju rubne boje zadanog preslikavanja.

```
mapper->SetScalarRange(-2.0, 7.0);
```

Klase `vtkLookupTable` i `vtkColorTransferFunction` služe za definiranje preslikavanja skalarnih vrijednosti u boje. Ukoliko preslikavanje nije zadano, maper će koristiti raspon boja kao na slici 3.7.



Slika 3.7: Graf funkcije $sin(x) + sin(y) + \frac{xy}{4}$. Skalarne vrijednosti točaka odgovaraju z koordinatama.

`vtkLookupTable` predstavlja tablicu boja. Boje su zadane pomoću HSVA[1] ili RGBA[2] vrijednosti. `vtkColorTransferFunction` definira funkciju za preslikavanje i za razliku od `vtkLookupTable` prihvaća boje u HSV i RGB obliku.

Sljedeći kod demonstrira kreiranje `vtkLookupTable` tablice sa tri boje, a primjer korištenja `vtkColorTransferFunction` može se naći u dodatku B.

```
vtkLookupTable *table=vtkLookupTable::New();
table->SetNumberOfColors(3);
table->Build();
table->SetTableValue(0, 1.0, 0.0, 0.0, 1.0); //crvena
table->SetTableValue(1, 0.0, 1.0, 0.0, 1.0); //zelena
table->SetTableValue(2, 0.0, 0.0, 1.0, 1.0); //plava
```

[1]HSVA – *hue, saturation, value, alpha*
[2]RGBA – *red, green, blue, alpha*

Poglavlje 4

Grafički prikaz i interakcija

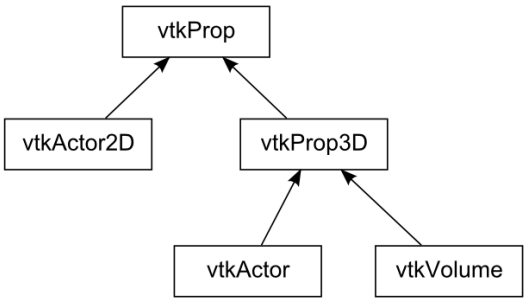
U prethodna dva poglavlja proučili smo podatkovne objekte i ulogu vizualizacijskog cjevovoda u njihovoj transformaciji do konačnog oblika koji želimo prikazati.

U ovom poglavlju proučit ćemo grafički model VTK. Grafički model nastavlja proces vizualizacije. Krajnji elementi vizualizacijskog cjevovoda povezuju se sa objektima grafičkog modela, koji se zatim grupiraju unutar scene. Jedna ili više scena iscrtavaju se konačno unutar prozora, koji je standardni dio grafičkog sučelja operativnog sustava.

4.1 Kreiranje scene

Glumci i rekviziti

Izgradnju scene započinjemo vidljivim elementima, a to su glumci i rekviziti, tj. potklase od `vtkProp`.



Slika 4.1: Dio klasne hijerarhije – vtkProp

Već smo u poglavlju o maperima objasnili na koji se način glumac povezuje sa maperom, te podatkovni objekt postaje dio grafičkog modela. Upotrebu klase `vtkVolume` za volumno iscrtavanje podatkovnih objekata opisat ćemo u odjeljku 4.2.

Dakle, scena može sadržavati jedan ili više rekvizita (*prop*). Neki od njih predstavljaju podatkovne objekte, dok su drugi pomoćni objekti pri vizualizaciji. Primjer takvih pomoćnih objekata su koordinatni sustav i indeks boja na slici 3.7 na stranici 17.

Objekti scene smješteni su u trodimenzionalnom prostoru. Svakom objektu potrebno je stoga moći zadati položaj i orijentaciju. `vtkProp3D`, kao natklasa svih trodimenzionalnih objekata na sceni, definira skup metoda koje nam to omogućavaju.

- `SetPosition(x,y,z)` – zadavanje položaja u globalnim koordinatama
- `AddPosition(deltaX, deltaY, deltaZ)` – pomak duž x, y i z osi
- `RotateX(theta)`, `RotateY(theta)`, `RotateZ(theta)` – rotacija za *theta* stupnjeva oko x, y i z osi
- `SetOrientation(x,y,z)` – zadavanje orijentacije rotacijom oko z, zatim x i konačno y osi
- `RotateWXYZ(theta,u,v,w)` – rotacija za *theta* stupnjeva oko vektora (u, v, w)
- `SetScale(sx,sy,sz)` – promjena veličine duž x, y i z osi za faktore sx, sy i sz
- `SetOrigin(x,y,z)` – postavljanje centralne točke za rotaciju objekta

Najuobičajeniji objekt na sceni je `vtkActor`. Ta potklasa od `vtkProp3D` grupira dodatna svojstva koja definiraju izgled glumca na sceni.

Geometrija objekta tipa `vtkActor` definirana je zadavanjem mapera.

```
vtkActor *actor=vtkActor::New();
actor->SetMapper(mapper);
```

Svojstva `vtkActor`-a poput boje, načina refleksije svjetlosti ili prozirnosti, grupirana su u jednom objektu tipa `vtkProperty`.

Mijenjanje nekog od svojstava možemo izvršiti na dva načina. Možemo pristupiti automatski kreiranom objektu pomoću metode `GetProperty()` kao u sljedećem primjeru:

```
actor->GetProperty()->SetColor(0.0,0.0,1.0);  //plava boja
actor->GetProperty()->SetOpacity(0.25);       //75% prozirnosti
```

Osim toga, možemo kreirati instancu klase `vtkProperty`, pridružiti joj željene vrijednosti, te zatim asocirati sa glumcem.

```
vtkProperty *property=vtkProperty::New();
property->SetColor(0.0,0.0,1.0);
property->SetOpacity(0.25);

actor->SetProperty(property);
```

Drugi način posebno je praktičan ukoliko imamo više glumaca kojima želimo pridružiti ista svojstva.

Važno je napomenuti da se boja koju definira svojstvo pridruženo glumcu koristi pri iscrtavanju samo ako boja već nije definirana pomoću mapera. To se događa u slučaju da podatkovnom objektu nisu pridruženi skalarne vrijednosti ili ako eksplicitno kažemo maperu da ne iscrtava boje.

Svjetla i kamere

Osim vidljivih objekata, na sceni se nalaze i objekti koje ne vidimo direktno, ali koji utječu na način na koji vidimo scenu. Radi se o kamerama i svjetlima.

Kamere i svjetla standardni su dio računalne 3D grafike. O svojstvima i položaju kamere ovisi koji dio scene vidimo, dok je svjetlo potrebno da bi objekti na sceni bili vidljivi. Svaka scena stoga mora sadržavati barem jednu kameru i jedno svjetlo. Ukoliko ih ne kreiramo, crtač (*renderer*) će ih kreirati sam.

Ipak, potrebno je znati na koji način možemo upravljati svjetlima i kamerama, pristupati im, ali i kreirati ih po potrebi.

Kamere u VTK predstavlja klasa `vtkCamera`. Kameru je moguće kontrolirati koristeći metode za direktno zadavanje elementarnih parametara, kao što pokazuje sljedeći primjer:

```
vtkCamera *camera=vtkCamera::New();
//polozaj kamere u prostoru
camera->SetPosition(0.32,-0.11,-0.25);
//smjer kamere odredjen je polozajem i tockom fokusa
camera->SetFocalPoint(0.05,-0.13,-0.06);
//kamera "vidi" samo objekte u zadanom rasponu udaljenosti
camera->SetClippingRange(0.04, 2.37);
```

Osim toga, postoje i pomoćne metode za upravljanje kamerom. Kamera se može pomicati oko točke fokusa, na fiksnoj udaljenosti, pomoću metoda `Azimuth()` i `Elevation()`. Za ekvivalentno pomicanje točke fokusa u odnosu na kameru, dostupne su metode `Yaw()` i `Pitch()`.

Ukoliko želimo da crtač koristi kameru koju smo kreirali, potrebno ju je asocirati sa crtačem pomoću metode `SetActiveCamera()`.

```
renderer->SetActiveCamera(camera);
```

Kameri koju je kreirao crtač, možemo pristupiti pomoću metode `GetActiveCamera()`.

```
//pomicanje kamere za 10 stupnjeva gore
renderer->GetActiveCamera()->Elevation(10);
//i 30 stupnjeva desno
renderer->GetActiveCamera()->Azimuth(30);
```

Svjetla su jednostavniji objekti od kamera. Najčešće definiramo samo boju, položaj i točku fokusa takvog objekta.

Na sljedećem primjeru vidimo standardne metode koje definira klasa `vtkLight` za tu svrhu.

```
vtkLight *light=vtkLight::New();
light->SetColor(1.0,0.0,0.0);       //boja
light->SetFocalPoint(0.0,0.0,0.0);  //tocka fokusa
light->SetPosition(0.0,0.0,5.0);    //polozaj
renderer->AddLight(light);
```

Uz navedeno, svjetlo možemo uključiti ili isključiti, pomoću metoda `SwitchOn()` i `SwitchOff()` ili mu regulirati jačinu pomoću metode `SetIntensity()`.

Scena, crtač, prozor

Scena nije poseban objekt unutar VTK, već elemente scene ujedinjuje crtač, tj. instanca klase `vtkRenderer`.

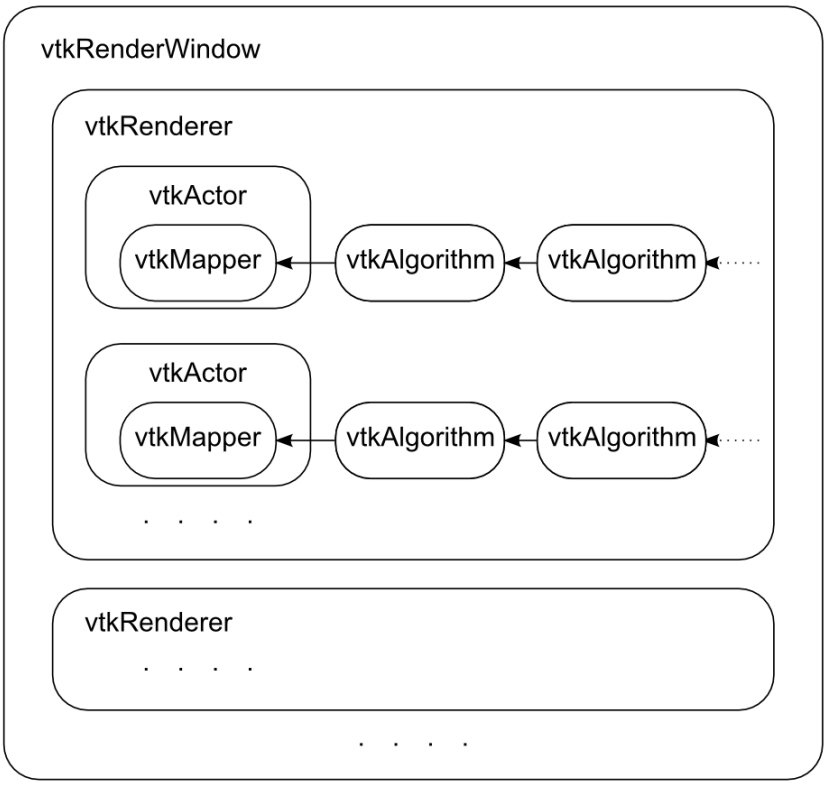
Crtač obavlja proces iscrtavanja slike na ekranu. Iscrtavanje uključuje pretvorbu geometrijskih objekata, svojstava svjetala i pogleda kamere, u dvodimenzionalnu sliku. Crtač kontrolira i pretvorbe između koordinata scene, te koordinata grafičkog sustava i slike na ekranu.

Kako bismo definirali scenu koju želimo prikazati, potrebno je željene objekte asocirati sa crtačem. Sljedeći kod kreira scenu sastavljenu od dva glumca, svjetla i kamere.

```
vtkRenderer *renderer=vtkRenderer::New();
renderer->AddActor(actor1);
renderer->AddActor(actor2);
renderer->AddLight(light);
renderer->SetActiveCamera(camera);
```

Ako svjetlo i kameru ne zadamo sami, crtač će ih kreirati automatski.

Zadnji element grafičkog modela je prozor unutar kojeg se slika iscrtava na ekranu. `vtkRenderWindow` predstavlja apstraktni model prozora unutar grafičkog sučelja operativnog sustava.



Slika 4.2: Grafički model

Prozor može prikazivati sliku jednog ili više crtača, koji se sa prozorom povezuju pomoću metode `AddRenderer()`. Pozivom metode `Render()` pokrećemo iscrtavanje svih scena asociranih sa danim prozorom.

Sljedeći primjer demonstrira kreiranje prozora za iscrtavanje sadržaja scene koju definira crtač.

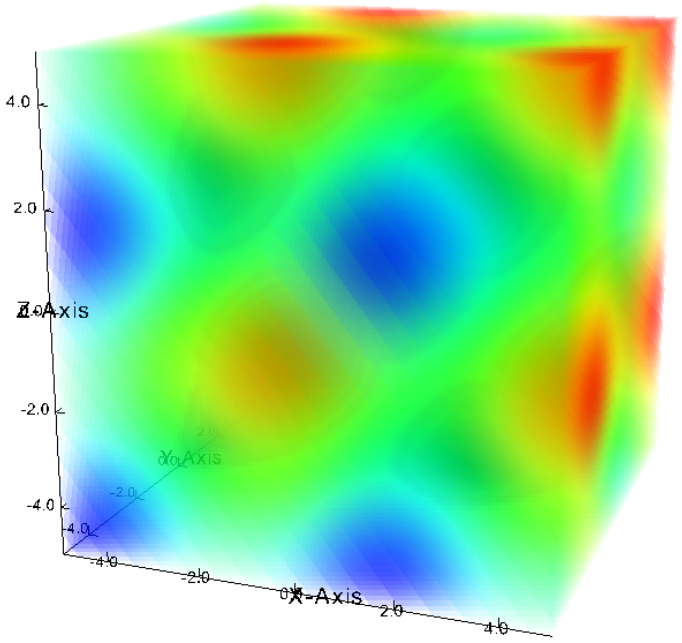
```
vtkRenderWindow *window=vtkRenderWindow::New();
window->AddRenderer(renderer);
window->SetSize(800, 600);
window->Render();
```

Na slici 4.2 vidimo prikaz cijelog grafičkog modela "izvana", krećući od prozora, prema glumcima i spoju sa vizualizacijskim cjevovodom. Prozor može sadržavati više crtača, a svaki crtač povezuje više elemenata jedne scene, tj. svjetla, kamere, glumce i rekvizite. Glumci na kraju predstavljaju poveznicu sa vizualizacijskim cjevovodom.

4.2 Volumno iscrtavanje

Volumno iscrtavanje (*volume rendering*) termin je koji se koristi za opis iscrtavanja podataka raspoređenih kroz 3D prostor, a ne samo na 2D plohama u 3D prostoru.

Skalarnim vrijednostima raspoređenim u prostoru, pridružuje se boja i razina prozirnosti, na temelju čega se iscrtava slika. Primjer volumnog iscrtavanja vidimo na slici 4.3.



Slika 4.3: Volumno iscrtavanje vrijednosti funkcije $sin(x) + sin(y) + sin(z)$

Podjela na volumne i geometrijske tehnike iscrtavanja nije stroga i često možemo dobiti vrlo slične rezultate koristeći dvije različite tehnike, od kojih jednu smatramo volumnom, a drugu geometrijskom.

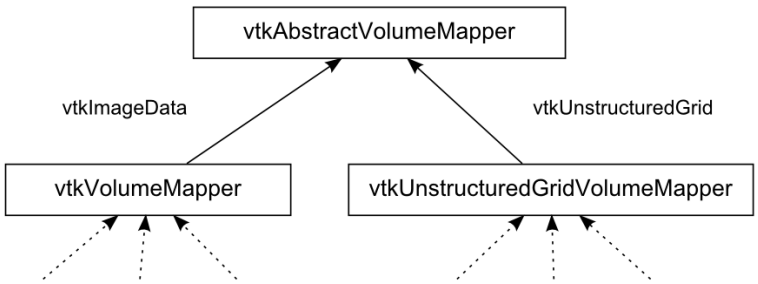
VTK ipak naglašava razliku među tim dvjema kategorijama iscrtavanja, kako bi svojstva pridružena podacima koje iscrtavamo bila bolje prilagođena tehnici iscrtavanja. Podacima koje volumno iscrtavamo se tako prvo pridružuje odgovarajući volumni maper,

koji se zatim povezuje sa specijalnom potklasom klase `vtkProp`, nazvanom `vtkVolume`. Svojstva pridružena takvom objektu spremaju se u instanci klase `vtkVolumeProperty`.

Prije detaljnijeg opisa spomenutih klasa, važno je istaknuti koje podatkovne objekte možemo volumno iscrtavati. Naime, VTK podržava samo volumno iscrtavanje podatkovnih objekata `vtkImageData` i `vtkUnstructuredGrid`.

O tipu podataka direktno ovisi i maper kojeg koristimo. Svi volumni maperi imaju zajedničku natklasu `vtkAbstractVolumeMapper`. Njena potklasa `vtkVolumeMapper` i daljnje potklase primjenjuju se na `vtkImageData`, dok je analogna potklasa za `vtkUnstructuredGrid` klasa `vtkUnstructuredGridVolumeMapper`.

Za oba tipa podataka definirane su specijalne potklase mapera koje se međusobno razlikuju po algoritmima koje koriste za kreiranje volumne slike.



Slika 4.4: Podjela volumnih mapera po tipu podataka

Volumni ekvivalent klase `vtkActor` je klasa `vtkVolume` (slika 4.1). Obje klase nasljeđuju metode za pozicioniranje i orijentaciju objekata u prostoru od klase `vtkProp3D`. Međutim, maperi i svojstva tih objekata bitno se razlikuju. `vtkVolume` preko metode `SetMapper()` prihvaća samo mapere tipa `vtkAbstractVolumeMapper` , a metoda `SetProperty()` prihvaća objekte koji pripadaju klasi `vtkVolumeProperty`.

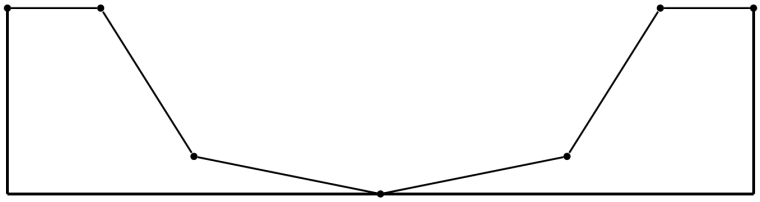
Dva su bitna svojstva definirana pomoću `vtkVolumeProperty` – boja i prozirnost pojedine skalarne vrijednosti.

Funkcija prozirnosti zadaje se kao instanca klase `vtkPiecewiseFunction`. Takav objekt omogućava nam da na jednostavan način definiramo funkciju po dijelovima, definiranjem vrijednosti u pojedinim točkama. Vrijednosti između točaka određuju se interpolacijom.

Sljedeći primjer demonstrira definiranje takve funkcije. Graf te funkcije prikazan je na slici 4.5.

```
vtkPiecewiseFunction *opacity=vtkPiecewiseFunction::New();
double distance=scalarBounds[1]-scalarBounds[0];
double segment=distance/8;
double min=scalarBounds[0];
opacity->AddPoint(min+segment, 1.0);
opacity->AddPoint(min+2*segment, 0.2);
opacity->AddPoint(min+4*segment, 0.0);
opacity->AddPoint(min+6*segment, 0.2);
opacity->AddPoint(min+7*segment, 1.0);
```

`vtkColorTransferFunction` definira bojanje skalara za volumne objekte. Tu klasu već smo spomenuli u odjeljku 3.3. Napomenimo sada još samo da se radi zapravo o



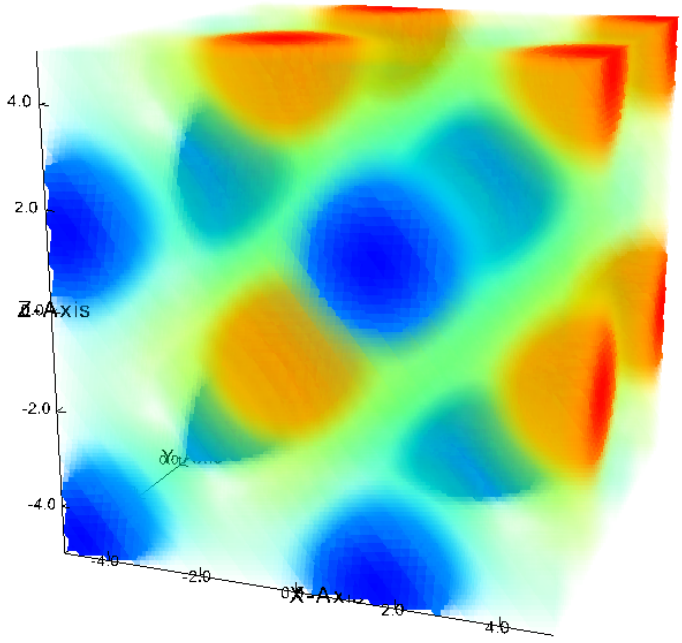
Slika 4.5: vtkPiecewiseFunction

funkciji sastavljenoj od tri funkcije `vtkPiecewiseFunction` – po jedna za svaku RGB komponentu.

Na kraju, `vtkVolume` povezuje sve elemente zajedno u jedan objekt koji možemo volumno iscrtati.

```
vtkVolume *volume=vtkVolume::New();
volume->SetMapper(volumeMapper);

vtkVolumeProperty *property=vtkVolumeProperty::New();
property->SetScalarOpacity(opacity);
property->SetColor(color);

volume->SetProperty(property);
```

Definicije pojedinih svojstava volumnog objekta sa slike 4.3 mogu se naći u dodatku B. Slika 4.6 prikazuje isti objekt iscrtan koristeći funkciju prozirnosti sa slike 4.5 definiranu na str. 23.



Slika 4.6: Funkcija sa slike 4.3 iscrtana sa drugom definicijom prozirnosti

4.3 Interakcija

Prirodni dodatak grafičkom modelu je mogućnost interakcije. Kada smo grafičke objekte iscrtali na ekranu, želimo na što jednostavniji način moći upravljati scenom. Interakcija sa scenom pomoću miša i tipkovnice sastavni je dio mogućnosti biblioteke VTK.

Interakcija u VTK se zasniva na korištenju uzorka dizajna promatrač/naredba (*observer/command design pattern*). Svaka klasa koja nasljeđuje `vtkObject` sadrži metodu `AddObserver()` pomoću koje pojedine događaje (*events*) povezujemo sa pozivom željene naredbe. Naredbu zadajemo nasljeđujući klasu `vtkCommand` i prerađujući metodu `Execute()`.

Međutim, najlakši način interakcije sa scenom je kreiranjem instance klase `vtkRenderWindowInteractor`. Takav objekt povezuje se sa prozorom za iscrtavanje i omogućava nam jednostavnu kontrolu nad prikazom 3D scene, pomoću miša i tipkovnice.

Definicija ponašanja `vtkRenderWindowInteractor`-a sadržana je u posebnom objektu klase `vtkInteractorStyle`. Ukoliko ne želimo koristiti jednu od postojećih potklasa te klase iz VTK, možemo način interakcije precizno definirati kreirajući vlastitu potklasu.

Standardna interakcija sa scenom uključuje kontrolu kamere pomoću miša. Na taj način možemo kameru rotirati, pomicati ili zumirati. Osim toga, interakcija sa scenom omogućena je i pritiskom na različite tipke na tipkovnici:

- e/q – Izlaz iz programa.
- r – Postavlja kameru tako da obuhvaća sve objekte na sceni.
- f – Pomiče kameru prema trenutnom položaju miša.
- w – Mijenja prikaz svih glumaca tako da im se iscrtavaju samo bridovi (*wireframe*).
- s – Mijenja prikaz svih glumaca tako da im se iscrtavaju plohe (*surface*).

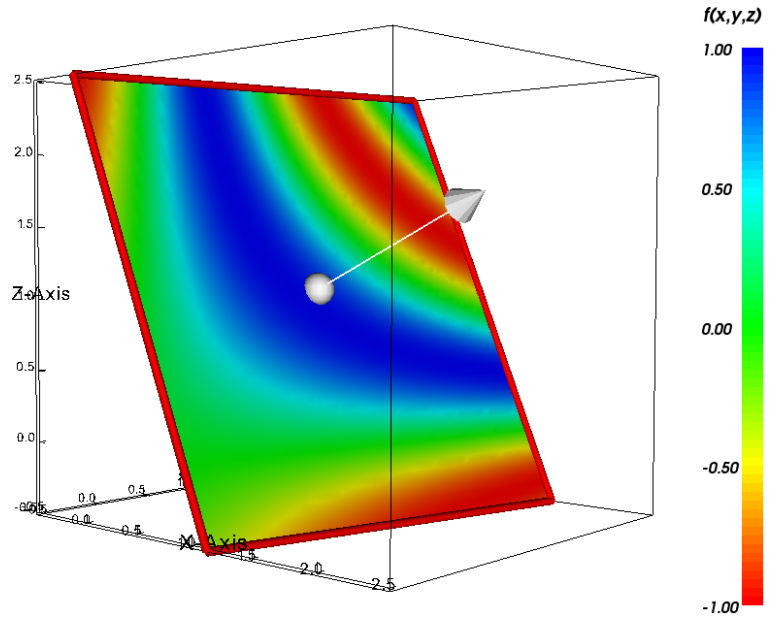
Primjer instanciranja objekta za interakciju i definiranja stila interakcije pomoću gotove klase dan je sljedećim kodom:

```
vtkRenderWindowInteractor *interactor=
   vtkRenderWindowInteractor::New();
interactor->SetRenderWindow(window);
vtkInteractorStyleTrackballCamera *style=
   vtkInteractorStyleTrackballCamera::New();
interactor->SetInteractorStyle(style);
style->Delete();

interactor->Initialize();
interactor->Start();
```

Osim stilova interakcije koji interpretiraju događaje koje prosljeđuje objekt za interakciju `vtkRenderWindowInteractor`, postoji još jedna kategorija objekata koje možemo koristiti za interakciju sa elementima scene. Te objekte nazivamo 3D kontrolama (*3D widgets*).

Poput stilova interakcije, 3D kontrole nasljeđuju klasu `vtkInteractorObserver`. Međutim, razlikuju se po tome što su 3D kontrole vidljive na sceni.

Na slici 4.7 vidimo 3D kontrolu `vtkImplicitPlaneWidget`, koja se koristi za interaktivno pozicioniranje ravnine u prostoru. Detalji korištenja te kontrole vidljivi su iz koda programa u dodatku B. Program presijeca podatkovni skup `vtkImageData` ravninom koja se zadaje interaktivno, pomoću spomenute kontrole.



Slika 4.7: Izgled 3D kontrole `vtkImplicitPlaneWidget` na sceni

VTK sadrži razne 3D kontrole, koje se uglavnom koriste za interaktivno pozicioniranje točaka koje definiraju linije, ravnine, kvadre i slične objekte. Takav interaktivno postavljen i oblikovan objekt, možemo tada iskoristiti za transformaciju objekata na sceni.

Kako bismo koristili jednu od kontrola, treba je instancirati i povezati sa objektom tipa `vtkRenderWindowInteractor`. Uz to, potrebno je kreirati objekte koji će promatrati događaje koje kreira kontrola (`StartInteractionEvent`, `InteractionEvent` i `EndInteractionEvent`). Kao što je spomenuto na početku ovog odjeljka, takvi objekti kreiraju se nasljeđivanjem klase `vtkCommand`.

Za kraj, demonstrirajmo korištenje 3D kontrole jednostavnim primjerom.

```
vtkImplicitPlaneWidget *planeWidget=vtkImplicitPlaneWidget::New();
planeWidget->SetInteractor(interactor);
planeWidget->PlaceWidget(reader->GetOutput()->GetBounds());
planeWidget->SetOrigin(reader->GetOutput()->GetCenter());

vtkMyCallback *callback=vtkMyCallback::New();
//vtkMyCallback nasljedjuje klasu vtkCommand

planeWidget->AddObserver(vtkCommand::InteractionEvent, callback);
planeWidget->EnabledOn();
```

Dodatak A

VTKvis3D

Programi u dodacima A i B opširniji su primjeri koncepata vizualizacije opisanih u prethodnim poglavljima.

VTKvis3D je program za iscrtavanje grafova funkcija dviju varijabli zadanih vrijednostima u pojedinačnim točkama.

Podaci o funkciji čitaju se iz tekstualne datoteke. Svaki redak datoteke sadrži vrijednosti x i y varijable, te vrijednost funkcije u toj točki (z koordinta). Program učitava podatke o točkama te kreira plohu u prostoru povezujući točke u mrežu trokutova pomoću filtera za triangulaciju. Svakoj točki pridružuje se skalarna vrijednost z koordinate.

Ploha se iscrtava na ekranu, obojana s obzirom na skalarne vrijednosti. Uz plohu program iscrtava i indeks boja i vrijednosti skalara, te koordinatni sustav.

Sa slikom iscrtanom na ekranu omogućena je standardna interakcija pomoću miša i tipkovnice. U sklopu interakcije dodana je mogućnost spremanja kreiranog objekta u datoteku. Proces spremanja datoteke pokreće se pritiskom na tipku "u".

Datoteka predstavlja `vtkPolyData` objekt zapisan u XML formatu. Uz već opisane obične tekstualne datoteke, program može pročitati i prikazati sadržaj i takvih datoteka.

Izgled programa može se vidjeti na slici 3.7 na strani 17.

VTKvis3Dmain.cxx

```
#include "CommandInterpretation.h"
#include "Message.h"
#include "vtkMySaveCallback.h"

#include <vtkPolyDataMapper.h>
#include <vtkActor.h>
#include <vtkProperty.h>
#include <vtkRenderer.h>
#include <vtkRenderWindow.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkInteractorStyleTrackballCamera.h>
#include <vtkXMLPolyDataWriter.h>
#include <vtkXMLPolyDataReader.h>
#include <vtkCubeAxesActor.h>
#include <vtkCamera.h>
#include <vtkScalarBarActor.h>
#include <vtkTextProperty.h>
#include <vtkSimplePointsReader.h>
#include <vtkDelaunay2D.h>
#include <vtkElevationFilter.h>

int main( int argc, const char* argv[] )
{
   CommandInterpretation *command=new CommandInterpretation(argc, argv);
   //NEPOZNATA NAREDBA
   if(command->isUnknown())
      Message::exitError("Nepoznati parametri. Pokrenite program sa opcijom -h za pomoc.");
   //HELP
   if(command->isHelp())
   {
      Message::help();
      return 0;
   }

   vtkPolyDataMapper *mapp=vtkPolyDataMapper::New();
   vtkMySaveCallback *save=vtkMySaveCallback::New();

   //CITANJE IZ TXT DATOTEKE
   if(command->isFileTxt())
   {
      //txt datoteku citamo pomocu vtkSimplePointsReader
      vtkSimplePointsReader *simpleReader=vtkSimplePointsReader::New();
      simpleReader->SetFileName(command->getFileName());

      double bounds[6];
      simpleReader->Update();
      simpleReader->GetOutput()->GetBounds(bounds);

      //tockama se dodjeljuju skalarne vrijednosti pomocu vtkElevationFilter-a
      vtkElevationFilter *elf=vtkElevationFilter::New();
      elf->SetInputConnection(simpleReader->GetOutputPort());
      elf->SetLowPoint(0,0,bounds[4]);
      elf->SetHighPoint(0,0,bounds[5]);
```

```
   elf->SetScalarRange(bounds[4], bounds[5]);

   simpleReader->Delete();

   //tocke trianguliramo pomocu vtkDelaunay2D filtera
   vtkDelaunay2D *del=vtkDelaunay2D::New();
   del->SetInputConnection(elf->GetOutputPort());

   elf->Delete();

   //povezujemo izlazne podatke sa mapp i save
   mapp->SetInputConnection(del->GetOutputPort());
   save->setInputConnection(del->GetOutputPort());
   del->Delete();
}

//CITANJE IZ XML DATOTEKE
if(command->isFileVtk())
{
   //XML datoteku citamo pomocu vtkXMLPolyDataReader
   vtkXMLPolyDataReader *xmlReader=vtkXMLPolyDataReader::New();
   xmlReader->SetFileName(command->getFileName());

   //povezujemo izlazne podatke sa mapp i save
   mapp->SetInputConnection(xmlReader->GetOutputPort());
   save->setInputConnection(xmlReader->GetOutputPort());

   xmlReader->Delete();
}

//ISCRTAVANJE PODATAKA

//raspon skalara je od Zmin do Zmax
double bounds[6];
mapp->GetBounds(bounds);
mapp->SetScalarRange(bounds[4], bounds[5]);

//surface actor
vtkActor *surface=vtkActor::New();
surface->SetMapper(mapp);

//scalar bar
vtkScalarBarActor *scalarBar=vtkScalarBarActor::New();
scalarBar->SetLookupTable(mapp->GetLookupTable());
scalarBar->SetTitle("f(x,y)");
scalarBar->GetPositionCoordinate()->SetCoordinateSystemToNormalizedDisplay();
scalarBar->GetPositionCoordinate()->SetValue(0.94, 0.1);
scalarBar->SetOrientationToVertical();
scalarBar->SetWidth(0.05);
scalarBar->SetHeight(0.9);
scalarBar->GetTitleTextProperty()->SetColor(0,0,0);
scalarBar->GetLabelTextProperty()->SetColor(0,0,0);
scalarBar->SetLabelFormat("%.2f");
scalarBar->SetTextPositionToPrecedeScalarBar();

//renderer
```

```
vtkRenderer *ren= vtkRenderer::New();
ren->AddActor(surface);
ren->AddActor(scalarBar);
ren->SetBackground( 1.0, 1.0, 1.0 );

surface->Delete();
scalarBar->Delete();

//cube axes actor
vtkCubeAxesActor *axes=vtkCubeAxesActor::New();
axes->SetBounds(mapp->GetBounds());
axes->SetCamera(ren->GetActiveCamera());
axes->SetFlyModeToStaticTriad();
axes->XAxisMinorTickVisibilityOff();
axes->YAxisMinorTickVisibilityOff();
axes->ZAxisMinorTickVisibilityOff();
axes->GetProperty()->SetColor(0,0,0);

ren->AddActor(axes);

ren->ResetCamera(axes->GetBounds());
axes->Delete();
ren->GetActiveCamera()->Roll(-20);
ren->GetActiveCamera()->Elevation(-75);

//render window
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(ren);
renWin->SetSize(800, 600);

ren->Delete();

//window interactor
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);
vtkInteractorStyleTrackballCamera *style=vtkInteractorStyleTrackballCamera::New();
iren->SetInteractorStyle(style);

style->Delete();

//user save
iren->AddObserver(vtkCommand::UserEvent, save);

iren->Initialize();
std::string name;
name="VTKvis3D - ";
name.append(command->getFileName());
renWin->SetWindowName(name.c_str());
iren->Start();

iren->Delete();
renWin->Delete();

save->Delete();
mapp->Delete();
delete command;
```

```
   return 0;
}
```

Message.h

```
#ifndef MESSAGE_H
#define MESSAGE_H

#include <string>
#include <iostream>

class Message
{
public:
   static void help();
   static void error(std::string err);
   static void exitError(std::string err);
   static void simple(std::string msg);
};

#endif
```

Message.cxx

```
#include "Message.h"

void Message::error(std::string err)
{
   std::cout << "Error: " << err << std::endl;
}

void Message::exitError(std::string err)
{
   error(err);
   exit(0);
}

void Message::help()
{
   std::cout << "Program je potrebno pokrenuti sa jednim parametrom." << std::endl;
   std::cout << "Parametar mora biti ime datoteke sa ekstenzijom .txt ili .vtk" << std::endl;
}

void Message::simple(std::string msg)
{
   std::cout << msg << std::endl;
}
```

CommandInterpretation.h

```
#ifndef COMMANDINTERPRETATON_H
#define COMMANDINTERPRETATON_H

#include <string>

class CommandInterpretation
{
private:
   int argc;
   const char** argv;
   bool unknown, help, filetxt, filevtk;
   std::string filename;
   void interpret();
   bool extension(std::string str, const char* cstr);
public:
   CommandInterpretation(int argc, const char* argv[]);
   bool isUnknown() const {return unknown;}
   bool isHelp() const {return help;}
   bool isFileTxt() const {return filetxt;}
   bool isFileVtk() const {return filevtk;}
   const char* getFileName() const {return filename.c_str();}
};
#endif
```

CommandInterpretation.cxx

```
#include "CommandInterpretation.h"

CommandInterpretation::CommandInterpretation(int argc, const char* argv[])
{
   this->argc=argc;
   this->argv=argv;
   unknown=false;
   help=false;
   filetxt=false;
   filevtk=false;
   filename="";
   interpret();
}

void CommandInterpretation::interpret()
{
   std::string tmp;
   if(argc==2)
   {
      tmp=argv[1];
      if(tmp.compare("-h")==0 || tmp.compare("-help")==0)
      {
         help=true;
         return;
      }
      if(extension(tmp, ".txt"))
      {
```

```
         filetxt=true;
         filename=tmp;
         return;
      }
      if(extension(tmp, ".vtk"))
      {
         filevtk=true;
         filename=tmp;
         return;
      }
   }

   unknown=true;
}

bool CommandInterpretation::extension(std::string str, const char *cstr)
{
   std::string tmp;
   size_t found;
   found=str.rfind(".");
   if(found!=std::string::npos)
   {
      tmp=str.substr(found, std::string::npos);
      if(tmp.compare(cstr)==0)
         return true;
   }
   return false;
}
```

---

vtkMySaveCallback.h

```
#ifndef VTKMYSAVECALLBACK_H
#define VTKMYSAVECALLBACK_H

#include <vtkCommand.h>
#include <vtkXMLPolyDataWriter.h>
#include <vtkAlgorithmOutput.h>
#include <vtkPolyData.h>
#include <vtkSmartPointer.h>
#include "Message.h"

#include <string>
#include <iostream>

class vtkMySaveCallback : public vtkCommand
{
private:
   vtkPolyData *inputpoly;
   vtkAlgorithmOutput *outputport;
   bool port;
public:
   vtkMySaveCallback() {port=false;}
   static vtkMySaveCallback *New() {return new vtkMySaveCallback;}
   virtual void Execute(vtkObject *caller, unsigned long, void*);
```

```
   void setInputConnection(vtkAlgorithmOutput *outputport);
};

#endif
```

---

vtkMySaveCallback.cxx

```
#include "vtkMySaveCallback.h"

void vtkMySaveCallback::Execute(vtkObject *caller, unsigned long, void *)
{
   std::cout << "Ime datoteke?" << std::endl;
   std::string fname;
   std::cin >> fname;
   if(!port)
   {
      Message::error("Nisu definirani ulazni podaci.");
      return;
   }
   vtkSmartPointer<vtkXMLPolyDataWriter> writer=vtkSmartPointer<vtkXMLPolyDataWriter>::New();
   writer->SetInputConnection(outputport);

   if(fname!="")
      writer->SetFileName(fname.c_str());
   if((writer->Write())==0)
      Message::error("Nije moguce spremiti podatke u zadanu datoteku.");
   else
   {
      Message::simple("Podaci spremljeni u datoteku:");
      Message::simple(fname);
   }

}

void vtkMySaveCallback::setInputConnection(vtkAlgorithmOutput *outputport)
{
   this->outputport=outputport;
   this->port=true;
}
```

Dodatak B

VTKvis4D

Program VTKvis4D sličan je programu iz dodatka A. Glavna razlika je u tipu podataka koji se prikazuje, odnosno u dimenzionalnosti funkcije koja se iscrtava.

VTKvis4D iscrtava funkcije sa tri varijable, a podaci su zadani na pravilnoj mreži podatkovnog skupa `vtkImageData`. Podaci se učitavaju iz odgovarajuće XML datoteke.

Kako graf funkcije više nije moguće iscrtati u 3D prostoru, program koristi nekoliko različitih metoda za vizualizaciju vrijednosti funkcije.

Prva metoda je presijecanje podatkovnog skupa ravninom, pri čemu se presječna ravnina boja u skladu sa skalarnim vrijednostima, tj. vrijednostima funkcije u pojedinim točkama u prostoru. Presječnu ravninu moguće je zadati interaktivno, korištenjem odgovarajuće 3D kontrole.

Slike 3.6 na strani 15. i 4.7 na strani 26. dobivene su korištenjem programa VTKvis4D na taj način.

Druga metoda prikaza podataka je iscrtavanjem jedne ili više izoploha. Primjer vidimo na slici 3.5 na 14. strani.

Na kraju, VTKvis4D nam pruža mogućnost volumnog iscrtavanja podatkovne strukture. Sve vrijednosti funkcije iscrtavaju se s određenom prozirnošću, a lagano su istaknute minimalne i maksimalne vrijednosti. Primjer volumnog iscrtavanja funkcije pomoću VTKvis4D vidi se na slici 4.3 na 22. strani.

VTKvis4Dmain.cxx

```
#include "CommandInterpretation.h"
#include "Message.h"
#include "vtkMyCallback.h"

#include <vtkXMLImageDataReader.h>
#include <vtkImageData.h>
#include <vtkPointData.h>
#include <vtkColorTransferFunction.h>
#include <vtkContourFilter.h>
#include <vtkPolyDataMapper.h>
#include <vtkActor.h>
#include <vtkPlane.h>
#include <vtkCutter.h>
#include <vtkPiecewiseFunction.h>
#include <vtkVolumeTextureMapper3D.h>
#include <vtkVolumeProperty.h>
#include <vtkVolume.h>
#include <vtkRenderer.h>
#include <vtkRenderWindow.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkInteractorStyleTrackballCamera.h>
#include <vtkCamera.h>
#include <vtkCubeAxesActor.h>
#include <vtkProperty.h>
#include <vtkScalarBarActor.h>
#include <vtkTextProperty.h>
#include <vtkImplicitPlaneWidget.h>
#include <vtkTextActor.h>

int main( int argc, const char* argv[] )
{
   CommandInterpretation *command=new CommandInterpretation(argc, argv);

   //NEPOZNATA NAREDBA
   if(command->isUnknown())
      Message::exitError("Nepoznati parametri. Pokrenite program sa opcijom -h za pomoc.");
   //HELP
   if(command->isHelp())
   {
      Message::help();
      return 0;
   }
   //ako smo tu, zadana je datoteka za citanje
   vtkXMLImageDataReader *reader=vtkXMLImageDataReader::New();
   if(!reader->CanReadFile(command->getFileName()))
      Message::exitError("Nije moguce citati zadanu datoteku.");
   reader->SetFileName(command->getFileName());
   reader->Update();
   //spremamo granice ucitanih podataka
   double dataBounds[6];
   reader->GetOutput()->GetBounds(dataBounds);
   //spremamo raspon pridruzenih skalara
   double scalarBounds[2];
```

```
reader->GetOutput()->GetPointData()->GetScalars()->GetRange(scalarBounds);

//bojanje skalara
vtkColorTransferFunction *scalarColor=vtkColorTransferFunction::New();
scalarColor->AddRGBPoint(scalarBounds[0], 1, 0, 0);
scalarColor->AddRGBPoint((3*scalarBounds[0]+scalarBounds[1])/4, 1, 1, 0);
scalarColor->AddRGBPoint((scalarBounds[0]+scalarBounds[1])/2, 0, 1, 0);
scalarColor->AddRGBPoint((scalarBounds[0]+3*scalarBounds[1])/4, 0, 1, 1);
scalarColor->AddRGBPoint(scalarBounds[1], 0, 0, 1);

//renderer
vtkRenderer *ren=vtkRenderer::New();
ren->SetBackground(1.0,1.0,1.0);
ren->ResetCamera(dataBounds);
ren->GetActiveCamera()->Roll(-20);
ren->GetActiveCamera()->Elevation(-75);


//izoplohe
if(command->isIso())
{
   vtkContourFilter *contour=vtkContourFilter::New();
   contour->SetInputConnection(reader->GetOutputPort());
   contour->GenerateValues(command->getIsoNo(), scalarBounds);
   vtkPolyDataMapper *contMapper=vtkPolyDataMapper::New();
   contMapper->SetInputConnection(contour->GetOutputPort());
   contMapper->SetLookupTable(scalarColor);
   vtkActor *contActor=vtkActor::New();
   contActor->SetMapper(contMapper);
   ren->AddActor(contActor);
   contour->Delete();
   contMapper->Delete();
   contActor->Delete();
}

//presjek
vtkPlane *plane=vtkPlane::New();
if(command->isCut())
{
   plane->SetOrigin(reader->GetOutput()->GetCenter());
   plane->SetNormal(0, -1, 0);
   vtkCutter *cutter=vtkCutter::New();
   cutter->SetInputConnection(reader->GetOutputPort());
   cutter->SetCutFunction(plane);
   vtkPolyDataMapper *cutMapper=vtkPolyDataMapper::New();
   cutMapper->SetInputConnection(cutter->GetOutputPort());
   cutMapper->SetLookupTable(scalarColor);
   vtkActor *cutActor=vtkActor::New();
   cutActor->SetMapper(cutMapper);
   ren->AddActor(cutActor);
   cutter->Delete();
   cutMapper->Delete();
   cutActor->Delete();
}

if(command->isVolume())
```

```
{
   //funkcija "prozirnosti" skalara
   vtkPiecewiseFunction *opacity=vtkPiecewiseFunction::New();
   double distance=scalarBounds[1]-scalarBounds[0];
   double segment=distance/8;
   double min=scalarBounds[0];
   opacity->AddPoint(scalarBounds[0], 0.6);
   opacity->AddPoint(min+2*segment, 0.3);
   opacity->AddPoint(min+4*segment, 0.2);
   opacity->AddPoint(min+6*segment, 0.3);
   opacity->AddPoint(scalarBounds[1], 0.6);

   //volumno iscrtavanje
   vtkVolumeTextureMapper3D *volMapper=vtkVolumeTextureMapper3D::New();
   volMapper->SetInputConnection(reader->GetOutputPort());
   double stranice[3];
   reader->GetOutput()->GetSpacing(stranice);
   volMapper->SetSampleDistance(stranice[0]);
   vtkVolumeProperty *volProp=vtkVolumeProperty::New();
   volProp->SetScalarOpacity(opacity);
   volProp->SetColor(scalarColor);
   vtkVolume *volume=vtkVolume::New();
   volume->SetMapper(volMapper);
   volume->SetProperty(volProp);
   ren->AddVolume(volume);

   volMapper->Delete();
   volProp->Delete();
   opacity->Delete();
   volume->Delete();
}

//koordinatne osi
vtkCubeAxesActor *axes=vtkCubeAxesActor::New();
axes->SetBounds(dataBounds);
axes->SetCamera(ren->GetActiveCamera());
axes->SetFlyModeToStaticTriad();
axes->XAxisMinorTickVisibilityOff();
axes->YAxisMinorTickVisibilityOff();
axes->ZAxisMinorTickVisibilityOff();
axes->GetProperty()->SetColor(0,0,0);
ren->AddActor(axes);
axes->Delete();

//scalar bar
vtkScalarBarActor *scalarBar=vtkScalarBarActor::New();
scalarBar->SetLookupTable(scalarColor);
scalarBar->SetTitle("f(x,y,z)");
scalarBar->GetPositionCoordinate()->SetCoordinateSystemToNormalizedDisplay();
scalarBar->GetPositionCoordinate()->SetValue(0.94, 0.1);
scalarBar->SetOrientationToVertical();
scalarBar->SetWidth(0.05);
scalarBar->SetHeight(0.9);
scalarBar->GetTitleTextProperty()->SetColor(0,0,0);
scalarBar->GetLabelTextProperty()->SetColor(0,0,0);
scalarBar->SetLabelFormat("%.2f");
```

```
scalarBar->SetTextPositionToPrecedeScalarBar();
ren->AddActor(scalarBar);
scalarBar->Delete();
scalarColor->Delete();

//render window
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(ren);
renWin->SetSize(800, 600);

ren->Delete();

//window interactor
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);
vtkInteractorStyleTrackballCamera *style=
   vtkInteractorStyleTrackballCamera::New();
iren->SetInteractorStyle(style);
style->Delete();

vtkImplicitPlaneWidget *planeWidget=vtkImplicitPlaneWidget::New();
if(command->isCut())
{
   //plane widget
   planeWidget->SetInteractor(iren);
   planeWidget->SetPlaceFactor(1.01);
   planeWidget->DrawPlaneOff();
   planeWidget->GetOutlineProperty()->SetColor(0,0,0);
   planeWidget->OutlineTranslationOff();
   planeWidget->PlaceWidget(dataBounds);
   planeWidget->SetOrigin(reader->GetOutput()->GetCenter());
   planeWidget->GetPlane(plane);
   planeWidget->EnabledOn();

   vtkMyCallback *mc=vtkMyCallback::New();
   mc->setPlane(plane);
   mc->setWidget(planeWidget);

   planeWidget->AddObserver(vtkCommand::InteractionEvent, mc);

   //text - podaci o ravnini
   vtkTextActor *text=vtkTextActor::New();
   mc->setText(text);
   text->GetTextProperty()->SetColor(0,0,0);
   text->GetTextProperty()->SetFontFamilyToCourier();
   text->GetPositionCoordinate()->SetCoordinateSystemToNormalizedDisplay();
   text->GetPositionCoordinate()->SetValue(0.01, 0.02);
   ren->AddActor(text);
   text->Delete();
   mc->Delete();

}
reader->Delete();

iren->Initialize();
std::string name;
```

```
   name="VTKvis4D - ";
   name.append(command->getFileName());
   renWin->SetWindowName(name.c_str());
   iren->Start();

   planeWidget->Delete();
   plane->Delete();
   renWin->Delete();
   iren->Delete();
   delete command;
   return 0;
}
```

Message.h

```
#ifndef MESSAGE_H
#define MESSAGE_H

#include <string>
#include <iostream>

class Message
{
public:
   static void help();
   static void error(std::string err);
   static void exitError(std::string err);
   static void simple(std::string msg);
};

#endif
```

Message.cxx

```
#include "Message.h"

void Message::error(std::string err)
{
   std::cout << "Error: " << err << std::endl;
}

void Message::exitError(std::string err)
{
   error(err);
   exit(0);
}

void Message::help()
{
   std::cout << "Program je potrebno pokrenuti sa barem jednim parametrom." << std::endl;
   std::cout << "Obavezni parametar mora biti ime datoteke." << std::endl << std::endl;
   std::cout << "    VTKvis4D [datoteka.vtk]" << std::endl << std::endl;
   std::cout << "Opcionalni parametri su sljedeci:" << std::endl << std::endl;
```

```
   std::cout << "-c  presjek" << std::endl << std::endl;
   std::cout << "    VTKvis4D -c [datoteka.vtk]" << std::endl << std::endl;
   std::cout << "-i  izoplohe" << std::endl << std::endl;
   std::cout << "    VTKvis4D -i [datoteka.vtk]" << std::endl;
   std::cout << "    VTKvis4D -i [broj_ploha] [datoteka.vtk]" << std::endl << std::endl;
   std::cout << "-v  volumno iscrtavanje" << std::endl << std::endl;
   std::cout << "    VTKvis4D -v [datoteka.vtk]" << std::endl;
}

void Message::simple(std::string msg)
{
   std::cout << msg << std::endl;
}
```

CommandInterpretation.h

```
#ifndef COMMANDINTERPRETATON_H
#define COMMANDINTERPRETATON_H

#include <string>
#include "Message.h"

class CommandInterpretation
{
private:
   int argc;
   const char** argv;
   bool unknown, help, filevtk;
   bool volume, iso, cut;
   int isoNo;
   std::string filename;
   void interpret();
public:
   CommandInterpretation(int argc, const char* argv[]);
   bool isUnknown() const {return unknown;}
   bool isHelp() const {return help;}
   bool isFileVtk() const {return (volume || iso || cut);}
   bool isVolume() const {return volume;}
   bool isIso() const {return iso;}
   bool isCut() const {return cut;}
   int getIsoNo() const {return isoNo;};
   const char* getFileName() const {return filename.c_str();}
};
#endif
```

CommandInterpretation.cxx

```
#include "CommandInterpretation.h"

CommandInterpretation::CommandInterpretation(int argc, const char* argv[])
{
   this->argc=argc;
   this->argv=argv;
```

```
   unknown=false;
   help=false;
   filevtk=false;
   filename="";

   volume=false;
   iso=false;
   cut=false;
   isoNo=7; //default broj izoploha
   interpret();
}

void CommandInterpretation::interpret()
{
   std::string tmp;
   if(argc==2)
   {
      tmp=argv[1];
      if(tmp.compare("-h")==0 || tmp.compare("-help")==0)
      {
         help=true;
         return;
      }
      else
      {
         filename=tmp;
         cut=true; //default to cut
         return;
      }
   }
   if(argc==3)
   {
      tmp=argv[1];
      if(tmp.compare("-c")==0)
      {
         cut=true;
         filename=argv[2];
         return;
      }
      if(tmp.compare("-v")==0)
      {
         volume=true;
         filename=argv[2];
         return;
      }
      if(tmp.compare("-i")==0)
      {
         iso=true;
         filename=argv[2];
         return;
      }
   }
   if(argc==4)
   {
      tmp=argv[1];
      if(tmp.compare("-i")==0)
```

```
      {
         if((isoNo=atoi(argv[2]))>0)
         {
            iso=true;
            filename=argv[3];
            return;
         }
         else Message::exitError("Drugi parametar mora biti cijeli broj veci od nule.");
      }
   }

   unknown=true;
}
```

---

vtkMyCallback.h

```
#ifndef VTKMYCALLBACK_H
#define VTKMYCALLBACK_H

#include <vtkCommand.h>
#include <vtkPlane.h>
#include <vtkImplicitPlaneWidget.h>
#include <vtkTextActor.h>
#include <sstream>

class vtkMyCallback : public vtkCommand
{
private:
   vtkPlane *plane;
   vtkImplicitPlaneWidget *widget;
   vtkTextActor *text;
public:
   static vtkMyCallback *New() {return new vtkMyCallback;}
   virtual void Execute(vtkObject *caller, unsigned long, void*);

   void setPlane(vtkPlane *plane);
   void setWidget(vtkImplicitPlaneWidget *widget);
   void setText(vtkTextActor *text);
};

#endif
```

---

vtkMyCallback.cxx

```
#include "vtkMyCallback.h"

void vtkMyCallback::Execute(vtkObject *caller, unsigned long, void *)
{
   widget->GetPlane(plane);

   std::ostringstream ss;
   ss<<"Ishodiste: ("<<(plane->GetOrigin())[0]<<", "<<(plane->GetOrigin())[1];
   ss<<", "<<(plane->GetOrigin())[2]<<")\n";
```

```
   ss<<"Normala: ("<<(plane->GetNormal())[0]<<", "<<(plane->GetNormal())[1];
   ss<<", "<<(plane->GetNormal())[2]<<")";
   //std::cout<<ss.str()<<std::endl;
   text->SetInput(ss.str().c_str());
}

void vtkMyCallback::setPlane(vtkPlane *plane)
{
   this->plane=plane;
}

void vtkMyCallback::setWidget(vtkImplicitPlaneWidget *widget)
{
   this->widget=widget;
}

void vtkMyCallback::setText(vtkTextActor *text)
{
   this->text=text;
}
```

Bibliografija

[1] W. Schroeder, K. Martin, B. Lorensen: *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Fourth Edition, Kitware Inc., 2006.

[2] Kitware Inc.: *The VTK User's Guide*, Updated for VTK Version 5, Kitware Inc., 2006.

[3] W. Schroeder, L.S. Avila, and W. Hoffman: *Visualizing with VTK: A Tutorial*, IEEE Computer Graphics and Applications, Vol. 20, No. 2, 20-27, 2000.

[4] S.B. Lippman, J. Lajoie, B.E. Moo: *C++ Primer*, Fourth Edition, Addison-Wesley, 2005.

[5] S.C. Brenner , L.R. Scott: *The Mathematical Theory of Finite Element Methods*, Springer, 2002.

[6] R.J. LeVeque: *Finite Volume Methods for Hyperbolic Problems*, Cambridge University Press, 2002.

[7] J. Strikwerda: *Finite Difference Schemes and Partial Differential Equations*, SIAM, 2004.