UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA
FACULTATEA DE ELECTRONICĂ, TELECOMUNICAŢII ŞI TEHNOLOGIA INFORMAŢIEI
Specializarea Tehnologii şi Sisteme de Telecomunicaţii

Modelling the Nervous System. 3D Visualization Algorithms

(Modelarea sistemului nervos. Algoritmi de vizualizare 3D)

Lucrare de licenţă

PREŞEDINTE COMISIE,
Prof.dr.ing. Gavril TODEREAN

DECAN,
Prof.dr.ing. Marina ŢOPA

CONDUCATOR,
Prof.dr.ing. Mircea Florin VAIDA

ABSOLVENT,
Teodora SZASZ

2011

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA
FACULTATEA DE ELECTRONICĂ, TELECOMUNICAŢII ŞI TEHNOLOGIA INFORMAŢIEI
CATEDRA DE COMUNICAŢII

Titlul temei: Modelarea sistemului nervos. Algoritmi de vizualizare 3D

Enunţul lucrării de licenţă:

Obiectivul lucrării îl constituie dezvoltarea unor algoritmi pentru modelarea semnalelor neuronale, prezentarea algoritmilor fundamentali de vizualizare, cât şi a celor utilizaţi în vizualizarea imaginilor medicale. Un alt obiectiv este crearea unui prototip de aplicaţie care să prezinte modalităţile prin care se pot vizualiza imaginile tridimensionale.
Finalitatea constă în descrierea algoritmilor implementaţi pentru procesarea semnalelor neuronale, descrierea principalilor algoritmi de vizualizare şi realizarea unei aplicaţii pentru vizualizarea imaginilor tridimensionale.

Locul de execuţie: Laborator Helios, Universitatea Tehnică din Cluj Napoca

Data emiterii temei: 24/02/2011

Termen de predare : 24/06/2011

ŞEF CATEDRĂ, **ABSOLVENT,**
Prof.dr.ing. Virgil DOBROTĂ **Teodora SZASZ**

CONDUCĂTOR,
Prof.dr.ing. Mircea Florin Vaida

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE ELECTRONICĂ, TELECOMUNICAŢII ŞI TEHNOLOGIA INFORMAŢIEI

Str. George Bariţiu, Nr. 26 – 28, 400027 Cluj-Napoca, ROMÂNIA

Tel/Fax: + 40-264-591689; Tel: + 40-264-401223, URL: http://www.etti.utcluj.ro

Declaraţia autorului

Subsemnata declar prin prezenta că ideile, partea de implementare şi experimentală, rezultatele, analizele şi concluziile cuprinse în această lucrarea constituie efortul meu propriu, mai puţin acele elemente ce nu-mi aparţin, pe care le indic şi le recunosc ca atare.

Declar de asemenea, că după ştiinţa mea, lucrarea în această formă este originală şi nu a mai fost niciodată prezentată sau depusă în alte locuri sau altor instituţii decât cele indicate în mod expres de mine.

În calitate de autor, cedez toate drepturile de utilizare şi modificare a lucrării de licenţă către Universitatea Tehnică din Cluj-Napoca.

Cluj-Napoca, 20 Iunie 2011

ABSOLVENT,
Teodora SZASZ

1. Numele şi prenumele studentului: Teodora SZASZ
2. Numele şi prenumele conducătorului: Prof. dr. ing. Mircea Florin VAIDA

SINTEZA LUCRĂRII DE LICENŢĂ

Enunţul temei: Modelarea sistemului nervos. Algoritmi de vizualizare 3D

Atunci când vorbim despre modelarea sistemului nervos, trebuie să ţinem cont de principalele semnale neuronale prezente în creierul uman, iar ele pot fi înregistrate fie considerând o singură celulă neuronală (numite „salturi neuronale”), fie extrăgând activitatea însumată a mai multor astfel de celule (numită „potenţial de câmp local”). Cele din urmă sunt cel mai frecvent analizate de către neurologi, deoarece pot conţine informaţie foarte utilă în diagnosticul şi tratamentul unor afecţiuni ale creierului. De aceea, aceste semnale au fost luate în considerare în implementarea algoritmilor pentru procesarea semnalelor în timp real. Cele mai întâlnite procesări sunt filtrările, convoluţiile şi corelaţiile. Lucrarea tratează modul de implementare a filtrelor, utilizând Transformata Fourier Rapidă şi convoluţia.

Extragerea informaţiilor utile din semnalele considerate este dependentă de modul în care se pot vizualiza imaginile medicale, deoarece în majoritatea operaţiilor pe creier, trebuie foarte bine precizat locul unde se află o anumită afecţiune. Pentru aceasta, diferite tehnici de vizualizare a imaginilor medicale au fost testate.

Mai mult, aceste tehnici au fost testate şi implementate în scopul de a fi un suport în operaţiile neuronale, fiind un început în crearea unui instrument în planificarea operaţiilor pe creier.

ABSTRACT

Enunţul temei: Modelling the Nervous System. 3D Visualization Algorithms

When we are talking about nervous system modelling, the most common neural signals must be considered, and these signals can be recorded from a single neural cell (being called “spikes”), or we can extract a sum of multiple neurons activity (being called “the local field potential”). The second ones are most examined by neurologists, because they carry a lot of useful information for very useful in the diagnosis and treatment of brain disorders. Therefore, these signals were considered in the implementation of the algorithms for real time signal processing. The most common techniques are the filtering, convolution and correlation. The paper discusses how we can implement some filters using Fast Fourier Transform method and convolution.

Extracting useful information from the considered signals is depended on how the medical images may be visualized, because during brain surgeries, we have to exactly take decisions about a certain disease is located. Therefore, different visualization techniques were tested.

Moreover, the visualization techniques have been implemented in order to be a support in during neurosurgeries, being a start point for developing a neurosurgery planning software.

CURRICULUM VITAE

Name and Surname: Teodora SZASZ
Birth Date: March 3rd, 1988

Education:

- 2007- 2011: student of Technical University of Cluj Napoca, Faculty of Electronics, Telecommunications and Information Technology, English line of study.
- 2003-2007: attended *Avram Iancu* Highschool, obtained the BAC Diploma and a certificate in computer operating skills and programming.

Experience:

- 2007 : Summer internship at SC Nokia SRL, Cluj Napoca for acquiring knowledge in mobile voice and data networking
- 2008-2012: Collaboration with Neurostar Company, Germany for acquiring knowledge in image processing and medical software development.

Technical Skills:

- Platforms: Windows XP/Vista/2007- experienced, Linux/Unix - basic.
- Programming Languages: C++ - medium; Java - basic; Phyton - basic.
- Scripting Languages: Tcl-basic, Praat - basic.
- Software and Technical Languages: Matlab - basic.
- Computer Aided Design Tools: OrCad - basic, LabView - basic.
- Basic knowledge in operating and configuration of electronics analysis equipment and telecommunications devices.
- Others: Asterisk – basic.

Foreign Languages:

- English: Speaking – Very good ; Writing – Very good ; Reading – Very good.
- German: Speaking – Fair; Writing – Good ; Reading – Very good.

Contact:

Address: Str. Aleea Gării, nr.3, bl. A4, sc.2, ap.14, Beclean, jud. Bistriţa-Năsăud, 425100.
Telephone: +40 754 644 571
E-mail address: dora.szasz@yahoo.com

Subsemnata Teodora SZASZ, prin prezenta declar că datele din acest Curriculum Vitae pot fi folosite de către Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei din Universitatea Tehnică din Cluj Napoca în scopuri de promovare şi orientare în cariera profesională, în următoarele condiţii:

- fără consultare prealabilă ☐
- după o consultare prealabilă cu subsemnatul ☒
- nu sunt de acord ☐

Table of Contents

Rezumatul Proiectului

1. Context

Medicina este un domeniu care evoluează şi se schimbă cu fiecare minut. Fiecare cercetare nouă contribuie la schimbarea tratamentelor şi a regimurilor care trebuie urmate într-o anumită ramură medicală. Cu ajutorul instrumentelor soft existente se pot face mult mai uşor diferite experimente, se pot simula diferite condiţii, sau chiar se pot coordona anumite operaţii care au nevoie de precizie.

Dacă ramura de interes este neurologia, fiecare programator care intenţionează să proiecteze o aplicaţie în acest domeniu, trebuie să înţeleagă foarte bine anumite principii care stau la baza neurologiei. Trebuie înţeles în primul rând modul în care percepem ceva, reacţionăm în anumite situaţii, învăţăm, sau suntem capabili să ne reamintim anumite lucruri. Într-adevăr, este foarte greu să înţelegem teoria din spatele acestor acţiuni relativ banale ale omului, însă au existat secole întregi de cercetare şi trudă în spatele întregii teorii neuronale. Acum, s-a ajuns într-un stadiu în care se studiază celulele neuronale la nivel de moleculă şi se încercă găsirea unei legături între comportamentul intern al moleculelor şi modul în care putem reacţiona în diferite situaţii. Mai mult, noile tehnici de prelucrare a imaginilor medicale, oferă posibilitatea de a vedea creierul uman şi de a corela regiunile creierului cu un anumit mod de a gândi, de a simţi. Mai mult, acest lucru se întâmplă şi prin extragerea anumitor tipuri de semnale neuronale; doar vizualizând un anumit semnal, putem prezice secţiunea de creier sau regiunea scalpului din care a fost extras, sau mai mult, se poate folosi în vederea manipulării anumitor dispozitive cu ajutorul semnalelor preluate de pe scalp (numite semnale EEG).

Există diferite sisteme de achiziţie a semnalelor neuronale, ele încercând să redea cât mai fidel posibil activitatea unei celule neuronale sau a unui grup de astfel de celule. De asemenea, prin procedeul invers, diverse sisteme de stimulare a celulelor neuronale au fost dezvoltate. Acestea sunt cel mai bun mod de a trata o afecţiune cronică a creierului (de exemplu boala Parkinson, care este asociată cu mişcări necontrolate repetate – tremurare). În acest sens, se stimulează zona afectată a creierului utilizând un generator care produce pulsuri de tensiune rectangulară, cu o amplitudine între 1 şi 3.5 V, o durată a pulsului între 60 şi 210 microsecunde şi o frecvenţă de 130-185 Hz. Atunci când zona afectată este stimulată, pacientul îşi revine din acea stare necontrolată, asemănătoare unui tremurat continuu. Această situaţie a fost prezentată pentru a putea realiza care este impactul pe care o anumită tehnică medicală, sau un anumit instrument soft îl poate avea asupra unui pacient şi pentru a conştientiza importanţa cu care domeniul medical trebuie privit.

Prin urmare, înaintea dezvoltării unei aplicaţii ce oferă suport în planificarea operaţiilor, este nevoie de acumularea a cât mai multor cunoştinţe despre tehnica folosită, şi înţelegerea scopului pentru care aplicaţia este construită. De asemenea trebuie testate toate modalităţile deja existente în care se poate proiecta acea aplicaţie, dar punctul de plecare trebuie să rămână cunoştinţele care sunt stăpânite cel mai bine, deşi nu ţin neapărat de domeniul medical.

De exemplu, pentru a putea crea un instrument care să proceseze semnale neuronale, fundamentele teoretice trebuie să fie bazate pe modul de procesare a semnalelor în general, putând fi aplicate, prin acumularea de cunoştinţe adiţionale, asupra anumitor tipuri de semnale. Totodată, dacă avem cunoştinţele de bază asupra anumitor prelucrări de imagini, şi înţelegerea modului în care acestea pot fi adaptate anumitor situaţii (de exemplu trecerea de la vizualizare bidimensională la una tridimensională).

2. Obiective

Obiectivele acestei lucrări constau în dezvoltarea unor algoritmi pentru modelarea semnalelor neuronale, prezentarea algoritmilor fundamentali de vizualizare tridimensională, cât şi a algoritmilor utilizaţi în vizualizarea imaginilor medicale. Totodată, s-a încercat evidenţierea dependenţei dintre procesarea semnalelor neuronale şi vizualizarea imaginilor medicale, pentru stabilirea şi tratarea secţiunii de creier care a fost afectată.

Pornind de la cunoştinţele deja existente în domeniul prelucrării numerice a semnalelor, lucrarea urmăreşte crearea unor algoritmi simpli de procesare în timp real a semnalelor neuronale. Pentru aceasta, diferite tipuri de semnale neuronale au fost considerate, accentul punându-se doar pe un anumit tip, „potenţialul de câmp local" (LFP), care reprezintă o însumare a activităţii a mai multor celule neuronale. Acest tip de semnal s-a constatat de-a lungul anilor a fi cel care conţine cea mai multă informaţie utilă despre activitatea celulelor neuronale dintr-o anumită secţiune de creier. Modul în care această informaţie poate fi extrasă utilizând diferite tehnici, este încă un domeniu de cercetare foarte activ, deoarece până acum s-a pus accentul mai mult pe modul în care semnalele LFP pot fi extrase utilizând anumite echipamente electronice. Această lucrare analizează acest aspect, dar şi instrumentele deja utilizate pentru procesarea semnalelor LFP. Cele mai cunoscute tipuri de filtrare: filtrare trece-jos, trece-sus, opreşte-bandă şi trece-bandă au fost implementate pentru procesare în timp real utilizând Transformata Fourier Rapidă şi Convoluţia dintre semnalul de intrare considerat şi răspunsul la impuls al filtrului considerat.

Apoi, s-au studiat şi testat diferiţi algoritmi de vizualizare 2D şi 3D, pentru a vedea care este cea mai bună soluţie pentru vizualizarea imaginilor medicale şi extragerii de date importante din aceste imagini, cât şi modalităţile prin putem manipula aceste date în vederea obţinerii altor date semnificative.

Deoarece s-au luat în considerare multiple direcţii atât pentru procesarea semnalelor neuronale, cât şi pentru vizualizarea imaginilor medicale, trebuie precizat că nu toate direcţiile investigate duc la rezultate viabile sau la o implementare finală.

3. Aspecte introductive

În această secţiune se vor prezenta scurt algoritmii utilizaţi pentru implementarea filtrelor descrise, dar şi aspecte importante privind modul de vizualizare a imaginilor medicale, şi ce anume presupune vizualizarea directă şi indirectă.

În primul rând, există diferite metode pentru măsurarea semnalelor biomedicale, cel mai important lucru în alegerea unei metode fiind minimizarea zgomotului şi mărirea vitezei de răspuns a sistemului. Patru metode sunt cel mai des folosite, acestea determinând şi tipul de semnal neuronal măsurat. Prima metodă constă în măsurarea activităţii unui anumit grup de celule neuronale, prin însumarea activităţii fiecăreia dintre ele. Aceasta este considerată cea mai simplă metodă, iar semnalele înregistrate sunt numite „potenţiale locale de câmp" şi au valori foarte mici (între 10 şi 500 de microvolţi). Deşi este cea mai simplă metodă, semnalul rezultat conţine informaţii foarte importante despre activitatea unei anumite secţiuni de creier. A doua metodă presupune detecţia activităţii unei celule neuronale dominante şi măsurarea acestui semnal rezultat (în acest caz semnalul rezultate este numit „salt"). Cea de-a treia metodă constă în măsurarea semnalelor mai multor celule neuronale, utilizând mai multe electrode ce sunt introduse pentru a putea înregistra semnalele. În final, cea de-a patra metodă este utilizată pentru înregistrarea potenţialului din interiorul unei celule neuronale.

Prima metodă din cele trei a fost considerată, rezultând „potenţialele locale de câmp" (LFP). Pentru extragerea acestui tip de semnal din semnalul original, se aplică un filtru trece jos,

cu scopul de a elimina „salturile” existente în semnal. Apoi, semnalul poate fi procesat pentru a obţine informaţiile dorite. Cele mai întâlnite procesări sunt filtrările, urmate de convoluţii şi autocorelaţii. Pentru obţinerea unei filtrări, semnalul considerat se împarte în segmente din care este extras spectrul de frecvenţă. Răspunsul la impuls corespunzător unei anumite filtrări este considerat, realizându-se mai apoi convoluţia dintre cele două semnale considerate (spectrul de frecvenţă a semnalului iniţial şi răspunsul la impuls al filtrului). Astfel, se va obţine spectrul semnalului filtrat, care poate fi mai apoi convertit în domeniul timp. Apoi, din răspunsul la impuls filtrului trece-jos, se poate obţine, prin inversiune spectrală, răspunsul la impuls al unui filtru trece-sus. Făcând convoluţia dintre cu spectrul semnalului original, vom obţine un semnal filtrat trece-sus. Pentru a obţine răspunsul la impuls corespunzător unui filtru opreşte bandă, vom aduna răspunsurile la impuls a unui filtru trece jos cu al unui filtru trece sus, iar apoi, prin inversiune spectrală vom putea obţine răspunsul la impuls al unui filtru trece-bandă. Pentru procesarea în timp real a semnalelor, aspecte legate de viteza procesării trebuie luate în considerare, de aceea, pentru transformarea din domeniul timp în domeniul frecvenţă a semnalelor, Transformata Fourier Rapidă a fost considerată, iar pentru transformarea din domeniul frecvenţă în domeniul timp, se foloseşte Inversa Transformatei Fourier Rapidă. Acesta este cea mai simplă descriere a modului în care aceste filtre au fost implementate. Multe alte aspecte legate de lungimea segmentului de intrare, lungimea răspunsului la impuls al filtrului şi modul în care se poate utiliza cel mai eficient au fost considerate. Mai multe informaţii despre modul în care au fost implementate filtrele se găsesc în **Capitolul 4** al acestei lucrări.

Trecând la partea de vizualizare a imaginilor medicale, câteva aspecte trebuie considerate înainte de a începe vizualizarea propriu-zisă. Câteva aspecte legate de modul în care se obţin imaginile tridimensionale CT şi MR sunt descrise în lucrare, şi formatul care este cel mai indicat să fie utilizat pentru scopul acesta. Mai mult, acest format afectează direct calitatea volumului 3D obţinut şi viteza cu care imaginile sunt procesate. În sensul acesta, aspecte legate de imaginile Dicom sunt prezentate în comparaţie cu alte tipuri de imagini care sunt capabile să stocheze 16 biţi de informaţie.

Apoi, câteva aspecte legate de grafica de pe calculator sun discutate. În mare, înainte de a începe orice tip de vizualizare 3D, trebuie sa ştim care sunt primitivele care din care vor fi alcătuit acel volum, modalităţile prin care poate fi setată lumina şi camera pentru o bună vizualizare a volumului şi sistemul de coordinate pe care poate fi reprezentat. Totodată, sunt descrise aspecte legate de maparea volumului în scenă.

Pentru a alege tehnicile de vizualizare folosite pentru implementarea aplicaţiei, au fost studiaţi diferiţi algoritmi pentru detecţia conturului volumului şi pentru vizualizarea directă a volumului. Multe alte proprietăţi au fost adăugate pentru a crea un aspect cât mai natural al pielii, a oaselor extrase şi a vizualizării întregului volum. Apoi, s-a precizat si modul în care din datele extrase putem obţine alte informaţii importante în luarea unor decizii pentru un anumit tratament aplicat, sau pentru detecţia unor structuri de creier afectate.

4. Implementarea algoritmilor pentru filtrare

În secţiunea anterioară s-a prezentat, succint, modul în care s-au dezvoltat algoritmii pentru filtrarea semnalului în timp real. **Figura 1** prezintă schema generală prin care se obţine semnalul filtrat. În implementare, pentru aplicarea Transformatei Fourier Rapidă, s-a folosit biblioteca C++ *FFTReal*, bibliotecă oferită de [L. de Soras, 2005]. Algoritmul din spatele acestei biblioteci elimină calculul părţilor imaginare, fiind considerate zero. Astfel, extragerea spectrului semnalului de intrare şi a filtrului se face mult mai repede decât dacă se utilizează algoritmul clasic de extragere a Transformatei Fourier Rapidă. Descrierea algoritmilor folosiţi pentru obţinerea filtrelor se află în secţiunea de implementare a lucrării.

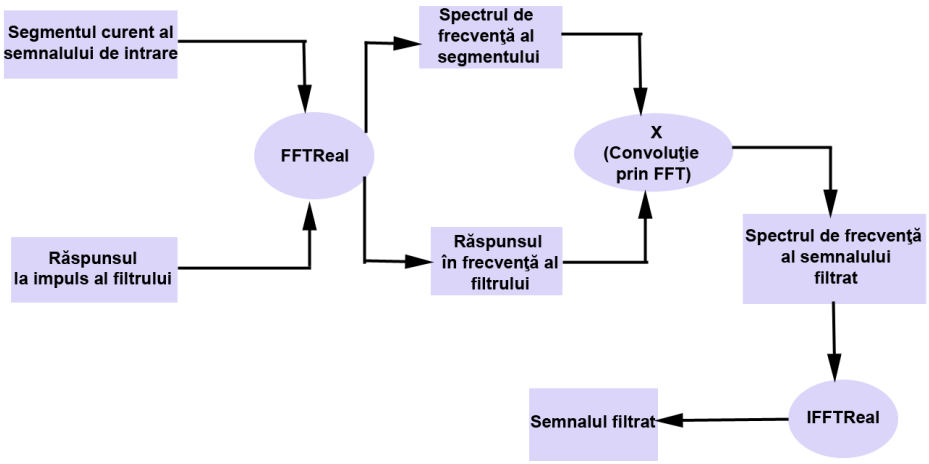


Figure 1 Schema-bloc generală pentru filtrarea semnalelor în timp real

S-a folosit limbajul C++ pentru implementarea filtrelor, creându-se clase pentru fiecare din filtrele implementate. În lucrare se prezintă şi dependenţele dintre filtre, cel mai bine ilustrate prin diagrama UML asociată lor.

5. Implementarea algoritmilor de vizualizare

Tehnicile de vizualizare au fost implementate folosind biblioteca de clase VTK. Această bibliotecă este construită în OpenGL, oferind o modalitate de vizualizare a structurilor de date şi diferiţi algoritmi pentru prelucrarea numerică a imaginilor. Deşi este dezvoltată în C++, se pot genera legături cu limbajele Tcl, Phyton şi Java. Procesul prin care se face această legătură se numeşte „împachetare" (în engleză, „wrapping") şi este foarte util în crearea interfeţelor grafice cu utilizatorul (GUI), cu costul de a scădea performanţa aplicaţiilor. Pentru vizualizare, s-a folosit limbajul Tcl, deoarece s-a dorit testarea lui în aplicaţii ce au la bază elemente de vizualizare. Totodată, acest limbaj este foarte uşor de utilizat pentru creare de GUI.

Pe lângă studiul asupra tehnicilor de vizualizare, s-a studiat modul în care anumite formate pentru imagini medicale poate influenţa rezultatul vizualizării. Imaginile utilizate au fost de tipul Dicom, acestea fiind transformate pe rând în fişiere RAW sau citite ca un fişier VTK. Mai multe detalii despre aceste formate se pot găsi în fundamentele teoretice ale acestei lucrări. De asemenea, se pot vedea diferenţe mari dintre utilizarea unui anumit format sau al altuia.

Figura 2 ilustrează diferenţa dintre citirea imaginilor Dicom, aşa cum sunt ele achiziţionate, incluzând fişierul-antet cu date despre pacient şi modul în care s-a realizat achiziţionarea imaginilor, şi transformarea acestora în format *.raw*, prin care informaţia din acest antet se înlocuieşte cu valori nule. Putem observa foarte uşor că pentru obţinerea unor rezultate bune pentru extragerea unor suprafeţe de contur, cum ar fi pielea, ştergerea informaţiei din aceste fişiere este cea mai bună metodă. Modul în care se pot extrage aceste contururi de suprafaţă a fost prezentat în secţiunea de implementare a acestei secţiuni, fiind descrise şi clasele şi metodele VTK utilizate. Pentru realizarea acestei extrageri, algoritmul *Marching Cubes* a fost utilizat, acesta fiind descris în [W.E.Lorensen et al., 1987]. Pe lângă algoritmul general, s-au folosit şi alte prelucrări oferite de VTK pentru a manipula modul de orientare a camerei, modul de vizualizare a volumului şi adăugarea unor funcţii de transfer pentru a da o culoare cât mai reală volumului generat. Mai multe informaţii privind modul în care s-a abordat acest proces (numit vizualizare indirectă) sunt oferite în partea de implementare şi de rezultate experimentale a acestei lucrări.

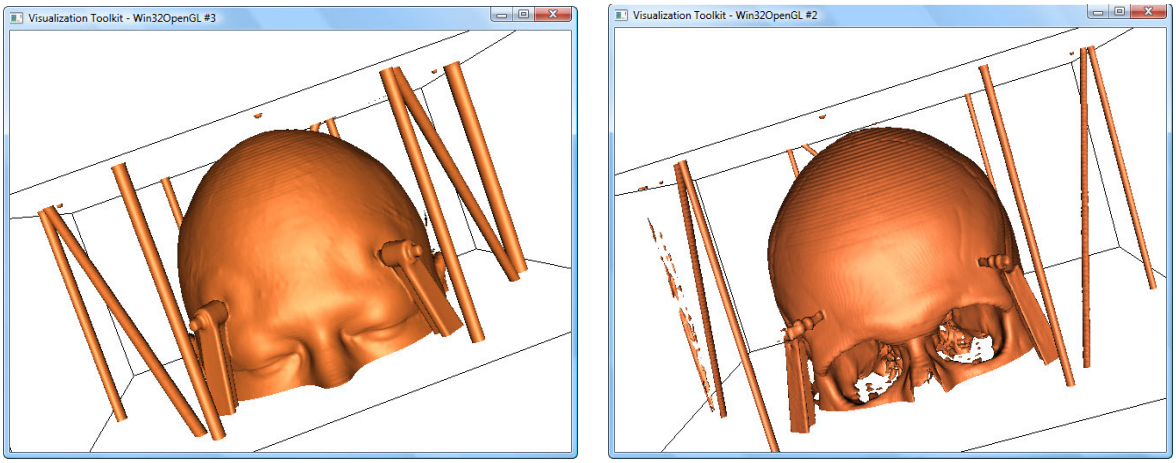


Figura 2 Extragerea conturului corespunzător pielii dintr-un set de imagini CT (stânga: utilizând imagini *.raw*; dreapta: utilizând imagini Dicom)

Totodată, a fost prezentat şi modul de vizualizare direct al volumelor. Se foloseşte o tehnică diferită de cea a extracţiei de contur, iar metoda folosită se numeşte „Ray Casting", fiind prezentată în partea de fundamente teoretice a lucrării. Şi în acest sens au fost considerate ambele metode de citire a fişierelor: Dicom şi fişiere RAW. În acest caz se poate observa foarte uşor că pentru realizarea unei aplicaţii ce implică vizualizarea volumelor, este de dorit eliminarea fişierului-antet conţinut de aceste imagini. **Figura 3** prezintă diferenţele dintre cele două metode, iar datorită vitezei foarte mici de procesare a acestor volume în cazul utilizării imaginilor Dicom, nu s-au mai adăugat alte proprietăţi volumului rezultat. Oricum, claritatea volumului rezultat în acest mod nu s-ar fi îmbunătăţit.

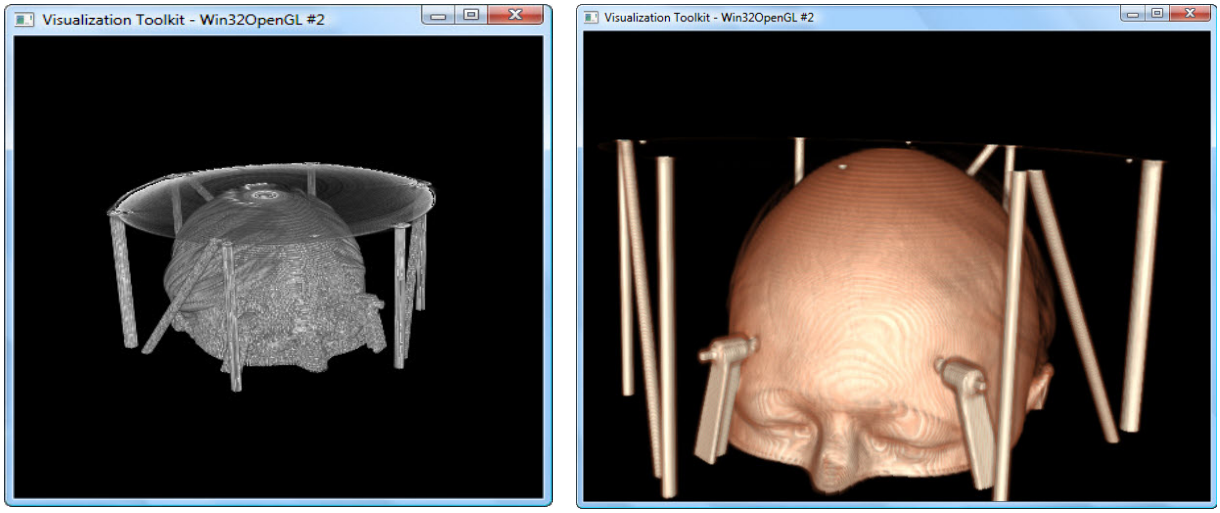


Figura 3 Vizualizarea directă a volumului obţinut din setul de imagini CT (stânga: utilizând imagini Dicom; dreapta: utilizând imagini *.raw*).

În secţiunea dedicată vizualizării 2D s-au prezentat modalităţile în care imaginile medical pot fi vizualizate, făcându-se o comparaţie între imaginile MR şi CT. Totodată s-a realizat vizualizarea celor trei moduri cele mai importante întâlnite în domeniul radiologiei: axial, sagital şi coronal. Câteva moduri de a manipula aceste imagini au fost prezentate (setarea contrastului şi a luminozităţii, setarea modului de vizualizare, mărirea/micşorarea imaginii, obţinerea imaginii oglindite). De asemenea, s-au prezentat pe scurt, clasele VTK care au stat la baza obţinerii acestor tehnici de prelucrare şi s-au evidenţiat informaţiile care cel mai des folosite de radiologi, cât şi a informaţiilor care ajută în extragerea suprafeţelor de contur sau de vizualizare a

volumului obţinut. Interfaţa propusă pentru vizualizarea 2D, dar şi opţiunile dedicate vizualizării 3D a imaginilor este prezentată în **Figura 4**, totodată fiind încărcată şi o imagine CT.

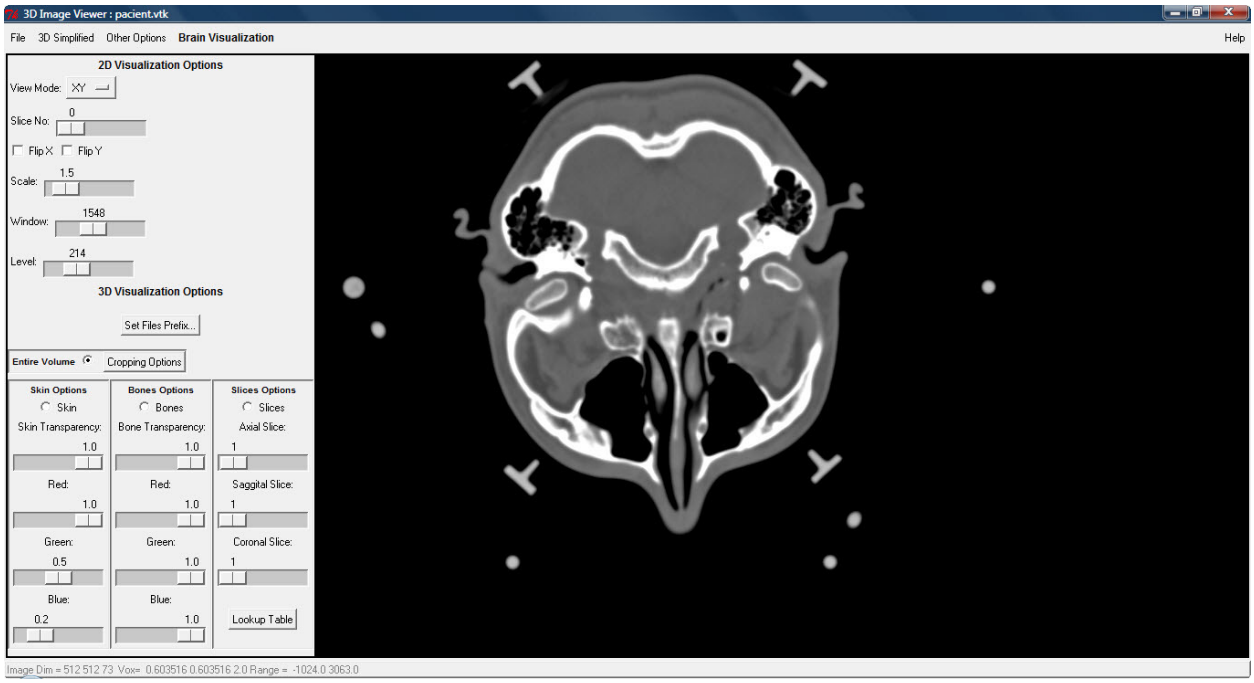


Figura 4 Imaginea numărul 0 dintr-un set de imagini CT, fără oglindire, mărită cu factorul 1.5, cu o valoare a luminozităţii de 1424 şi o valoare a contrastului de 214.

După încărcarea diferitelor tipuri de imagini medicale: CT şi MR, s-a putut constata şi diferenţa dintre cele două tipuri. Se poate observa în secţiunea de rezultate experimentale că decizia de a face un anumit tip de imagine depinde de ceea ce se doreşte a fi analizat. MRI ne oferă posibilitatea de a vizualiza tendoanele, ligamentele existente într-un anumit organ, pe când CT ne oferă posibilitatea de a vizualiza acel organ. De exemplu sângerarea unui anumit organ se poate observa mult mai uşor dacă se achiziţionează imagini CT, pe când, o tumoare pe acel organ este vizibil achiziţionând imagini MR. Mai multe detalii despre modul de achiziţionare a acestor imagini şi modul de vizualizare a lor sunt prezentate în secţiunea de rezultate experimentale.

6. Concluzii

Această lucrare a fost structurată în două mari secţiuni, urmărindu-se pe parcursul dezvoltării modul în care acestea pot fi îmbinate pentru a se ajunge la ceea ce se urmăreşte de fapt când se proiectează astfel de aplicaţii: realizarea unui instrument complex de preluare a semnalelor neuronale, vizualizare a lor şi aplicarea diferitelor tehnici de procesare pentru a obţine informaţia dorită de către neurolog. Apoi, pe baza acestei informaţiei, şi a informaţiei extrase din imaginile medicale achiziţionate se pot lua anumite decizii foarte importante în tratamentul unor boli cronice, cum ar fi locul de amplasare unei electrode cu rolul de a stimula secţiunea de creier afectată şi proprietăţile pe care impulsurile care stimulează această zonă trebuie să le îndeplinească pentru un anumit pacient.

În realizarea acestei lucrări, s-a pornit de la cunoştinţele deja existente în domeniul prelucrării numerice a semnalelor şi a imaginilor ce au fost dobândite de-a lungul anilor de studiu. Apoi, cercetând domeniul neurologiei, s-au dobândit cunoştinţe despre tipul de semnale neuronale existente, despre importanţa informaţiei pe care o conţin aceste semnale, şi modul în

care se pot extrage informaţii adiţionale care pot fi oferite ca suport pentru luarea diferitelor decizii legate de activitatea unei anumite secţiuni de creier. Apoi, cercetând modul de obţinere de a imaginilor medicale şi modul în care acestea pot fi vizualizate, se pot extrage informaţii vizuale despre acea secţiune afectată, şi despre localizarea ei în creier. Pentru vizualizarea secţiunilor de creier, diferite atlase ale creierului sunt oferite ca sprijin. Acestea pot fi tridimensionale, obţinute prin segmentarea imaginilor medicale, sau bidimensionale. Şi imaginile din atlasele bidimensionale pot oferi la fel de mult suport ca şi cele tridimensionale, însă trebuie integrate cu foarte mare acurateţe în aplicaţia proiectată, astfel încât să ilustreze exact poziţia acestora în interiorul creierului. Oricum, aceste atlase vin împreună cu informaţiile despre modul lor de achiziţionare şi modul în care acestea pot fi aliniate între ele.

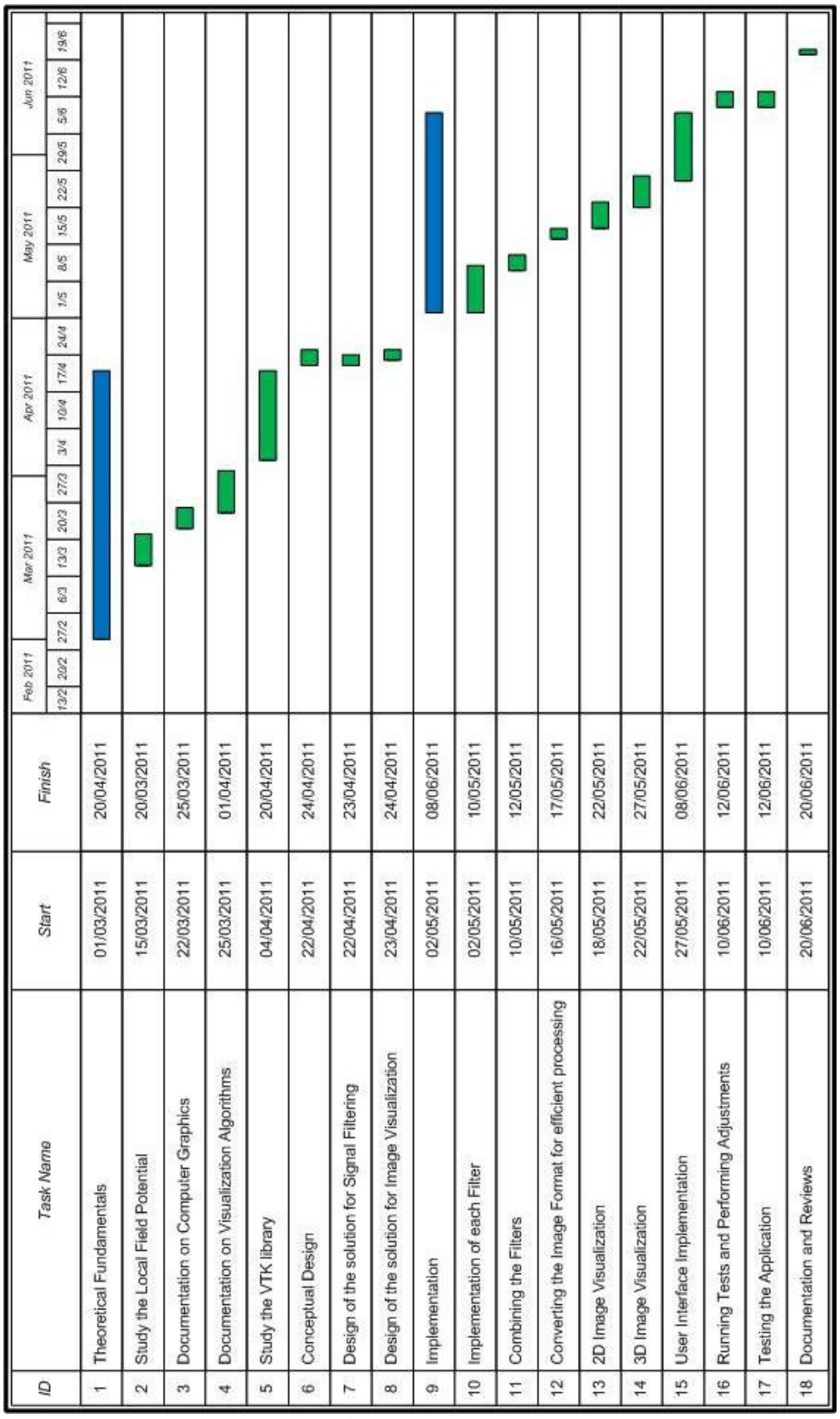
Studiul diferiţilor algoritmi de vizualizare a avut rolul familiarizării cu tehnicile de vizualizare existente în vederea găsirii unor soluţii optime de vizualizare pentru acest tip de imagini. Diverse metode au fost testate, şi mai mult, au fost îmbinate în vederea creării unei aplicaţii care poate fi startul proiectării unui instrument de planificare în domeniul neurochirurgiei.

În cazul studiului efectuat asupra semnalelor neuronale, trebuie aprofundat modul în care acestea furnizează informaţii utile despre activitatea unei celule neuronale sau a unei secţiuni de creier, noi metode de procesare a acestor semnale trebuie adăugate, cum ar fi corelaţia, autocorelaţia, precum şi crearea unei interfeţe cu aceste semnale şi rezultatul procesării lor. Totodată, trebuie achiziţionate astfel de semnale în diverse formate, citite şi vizualizate. Apoi, trebuie realizată sincronizarea acestora cu secţiunile de creier care sunt vizualizate cu ajutorul atlaselor de creier. Pentru aceasta, volumele vor fi procesate şi aliniate, astfel încât să ilustreze cât mai bine situaţia reală în care sunt dispuse în creier.

În cazul vizualizării de imagini medicale, se pot găsi metode pentru o procesare mai rapidă a lor, eventual implementarea unei aplicaţii utilizând limbajul C++, limbajul sub care biblioteca VTK a fost dezvoltată. Totodată, se pot cerceta tehnici de segmentare a imaginilor, în special pentru atlasele de creier bidimensionale, care în momentul de faţă sunt cele mai utilizate. Astfel, se vor putea genera volume tridimensionale pornind de la aceste imagini. Există deja numeroase instrumente soft care realizează acest lucru, însă majoritatea nu au opţiuni pentru filtrarea şi procesarea volumelor rezultate în diverse situaţii.

În principiu, se doreşte crearea unui instrument soft care să servească la planificarea operaţiilor pe creier, incluzând atât analiza semnalelor neuronale, cât şi a imaginilor medicale. Acest instrument trebuie să fie la început destinat unei anumite tehnici în neurochirurgie (de exemplu, DBS), urmând mai apoi să fie adaptat şi pentru tehnicile adiacente ei.

Work Planning



ID	Task Name	Start	Finish
1	Theoretical Fundamentals	01/03/2011	20/04/2011
2	Study the Local Field Potential	15/03/2011	20/03/2011
3	Documentation on Computer Graphics	22/03/2011	25/03/2011
4	Documentation on Visualization Algorithms	25/03/2011	01/04/2011
5	Study the VTK library	04/04/2011	20/04/2011
6	Conceptual Design	22/04/2011	24/04/2011
7	Design of the solution for Signal Filtering	22/04/2011	23/04/2011
8	Design of the solution for Image Visualization	23/04/2011	24/04/2011
9	Implementation	02/05/2011	08/06/2011
10	Implementation of each Filter	02/05/2011	10/05/2011
11	Combining the Filters	10/05/2011	12/05/2011
12	Converting the Image Format for efficient processing	16/05/2011	17/05/2011
13	2D Image Visualization	18/05/2011	22/05/2011
14	3D Image Visualization	22/05/2011	27/05/2011
15	User Interface Implementation	27/05/2011	08/06/2011
16	Running Tests and Performing Adjustments	10/06/2011	12/06/2011
17	Testing the Application	10/06/2011	12/06/2011
18	Documentation and Reviews	20/06/2011	20/06/2011

Figura5 Gantt Chart

List of Figures

List of Abbreviations

2D two- dimensional
3D three-dimensional
BPF Band Pass Filter
BSF Band Stop Filter
CT X-ray Computed Tomography
DBS Deep-Brain Stimulation
DICOM Digital Imaging and Communications in Medicine
DFT Discrete Fourier Transform
DSP Digital Signal Processing
DTI Diffusion Tensor Imaging
FFT Fast Fourier Transform
GUI Graphical User Interface
HPF High Pass Filter
IFFT Inverse Fast Fourier Transform
ITK Insight Toolkit
LFP Local Field Potential
LPF Low Pass Filter
MIP Maximum Intensity Projection
MRI Magnetic Resonance Imaging
MPR Multi-Planar Reconstruction
PD Parkinson's disease
PET Position Emission Tomography
EEG electroencephalography
TCL Tool Command Language
VTK Visualization Toolkit

Chapter 1. Introduction

Medical imaging is one of the most important fields in constant evolution, being a set of techniques and processes that are employed to create images of the human body for medical and clinical purposes. Its rapid development and proliferation has changed how medicine is practiced in the modern world. Medical imaging allows doctors and medical practitioners to non-invasively observe and analyze life saving information. It has gone beyond mere visualization and inspection of anatomic structures. Its application has expanded to include its use as a tool for surgical planning and simulation, intra-operative navigation, radiotherapy planning and for tracking the progress of a disease. There exist many perspectives in preventive medicine, diagnosis, or treatment. Nowadays, many imaging techniques are in continuous development, but even if we can get very accurate images, image processing is necessary to use these data, especially when we are looking for a diagnosis or a treatment of a disease. The main purpose of the research in the visualization field was to test some visualization algorithms and their rendering speed, and also the information that can be extracted from medical images and the mode in which this information can be used.

Dependent to medical visualization, the problem of predicting behavior from the hard or brain activity has attracted considerable attention over the past decades. If we are referring especially to brain activity, two main types of signals are most considered: *the spikes* (extracted from a single neural cell – Single Cell Recording) and the *local field potentials* (viewed as a sum of multiple neurons activity – Multiple Cell Recording). If we are referring to spikes, acquiring and holding single cells for long periods of time is considered a great experimental challenge, so there exists the need for more reliable control signal, and these are the local field potentials. The Local Field Potential (LFP) refers to the low frequency component of the recorded neural activity, which is supposed to reflect the combined synaptic activity of multiple neurons. These signals are easier to be measured than single units, and have been shown to carry a lot of useful information. However, algorithms to extract information from the LFP are still a research subject. Considering this, some common algorithms were implemented in order to be used with LFP, under the real-time acquisition mode.

The research activity for this thesis is structured in two parts. *In the first part*, some action potential recording techniques are presented together with the most common algorithms that are used to process local field potential signals. We must consider that, even if we will deal with simple algorithms, especially the algorithms for filter designing, they were not implemented yet in C++ programming language for real time proposes, maybe because of the complexity this programming language implies, and the complexity of the FFT (Fast Fourier Transform) algorithm. Anyway, working in C++ language is the best option for developing any medical application (for both signals and image processing) because of the higher processing speed it can provide.

In the second part, the basic visualization algorithms are presented using a C++ library class - Visualization Toolkit. I chose to study different visualization algorithms in order to observe which ones are the best options for medical visualization. The main used techniques are presented and some comparisons between them are described. Then, most of algorithms were integrated in an application that has the goal in visualization of DICOM images (especially the CT ones) and in creating isosurfaces and visualizing the reconstructed volume. In medical treatment planning many data entries of different types are encountered, and the data sets are usually quite large. Nevertheless, fast and accurate visualization methods are needed for successful treatment planning. Often it is important to visualize different data in combination. This allows the user, for example, to judge how good a reconstructed geometry approximates the real anatomic situation, how numerically calculated quantities depend on the underlying patient model, or which tissue compartments are primarily affected by certain effects.

Every year, millions of images of patients are taken, in varying dimensions and size. Most of them are three-dimensional or four-dimensional images of patients taken in order to assist in diagnosis and therapy. Every year, the neuroscientists are increasingly gathering large time series data sets in the form of multichannel electrophysiological recordings. The availability of such data has brought with it new challenges for analysis, and has created a pressing need for the development of software tools for storing and analyzing neural signals.

Even we are referring to medical imaging, or to different neural signal processing techniques, the most important thing in the neurosurgery domain is the accuracy placement of the recording or stimulation electrode in the appropriate neuroanatomical target. Nowadays, we can easily observe that neurosurgical navigation systems that exist for clinical applications are lacking beginning with the area of non-human primate research. It is very difficult to design a good system, which it includes both visualization techniques, and signal processing methods, considering the aspects mentioned above. Moreover, there exist controversies regarding different processes that are present in the human body - for example, in Deep Brain Stimulation (DBS), an invasive neurosurgical procedure described in the **Chapter 2**, we cannot know for sure if in the affected structure of the brain, the signals must be inhibited, or stimulated. Under these conditions it is very hard, or even impossible to satisfy the all criteria a good neurosurgical software system needs, and to be used for neurosurgeons over the entire world.

A variety of research modalities, have been employed to study different neurosurgical mechanisms. Electrophysiological recordings of neural activity, biochemical analysis, computer modeling and imaging studies provided a lot of methods to analyze the brain. The main goal remains to consider this contrasting results from these methods and to try to understand which are the most important aspects of a certain method. Then, to try to design a software application that is able to meet the all critical situations, to be flexible, so to be easily integrated on different platforms, and to have a very fast processing time.

Concluding, the main goal of the thesis was to understand the neural signaling principles, and how these signals are recorded, which are the most used neural signals, and the best modalities they can be processed. For this purpose, some typical algorithms for real-time signal processing were studied and implemented. These algorithms are not yet integrated with neural signals, because it is hard to find a library that is able to read the signals provided by the recording system. For this, more research need to be done in order to see how the signal may be read (maybe there exists some C++ dedicated) libraries. Moreover, more medical imaging visualization techniques and platforms were studied in order to test their capabilities and to gain the basic knowledge regarding the medical visualization. The 3D visualization algorithms were tested using VTK C++ library and Tcl scripting language described in the **Chapter 3.** This is not the best solution if we consider the rendering time of the volume, but it is a great solution if we want to familiarize with the medical visualization, and to test different visualization techniques.

Chapter 2. State of the Art

This chapter outlines part of the research I have made for the thesis. Because of the two different fields, the neural part and the visualization part, the research focuses on the two main directions.

The first line of the research investigates the existing documentation and systems involved in surgical planning, the focus being on the surgical therapy in the treatment of movement disorders, therapy called Deep brain stimulation (DBS) (**Figure 2-1**).

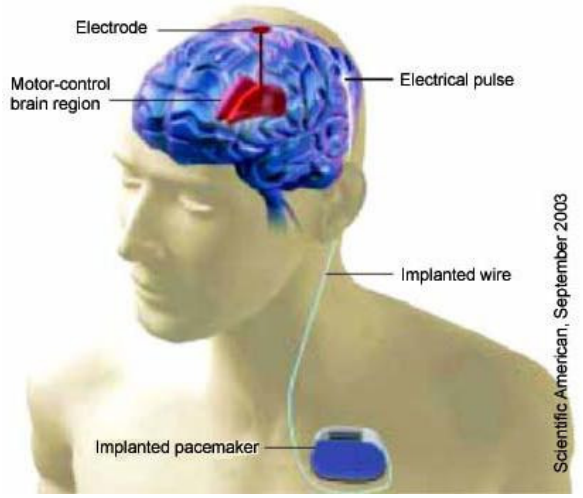


Figure 2-1 Deep brain stimulation. Electrode is implanted permanently in motor control region of the brain. Electrical pulse generator is placed underneath the skin and connected to the electrode, providing constant stimulation.

It is an invasive surgical procedure, used as a treatment for severe forms of epilepsy, Parkinson's disease, obsessive-compulsive disorder and depression [Benabid99]. The neurosurgical process consists in application of a high frequency electrical stimulation through a four-contact electrode placed permanently in the brain. Then, the electrode is connected to a battery-powered pulse generator implanted in the patient's chest. Before the electrode implantation, the neurologist must determine which is the best position the electrodes may be placed. Usually, the patient reacts in different ways because of the stimulation (for example less tremor, lost of the consciousness), but it is very important for the neurologist to analyze the signals that are coming from the patient's brain. Actually, only a small area of the brain is analyzed, the one corresponding with the disease the patient suffers. Like in the case of medical imaging, when the images must be processed in different ways before being analyzed, the signals (named local field potentials) must be processed in order to help the neurologist in taking some decisions regarding the best place to determine the final position of the electrode (also called "lead") in the brain. For this we need a recording system, as accurate as possible and some fast, real-time signal processing algorithms able to analyze the entire signal that is coming.

The second line of the research investigates which are the best solutions in medical visualization. A surgery planning software needs good visualization methods. For this part the best visualization solutions are briefly presented, and their main application area in medicine.

2.1. LFP Systems

All the existing recording systems include, besides the data acquisition electronic circuit, some tools for processing different signals. Signals from multiple electrodes (channels) can be amplified, recorded, and analyzed. Usually, such a system can record multiple types of neural signals, such as spikes, LFP, EEC, ECG, and ECoG. Spike activity from acute or chronically implanted electrodes can be monitored, depending on the channel. Of course, the best results will bring the monitoring tool that is dedicated for few such a signals (for example, only for spikes and LFPs).

One such a system is described in [Y. Perelman, R.Ginosar, 2007], in which a mixed-signal processor for multichannel neuronal recording design is proposed. They developed an electronic system, which is able to receive 12 differential-input channels of implanted recording electrodes. The signals are split at about 200 Hz to low-frequency LFP and high-frequency spike. A programmable cutoff filter has the role to eliminate the dc component at the input. But, we can easily observe that all the signal processing tasks are electronically adjusted, and it is not mentioned any software tool for manipulating the recorded signals.

Another example of such a system is one LFP dedicated and is the property of "Thomas Recording" company [Thomas Recording Gmbh]. It is an electronically filter/amplifier system, including a low-noise differential amplifier with adjustable gain, a low-pass filter with internal switchable cu-off-frequency, and an adjustable cut-off-frequency. As we may predict, they provide no software tool for manipulating the LFP signals. So, while sophisticated methods for analyzing multichannel time series have been developed over the past several decades in statistic and signal processing, the lack of a unified, user-friendly platform that implements these methods is a critical bottleneck in mining large neuroscientific datasets.

The most known toolbox for LFP analysis is *Chronux* (www.chronux.org). It is a widely used open-source Matlab Toolbox for LFP analysis, including a set of Matlab scripts. It has been developed for the analysis of neural data, being implemented as a Matlab library [P. Andrews, et al.]. The current version of Chronux includes a Matlab toolbox for signal processing of neural time series data, some specialized mini-packages for spike sorting, local regression, audio segmentation and other tasks. A GUI is also provided, containing a number of features to the analysis of EEG data. The principal toolbox of the Chronux is the spectral analysis toolbox, being the most used component. It is able to compute the spectrum of one or more time series data as well as the coherence between two simultaneously measured time series.

In the **Figure 2-2** we can see how Chronux works with LFP signals. The LFP signals that are provided at the toolbox download, and for visualization, the tutorial from [K.P. Purpura, 2008] was taken into account. As the tutorial describes, a first step to take in exploratory data analysis is to construct a summary of neural activity that is aligned with the appearance of a stimulus or some behaviorally relevant event. In this case, the signals were recorded from a monkey brain which is challenged with a delay period during which it must remember the location of a visual target that was cued at the beginning of a trial. We can observe that each 3 seconds trial is composed of a 1 second baseline period followed by a 2 second period containing the delay and response periods. We can observe that these signals demonstrate a change in activity at the start of the delay period. Then, the Chronux toolbox will be further be used to characterize the neural activity in these recordings (more information about how the signals may be processed using Chronux, we can find in the [K.P. Purpura, 2008] tutorial.

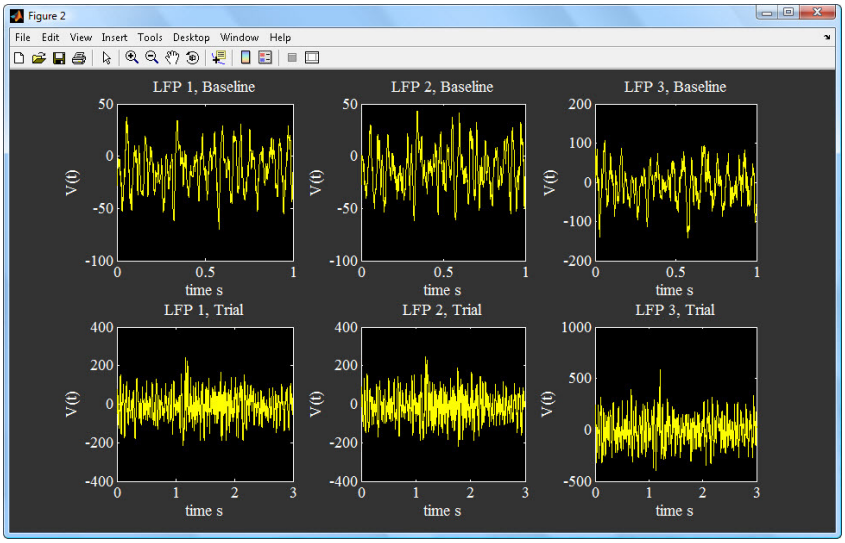


Figure 2-2 LFP signals visualized with the Chronux toolbox

2.2. Visualization Software

Frequently used software tools are the Visualization Toolkit and the Insight Segmentation and Registration Toolkit (more about VTK and ITK can be found in **Chapter3)**, they being also used as a solid foundation for many other systems. These two software tools are Kitware company products, and besides these, they also have some good visualization tools that include also powerful interfaces: ActiViz, VolView, MIDAS, CDashPro, ParaView and IGSTK. Other tools, which are not belonging to Kitware, and need to be mentioned, are: SciRun, Amira, MeVisLab, MITK, VolV, BioImageSuite and 3D Slicer. The most of these tools include powerful interface and interactive systems for volume visualization and are designed to be flexible for different purposes. Following, there is a briefly description of the tools that are most used in medical visualization.

3D Slicer – is a tool created for the purpose of subject specific image analysis and visualization. It is widely used in clinical research, especially for image-guided therapy research, because is able to construct and visualize collections of MRI data that are available pre- and intra-operatively to allow for the acquisition of spatial coordinates for instrument tracking. It has also the option for manual segmentation.

ParaView – is a program built using VTK libraries. The main feature is that it is able to analyze extremely large datasets using distributed memory computing resources. If it is used on supercomputers it can analyze datasets of terascale. In the **Figure 2-2** a 3D segmented brain model atlas is visualized using ParaView. Different brain sections can be labeled with different colors, or can be visualized one by one. The atlas contains *.vtk* format files including 148 brain structures and is achieved from [Halle M. et al., 2011]. More information about this 3D brain atlas can be found in the **Chapter 4**.

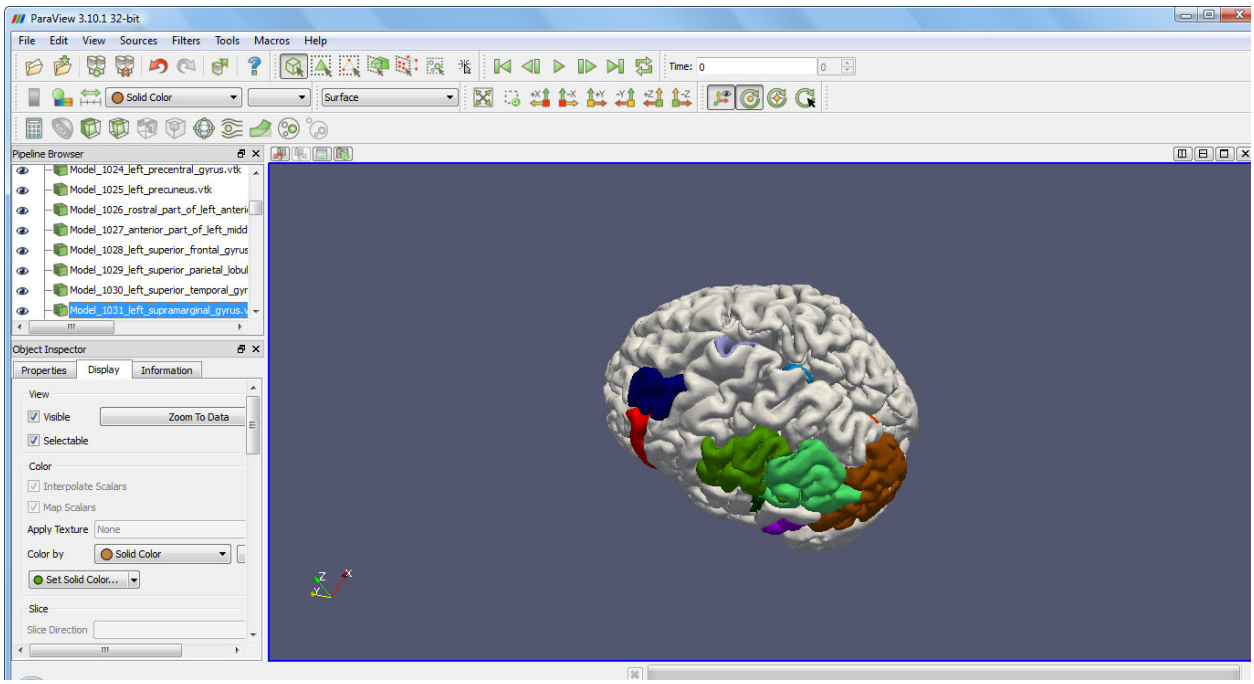


Figure 2-3 Visualization of a 3D atlas using Paraview tool.

BioImageSuite – it belongs to *National Institutes of Health (NIH) from Yale* and it comes with a great support for the beginners in medical visualization field using VTK and Tcl, providing the free online book *An Introduction to Programming for Medical Image Analysis with the Visualization Toolkit* [Yale09]. It is dedicated for neural, cardiac and abdominal image analysis and supports manual segmentation and registration.

Amira – the main feature of this tool is that it includes the Open-Inventor library that offers a powerful set of 3D viewer classes that are supporting interaction.

All the software instruments described above are powerful toolboxes, and they are very used in the research area of neuroscience domain. Most of them were developed for general tridimensional visualization, not for neuroscience purposes, but they can provide good results in volume visualization of the brain, or segmented volume visualization. They also include the most used image processing techniques in the field of medical visualization, like: zooming, padding, contour extraction, and the interaction with the volume.

Chapter 3. Theoretical Fundamentals

In this chapter are presented the most important theoretical aspects regarding the signaling between neurons, the way the neural signals are recorded, and the way we can design filters for extracting useful information from these signals. Furthermore, it is described the most important aspects of visualization, computer graphics aspects, the most used algorithms for isocontour extraction and direct volume visualization. The most important aspects are illustrated, very helpful in the implementation parts, so we will often see references to the chapters that include the implementation and the experimental results.

Even if is hard to believe, maybe the most time of designing a certain application is spent trying to understand the theoretical aspects of a technique, or an algorithm intended to be used. After we are able to understand the basic principles of a certain domain, we can try to find which are the main problems a certain domain deals with. Then, we try to find some solutions already implemented, or some directions to solve them. Finally, we can implement an original application based on the experience accumulated in the previous steps.

3.1. Brain Model

The most known book for neuroscientists, where is presented the entire neural system, from the structure of the brain to neural cells and the networks that are formed within these cells is [E.R.Kandel et al., 2000]. In the following is a synthesis of the most important parts which must be considered when we have to develop applications in the neuroscience domain.

3.1.1. Structural and Functional Properties of Neurons

The nervous system has two main classes of cells: *nerve cells* (neurons) and *glial cells* (glia). There are between 10 and 50 times more glia than neurons in the central nervous system of vertebrates. The name "glia" derives from the Greek term for glue, but their role is not to hold nerve cells together. They are divided into two major classes (microglia and microglia) and they surround the cell bodies, axons and dendrites of neurons. Glia is not directly involved in information processing, but they have some vital roles in supporting the neurons and promoting efficient signaling between neurons.

The cells of the nervous system vary more than those in other part of the body. All neurons have common features that distinguish them from cells in other tissues. An example is that they are highly polarized. Then, the cell functions are compartmentalized, and arranged that contributes significantly to the processing of the electrical signals. Typically a neuron has four morphological defined regions: *the cell body* (the storehouse of genetic information), *dendrites* (the input elements of the neuron), *the axon* (the transmitting element of neurons), and *the presynaptic terminals*. Each of these regions has a distinct role in the generation of signals and the communication of signals between nerve cells.

The cell body, named *soma*, is the metabolic center of the cell. Its structure contains the nucleus, which stores the genes of the cell, the endoplasmic reticulum: an extension of the nucleus where the cell's proteins are synthesized. The cell body usually gives rise to two kinds of processes: several short *dendrites* and one, long, tubular *axon*. The dendrites are the main apparatus for receiving incoming signals from other nerve cells. The axon extends away from the

cell body and is the main conducting unit for carrying signals to other neurons. The distance along which the axon can convey electrical signals is ranging from 0.1 mm to 3m. These electrical signals are called *action potentials* and are rapid, transient, all-or-none nerve impulses, with amplitude of 100mV and duration about 1 ms. Action potentials are initiated at a specialized trigger region, called *axon hillock* (initial segment of the axon). Then, the electrical signals are conducted down the axon without failure or distortion at rates of 1-100 meters per second. Because the action potential is an all-or-none impulse that is regenerated at regular intervals along the axon, the amplitude remains constant during the traveling.

The action potentials are very important because they constitute the signals by which the brain receives, analyzes, and conveys information. These signals are highly stereotyped throughout the nervous system, even though they are initiated by a great variety of events in the environment that impinge on our bodies – from light to mechanical contact, from odorants to pressure waves. For example, the signals that convey information about vision are identical to those that carry information about odors. This result describes another key in understanding the brain functions: the information conveyed by an action potential is determined not by the form of the signal but by the path the signal travels in the brain. The brain analyzes and interprets these patterns of incoming electrical signals and in this way creates the human sensations: sight, touch, taste, smell, and sound.

The *myelin sheath* has the insulation role and also increases the speed by which action potentials are conducted. This sheath is interrupted at regular intervals by the nodes of Ranvier. At these nodes the action potential is regenerated.

Near its end, the tubular axon divides into fine branches that form communication sites with other neurons. The point of communication is called a *synapse*, the nerve cell that transmits a signal being known as the *presynaptic cell.* The swollen ends of the axon's branches are the terminals from which the transmission is initiated, and these terminals are called *presynaptic terminals*. The cell that receives the signal is called *postsynaptic cell*. Since the two cells are separated by a space named *synaptic cleft*, the presynaptic cell does not actually communicate or touch with the postsynaptic cell.

Ramón y Cajal was the first anatomist which illustrated the structure of the neuron and was able to describe the differences between classes of nerve cells and to map the precise connections between a good many of them. He announced two principles of neuronal organization, very important for studying communication in the nervous system.

The first one is called *the principle of dynamic polarization* and it states that the electrical signals within a nerve cell flow only in one direction: from the receiving sites of the neuron (the dendrites and the cell body) to the trigger region at the axon. From there, the action potential is propagated unidirectional along the entire length of the axon to the cell's presynaptic terminals.

The second one, *the principle of connectional specificity,* states that the nerve cells do not connect indiscriminately with one other to form random networks. Each cell makes specific connections with certain postsynaptic target cells but not with others.

Ramón y Cajal was the first what realized that the feature that most distinguishes one neuron from another is the shape. He also classified the neurons in three groups: *unipolar*, *bipolar*, and *multipolar* according to the number of processes that originate from the cell body. The *unipolar neurons* have a single primary process which gives rise to many branches (one branch which serves as the axon and the other branches function as dendritic structures). Bipolar neurons give rise to two processes: a dendrite that conveys information from the periphery of the body, and an axon that carries information toward the central nervous system. The *multipolar neurons* have a single axon and many dendrites emerging from various points around the cell body.

Another classification of neurons was given according the functional group they belong: *sensory, motor*, and *interneuronal*. The *sensory neurons* carry information from the body's periphery into the nervous system for the purpose of perception and motor coordination. *Motor neurons* carry commands from the brain or spinal cord to muscles and glands and *interneurons*, the largest class, represent the all nerve cells that are not specifically sensory or motor. The

interneurons can be *relay (projection) interneurons* and they have long axons, conveying signals over large distances, from one brain region to another; the other group of interneurons is *local interneurons* which have short axons and process information within local circuits.

All the behavioral functions of the brain - the processing of sensory information, the programming of motor and emotional responses, memory - are carried out by specific interconnected neurons.

Any behavior (for example a stretch reflex) is formed by the participation of the sensory and nerve cell which sequentially generates four different signals at different sites within the cell: an input signal, a trigger signal, a conducting signal, and an output signal. Regardless of cell size and shape, almost all neurons can be described by a model neuron that has four functional components, or regions, that generate the four types of signals: a local input component, a trigger (summing or integrative) component, a long-range conducting (signaling) component, and an output component. This model is the description of the Ramón y Cajal's principle of dynamic polarization.

3.2. Brain Signaling Model

3.2.1. Neural Signaling

Neural signaling depends on rapid changes in the electrical potential difference across nerve cell membranes. The changes in the membrane potential can be generated by individual sensory cells in response to very small stimuli: receptors in the eye respond to a single photon of light; olfactory neurons detect a single molecule of odorant. So, signaling in the brain is proportional with the ability of the neurons to respond to these small stimuli, by producing rapid changes in the electrical potential difference across nerve cell membranes.

The action potential is the conducting signal of the neuron and is all-or-none. During an action potential the membrane potential changes quickly, up to 500 volts per second. The “all-or-none” property means that while stimuli are below a threshold will not produce a signal. All stimuli above the threshold produce the same signal. It does not matter how much the stimuli vary in intensity or duration, the amplitude and duration of each action potential are pretty much the same. In addition, the action potential does not decay as it travels along the axon to its target- a distance that can measure three meters in length- because it is periodically regenerated. The rate of travelling can be as fast as 100 meters per second.

Professor Kandel formulates the following question in [E.R.Kandel et al., 2000]: How do the neuronal signals carry specific behavioral information, if the action potentials are stereotyped? In particular, how we can distinguish a signal that carries pain information about a bee sting, for example from a signal that carries visual information or voluntary movement? The answer is the pathways along which each signal travels.

There are also neurons that do not generate action potentials, for example local interneurons without a conductive component—they have no axon or such a short one that a conducted signal is not required. In these neurons the input signals are summed and spread passively to the nearby terminal region, where transmitter is released. There are also neurons that lack a steady resting potential and are spontaneously active. Even cells with the same structure can have different molecular components, resulting in different combinations of ion channels providing neurons with various thresholds, excitability properties and firing patterns. So, the neurons with different ion channels can convey different signals.

Neurons also differ in the chemical transmitters they use to transmit information to other neurons, and the receptors used for receiving information from other neurons. Many drugs that

act on the brain affect the actions of a specific chemical transmitter or a type of receptor for a given transmitter. For example, in Parkinson's disease, a disorder of voluntary movement there is damaged a small population of interneurons that use dopamine as a chemical transmitter. Other diseases may be selective even within the neuron, affecting only the receptive elements, the cell body or the axon. As we can anticipate, because the nervous system has so many cell types and variations at the molecular level, it is susceptible to more diseases than any other organ of the body.

As we saw, electrical and chemical signals ensure the propagation of information within and between neurons. Transient electrical signals are important for carrying time-sensitive information rapidly and over long distances. The electrical signals are: *receptor potentials*, *synaptic potentials* and *action potentials* and they all are produced by temporary changes in the current flow into and out of the cell that derive the electrical potential across the cell membrane away from its resting value. The current flow is controlled by the ions channels in the cell membrane. There are two types of ion channels: resting and gated, and are grouped according to their role in the neuronal signaling. *Resting channels* normally are open, and are not influenced significantly by potential across the membrane. They are very important in maintaining the resting membrane potential (the electrical potential across the membrane in the absence of signaling). The *gated channels* are closed when the membrane is at rest and their probability of opening depends on the changes in the membrane potential.

Every neuron has a separation of charges across its cell membrane consisting of a thin layer of positive and negative ions spread over the inner and outer surfaces of cell membrane. At the rest, the nerve cell has an excess of negative charges inside the membrane and an excess of positive charges inside the membrane. From this charge separation results a difference of electrical potential (voltage) across the membrane, which is called *membrane potential*, denoted Vm, and is defined as: Vm = Vin – Vout, where: Vin is the potential inside the cell, and Vout is the potential on the outside. At the rest, the potential is called *resting membrane potential* and Vm=Vin, because in this case the potential outside the cell is zero. In this particular case, its range is -60mV to -70mV. All electrical signaling involves brief changes from the resting membrane potential due to alterations in the flow of electrical current across the cell membrane resulting from the opening and closing of ion channels.

The ions carry the electric current that flows into and out of the cell and they can be positively charged (cations) and negatively charged (anions). It results that in an ionic solution, cations move in the direction of the electric current and anions in the opposite direction. Despite of the net flow of cations or anions into or out of the cell, the charge separation across the resting membrane is disturbed, so the polarization of the membrane is disturbed. If there is a reduction of charge separation, we have a less negative membrane potential and this is called *depolarization*. The inverse process, the increase in charge separation leads to a more negative membrane potential and it is called *hyperpolarization*. When depolarization approaches a given threshold, the cell responds actively with the opening of voltage-gated ion channels, which at the threshold produces an *action potential*.

3.2.2. Techniques for Recording Action Potential

There are several recording techniques used for measuring bioelectric signals. These techniques range from simple voltage amplification (extracellular recording) to sophisticated closed-loop control using negative feedback (voltage clamping). The most important thing for a designer of such a system is to minimize the noise and to have a faster response.

As we saw, a cell derives its electrical properties mostly from the electrical properties of its membrane. A membrane, in turn, acquires its properties from its lipids and proteins, such as ion channels and transporters. An electrical potential difference exists between the interior and exterior of cells. This potential is also termed *voltage*. There exists electrophysiological equipment that is able to measure the potential differences in biological systems. It can also measure current, which is the flow of the electrical charge passing a point per unit time. Usually, currents measured by electrophysiological equipment range from picoamperes to microamperes.

There are two important rules about currents in order to understand the electrophysiological phenomena: (1) current is conserved at a branch point and (2) current always flows in a complete circuit. The **Figure 3-1** illustrates an example of such a circuit.

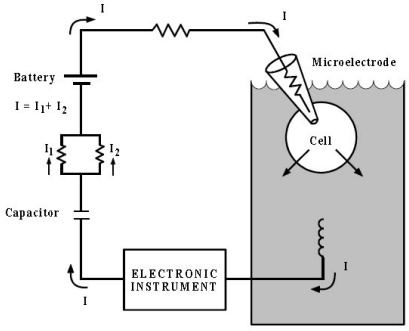


Figure 3-1 The functional model of an electrophysiological circuit [R. Sherman, 1993]

An electrophysiological measurement depends on two factors: de design of the electronic instrumentation and the properties and fabrication of glass micropipettes (microelectrodes). The microelectrodes are used for both intracellular recording and for patch recording. Actually, the intracellular electrodes measures the resting potential of a cell whose membrane contain only open K+ channels. The equivalent circuit looks like the one in the **Figure 3-2**.

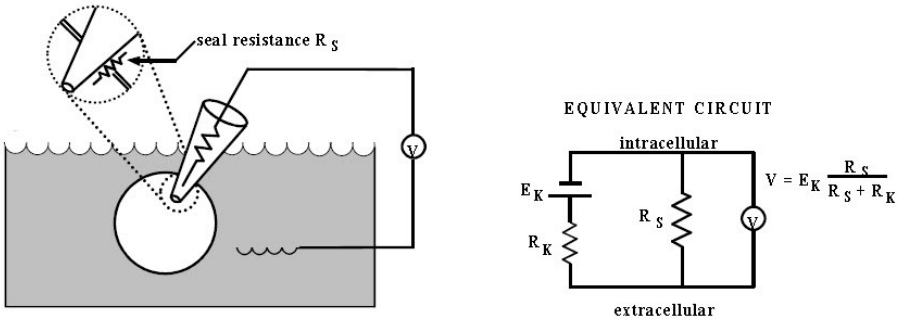


Figure 3-2 The equivalent circuit of an electrophysiological circuit [R. Sherman, 1993]

An electrophysiological setup has four main requirements:

- *Environment*: the preparation must be maintained healthy;
- *Optics*: to visualize the preparation;
- *Mechanics*: the microelectrode must be precisely positioned;
- *Electronics*: the amplification and the recording of the signal.

The entire setup is used for recording field potentials in brain slices. The optical and mechanical requirements help in accuracy placement of the electrode on a particular region of the brain, and a particular cell. Electrodes convert ionic current in solution into electron current

in wires; they are made of materials that can participate in a reversible reaction with one of the ions in the solution. There exists also a microscope that magnifies up to 300 or 400 fold, equipped with some kind of contrast enhancement.

A micromanipulator keeps the microelectrode stable and permits fine, smooth movement down to a couple of microns per second, at most. Currently, there are three main types of remote-controlled micromanipulators available: motorized, hydraulic/pneumatic and piezoelectric. The most used are the motorized ones.

Extracellular recording

Extracellular recording is the simplest one. In this case, the field potentials outside cells are amplified by an AC-coupled amplifier to levels that are suitable for recording on a chart recorder or a computer. These recorded signals are very small (typically in the order of 10-500 μV), because they arise from the flow of ionic current through extracellular fluid. So, to design an extracellular amplifier, the single problem remains to keep a low instrumentation noise.

Single-Cell Recording

In this type of recording the microelectrode is advanced into the preparation until a dominant extracellular signal is detected. This signal arrives due the activity of one cell. The microelectrode can be made of metal (glass-insulated platinum) or can be a saline-filled glass micropipette. There exists also an AC-coupled amplifier with a gain of 100.

Multiple-Cell Recording

In multiple-cell extracellular recording the recording is made from many neurons simultaneously in order to study their concerted activity. Several microelectrodes are inserted into one region of the preparation, but each electrode in the array must have its own amplifier and filters. There can be tens or hundreds of microelectrodes used, but they require special fabrication techniques in order to produce integrated pre-amplifiers.

Intracellular Recording

The current-clamp technique or "Bridge" recording is the traditional method for recording the cell interior potential. The principle of this technique is the connection of a micropipette to a unity gain buffer amplifier that has an input resistance many orders of magnitude greater than that of the micropipette and the input resistance of the cell. The output of the buffer amplifier follows the voltage at the tip of the electrode.

If a high-quality current injection circuit is connected to the input node, all of the injected current flows down the micropipette and into cell and this can be used to inject a pulse of current to stimulate the cell, a DC current to depolarize or hyperpolarize the cell, or any waveform that users introduces into the control input of the current source.

3.2.3. Algorithms Applied to LFP

After an extracellular recording, two types of data are indentified: spikes and local field potentials (LFPs). The data from different structures of the rat brain is presented in the **Figure 3-3**.

Local field potentials (LFPs) are very important if we want to understand the brain activity and network and they are widely studied. LFPs are extracted by placing an extracellular microelectrode in the middle of a group of neurons, without being too close to any particular cell. A local field potential reflects the sum of action potentials and slower ionic events. The microelectrode measures the electrical potential difference (in volts) between the tissue and a reference electrode placed relatively near the recording electrode. To extract the LFP from the acquired signal, a low pass filter is used to remove the spike components (the spike data represents the activity of a particular cell under a given

situation, and is situated at high frequencies). Also, if some neuroscientists prefer to analyze the spikes from a particular neural cell, a high pass filter must be used. From these two types of filters, may be useful to develop the algorithms for the other filters: band pass filter and band stop filter. The convolution and the correlation of these types of signals may be also used to extract important information [N. Wisniewski, 2006].

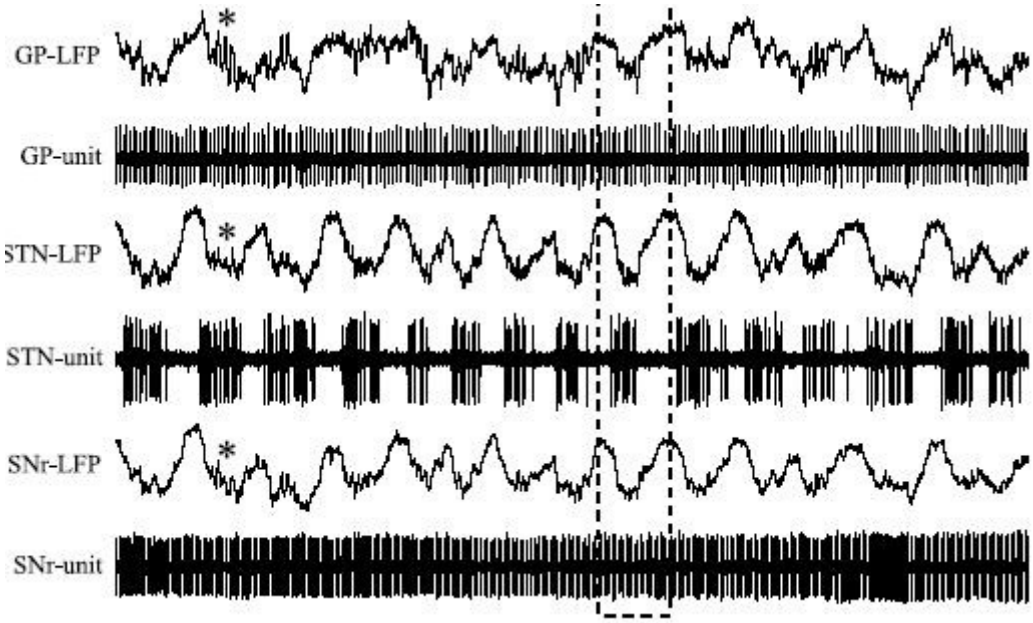


Figure 3-3 Brain activity recording from different structures of the brain (GP-unit, STN-unit, and SNr-unit). LFP extraction from the basic units (GP-LFP, STN-LFP, SNr-LFP) [P.Brown et al., 2004]

As we saw, after the signals are recorded, they pass through some modules including several filters and amplifiers. These modules can be either analogical or digital, depending on the purpose of the recording system. The final form of the signal results in a certain format (for example *.smr*) with a digitized form of the signal.

The most neurologists prefer to visualize the recorded signal on a screen, and because is very hard to manually extract some parameters, or to apply different processing techniques, they prefer applications that can read these signals and are also able to perform different operations on the recorded signals, in order to extract important features. In the following, a briefly description of the implementation mode of these algorithms (filters: low-pass filter, high-pass filter, band-pass filter, band-stop filter, and convolution) is described, accentuating the theoretical parts and some digital processing aspects.

The most digital signal processing (DSP) techniques are based on the *superposition strategy*, which breaks the signal being processed into simple components, each components being processed individually, and the results being then reunited. There exists five main ways to decompose signals in signal processing: *impulse decomposition*, *step decomposition*, *even/odd decomposition*, *interlaced decomposition*, and *Fourier decomposition*.

- *Impulse Decomposition*: - breaks an N samples signal into N component signals, each containing N samples. Each of the component signals contains one point from the original signal, with the remainder of the values being zero. A single nonzero point in a string of zeros is called *impulse*. It is important because it allows signals to be examined one sample at a time. This method is widely used in convolution.
- *Step Decomposition*: - breaks an N sample signal into N component signals, each composed of N samples. The first sample has a value zero, while the last samples are some constant value.

- *Even/Odd Decomposition*: - breaks the signals into two component signals: one having even symmetry and the other having odd symmetry: (even symmetry – if it is a mirror image around point N/2); (odd symmetry – the matching points have equal magnitudes but are opposite in sign).
- *Interlaced Decomposition*: - breaks the signal into two component signals: the even sample signal and the odd sample signal. To find the even sample signal, we start with the original signal and set all of the odd numbered samples to zero. To find the odd sample signal, we start with the original signal and set all of the even numbered samples to zero. This is the base of Fast Fourier Transform (FFT).
- *Fourier Decomposition*: - Any N point signal can be decomposed into N+2 signals, half of them sinus waves and half of them cosine waves.

As we may know, a system is any process that produces an output signal in response to an input signal. The most common systems are the *linear systems*, having three basic properties:

- *Homogeneity*: - meaning that a change in the input signal's amplitude leads to a change in the output signal's amplitude;
- *Additivity*: - if we have two input signals x1[n] and x2[n] with the corresponding output y1[n] and y2[n], then x1[n] + x2[n] = y1[n] + y2[n]. This property is important and is most used in telephony when added signals pass through the system without interacting;
- *Shift invariance* - is an optional property, so if a system has the first two properties, we can say it is a linear system. This property states that a shift in the input signal will lead into a shift in the output signal.

The *impulse response* of a linear system is denoted by h[n] and represents the output of the system when the input is a delta function. A delta function (also called unit impulse), δ[n], is a normalized impulse in which sample number zero has a value of one, while all other samples have a value of zero.

If we are dealing with filters, the common name of the impulse response of the system is *kernel*. There exist different types of kernels, for example the *exponential kernel*, the *square pulse kernel* and the *sinc kernel*. These are common low-pass filter kernels. By making the filter kernel wider or narrower, we can change the cutoff frequency of the filter. If we want to obtain a high-pass filter kernel, we must simply transform the low-pass filter kernel. By superposition, a filter kernel consisting of a delta function minus the low-pass filter kernel will pass the entire signal minus the low-frequency components. In this way we can obtain a high-pass filter.
In linear systems, the *convolution* is used to describe the relationships between three signals of interest: the input signal, the impulse response, and the output signal and it can be represented as:

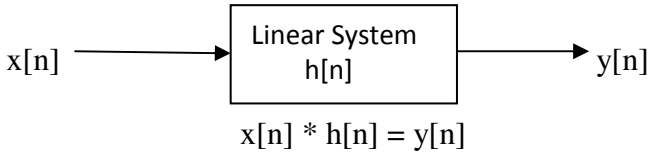


Figure 3-4 The representation of convolution

Figure 3-4 expressed in words can be translated as: the input signal convolved with the impulse response is equal with the output signal (in this case, the sign "*" represents convolution, not multiplication). The length of the output signal is always expressed as the length of the input signal added with the length of the kernel, minus 1.

In DSP applications, the length of the impulse response is much shorter than the length of the input signal. The convolution is mathematically described as:

$$y[i] = \sum_{j=0}^{M-1} h[j] \times [i - j]$$

The equation is called *convolution sum*, and it allows for each point in the output signal to be calculated independently of all other points in the output signal. According to [S.W.Smith, 1997] the index, *i*, determines which sample in the output signal is being calculated, and therefore corresponds to the left-right position of the convolution machine. In computer programs performing convolution, a loop makes this index run through each sample in the output signal. To calculate one of the output samples, the index *j* is used *inside* of the convolution machine. As *j* runs through 0 to *M*-1, each sample in the impulse response, *h*[*j*], is multiplied by the proper sample from the input signal, *x* [*i* - *j*]. All these products are added to produce the output sample being calculated.

As described in [N. Wisniewski, 2006], the best method when we are dealing long signals is to use FFT convolution method. The standard convolution is avoided for two reasons. First, convolution is mathematically difficult to deal with, because of the *deconvolution* (to find the input signal if the impulse response and the output signal are given) that is impossible to understand in time domain. However, deconvolution can be carried out in the frequency domain as a simple division, the inverse operation of multiplication. The frequency domain is used whenever the complexity of the Fourier Transform is less than the complexity of the convolution. The second reason for avoiding standard convolution is the computation speed. As an example from [N. Wisniewski, 2006], if we design a digital filter with a kernel containing 512 samples using a 200 MHz personal computer with floating point numbers, each sample in the output signal requires about one millisecond to calculate, using the standard convolution algorithm. The standard convolution algorithm is too slow because of the large number of multiplications and additions that must be calculated. Unfortunately, simply bringing the problem into the frequency domain via the DFT (Discrete Fourier Transform) does not help at all, because just as many calculations are required to calculate the DFTs, are required to directly calculate the convolution. The solution is the FFT.

Briefly, DFT changes an N point signal into two N/2 plus 1 point output signals. The input signal is the signal being decomposed (in time domain) and the two output signals are the amplitudes of the component sine and cosine waves (frequency domain). The role of the forward DFT is to transform a time domain signal into a frequency domain signal. The inverse transformation (from frequency domain into time domain) is called synthesis or the inverse DFT. The most efficient algorithm for calculating the DFT is FFT and usually operates with N that is power of 2. DFT is one of the most important tools in DSP, because it calculates a signal's *frequency spectrum*. This is a direct examination of information encoded in the frequency, phase, and amplitude of the component sinusoids. For example, human speech and hearing use signals with this type of encoding. Second, the DFT can find a system's frequency response from the system's impulse response, and vice versa. This allows systems to be analyzed in the *frequency domain*, just as convolution allows systems to be analyzed in the *time domain*. Third, the DFT can be used as an intermediate step in more elaborate signal processing techniques. The classic example of this is *FFT convolution*, an algorithm for convolving signals that is hundreds of times faster than conventional methods.

The FFT is another method for calculating the DFT. While it produces the same result as the other approaches, it is incredibly more efficient, often reducing the computation time by *hundreds*. The FFT operates by decomposing an *N* point time domain signal into *N* time domain signals each composed of a single point. The second step is to calculate the *N* frequency spectra corresponding to these *N* time domain signals. Lastly, the *N* spectra are synthesized into a single frequency spectrum. The FFT decomposition is figured in **Figure 3-5**. In this example, a 16

point signal is decomposed through four separate stages. The first stage breaks the 16 point into two signals, each consisting of 8 points. The second stage decomposes the data into four signals of 4 points and so on, until there are N signals composed of a single point. An interlaced decomposition is used each time and it requires $\log_2 N$ stages (in our example we have a 16 point signal, so 4 stages are required for decomposition).

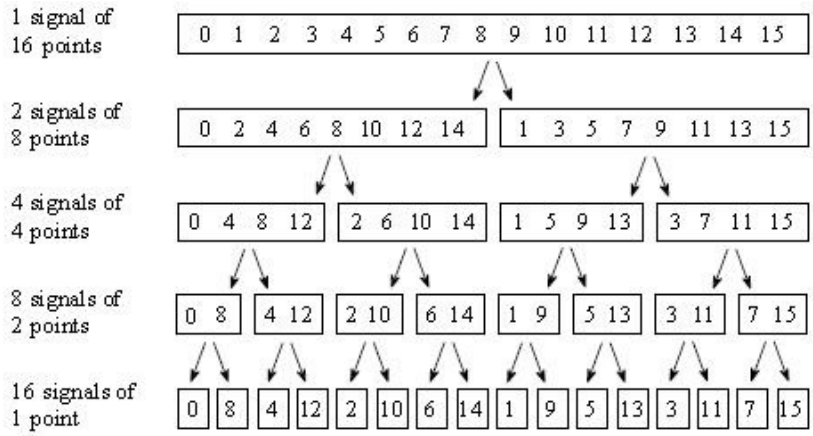


Figure 3-5 FFT decomposition [N. Wisniewski, 2006]

If we return to filters, they have two general purposes. The first one is the *separation of signals* that have been combined, and the second one is the *restoration of the signals* that have been distorted some way. Every filter has an *impulse response*, a *step response*, and a *frequency response*. Each of these responses contains complete information about the filter, but in a different form. According to [N. Wisniewski, 2006], all three of these representations are important, because they describe how the filter will react under different circumstances. The *step response* describes how information represented in the *time domain* is being modified by the system. In contrast, the *frequency response* shows how information represented in the *frequency domain* is being changed. This distinction is absolutely critical in filter design because it is not possible to optimize a filter for both applications. Good performance in the time domain results in poor performance in the frequency domain, and vice versa.

The most straightforward way to implement a digital filter is by convolving the input signal with the digital filter's *kernel*. The number of samples used to represent the impulse response can be arbitrarily large. For example, if we want to find the frequency response of a filter kernel that consists of 80 points, because FFT works only with the signals that are power of two, we need to add 48 zeros to the signal to bring it to a length of 128 samples. But this padding with zeros does not change the impulse response.

High – pass, band – pass and band – reject filters are designed by starting with a low – pass filter, and then converting it into the desired response. Two methods are described in [N. Wisniewski, 2006] for the low – pass to high – pass conversion: *spectral inversion* and *spectral reversal*.

- *Spectral inversion* starts with a filter kernel and in the first step, changes the sign of each sample in the filter kernel. In the second step, it adds one to the sample at the center of symmetry, so it results a high – pass filter kernel.
- *Spectral reversal* changes the sign of alternate kernel components. The effect is the multiplication of the filter kernel by a sinusoid with a frequency one half the sampling rate, which shifts the filter's frequency response by $f_s/2$. This results in a frequency response that is flipped left to right.

Low-pass and high-pass filter kernels can be combined to form band-pass and band-reject filters. Adding the filter kernels produces a band-reject filter, while convolving the filter kernels

produces a band-pass filter. The *sinc function* is the perfect kernel for LPF. The *sinc* function is mathematically described by:

$$h[i] = \frac{\sin(2\pi f_c i)}{i\pi}$$, where: h[i] is the impulse response of the filter

f_c is the cutoff frequency of the filter

Convolving an input signal with this kernel will result in a perfect LPF. The only problem is appears is that the *sinc* function continues to both negative and positive infinity without dropping to zero the amplitude. This infinity is a problem for computers, and for this, two modifications are made to the *sinc* function. The first one, it is truncated to M+1 points, symmetrically chosen around the main lobe, where M is an even number. All samples outside these *M*+1 points are set to zero, or simply ignored. Second, the entire sequence is shifted to the right so that it runs from 0 to *M*. This allows the filter kernel to be represented using only *positive* indexes. A problem will rise up, because the frequency response of such a kernel is not ideal anymore. To improve this situation, the *truncated-sinc* is multiplied with a *Blackman or Hamming window*, resulting a *windowed-sinc kernel*. The idea of a Blackman or Hamming window is to reduce the abruptness of the truncated ends and thereby improve the frequency response. The equations of the two windows are given by:

$$w_{Ham\min g}[i] = 0.54 - 0.46\cos(2\pi i / M)$$

$$w_{Blackman}[i] = 0.42 - 0.5\cos(2\pi i / M) + 0.08\cos(4\pi i / M)$$

The algorithms for the designed filters using windowed-sinc kernel are described in the **Chapter 4.** The FFT convolution algorithm is also described there. FFT convolution uses the principle that multiplication in the frequency domain corresponds to convolution in the time domain. The input signal is transformed into frequency domain by taking the DFT of the filter kernel (using the FFT), multiplied by the frequency response of the filter and then transformed back into the time domain.

One thing must be specified here. The FFT convolution algorithm uses the *overlap-add method* in order to break long signals into smaller segments for easier processing. This technique is very useful in the real time processing because the system needs to process the data segment-by-segment.

Overlap-add method is done in three steps: (1) decomposition of the signal into simple components, (2) processing each of the components, and (3) recombine the processed components into the final signal. The key of this method is how the lengths of these signals are affected by the convolution. Assuming that a N sample signal is convolved with a M sample filter kernel, the output signal will be N+M-1 samples long. So, when a N sample signal is filtered, it will be expanded by M-1 points to right.

For example, if we have a signal that is broken into segments, with each segment having 100 samples from the original signal, in the next step, each segment is individually filtered by convolving it with the filter kernel. Assuming that the filter kernel is 101 samples long, each output segment will be 200 (100 + 101 – 1) samples long.

3.3. Visualization Model

"The purpose of visualization is insight, not pictures", states the professor McCormick in [B.H.McCormick et al., 1987]. From this statement we may deduce that it is essential to understand what kind of "insight" particular users want to achieve. For medical visualization systems, an in-depth understanding of diagnostic processes, therapeutic decisions, and intraoperative information needs is indispensable to providing dedicated computer support. It is also essential to consider organizational and technical constraints [B.Preim, 2007].

Interaction methods support users in the storage of results (static pictures and image sequences, along with annotations). Medical visualization is primarily based on 3D volume data. The interaction facilities must be focused on 3D interaction techniques which allow the immediate exploration of 3D data.

Computer graphics and visualization play an important role in *intelligence amplification* (IA), term introduced by Dr. Brooks (ACM SIGGRAPH '94) to compare this field with artificial intelligence (AI). The goal of AI is to develop computer systems that could replace humans in some applications, while the goal of IA is to assign the role of computers as amplifiers and assistants to humans.

Visualization is *the act or process of interpreting in visual terms or of putting into visual form* (Webster's Ninth New Collegiate Dictionary). This process engages *vision*, the primary human sensory apparatus, and the processing power of the mind. The result is a medium to communicate complex and voluminous information.

Scientific visualization is the formal name given to the field in computer science that encompasses user interface, data representation and processing algorithms, visual representation, and other sensory presentation such as sound or touch [B.H.McCormick et al., 1987].

Data visualization is a generalization of scientific visualization, because it implies treatment of data sources (like financial, business, or marketing data) beyond the sciences and engineering [L.Rosenblum et al., 1994].

Information visualization deals with the visualization of abstract information such as hypertext documents on the World Wide Web, directory/file structures on a computer, or abstract data structures [InfoVis, 1995].

The base of the visualization process is the *data*. First, the data is acquired from some source, then it is transformed using various methods, and then mapped to a form appropriate for presentation to the user. Finally, the data is rendered or displayed, completing the process. This process can be repeated and new models can be developed.

3.3.1. CT and MRI Imaging Techniques

Medical image data is usually represented as a stack of individual images. Each image represents a thin slice of scanned body part and is composed of individual pixels, which are arranged on a two-dimensional grid, where the distance between two pixels (picture elements) is typically constant in each direction. Besides 2D representation, the volumetric data combine individual images into a 3D representation on a 3D grid. The data elements are now called voxels (volume elements, and they are located on the grid points. The two discussed cases are represented in the **Figure 3-6**.

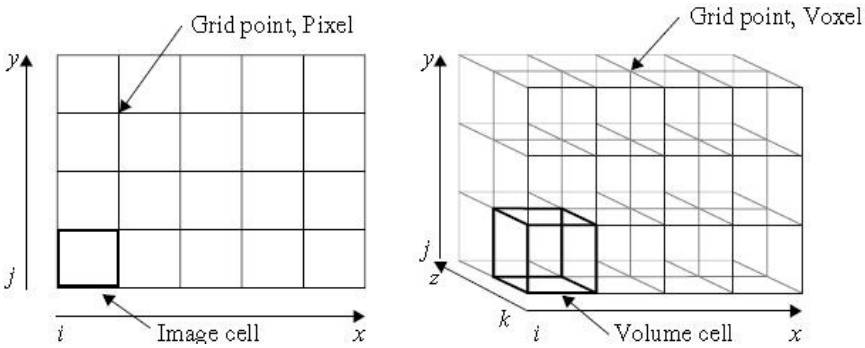


Figure 3-6 2D (left) and 3D (right) representations of medical images [B.Preim, 2007]

The common used methods for the acquisition of the medical images are X-ray Computed Tomography (CT) and Magnetic Resonance Imaging (MRI) and are important diagnostic tools in the medicine field. They use a sampling or data acquisition process that captures information about the internal anatomy of a patient. The information is organized in the form of *slice-planes* or cross-sectional images of the patient. Actually, CT technique uses many pencil thin X-rays to acquire the data. MRI technique combines large magnetic fields with pulsed radio waves. There are a lot of mathematic techniques used to reconstruct the slice-planes. As acquired from the imaging system, a slice is a series of numbers arranged in a matrix, representing the attenuation of X-rays (CT) or the relaxation of nuclear spin magnetization (MRI) [Krestel90]. A grayscale value is assigned to this large amount of values (because it cannot be understood in its raw form) and then is displayed to a computer screen. So, what the computer represents as a series of numbers, we see as a cross section through the human body: skin, bone and muscle. These sections can further be gathered into volumes and the volumes can be processed to reveal complete anatomical structures. We can view the entire brain, skeletal system or the vascular one on a living patient without interventional surgery.

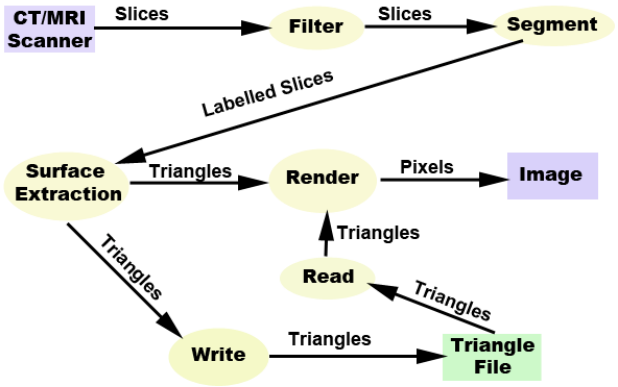


Figure 3-7 MRI/CT data flow diagram

The series of cross-sectional slices provided by the CT or MRI scanner are processed by image processing filters to enhance features in the gray scale slices. The segment process identifies tissues and produces labels for the various tissues present in the slices. The labeled slices are then passed through a surface extraction process to create triangles that lie on the surface of each tissue. The role of rendering is to transform the geometry into an image. A write process can be used to store the triangles in a file, and after that the triangles can be read and rendered into an image [VTK3rd]. More about CT and MRI techniques can be found in [B.Preim, 2007].

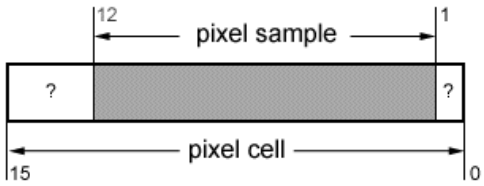
Of course, there exists more imaging techniques (for example ultrasound imaging or position emission tomography - PET), but CT and MRI are the most used in nowadays. An important MRI variant, MRI diffusion tensor imaging (DTI) has recently become a modality of

high interest, because it exploits the anisotropic nature of water diffusion to estimate the fiber direction of neural pathways (for example the communication network of the brain) and heart muscles [B.Preim, 2007].

3.3.2. DICOM standard

The CT and MRI images are usually stored in many medical formats, the common one being the *DICOM format*. This standard was created by the National Electrical Manufacturers Association (NEMA) to aid the distribution and viewing of medical images. A DICOM file contains both *a header* (where the information about patient is stored: patient's name, the type of scan, image dimensions and other) and *all of the image data* (which can contain information in three dimensions). The number of bytes allocated for header depends on how much header information is stored. The header is a very important element in processing medical images, because we can also find information about how the data is organized, or if the data has been compressed – there exists DICOM viewers that can only handle the uncompressed raw data. Here is also specified the byte order for raw data, because different computers store integer values differently: *big endian* or *little endian* ordering. For example, if we have a 16-bit integer with the value 257, the most significant byte stores the value 01 (=255), while the least significant byte stores the value 02. So, some computers would save the value 255 as 01:02, while others will store it as 02:01.

In the header we can see information regarding the bits allocated, bits stored and high bit for a pixel cell. The distribution of bits into a pixel cell can be figured as in **Figure 3-8**:

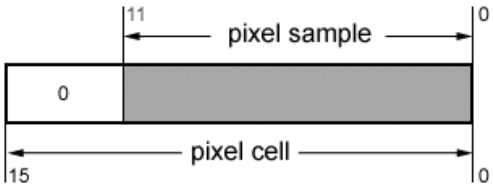


A pixel cell:
Bits Allocated = 16
Bits Stored = 12
High Bit = 11

Figure 3-8 Bits organization is a pixel cell of a DICOM image

As we can see, every pixel is made of one or more samples, each sample being described by a given number of bits ("Bits Stored"), and is packed in a cell that can use an even greater number of bits ("Bits Allocated"), so the user can include overlay data within each sample, making the DICOM format very flexible.

On the other hand, the flexibility means complexity, because we have to deal of all possible combinations of allocated, stored and high bits. Because of this, sometimes it is recommended to transform the DICOM files format into another format that "reorganize" the pixel data and provide them with raw file, so that the files will not include other information except the "cleaned" and "padded" pixel data (an example is the *.raw* format). The *RAW format* one of the common formats that can store images on 16 bits (for example TARGA JPEG and BMP do not support more than 8 bits per sample, which is not sufficient to achieve the precision commonly encountered in standard medical image formats). The **Figure 3.9** shows what happens when a DICOM file is converted into an RAW file.



A pixel cell:
Bits Allocated = 16
Bits Stored = 12
High Bit = 11

Figure 3-9 Bits organization is a pixel cell of a RAW image

3.3.3. Computer Graphics Primer

Computer graphics (rendering) is the process of generating images using computers [W.Schroeder, et al., 2002]. This process consists in converting the graphical data into an image. Before obtaining the image, the graphical data must be transformed into *graphics primitives*, which are then rendered.

To understand this we need to know the way in which lights (for example sun), cameras, and objects (or actors) interact in the world around us (the way we can see the objects). This is very hard to simulate with a computer, but there exists a technique for 3D computer graphics called *ray-tracing* or *ray-casting* (presented in this chapter), that simulates the interaction of light with object, by following the path of each light ray. This technique is not widely used because it is a relatively slow image generation method, being typically implemented in software. There exist other methods that make use of the computer hardware and are faster.

The rendering process contains two categories of processes: *image-order processes* (for example ray-tracing) and *object-order processes* (oriented on objects). The computers graphics field is object-order process based. There exist two rendering models based on object-order process: *surface rendering* (to render the surfaces of an object) and *volume rendering* (to render the surface and the interior of an object). To render the object using *surface rendering techniques* means to mathematically model the object with a surface description: points, lines, triangles, polygons or 2D and 3D splines. The object interior is not described.

Using *volume rendering techniques* we can see the homogeneity inside the objects. If we think about *ray-tracing* we can think about the rays that does not interact only with the surface of the object, but they also interact with the interior.

The rendering process is affected by the interaction of light with the actors in the scene. If there are no lights, we will have a black image and rather uninformative. Once the rays of light interact with the actors in a scene, we have something for our camera to view.

There are two color systems most used: RGB and HSV. The *RGB system* represents colors based on their red, green and blue intensities and *HSV system* represents colors based on his hue (the dominant wavelength color), saturation (how much of the hue is mixed into the color), and value (brightness or intensity component: from 0.0 - black to 1.0 – something bright).

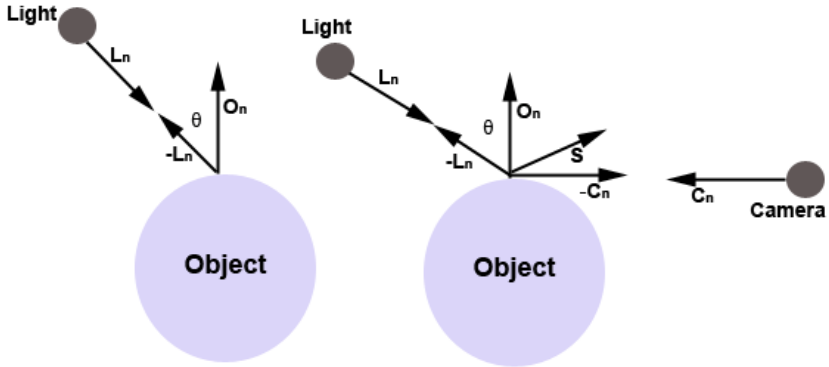


Figure 3-10 Diffuse lighting compared with specular lighting

Some of the rays that are travel through space intersect the actors, so they interact with the surface of the actor, producing a color. But, part of this resulting color is actually not due to direct light, but rather from ambient light that is reflected or scattered from other objects. The ambient lighting equation is:

$R_c = L_c * O_c$, where: R_c - resulting intensity curve
L_c - intensity curve of the light
O_c - color curve of the object

There are two components of the resulting color that depend on direct lighting: the *diffuse lighting* (also known as Lambertian reflection) and the *specular lighting*.
The *diffuse lighting* takes into account the angle of incidence of the light onto an object, the contribution from *diffuse lighting* is expressed as:

$R_c = L_c * O_c[\overrightarrow{O_n} * (-\overrightarrow{L_n})]$, where: R_c - resulting intensity curve
L_c - intensity curve of the light
O_c - color curve of the object
L_n - incident light vector
O_n - the surface normal of the object

The *specular lighting* represents direct reflections of a light source off a shiny object. The specular power, O_{sp}, specifies how shiny an object is (how quickly specular reflections diminish as the reflection angles deviate from a perfect reflection (VTK 3rd)). The equation for specular lightning is:

$R_c = L_c * O_c[\vec{S} * (-\overrightarrow{C_n})]^{Osp}$, where: $\overrightarrow{C_n}$ - the direction of projection for the camera
$\vec{S} = 2[\overrightarrow{O_n} * (-\overrightarrow{L_n})]\overrightarrow{O_n} + \overrightarrow{L_n}$ $\vec{S}$ - the direction of specular reflection

Besides light sources, actors and color, we need a camera in order to render the scene. The most important attributes of the *camera* are: the position, orientation, and focal point, the method of camera *projection*, and the location of the camera *clipping planes* (front and back clipping planes-intersect the direction of projection). The position and the focal point of the camera define the location of the camera and where it points. The *direction of projection* is the vector defined from the camera position to the focal point. In computers graphics the camera can be manipulated by directly setting these attributes, but there are a lot of operations that make the job easier: *the azimuth* (rotation around the direction of projection), *the elevation* (perpendicular to the direction of projection) and *roll* (rotation of the view up vector). There are two methods used to change the focal point: *yaw* and *pitch* (like the azimuth and elevation, but the focal point is changed instead of the camera). *Dollying* in and out moves the camera closer or further the focal point. Finally, *zooming* changes the camera's view angle, so that more or less of the scene falls within the view frustum [W.Schroeder, et al., 2002]. Once the camera is situated, we can generate our 2D image.

Four coordinate systems are commonly used in computer graphics: *model*, *world*, *view*, and *display*. The *model* coordinate system is the coordinate system in which the model is defined, a local Cartesian coordinate system. The *world* coordinate system is the 3D space in which the actors are positioned. One of the roles of the actor is make the conversion from model's coordinates into world coordinates. So, any model can have its own coordinate system,

but there can be only one world coordinate system. The *view* coordinate system represents what is visible to the camera and it consists of a pair of x and y values, ranging between (-1, 1), and a z depth coordinate. The x and y values are used to specify the location in the image plane, and z coordinate specifies the distance from the camera. The properties of the camera are represented by a four by four transformation matrix (where the perspective effects of the camera are introduced), used to convert from world coordinates into view coordinates. This transformation matrix is widely used in computer graphics, allowing the translation, scaling, and rotation of objects by repeated matrix multiplication.

The *display* coordinate system is similar with the view coordinate system, but the x and y values are the pixel locations on the image plane, and is not restricted by (-1,1) range.

Beside the transformation matrix of the camera, there exists a transformation matrix for the actor and is used to control the actor's location using orientation, position, and scale factors along the coordinate axes.

3.3.4. Fundamental Visualization Algorithms

According with [W.Schroeder, et al., 2002], the methods to transform the image data to and from various representations of it are called *algorithms* and represent the heart of data visualization. These algorithms are divided according to the *structure* and *type* of transformation. The *structure* of a transformation represents the effects that a transformation has on the topology and geometry of a image data; the *type* of the transformation is related to the type of the data that the algorithms operates on.

The structural transformations are classified in four ways, depending on the effects they have on the image data [W.Schroeder, et al., 2002]:

- *Geometric transformations* can be: translation and rotation. So, they alter the input geometry (for example the point coordinates), but they do not change the topology of the input data image.
- *Topological transformations* are complementary of the geometric transformation because they change the input topology, but they do not change the geometry and attribute data (for example, when we change an image data to an unstructured grid).
- *Attribute transformations* are used to convert the data attributes from one form to another, but can create new attributes from the input data. They do not alter the structure of the data (for example, when we compute the vector magnitude based on elevation).
- *Combined transformations* can be: the computation of the contour lines or surfaces. They change both the data structure and the attribute data.

According to the type of data they operate on or they generate, the algorithms can be classified as:

- *Scalar algorithms* – operate on scalar data. An example given by [W.Schroeder, et al., 2002] is the generation of contour lines of temperature on a weather map.
- *Vector algorithms* – operate on vector data. They are used every time we have to compute the direction and magnitude of the data (for example, the direction and magnitude of airflow).
- *Tensor algorithms* – operate on tensor matrices. An example can be: illustrating the components of stress or strain in a material [W.Schroeder, et al., 2002].
- *Modelling algorithms* – combined scalar/vector algorithms. They may generate data topology, geometry, surface normals or texture data.

The *scalar algorithms* are the most used in developing applications because the scalar data is commonly found in the real-world and is easy to work with. Also, a lot of algorithms were developed to visualize it. In [W.Schroeder, et al., 2002] all these algorithms are detailed described, and in the following are presented briefly:

- *Color Mapping* technique is used to map the scalar data to colors that are further displayed on the computer system. For this, a *color lookup table* is created and the scalar values are indices to this table.

Briefly, the mapping process follows these steps: (1) the lookup table holds an array with colors (red, green, blue components or other representations); (2) there exists a minimum and maximum *scalar range (min, max)* within the scalar values are mapped. If a scalar value is less than the minimum range, it is clamped to the minimum color value and if it is greater than the maximum range it is clamped to the maximum color; (3) for each scalar value s_i, the index *i* into the color table with *n* entries is given by [W.Schroeder, et al., 2002]:

- if $s_i < min$, $i = 0$
- if $s_i > max$, i = n-1
- if $s_i \in (min, max)$, $i = n\left(\frac{s_i - \min}{\max - \min}\right)$

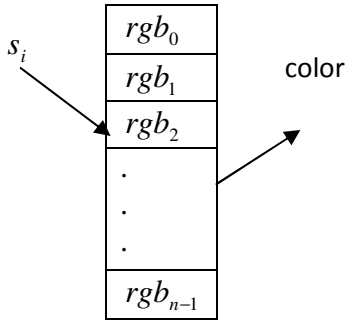


Figure 3-11 Mapping scalars to colors using lookup table

- *Contouring* technique is an extension to color mapping and it rises because of the property of the human eye to separate different colored areas into distinct regions. To contour the data means to construct the boundary between these regions. We can have *contour lines* (2D): for example, the weather maps which are annotated with lines of constant temperature; or *contour surfaces* (3D), also called *isosurfaces*: for example, [W.Schroeder, et al., 2002] constant medical image intensity corresponding to body tissues such as skin, and bone.

  In [M.Gordan, 2011] are presented some of the basic algorithms for contour lines extraction like Laplacian, Sobel or Prewitt algorithms.

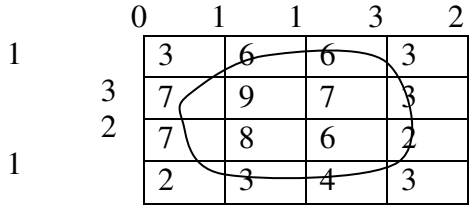


Figure 3-12 Contouring using a contour line with value 5

Briefly, if we have a 2D structured grid (**Figure 3-12**): every point of the grid has a weight and the scalar values are defined next to the points that define the grid, the contouring process consists in the following steps: (1) selection of a scalar value (contour value), corresponding to the contour lines; (2) for contour generation, a form of interpolation is used, because of the contour value that may lie between some ranges (we apply linear interpolation along the edges); (3) once we compute the points on the cell edges, we can connect these points into contours.

Other common used methods are the *marching squares* algorithm in 2D, and marching cubes [W.E.Lorensen et al., 1987] in 3D. If we use the same 2D structured grid from the **Figure 3-13**, one approach in displaying the contour is *marching square method*. It consists on the assumption that a contour can only pass through a cell in a finite number of ways. So, a table with all possible topological states of a cell is defined. The number of topological states of a cell depends on the number of cell vertices, and the number of inside/outside relationships a vertex

can have with respect to the contour value. For a vertex to be inside a contour, it must have the scalar value larger than the scalar value of the contour line. In order to be outside the contour, it must have a scalar value smaller than the contour value. So, because our cell has four vertices and each of them can be either inside or outside the contour, we have $2^4 = 16$ possible ways for a contour to pass through the cell. The sixteen combinations are:

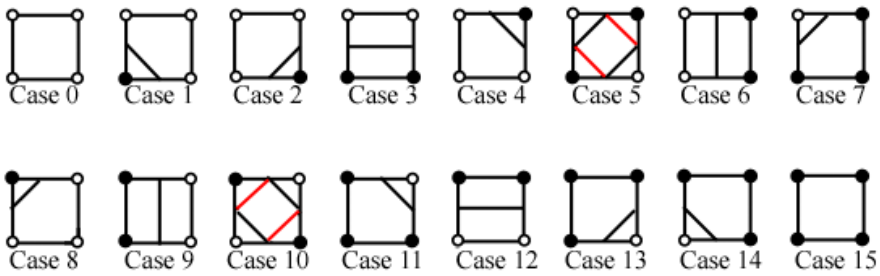


Figure 3-13 Sixteen different marching squares cases [W.Schroeder, et al., 2002]

The steps used in *marching squares* algorithm are: (1) selection of a cell; (2) for each vertex of the cell it calculates the inside/outside state; (3) an index is computed by encoding the state of each vertex as a binary digit (in our example we have 16 cases, so we will have a 4 bit index); (4) the index is used to search the topological state of the cell in a case table; (5) the contour location is calculated, using interpolation, for each edge in the case table. After a cell is processed, the algorithm *marches* (moves) to the next cell. Obviously, the interpolation along each edge should be done in the same direction.

Because this algorithm is easy to implement, it can be extended for the case when we have three dimensions (marching cubes algorithm).

On the other hand, we can observe that in the case 5 and case 10, a cell can be contoured in more than one way. We can observe that this ambiguity arises when adjacent edge points are in different states, but the diagonal ventricles are in the same state. This problem is very easy to treat in the case of 2D because for each ambiguous case just one of the two possible cases is implemented. In this situation, the contour may either extend or break the current contour. The two situations are represented in the **Figure 3-14**.

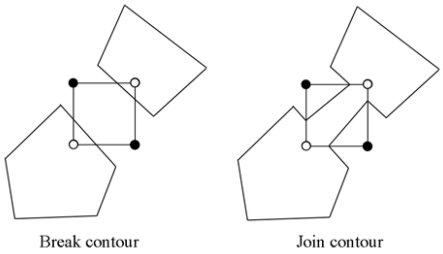


Figure 3-14 Treating the ambiguous cases [W.Schroeder, et al., 2002]

If we have to extract an isosurface, then we can use *marching cubes algorithm*. Because there are eight points in a cubical cell, we will have $2^8 = 256$ different combinations of scalar value. Because of the symmetry between them, the number of possible cases can be reduced to 15 (represented in the **Figure 3-15**):

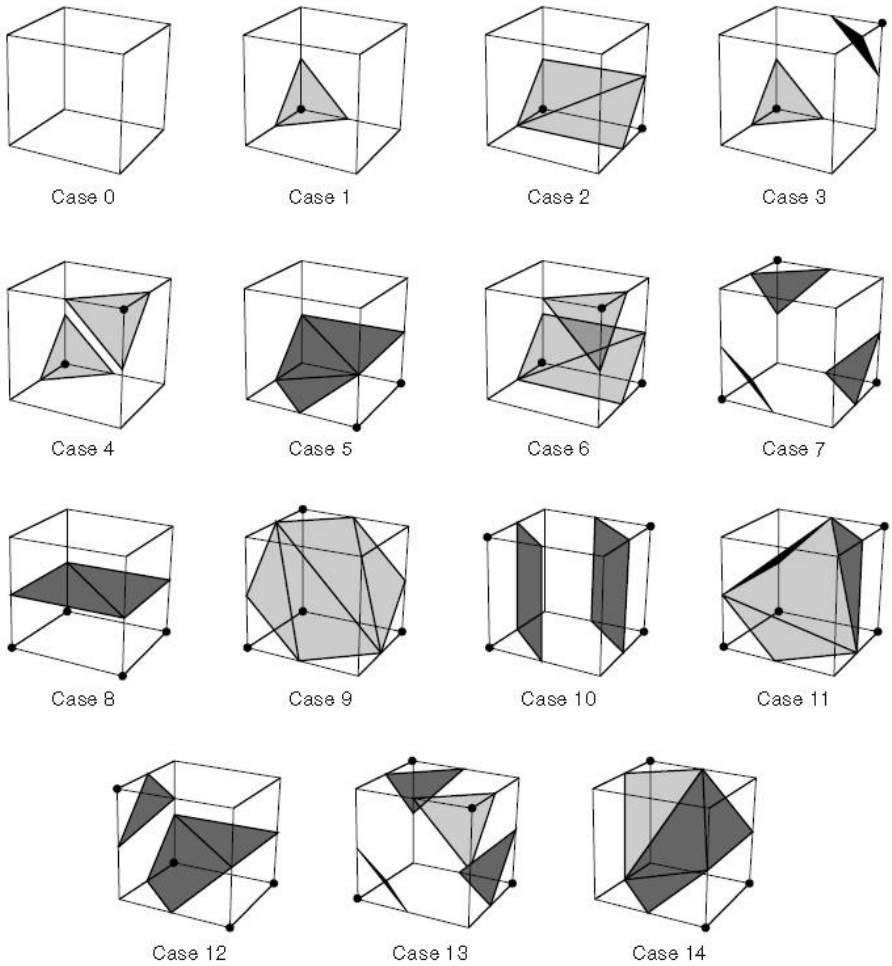


Figure 3-15 Marching cubes cases for 3D isosurface generation
[C.R. Johnson, C.D. Hansen, 2005]

The ambiguous cases in this situation are the cases with the number: 3, 6, 7, 10, 12 and 13 (counting from the top-left of the **Figure 3-15**. Unfortunately, this problem is more complex in 3D case, because we cannot simply choose an ambiguous case independent of all the others ambiguous cases (if we would do that, holes in the isosurface can appear). There were proposed several approaches to remedy this problem and the one method is to use the *marching tetrahedra* technique, because this method does not exhibit ambiguous cases. But the price paid for this is that the marching tetrahedral algorithm generates isosurfaces consisting of more triangles and this may affect the result, because it creates artificial "bumps" in the isosurface because of the interpolation along the diagonals.

The most simple and effective solution is described in [W.Schroeder, et al., 2002] and it consists in an extension of the original 15 marching cubes cases by adding six complementary cases (corresponding to the marching cases 3,6,7,10,12 and 13) in such a way to be compatible with neighboring cases and to prevent the creation of the holes in the isosurface.

Besides the ambiguity problem, another problem of the *marching cubes* algorithm is its interpolation, which is composed of a linear interpolation for the edge vertices and a Gouraud interpolation of the color values inside the triangles (discussed in [B.Preim, 2007]). This is not equivalent to cubic trilinear interpolation (discussed in **Chapter 5)** and results in visual artifacts if the individual triangles become visible.

Another scalar algorithm used for contour extraction is the *dividing cubes algorithm*. It is similar with the marching cubes algorithm, but unlike marching cubes, it generates point primitives as compared to triangles (3D) or lines (2D). If the number of points on the contour surface is large [W.Schroeder, et al., 2002], the rendered appearance of the contour surface

appears “solid”. To obtain this solid appearance, the density of the points must be at or greater than the screen resolution.

The advantage of using this algorithm is that the rendering points, is much faster than rendering polygons. Obviously, the speed depends on the hardware used.

One disadvantage of creating contours with dense point clouds is that magnification of the surface reveals the disconnected nature of the surface.

1) *Scalar Generation* is a technique used when the data is not single-valued or it is a mathematical or a complex expression. In this case we must convert data into a form we can visualize. Some examples of scalar mapping are the computation of the vector magnitude, evaluating the surface curvature or determining the distance between points.

The *vector algorithms* deal with the vector data which is a 3D representation of direction and magnitude. The most used techniques are described very briefly, because they are not used in the developed application:

1) *Hedgehogs and oriented Glyphs* – represent the natural way to create a vector and it consists in drawing an oriented, scaled line for each vector.
2) *Warping* – is a technique used in order to display the motion that is in the form of velocity or displacement. Actually it deforms the geometry of the data according to the vector field (for example, the visualization of the flow of fluid can be simulated by distorting a straight line inserted perpendicular to the flow.
3) *Displacement Plots* – are used in the visualization of the motion of an object in the direction perpendicular to its surface. The most used application of this technique is the study of vibration, where we need to know the eigenvalues (for example the natural resonant frequencies) and the eigenvectors (for example the mode shapes) of a structure.
4) *Streamlines* – is an extension of the displacement plots technique and it includes: *particle traces* (trajectories traced by fluid particles over time); streaklines (set of particle traces at a particular time t_i that have previously passed through a specified point x_i; *streamlines* (integral curves along a curve *s* satisfying the equation) [W.Schroeder, et al., 2002]:

$$s = \int_t \vec{V} ds \quad \text{, with } s = s(x, t) \text{ for a particular time.}$$

The *tensor visualization* area is in continuous research and there are a few simple techniques used to visualize 3 x 3 real symmetric tensors. They are used to describe the state of displacement or stress in a 3D material. The stress and strain tensors for an elastic material are given by (**Figure 3-16**):

$$\begin{bmatrix} \delta_x & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \delta_y & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \delta_z \end{bmatrix}$$

(a) *Stress tensor*

$$\begin{bmatrix} \frac{\partial u}{\partial x} & \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial z}\right) & \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\ \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial z}\right) & \frac{\partial v}{\partial y} & \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) \\ \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) & \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) & \frac{\partial w}{\partial z} \end{bmatrix}$$

(b) *Strain tensor*

Figure 3-16 Stress and strain tensors [W.Schroeder, et al., 2002]

In the described tensors the diagonal coefficients are called *normal stresses and strains* (they act perpendicular to a specified surface), and the off-diagonal terms are the *shear stress*

and strains (they act tangentially to the surface). The normal stress can be either compression or tension, depending on the sign of the coefficient.

Finally, the *modelling algorithms* have only one thing in common: they can create or change the geometry or topology of the data. The bases of these algorithms are so called *implicit functions*, which have the form:

$F(x, y, z) = c$, where c is an arbitrary constant

The implicit functions are very used in description of the common geometric shapes (planes, spheres, cylinders, cones, ellipsoids, and quadrics). They have also the role of separation of the 3D Euclidean space into three distinct regions: inside, on, and outside the implicit function. We can define these regions mathematically, as:

$F(x, y, z) < 0$, $F(x, y, z) = 0$, and $F(x, y, z) > 0$, respectively.

Another role of the implicit functions is to convert a position in space into a scalar value. If we have an implicit function we can sample it at a point (x_i, y_i, z_i) to generate a scalar value c_i. An example of an implicit function is the equation for a sphere of radius R:

$$F(x, y, z) = x^2 + y^2 + z^2 - R^2$$

Implicit functions can be used alone or in combination to model geometric objects. For example, if we want to model a surface describes by an implicit function, we have to sample F on a dataset and generate an isosurface at a contour value c_i. The result will be a polygonal representation of the function. If we want to obtain more complex objects, then we can use boolean operations like union, intersection, and differences.

Based on the implicit functions, *cutting algorithm* was developed. This operation requires two input information: a definition of the surface and a dataset to cut. The cutting algorithms has the following steps: (1) by evaluating $F(x, y, z)$ for each cell point, we generate function values for each cell; (2) if all the points evaluate positive or negative, then the surface does not cut the cell; (3) however, if the points evaluate positive and negative, then the surface passes through the cell; (4) in order to generate the isosurface $F(x, y, z)$, we can use the cell contouring operation; (5) finally, data attribute can be computed by interpolating along cut edges.

3.3.5. Volume Visualization Algorithms

The first step in medical imaging is the visualization of the data. This step is very important in obtaining the results of an experiment and in testing different methods that can be beneficial for a patient. The volumetric data is composed of a very large number of individual voxels. There exits two main methods of volume rendering in medical visualization [B.Preim, 2007]: *indirect volume rendering* and *direct volume rendering*.

Indirect Volume Visualization:

Two methods are common used in indirect volume visualization:

a. Plane-based volume rendering:

This is a widely used technique of visualization in medical imaging, also called the "cine mode", because individual slices of a volumetric dataset are examined subsequently. Using this visualization we can easily visualize a plan aligned with the cuboid of volume. In this way we will obtain the axial, coronal, and sagittal views. If we visualize the data with more than one slice orientation, we will have a multi-planar reconstruction (MPR). In this case, the visualized slices only use the points of the grid.

An oblique volume slice can provide better representation since we have the flexibility to choose arbitrarily its orientation. The slice is defined by a position vector and the normal vector of the plane and it is the visual representation of the voxels of the volume dataset intersected by the plane.

b. Surface-based volume rendering

The goal of this technique is to represent boundary or a material surface of the object (for example a human organ). Typically, we try to display voxels that have a similar intensity value. The resulting surface is called an isosurface and the value used for extraction is called isovalue.

The methods for contour extraction (obtaining the isosurface) were already presented: the *contour tracing* and *marching squares* algorithms. Usually, there is just one isosurface, but in the case of the visualization of multiple segmented anatomical structures, we need to distinguish between different objects. The basic distinction is the color and the texture, but if one object contains another, we cannot see both. In this case, the most used solution is the use of transparency. An object is rendered semi-transparently with an α coefficient between 0 and 1.

Direct Volume Visualization:

This method consists in visualization of the volume using all the voxels. All direct volume rendering algorithms can be classified into two groups: *image space approaches* (also called backward-mapping approaches), and *object-space approaches* (also called forward-mapping approaches). In each classification class there are many variations of the basic algorithms, but there were not implemented in this application.

a. Ray Casting

The classic direct volume rendering algorithm is the image-oriented *ray casting*. This is a variation *of ray-tracing* and is presented in [B.Preim, 2007]. In both algorithms, rays are casted from the eye or viewpoint through the image-plane and through the dataset. In the case of *ray tracing*, contributions are typically computed at the intersections of the rays with the objects of the scene, and depending on the material properties of the objects, the rays are refracted or reflected. In the case when both effects take place, secondary rays are cast from the intersection point, while the primary rays proceed in the current or new directions.

Since all voxels of the dataset may be contributing, *ray casting* does not compute intersections with objects. The ray samples the volumetric dataset along its path and depending on the specified properties, the samples may be contributing (if they have opacity values larger than zero) or not contributing (if they have a zero opacity). The positions of the samples on the ray within the data volume depend on the direction of the ray and the sampling rate, which governs the distance between neighboring samples on that ray. Since in most cases these

positions will not be located directly at voxel positions, we need to compute the sample values by an interpolation method, the most used method being the trilinear interpolation briefly presented in **Chapter 4.**

b. Shear Warp

This volume rendering approach is a highly optimized version of the basic ray casting algorithm. The main goal of this method is to simplify the volume sampling to reduce the memory access costs. Ray casting samples the volume in arbitrary directions, depending on the position and orientation of the image-plane.

In contrast, *shear warp algorithm* casts the rays from a base-plane, which is always parallel to the side of the data volume that faces the viewpoint most. The resulting ray direction will be perpendicular to the respective volume face, enabling a very regular data access pattern [B.Preim, 2007].

In [S. Zambal] the algorithm is illustrated as follows: if we have a fat book and we look directly at the front of the book, we can see only the front page of it. We cannot see a side of it without rotating. The "trick" that is applied in this algorithm is to shear the pages of the book so that all the pages remain parallel, but their relative positions change, then we can see the side of the book.

In the volume visualization, the slices of the volume are sheared and after the shearing, the samples are projected onto a so-called immediate image. Because this image is distorted because of the projection, the image must be warped to get the correct final image.

c. Texture-Mapping

This volume rendering approach is based on the texture-mapping support of computer graphics hardware. In essence, a volume dataset is loaded into texture memory and is subsequently resampled by the texture-mapping functions of the graphics hardware and mapped to a rasterized polygon on the screen. The resampling through the texture-mapping capability means reslicing the volume dataset and combining the contribution of the newly resampled slices through VTK's alpha-blending functions in the frame buffer. The **Figure 3-17** presents the main idea of the texture-mapping algorithm.

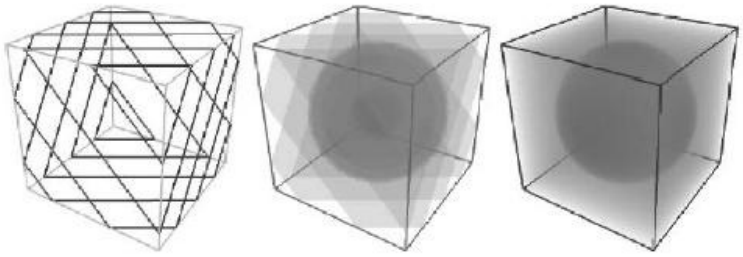


Figure 3-17 Volume rendering with 3D texture-mapping [B.Preim, 2007]

The slicing process is the following: a slice which is parallel with the view plane and perpendicular to the view vector is computed through the resampling process of the volume dataset. In the mapping process, the farthest slices are mapped first through a rectangle rasterized to the framebuffer (**Figure 3-17** - left). Because of the clipping process it results many triangles. The middle image shows how the clipped polygons are texture-mapped with the slice information. Because of the small number of slices, in this example, we can observe artifacts. If the number of the slices is increased, so the distance between the slices is reduced, we can reduce the staircast artifacts (right).

The main disadvantage of this technique is that for optimal performance, the volume dataset has to be fitted into the texture memory or at least to the graphics memory onboard the graphics accelerator and this is the case only for relatively small datasets. Therefore, the parts of

the data need to be swapped from main memory, if the dataset does not fit into main memory. This results in slowing the application. Many approaches were considered to solve this problem, the common one is to reorganize the volume dataset into data bricks that are exchanged, depending on which part of the dataset will be rendered next.

These are not the only techniques for direct volume rendering, but are the one that are used most of the time. There exist other techniques that are well suited for specific circumstances (for example *Cell Projection*, *Fourier Domain Volume Rendering*), but they are not the commonly ones, and are described in detail in [B.Preim, 2007].

3.3.6. Using Visualization and Insight Toolkits

The Visualization Toolkit (VTK) is an open source software able to implement visualization data structures, algorithms and image processing. Even if it consists of a C++ class library, it also generates language bindings to the interpreted languages Tcl, Phyton, and Java. This process is called wrapping and it is able to create a layer between the native C++ VTK library and the interpreter. This is very useful in creating GUIs, many of the C++ complexities being hidden using an interpreted language. Of course, for creating higher performing applications, C++ is the best solution, but it requires more time spend on developing such of applications.

The data information in VTK library is represented using primitives like polygons, triangles and lines. The base building block of these primitives is a point, named vertex. The properties of a vertex are: *position*, *normal*, and *color*, each of which being a three element vector. The position gives the location of the vertex, its normal specifies the direction, and the color specifies its red, green, and blue components. So, a polygon is obtained by connecting a series of vertices.

The steps in creating the visualization using VTK are described in the functional model in [W.Schroeder, et al., 2002]:

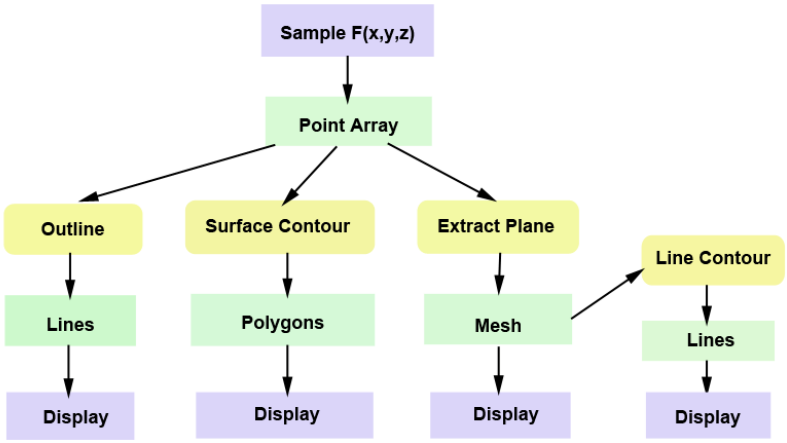


Figure 3-18 Functional Model of VTK

In the **Figure 3-18** the yellow blocks represent the operations that are performed on the data and the purple blocks indicate the objects that represent and provide access to the data; the arrows indicate the direction of data movement. If we have processes that create data without any input, they are called *data source objects* (or sources), and if the processes consume data with no output, they are called *sinks*. If a process has both an input and an output, it is called a *filter*.

The objects can be divided in two categories: the *data objects* and the *process objects*. *Data objects* represent information that is processed by the visualization pipeline and provide

methods to create, access, and delete this information. We cannot directly modify the data represented by the data objects, the modification being reserved for the *process objects*. The *process objects* operate on input data to generate output data, and they are classified depending on the data object types they process: source objects, filter objects, or mapped objects.
Practically, a conceptual view of the pipeline execution can be represented as following:

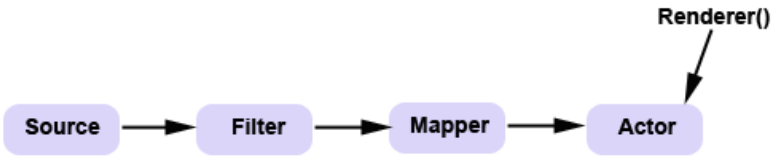


Figure 3-19 Conceptual view of pipeline execution

Figure 3-19 shows how filters and data objects are connected together to form a visualization network. The input instance variable is set using *SetInput()* method and the output instance variable is accessed using the *GetOutput()* method. So, if we have more filter objects *filter1* and *filter2* of compatible types, the C++ statement for connecting them will be [W. Schroeder, 1998]:

filter2 - >SetInput (filter1 ->GetOutput());

Because VTK was developed for interactive applications, it also contains the *command/observer* design pattern. The base of this design patters is the concept of *events*. An *event* signals when an operation occurred in the software (for example, the LeftButtonPress Event is invoked when the user presses the left mouse button in the renderer window). There exist special objects, named observers, which process the events and the commands are realized.

The Insight Toolkit

The Insight Toolkit (ITK) is an open source software toolkit for registration and segmentation. It is implemented in C++ and uses CMake build environment (briefly presented in **Chapter.**). ITK is considered the "first-cousin" of VTK, but there exists a difference between them: ITK is implemented using generic programming principles. It uses templates both for the algorithm implementation and the class interfaces. This type of heavily templated C++ code challenges many compilers and it can take much longer to compile. The other difference, is that the memory model depends on "smart pointers" that maintain a reference count to objects. Smart pointers can be allocated on the stack, and when scope is exited, the smart pointers disappear and decrement their reference count to the object that they refer to. There is no need to call itkFilter->Delete(), unlike VTK filters.

ITK can be used with VTK, but the main disadvantage is that it cannot be wrapped yet with Tcl, Phyton and Java. For GUIs, ITK uses Qt and FLTK (cross-platforms C++ GUIs toolkits). Because this application uses Tcl, the ITK classes were not included in the application; they were used independently to the application (more about the scope of using ITK for this project we can find in **Chapter 5**).

3.3.7. Using Tcl and Tk for Graphical User Interface

The development tools used to create the GUI are the Tcl scripting language and the Tk extension for creating GUI components, also containing some extension libraries of Tcl. *Tcl scripting language* was created by John Ousterhout, and it consists of a central core language interpreter that provides general functionality and a library of packages that provides extra "plug-in" functionality when needed. The main three goals of the Tcl language are: extensibility, simplicity, and ease of integration, being a really user friendly language.

Tk is a set of GUI components and acts like an extension to Tcl. Because this library is now an inseparable part of Tcl, they were glued together with the name "Tcl/Tk". Tk provides simple functionality to quickly build GUIs using the many ready-made "widgets" that acts as GUI building blocks for programmers.

In [S.Svensson] are briefly presented the advantages and disadvantages of programming with Tcl. Even if it has many extensions and the functionality is compared with Java or C++ programming languages, the efficiency is not so good. However, Tcl/Tk is very straightforward and the applications can be built fast and very simple, because of the simple syntax it has. But, on the backside also comes the drawback of less control over the code, so the programmer is limited. Furthermore, compared with C++ and Java, in Tcl the code is harder to debug.

Even if Tcl is easily to learn, small accessible documentation is available, especially for the integration of the *Tk* graphical components. One of the most considerable book for learning Tcl and Tk is [B.B. Welch,1999], and was extensively used in designing the graphical interface part of the application. Anyway there is not a reference which describes how the VTK C++ classes can be used with Tcl, but if we are considering some Tcl aspects, and then we try to understand the Tcl examples provided by VTK, we will be able to design our own applications.

Chapter 4. Implementation Methods

4.1. Digital Filtering Algorithms

Because of the complexity of the FFT algorithm, the *FFTReal* C++ library, from [L. de Soras, 2005] was used. The library is able to compute FFT on vectors of real numbers and their inverse FFT. It is very fast and LGPL (GNU Lesser General Public License), meaning that it can be used anywhere, including the commercial software.

The FFT for real signals algorithm is also described in [N. Wisniewski, 2006], but the *FFTReal* C++ library developed by Laurent de Soras is an optimized version of the algorithm described in [N. Wisniewski, 2006]. The algorithm eliminates the wasted calculations associated with the imaginary part of the time domain being zero, and the frequency spectrum being symmetrical. In this case, the algorithms are about 30% faster than the conventional FFT routines. Unfortunately, the code is about twice as long and the debugging is slightly harder, because the symmetry cannot be used to check if the algorithm works fine. This algorithm forces the imaginary part of the time domain to be zero, and the frequency domain to have left-right symmetry. But, because for *FFTReal* C++ library the functionality was checked, it is the best solution to use it; otherwise it is a waste of time and energy to try to develop such an algorithm if we are not some great mathematicians and programmers.

In the following the other algorithms that were developed for this project are presented, and includes: the FFT Convolution (the best way to compute the convolution), the Low-Pass Filter, High-Pass Filter, Band-Pass Filter and Band-Stop Filter algorithms. The FFT Convolution algorithm is used in performing the discussed types of filtering. The general block – diagram is presented in the **Figure 4-1**.

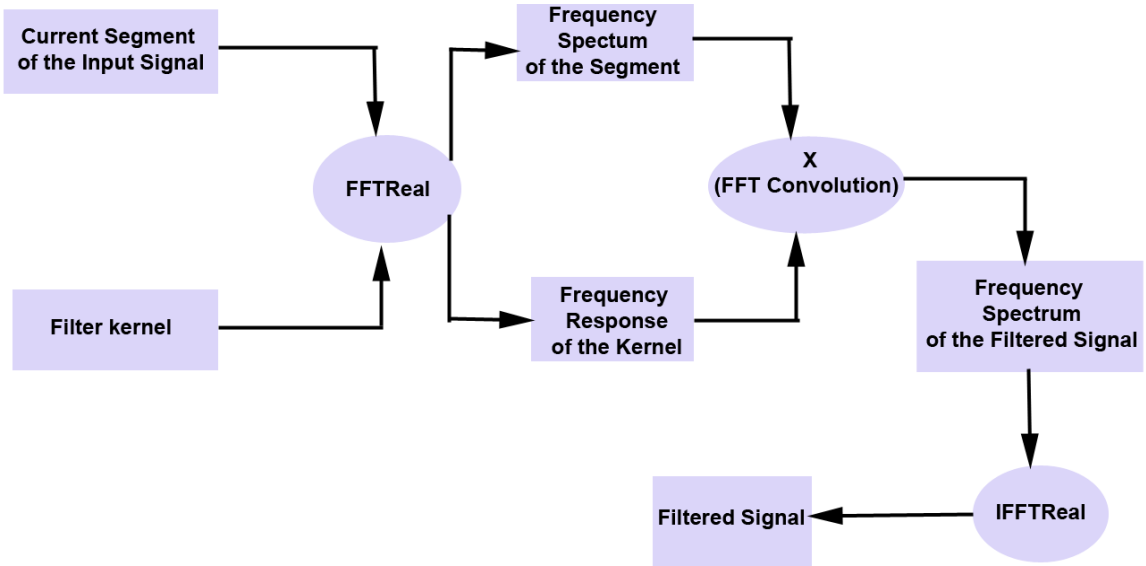


Figure 4-1 General block diagram of obtaining a filtered signal

As we saw in **Chapter 3** the strongest limitation of the windowed-sinc filter is the execution time; if there exists many points in the kernel and the standard convolution is used, it can be unacceptably long. A high-speed alternative algorithm for such a filter is the FFT convolution. In [N. Wisniewski, 2006] it is specified that for kernels longer than about 64 points, FFT convolution is faster than the standard convolution, while the same result is obtained. Besides this, recursive filters also provide good frequency separation and are a reasonable

alternative to the windowed-sinc filter. The recursive filters execute very rapidly, but according to [N. Wisniewski, 2006] they have less performance and flexibility than other digital filters.
In the following, the algorithm for FFT convolution is provided. It uses the overlap-method presented in the **Chapter 3** together with the FFT allowing the signals to be convolved by multiplying their frequency spectra.

The FFT convolution algorithm works as follows:

1. The frequency response of the filter is found by taking the DFT of the filter kernel, using the FFT;
2. The FFT converts this into the real and imaginary parts of the frequency response. These real and imaginary parts are stored in the computer for use when each segment is being calculated;
3. FFT is used to find the frequency spectrum of the input segment of the signal being processed;
4. The FFT converts this segment into the real and imaginary parts.
5. The frequency spectrum of the output segment is found by multiplying the filter's frequency response, by the spectrum of the input segment. The multiplication of frequency domain signals in rectangular form is carried out using the following formula:

$$\mathrm{Re}\,Y[f] = \mathrm{Re}\,X[f]\,\mathrm{Re}\,H[f] - \mathrm{Im}\,X[f]\,\mathrm{Im}\,H[f]$$
$$\mathrm{Im}\,Y[f] = \mathrm{Im}\,X[f]\,\mathrm{Re}\,H[f] + \mathrm{Re}\,X[f]\,\mathrm{Im}\,H[f]$$

6. The inverse FFT is then used to find the output segment, from its frequency spectrum.

Suppose that we have a 10 million point signal, and a 400 point filter kernel. We break the input signal into 16000 segments, with each segment having 625 points. This will results in an each output segment having 625+400-1=1024 points, so the 1024 points FFTs are used.

- Initialization of the arrays:
  Initialization of the time domain signal: XX[1023]
  Initialization of the real part frequency domain: REX[512]
  Initialization of the imaginary part of the frequency domain: IMX[512]
  Initialization of the real part of the filter's frequency response: REFR[512]
  Initialization of the imaginary part of the filter's frequency response: IMFR[512]
  Hold the overlapping samples from segment to segment: OLAP[398]

- Putting zeros in the array that holds the overlapping samples:
  FOR I = 0 TO 398
  OLAP[I] = 0
  ENDFOR
- Calculate and store the frequency response of the kernel, using FFT
- Load the filter kernel into XX[0] through XX[399] and sets XX[400] through XX[1023] to 0
- Apply FFT to transform the 1024 samples held in XX[] into 513 samples held in REX[] and IMX[], the real and imaginary part of the frequency response
- Transfer REX[] and IMX[] to REFR[] and IMFR[] to be later used in the program:
  FOR F = 0 TO 512
  REFR[F] = REX[F]
  IMFR[F] = IMX[F]
  ENDFOR
- Processing each of the 16000 segments:

FOR SEGMENT = 0 TO 15999

- We load the next segment being processed into XX[0] through XX[624] and we set XX[625] through XX[1023] to 0.
- We apply the FFT algorithm to find the frequency spectrum of the analyzed segment with the real part being placed in the 513 points of REX[], and the imaginary part being placed in 513 points of IMX[]
- We multiply the frequency spectrum, held in REX[] and IMX[] with the frequency response of the kernel, held in REFR[] and IMFR[](according to **Equation.**). The result of the multiplication is stored in REX[] and IMX[], overwriting the data previously there:

```
FOR F = 0 to 512
    TEMP = REX[F]*REFR[F] – IMX[F]*IMFR[F]
    IMX[F] = REX[F]*IMFR[F] + IMX[F]*REFR[F]
    REX[F] = TEMP
ENDFOR
```

- Because we have now the frequency spectrum of the output segment, we can apply the IFFT algorithm to obtain the time domain output signal. This is used to transform the 513 points held in REX[] and IMX[] into 1024 points held in XX[], the output segment.
-We add the last segment's overlap to this segment. Each output segment is divided in two sections. The first 625 points (0 to 624) need to be combined with the overlap from the previous output segment, and then written to the output signal. The last 399 points (625 to 1023) need to be saved so that they can overlap with the next output segment.
- OLAP[] array is used to hold the 399 samples that overlap from one segment to the next. Values in this array (from the previous output segment) are added to the output segment currently being working on, held in XX[].

```
FOR I = 0 TO 398
    XX[I] =XX[I] + OLAP [I]
ENDFOR
```

-Storing the 399 samples of the current output segment that need to be held over the next output segment

```
FOR I = 625 TO 1023
    OLAP[I-625] = XX[I]
ENDFOR
```

-Output the 625 samples stored in XX[0] to XX[624] to the file holding the output signal

ENDFOR

- After all 16000 (0 to 15999) segments have been processed, the array OLAP[] will contain the 399 samples from the last segment: 15999, that should overlap the segment 16000, but this segment does not exists, so the 399 samples are written to the output signal. So, the length of the output signal will be 16000 x 625 + 399 = 10 000 399 points (it matches the length of the input signal + the length of the filter kernel -1).

END

In [N. Wisniewski, 2006] it is specified that the FFTs must be long enough that the *circular convolution* does not take place. This means that the FFT should be the same length as the output segment (for example if we have a filter kernel with 129 points and each segment contains 128 points, the output segment must have 256 points, resulting that a 256 point FFT must be used. So, the filter kernel must be padded with 127 zeros to bring it to a total length of 256 points. In the same manner, the input segments must be padded with 128 zeros.

Now, let us see how the FFT convolution is used to obtain different filters. Before starting to design a filter, 2 parameters must be selected: f_c (the cutoff frequency) and M (the length of the kernel). After f_c and M have been selected, the filter kernel is calculated. For this case a Hamming window was selected, having the relation:

$$w_{Hamming}[i] = 0.54 - 0.46\cos(2\pi i / M)$$

The filter kernel will be mathematically described as:

$$h[i] = K\frac{\sin(2\pi f_c(i - M/2))}{i - M/2}\left[0.54 - 0.46\cos\left(\frac{2\pi i}{M}\right)\right]$$

In the **Chapter 3** are described the main components from this equation the sinc function, the M/2 shift, and the Hamming window. As we saw, we have obtained a low-pass filter kernel, and using the convolution with the input signal we are able to extract the low frequency components of the signal. In the following the *algorithm for the low-pass windowed-sinc filtered* is presented. Supposing that we have 20000 samples with a 801 point windowed-sinc filter, resulting a 19200 samples of filtered data. We consider the cutoff frequency 500 Hz.

The way the four filters depend on each other is illustrated in the **Figure 4-2**.

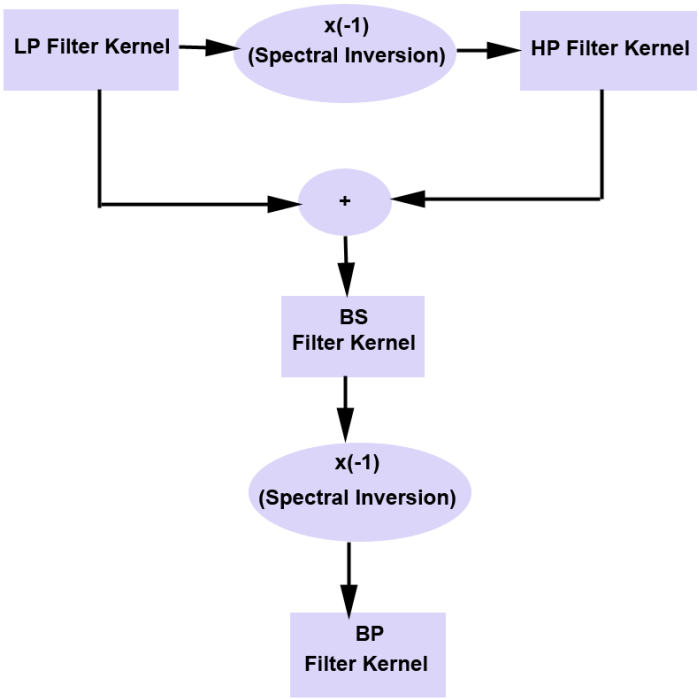


Figure 4-2 The dependency filters block

The Low-Pass and High-Pass Windowed-Sinc Filters Algorithms:

Hold the input signal: X[19999]
Hold the output signal: Y[19999]
Hold the filter kernel: H[800]
Initialize PI=3.14159265
Initialize cutoff frequency: Fc = 500
Set the filter kernel length to 800: length(M) = 800

- Here we calculate the low-pass filter kernel.

FOR I = 0 TO 800

*IF (I-M/2) = 0 THEN H[I] = 2*PI*Fc*
*IF (I-M/2) <> 0 THEN H[I] = SIN(2*PI*FC*(I-M/2))/(I-M/2)*
*H[I] = H[I] * (0.54 – 0.46 * COS(2*PI*I/M))*
ENDFOR

- Normalization of the low pass filter for unity gain at DC:

SUM = 0
FOR I = 0 TO 800
SUM = SUM + H[I]
ENDFOR

FOR I=0 TO 800
H[I] = H[I] / SUM
ENDFOR

- This is needed only if we want to obtain a HPF. We use the spectral inversion described in **Chapter 3** to obtain a high-pass filter kernel:

FOR I = 0 TO 800
*H[I] = (-1)*H(I)*
ENDFOR
H[800/2] = H[800/2] + 1

- We make the FFT convolution between the input signal and the high-pass/ low-pass filter kernel.

END

Let us consider that we already implement the high-pass and low-pass filter algorithms. Then, a band-stop kernel can be obtained if we add the two kernels of the low-pass filter and the high-pass filter. From here we apply the spectral inversion to the band-stop kernel in order to obtain a band-pass kernel.

The Band-Stop and Band-Pass Windowed-Sinc Filters Algorithms:

- Here we allocate memory for the lower cutoff (for example, of 500 Hz) and the upper cutoff (for example 1500 Hz). We assume that the band-pass kernel will have 801 samples (from 0 to 800):

Workspace for the lower cutoff: A[800]
Workspace for the upper cutoff: B[800]
Hold the band-pass kernel: H[800]
Initialize PI = 3.1415926
Set the filter kernel length: length(M) = 800
Initialize the lower cutoff frequency = 500

- Storing in A[I] the low-pass filter kernel already determined
- Make the normalization of the low-pass filter kernel for unity gain at DC
- Storing in B[I] the high-pass filter kernel already determined
- Make the normalization of the high-pass filter kernel for unity gain at DC
- Here we create a band-reject filter kernel by adding the two kernels for low-pass filter and high-pass filter:

FOR I = 0 TO 800
H[I] = A[I] + B[I]
ENDFOR

- This is needed only if we want to obtain a BPF. We use the spectral inversion described in **Chapter 3** to obtain a band-pass filter kernel:

FOR I = 0 TO 800
H[I] = 0 – (A[I] + B[I])
ENDFOR; END

4.1.1. C++ Implementation of the Algorithms

The application is developed using seven classes. Four classes are used to implement the four algorithms for the low-pass filter, high-pass filter, band-pass filter, and band-reject kernels (*CLPFilter1*, *CHPFilte1*, *CBPFilter1*, and *CBSFilter1*).

The dependencies between them are provided in the **Figure 4-3**, such that *CBPFilter1* and *CBSFIlter1* uses the kernels implemented in *CLPFilter1* and *CHPFilter1*. The main class, *CSFilter*, is used as an interface for all the filters applied to real-time or recorded signals, and it also contains the methods for frequency convolution. The *CSWindow* class is used as an interface for the *CHammingWindow* class. *CSWindow* was created because in this class there exists more windows implemented, not only the Hamming Window, but the best result is given by using the Hamming Window. More details about the classes are described in the **Appendix A**, where appears the entire C++ code for the provided algorithms.

In the **Figure 4-3**, the UML (Unified Modeling Language) class diagram is provided:

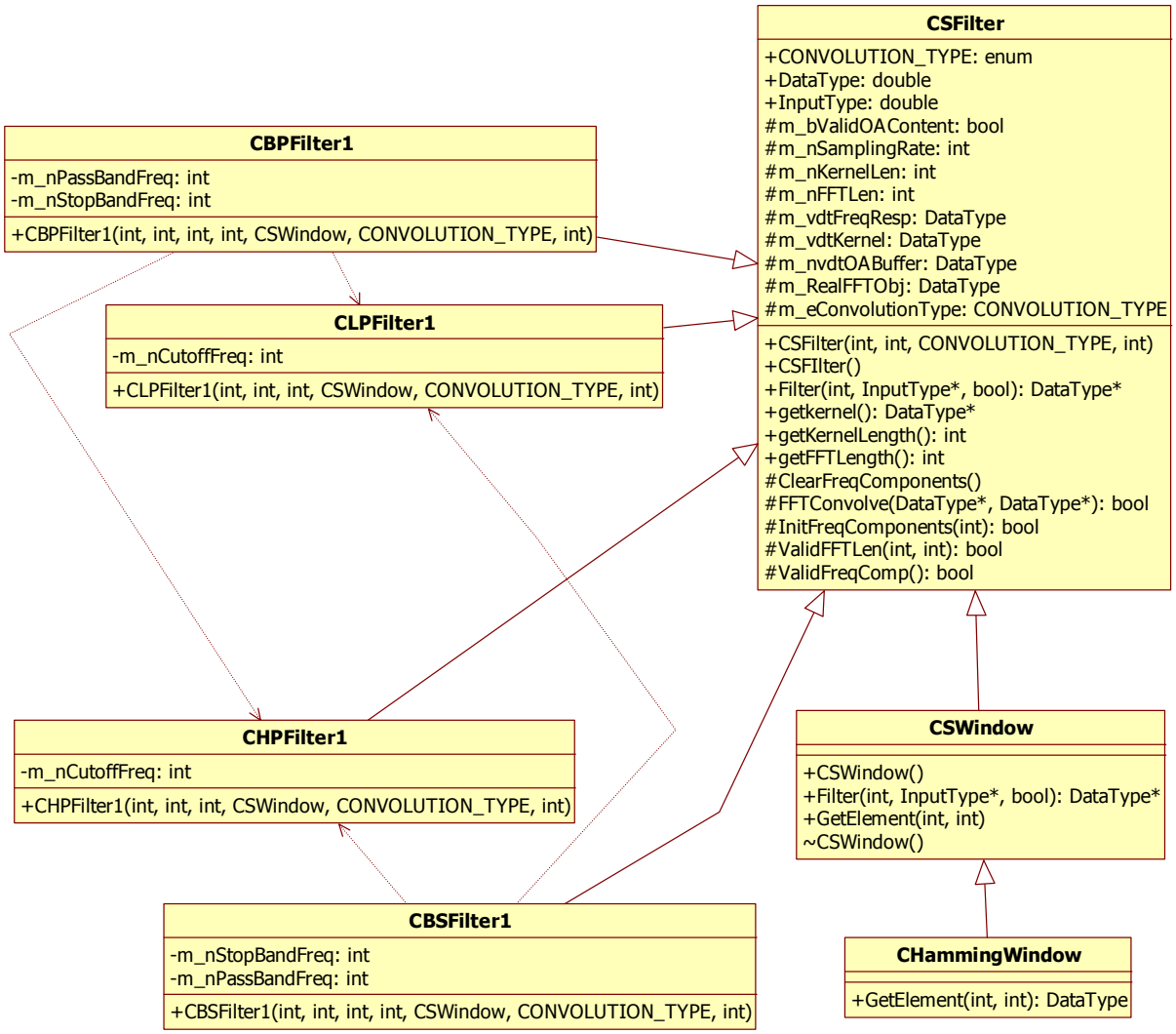


Figure 4-3 UML (Unified Modeling Language) class diagram

4.2. Medical Imaging Visualization

The Visualization Toolkit API is written in C++. In order to use VTK while coding in Tcl VTK needs to be wrapped into Tcl. The concept of "wrapping" code is used to provide a language specific abstraction layer to code written in another language. The entire wrapping process is described in [W. Schroeder, 1998] and was done using CMake 2.6 software, a cross-platform, open-source build system. CMake is a family of tools designed to build, test and package software and also to control the software compilation using simple platform and compiler independent configuration files [W. Schroeder, 1998].

After the wrapping process, all we need to do is to use the VTK classes in our Tcl application. This application has two main goals: the 2D visualization of the Dicom images (both CT and MRI), the extraction of the isosurfaces from these images, and the volume visualization based on these images (3D visualization). The main interface for both 2D and 3D visualization is pictured in the **Figure 4-4** and will be described in the next sections.

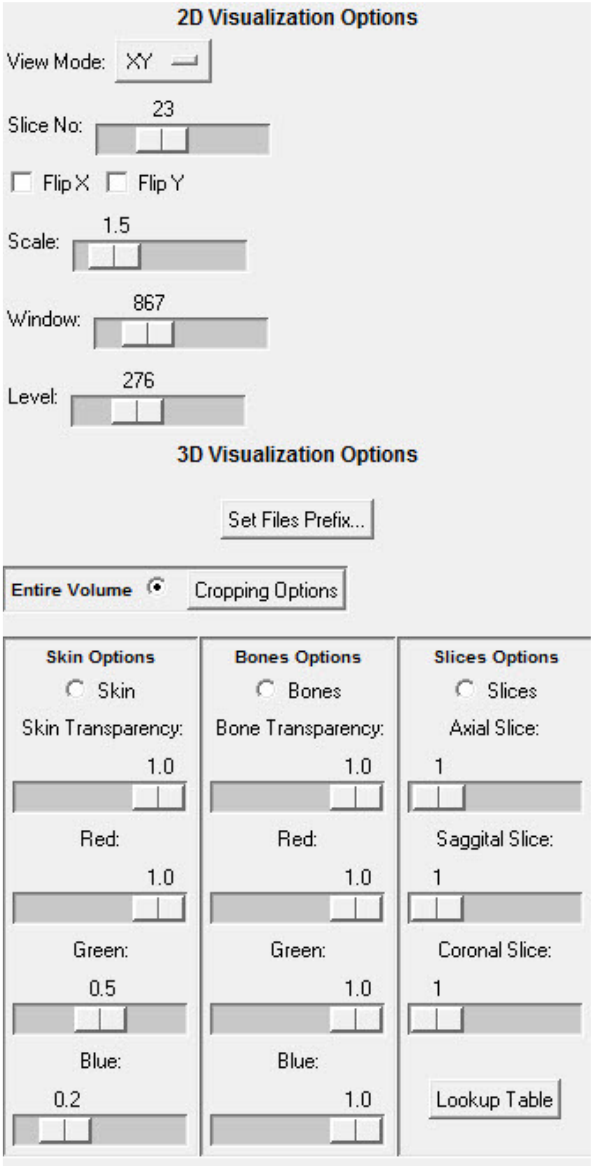


Figure 4-4 Main options for 2D and 3D visualization

4.2.1. 2D visualization

The application deals with Dicom images (especially CT data). The computed tomography measures the attenuation of X-rays as pass through the body. A CT image is a 3D image consisting in levels of gray that vary from black (for air), to gray (for soft tissue), and to white (for bone).

There are three main methods to read the input using VTK: (1) directly reading the Dicom files (*.dcm* format), (2) embed the Dicom files into a *.vtk* file, or (3) transform all the Dicom files into another (for example *.raw*) format in order to remove the header.

For 2D visualization, the second method: (2), was used. Using *itkDICOMSeriesFileNames* class a *.vtk* file format was obtained containing all the CT or MRI slices (for 3D visualization, the result is similar with the result when we use *vtkDICOMReader* class for reading *.dcm* files, the only difference being the extremely lower speed of the volume rendering in the case when we use the *itkDICOMSeriesFileNames)*. The C++ code for transforming a set of DICOM images into a *.vtk* file is provided in the **Appendix B**. The pipeline execution of this transformation is described in the **Figure 4-5.**

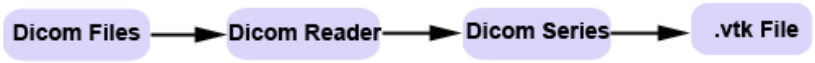


Figure 4-5 Pipeline execution for obtaining a *.vtk* file

When we open the application, a loading window automatically appears where one of the .vtk files can be selected. First, the filename is obtained, then the image is loaded, and the display is update. This is done using the Tk option, *tk_getOpenFile*, and the file is loaded using the *vtkStructuredPointsReader* class, a class that reads ASCII or binary structured points data files in vtk format. A copy of the current slice is stored in a variable, and then the parameters are reseted. The frame where the current slice is displayed is created with the class *vtkTkImageViewerWidget*, and it uses the method *GetImageViewer()*. In the **Figure 4-6** the pipeline for displaying an image is provided:

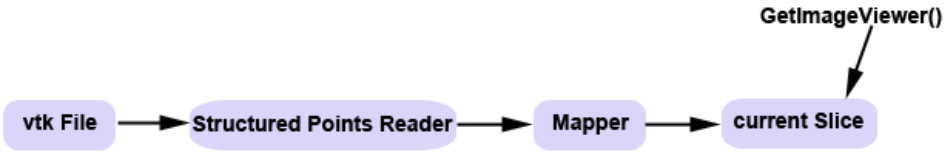


Figure 4-6 Pipeline execution for slice visualization

Some parameters of the *.vtk* file can be exctracted using the *vtkImageData* class. The image dimensions (using the *GetDimensions()* method), the pixel spacing (*GetSpacing()* method), the slice thickness (*GetScalars()* method), the level range and the window range (*GetRange()*). These values are displayed at the bottom of the main window.

All the *view modes* can be selected: Axial, Coronal and Sagittal, using the *vtkImagePermute* class, a class used to reorder the axes of the input, and its method *SetFilteredAxes(int, int, int).* After a view mode is selected, image is updated in the section for slice visualization. All the needed parameters are extracted using vtkImageData class and are updated for each *.vtk* loaded file.

The *visualization of each slice* is done using the *SetZSlice(z)* method of the *vtkActor* class, because each slice is seen as an actor, and the visualization of each slice is done in the image viewer Tk widget.

The *flipping options* are done using the *vtkImageFlip* class, and the *SetFilteresAxis()* method. The scaling is done using *vtkImageResample* class, with a scale range from 0.5 to 10.

The VTK used classes for 2D visualization is provided in the **Figure 4-7** In Tcl we do not create class, only procedures that may have global parameters, or not. Objects for the described VTK classes are created inside the procedures and then are used in the member functions.

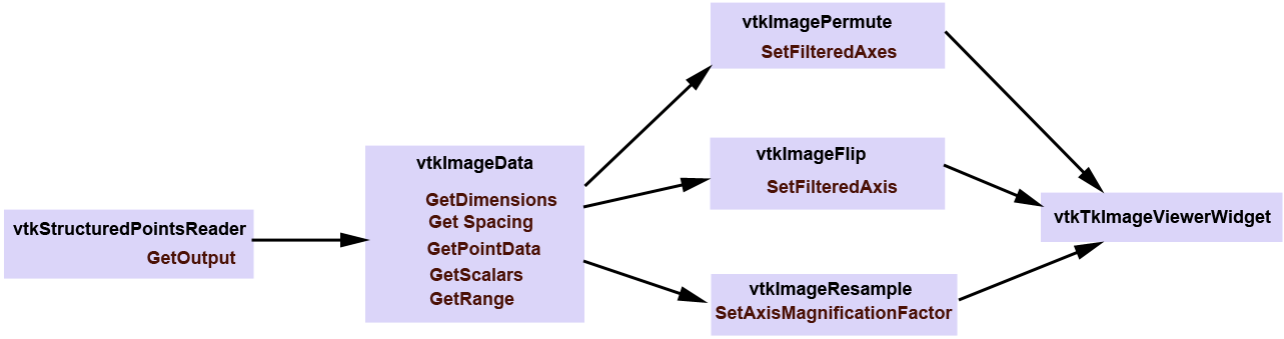


Figure 4-7 The VTK classes used for 2D visualization of the slices

4.2.2. Isosurface Extraction

The objective of the Dicom images is to take the gray scale data and to convert it into information that will aid the surgeon. For this, isocontouring techniques are used to extract the skin and bone surfaces and to display orthogonal cross-sections to put the isosurface in context. In [W.Schroeder, et al., 2002] is defined that a density value of 500 will define the air/skin boundary, and a value of 1150 define the soft tissue/bone boundary. The main steps in representing this are:

1. Read the input;
2. For each anatomical feature of interest, create an isosurface;
3. Transform the models from patient space to world space;
4. Render the models.

In the following all these steps are explained:

1. Reading the input

As we know, medical images may come in many formats. The usual format for CT and MRI types of images is the DICOM format *(.dcm*). This format introduces in the structure of each image an header, containing the information about the patient (name, average, birth day, gender physician's name, country, and other information) and about the extracted slices (slice thickness, pixel spacing, rows, columns, slice location, acquisition time and date and other information). So, the next step will be to try to remove this header, because will affect the result of extracting the isosurface.

Preparing the datasets:

For thesis, a multiple datasets with CT and MRI images were obtained from *Department of General Neurology, University of Tübingen, Tübingen,* Germany. The datasets were obtained from different patients during surgery. VTK contains a C++ class vtkDICOMReader able to read DICOM formats, but because of the confidentiality, the header file was removed for this application.

The confidentiality was not the only criteria considered in removing the header files and transforming DICOM in another file format: *.raw*. The isosurface of the data is affected if we will not "clean" the pixel cell (topic discussed in **Chapter 3**), as we can see in the experimental

results of **Chapter 5**. If the pixel cell is not "cleaned", the 16-bit pixel is used to mark connectivity between voxels, and the isosurface extraction is not very accurate.

Another thing about the processing of medical image data is that it can be acquired in a variety of orders that refer to the relationship of consecutive slices to the patient. Radiologists view an image as though they were looking at the patient's feet, so on the display the patient's left appears on the right. For CT images, there exist two standard orders: *top to bottom* and *bottom to top*. For a top to bottom acquisition, slice *i* is farther from patient's feet than slice *i*- 1. This is important, because if we ignore the slice acquisition order a flipping of left and right can result.

Removing the header files:

In order to remove the header files from DICOM files, the *dicom2* program was used. This is a free command-line driven program which allows us to convert medical images and DICOM files to various other formats (*.bmp*, *.png*, *.raw*), while performing some rudimentary image processing tasks [S.Barre, 2003]. The files format in which were DICOM images converted is *.raw* format.

Two commands were typed in the command line in order to separate the two information types of the DICOM format:

```
dicom2 –to=textfiles –t *
dicom2 –to=rawdir –r *
```

The first command extracts each header file and save them as *.txt* files in the "textfile" and the second command extracts each image data and save them as *.raw* files in the "rawdir" directory.

Reading the input:

There exists a class provided by VTK that can read 16-bit file formats – *vtkVolume16Reader*. To read data, this class was instantiated and the instance variables can be introduced by the user (the variables are market by $):

```
vtkVolume16Reader v16
        v16 SetDataDimensions $params(col) $params(row)
        v16 SetDataByteOrderToLittleEndian
        v16 SetFilePrefix  $params(prefix)
        v16 SetFilePattern %s%d.$params(type)
        v16 SetImageRange $params(ssli) $params(esli)
        v16 SetDataSpacing  $params(pixx) $params(pixy) $params(pixz)
```

The user can read these variables from the *.txt* header files that were obtained after using the *dicom2.exe* program. The **Figure 4-8** illustrates the mode in which the data can be introduced in parallel with the information needed from one header file (left: the interface for introducing the information about the slices; right: part of the header file that is used in creating the isosurface - all the needed data is stored in the square brackets).

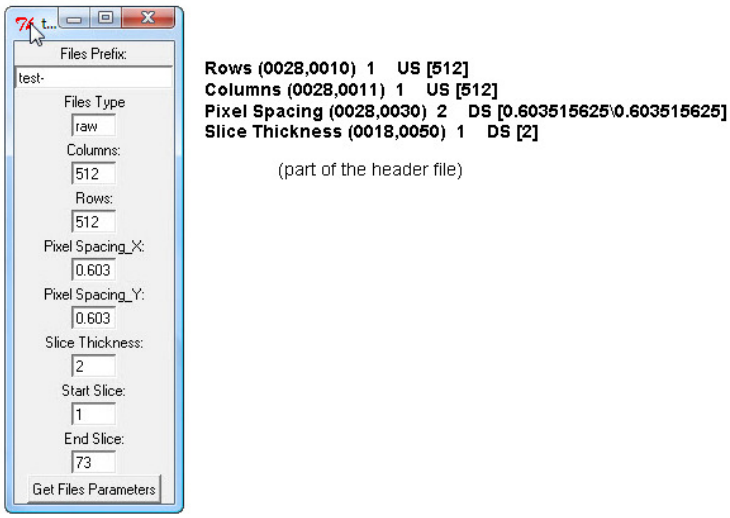


Figure 4-8 Introducing the data for 3D visualization

The "Files Prefix" is the prefix of the *.raw* files (in our case the files are saved with the name from test-1.raw to test-73.raw). There are 73 slices in this example.

2. Creating the isosurface

As we saw in the **Chapter 3**, this method is called "indirect volume visualization" and there exist three methods for isosurface visualization: Volume Rendering, Marching Cubes and Dividing Cubes. The Volume Rendering is not recommended in the case when we want to interact with our data at the highest possible speed. Dividing Cubes does not produce a surface, instead it outputs a point set that represents a surface, but the points are not connected. Dividing Cubes was briefly experimented with, but due to the huge amount of points it produced, it was discarded. Finally, the Marching Cubes algorithm is the best option. Then, the class *vtkPolyDataNormals* is used to generate surface normals for the data. The procedure is like the one illustrated in the **Chapter 3**: the output of the *vtkMarchingCubes* filter was connected to a mapper and an actor using *vtkPolyDataMapper* and *vtkActor* classes. In our case the **Figure 4-5** will be transformed in the following pipeline (**Figure 4-9**):

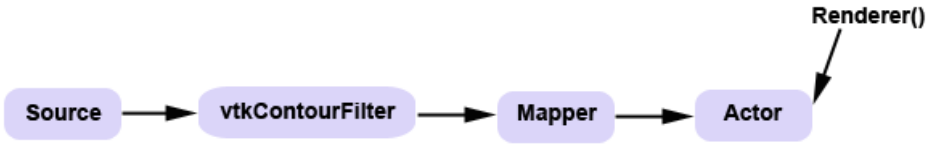


Figure 4-9 Simplified pipeline execution for extracting the isosurface

The *vtkMarchingCubes* filter is the best solution in our case. In VTK library there exists another filter called *vtkContourFilter*. This filter also takes as input any dataset and generates an output isosurfaces, and in [W.Schroeder, et al., 2002] is described that it automatically create an instance of *vtkMarchingCubes*. In the **Chapter 5** we can see that there is no difference between them. The class *vtkPolyDataNormals* is used to generate the better surface normals for data. *vtkMarchingCubes* can also generate normals, but better results are achieved when the normals are taken directly from the surface (if we use *vtkPolyDataNormals*), than from the data (if we use *vtkMarchingCubes*). In the **Chapter 5** we can see the comparison between them.

The *vtkOutlineFilter* provides context for the isosurface, creating an outline around the data. An initial view is set up in a window size of 400 x 400 pixels.

The *SetValue()* method of the *vtkContourFilter* class allows us to select the needed isosurface. For example, a value of 500 is used to select the skin of the patient, and a value of 1150 is used to select the bones of the patient. The visualization can be improved in a number of ways. First, a more appropriate color for the skin can be chosen. For example, using the

vtkProperty method *SetDiffuse Color()*, we can obtain a fleshy tone for the skin color. Finally, we can improve the rendering performance on the system, by adding *vtkStripper* that creates triangle strips from the output of the contouring process. For both skin extraction and bone extraction, in the **Figure 4-10** is provided a diagram with the used classes and the associated methods for extracting the isosurface.

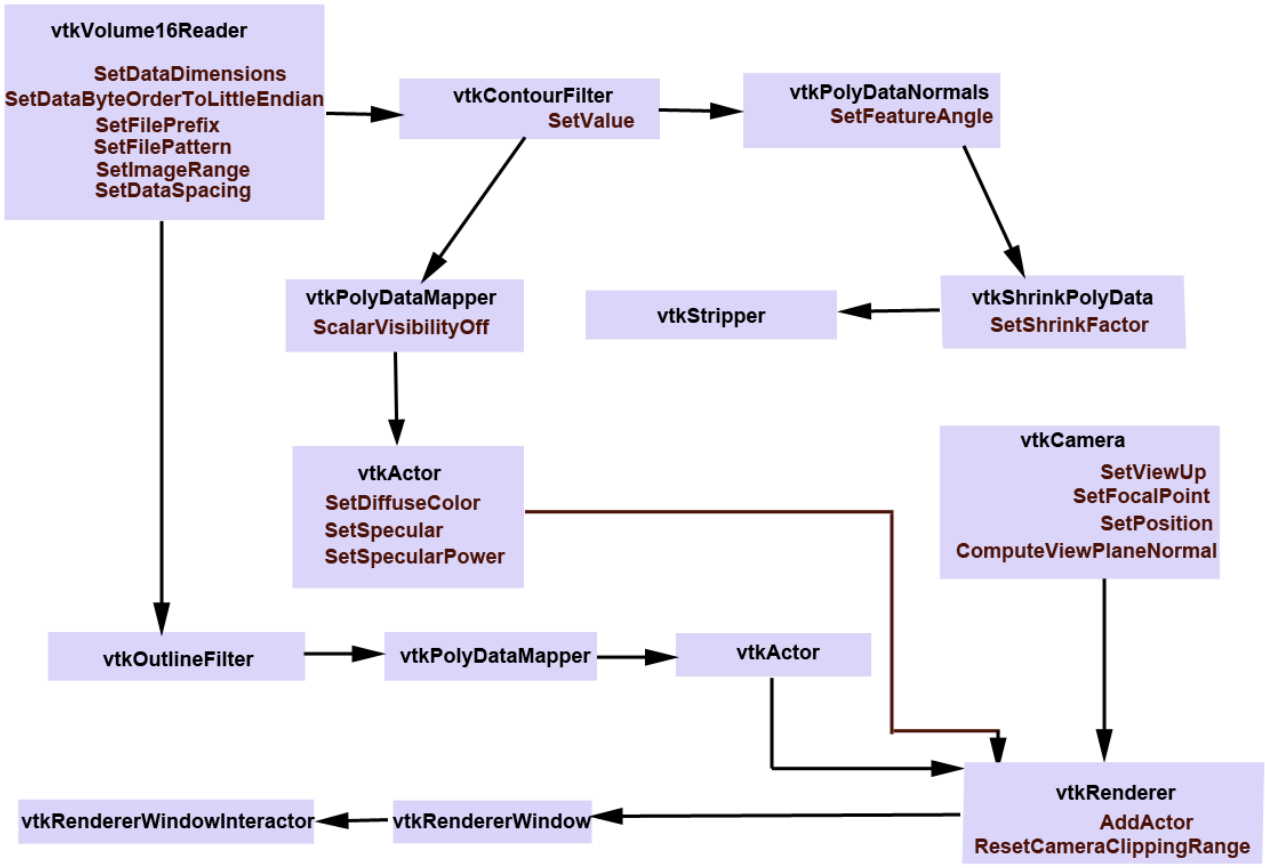


Figure 4-10 Application pipeline execution for extracting the isosurface

Besides isocontouring, VTK provides other useful techniques for indirect exploring the volume data. For example, we can view orthogonal slices or planes through the data. One approach is to use the *Texture Mapping* technique described in the **Chapter 3**. This technique gives the best result in terms of interactive performance.

Texture-mapped volume renderers sample and blend a volume to produce an image by projecting a set of texture-mapped polygons that span the entire volume. In 2D texture-mapped volume rendering the dataset is decomposed into a set of orthographic slices along the axis of the volume most parallel to the viewing direction [W.Schroeder, et al., 2002]. The basic rendering algorithm consists of a loop over the orthogonal slices in a back-to-front order, where for each slice a 2D texture is copied into texture memory. Each slice, which is a rectangular polygon, is projected to show the entire 2D texture. If neighboring slices are far apart relative to the image size, then it may be necessary to use a software bilinear interpolation method to extract additional slices from the volume in order to achieve a desired accuracy.

The performance of this algorithm consists, according to [W.Schroeder, et al., 2002], in three parts: *software sampling rate*, the *texture download rate*, and the *texture-mapped polygon scan conversion rate*. The software sampling step is required to create the texture image, being dependent on the view direction due to cache locality when accessing volumetric data stored in a linear array. The texture download rate is the rate at which the image can be transferred from main memory to texture mapping memory. The scan conversion of the polygon is limited by the rate at which the graphics hardware can process pixels in the image, or the pixel fill rate.

Three orthogonal planes can be extracted, corresponding to axial, sagittal, and coronal cross sections, very familiar to any radiologist. As we know, the axial plane is perpendicular to the patient's neck, sagittal passes from left to right, and coronal passes from front to back. Each of these planes can be rendered with a different color lookup table, in order to distinct them. In the **Chapter 5** we can see the result of applying this technique. The lookup table can be changed from the main options window. In order to make visible the coronal cross sections, there exist options for translucent rendering of the skin and the bones.

The image data consisting in slices mapped to colors using the filter *vtkImageMapToColors* in combination with the lookup tables. The actual display of the slice is performed using *vtkImageActor*, class that is able to combine a quadrilateral, polygon plane with a texture map. The image data of type unsigned char required from *vtkImageActor* is provided by the *vtkImageMapToColors*. To avoid copying the data and to specify the 2D texture to use, the *DisplayExtent* of each *vtkImageActor* can be set, using the interface of slide bars for each of the axes. In the **Figure 4-11** is provided the pipeline execution of one orthogonal plane. If we need three orthogonal planes, we need to use this pipeline three times. For this pipeline to work, it must be connected with the pipeline for extraction the isosurface.

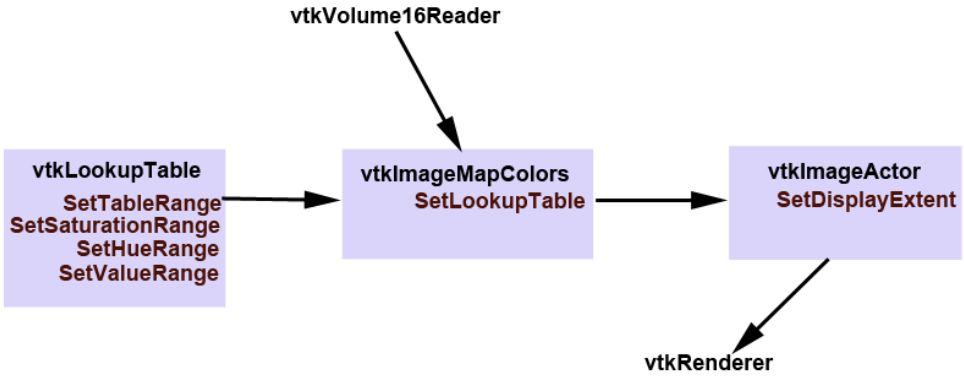


Figure 4-11 Pipeline execution for obtaining one orthogonal plane

A special attention requires the *vtkRendererWindowInteractor* class, because as we saw in the **Chapter 3**, interaction is an essential feature of systems that provides methods for data exploration and query. This class is used to capture windowing-system specific events in the renderer window, translate them into VTK events, and then take action as appropriate to the event invocation.

4.2.3. Direct Volume Visualization

Direct Volume Rendering or object-order volume rendering methods process samples in the volume based on the organization of the voxels in the dataset and the current camera parameters. The voxels must be transverse in either a front-to-back or back-to-front order to obtain correct results. When graphics hardware is employed for compositing, a back-to-front ordering is typically preferred. If a software compositing method is used, a front-to-back ordering is more common since partial image results are more visually meaningful, and can be used to avoid additional processing when a pixel reaches full opacity. Voxel ordering based on distance to the view plane is not always necessary since some volume rendering operations, such as MIP or average, can be processed in any order and still yield correct results.

In the **Chapter 3** three algorithm for direct visualization were described: Ray Casting, Shear Warp and Texture-Mapping. The Shear Warp algorithm is not currently implemented in VTK in the version 5.6, which was used for this application. Anyway, this algorithm is expected to bring fast data access due to the three precomputed rotated versions of the volume. This is one

of the fastest rendering algorithms in image space, because of the viewing directions that are not parallel to one of the coordinate axes – so reduces the costs for memory access.

Unlike 2D hardware (described in the **Creating the isosurface** part), 3D texture hardware is capable of loading and interpolating between multiple slices in a volume by utilizing 3D interpolation techniques such as *tri-linear interpolation*. If the texture memory is large enough to hold the entire volume, then the rendering algorithm is simple. The entire volume is downloaded into texture memory once as a preprocessing step. To render an image, a set of equally spaced planes along the viewing direction and parallel to the image plane is clipped against the volume. The resulting polygons are then projected in back-to-front order with the appropriate 3D texture coordinates. For large volumes, like our case, it is not possible to load the entire volume into 3D texture memory, so for using this technique we must break the dataset into small enough subvolumes, or bricks, so that each brick will fit in texture memory. The subvolumes must then be processed in back-to-front order while computing the appropriately clipped polygon vertices inside the subvolume. Then, special care must be taken to ensure that boundaries between bricks do not result in image artifacts, which may be time consuming. These are the reasons why for this application this technique was not applied. However, 3D texture mapping is superior to the 2D version in its ability to sample the volume, general yielding higher quality images with fewer artifacts.

Besides Ray Casting and Texture-Mapping there exists another visualization method called *Maximum Intensity Projection* (MIP) and is implemented using the *vtkRayCastMIPFunction* class. The role of this algorithm is to project in the visualization plane the voxels with maximum intensity that fall in the way of parallel rays traced from the viewpoint to the plane of projection. MIP is used for the detection of lung nodules in lung cancer screening programs which utilize computed tomography scans. MIP enhances the 3D nature of these nodules, making them stand out from pulmonary bronchi and vasculature.

For now, let us consider the Ray Casting algorithm that is implemented in VTK using the *vtkVolumeRayCastCompositeFunction* class and *vtkVolumeRayCastMapper* class. In theory, a 3D texture-mapped volume render and a ray casting volume render perform the same computations, have the same complexity $O(n^3)$, and produce identical images. Therefore, we can view 3D texture mapping and standard ray casting methods as functionally equivalent. The main advantage of using a texture mapping approach is the ability to utilize relatively fast graphics hardware to perform the sampling and blending operations. However, there are several drawbacks to using graphics hardware for volume rendering. Hardware texture-mapped volume renderings tend to have more artifacts than software ray casting techniques due to limited precision within the frame buffer for storing partial results at each pixel during blending. Then, only few ray functions are supported by the hardware, and advanced techniques such as *Shading* are more difficult to achieve. Anyway, the texture mapping techniques is in continuing evolution, and all the drawbacks will disappear.

In *vtkVolumeRayCastMapper* class, there exists methods for cropping the entire volume, like *CroppingOn()* and *SetCroppingRegionPlanes*. These methods were used in this application for cropping the volume. Another class is used, *vtkColorTransferFunction*, to map the voxel intensities to colors. It is modality specific, and anatomy-specific, because there can be selected colors for flesh (between 500 and 1000) and colors for bones (1150 and over). This method is used to enhance the 3D volume visualization. In the **Chapter 5** we can see a direct volume visualization that it not includes any of these methods. Also, the opacity transfer function is controlled using the *vtkPiecewiseFunction* class and is often used to control the opacity of different tissue types. This class can be used also to decrease the opacity in the "flat" regions of the volume, while maintaining the opacity at the boundaries between tissue types. The class that is used to attach the color and opacity functions to the volume is *vtkVolumeProperty*. Here, there exist methods to set the interpolation mode. Two types of interpolation are implemented: the Nearest Neighbor and the Tri-linear interpolation.

Briefly, *tri-linear interpolation* is one of the most popular interpolations used in computer graphics, and it is composed of seven linear interpolations. In the first computation step, four linear interpolation compute the weighted sample points between two neighboring voxels on an edge of a volume cell along one direction (either x, y, or z). In the second computation step, the result of two such linear interpolations between voxels located on the same face of the volume cell, are combined by interpolation to compute the result of a bilinear interpolation. Finally, the result of the two bilinear interpolations on opposite faces of the volume cell, are combined by a linear interpolation into the final trilinear sample point within the volume cell. So, three levels of linear interpolations are chained into one function. This method can be selected if we want a very accurate rendering of the volume.

The *nearest-neighbor interpolation*, as we know, is the simplest method of interpolation. It simply selects the value of the nearest point and does not consider the values of the neighboring points at all. The algorithm is very simple to implement and is commonly used in real-time 3D rendering because of the speed of rendering and because if we have a good resolution of the data, the artifacts are not so visible.

Returning to the *vtkVolumeProperty* class, there exists also the *ShadeOn* method, used to enhance the appearance of the volume and make it more "3D", because it turn on the directional lighting. The impact of the shading can be decreased by increasing the *Ambient* coefficient while decreasing the *Diffuse* and *Specular* coefficients.

In the **Figure 4-12** there are presented the classes used for direct volume visualization. Another observation need to be done. The interaction with the actors in the renderer window can be faster if the *vtkLODActor* class is used. This class has the role to store multiple levels of detail (LOD) and can automatically switch between them. It is able to select which level of detail to use based on how much time it has been allocated to render.

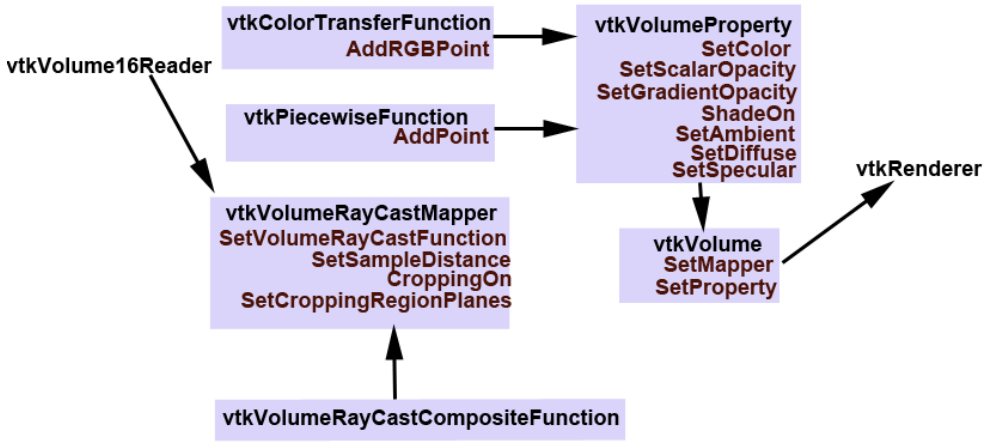


Figure 4-12 Pipeline execution for direct visualization of the volume

When a voxel is processed, its projected position on the view plane is determined and an operation is performed at that pixel location using the voxel and image information. This operator is similar to the ray function used in image-order ray casting techniques. This approach to projecting voxels is fast and efficient, but it often yields image artifacts due to the discrete selection of the projected image pixel. For example, as we move the camera closer to the volume, neighboring voxels will project to increasingly distant pixels on the view plane, resulting in distracting "holes" in the image (illustrated in the **Chapter 5**).

This problem is addressed by a volume rendering technique called *Splatting*, which distributes the energy of a voxel across many pixels. A kernel with finite extent is placed around each data sample. The footprint is the projected contribution of this sample onto the image plane, and it is computed by integrating the kernel along the viewing direction and storing the results in a 2D footprint table. The evaluation of the footprint table and the image space extent of a sample can be performed once as a preprocessing step to volume rendering. Splatting is difficult for

perspective volume rendering because the image space extent is not identical for all samples. Also, several considerations must be taken into account: the type of the kernel, the radius of the kernel, and the resolution of the footprint table will all impact the appearance of the final image.

4.2.4. Segmented Data Visualization

There exist many 3D segmented atlases available for download. For testing how the segmented data can be visualized using VTK, a multi-modality MRI-based atlas of the brain was achieved from The Neuroimage Analysis Center's Computational Clinical Anatomy Core and the Surgical Planning Lab from Brigham and Women's Hospital. As described in [Halle M. et al., 2011] the segmented slices have the *.vtk* format, and were obtained from a healthy subject. There exist 148 brain sections, each of them included in a *.vtk* file.

These atlases need a special treatment, because they cannot be read using the general way, with *vtkStructuredPointsReader* class. Instead, the *vtkPolyDataReader* class must be used in parallel with *vtkDataSetMapper* class as a mapper. Otherwise, the rendering process is similar with the one described for direct volume visualization.
The simplest diagram for reading such data is provided in the **Figure 4-13**.

Figure 4-13 Pipeline execution for visualization of segmented data

Chapter 5. Experimental Results

5.1. Filter Design

For signal visualization, the *Igor Pro* software tool, 6.11 version, was used. *Igor* is an integrated program for visualizing, analyzing, transforming and presenting experimental data, and it includes the ability to handle large data sets. Image display and processing can be done, having a combination of graphical and command-line interface. This tools were consider just for producing high-quality, and finely-tuned graphics.

For exemplification, the input signal consists in addition of three separate sinusoidal signals, with the amplitude of 10 and the frequencies: 200Hz, 1000Hz and 5000Hz. The sampling rate is selected at 20000 Hz. In the **Figure 5-5** the input signal is presented (a little bit expanded, because of the large sample rate) and its spectrum is presented.

The length of the kernel, M was selected 801 (from 0 to 800), and the filter's cutoff frequency 500 Hz. The kernel can be visualized in the **Figure 5-1**, being a windowed-sinc kernel (a low-pass filter kernel – more about the filters kernels was discussed in the theoretical fundamentals of this paper).

Three sinus signals were generated: with the frequencies 200 Hz (**Figure 5-2**), 1000 Hz (**Figure 5-3**), and 5000 Hz (**Figure 5-4**). The sampling rate is 20000 Hz and the length of the signals is one second. These signals were added into a single signal, considered the input signal of the system.

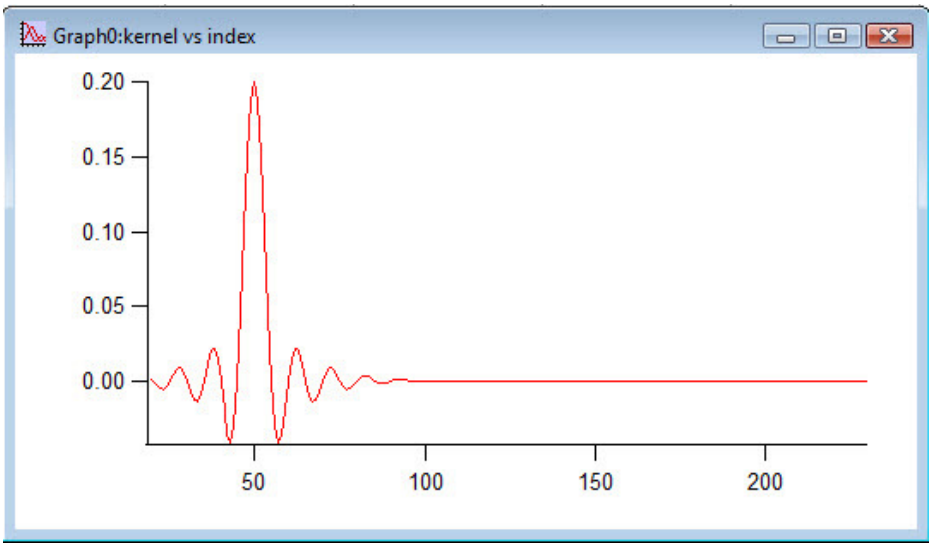


Figure 5-1 Windowed-sinc kernel visualization

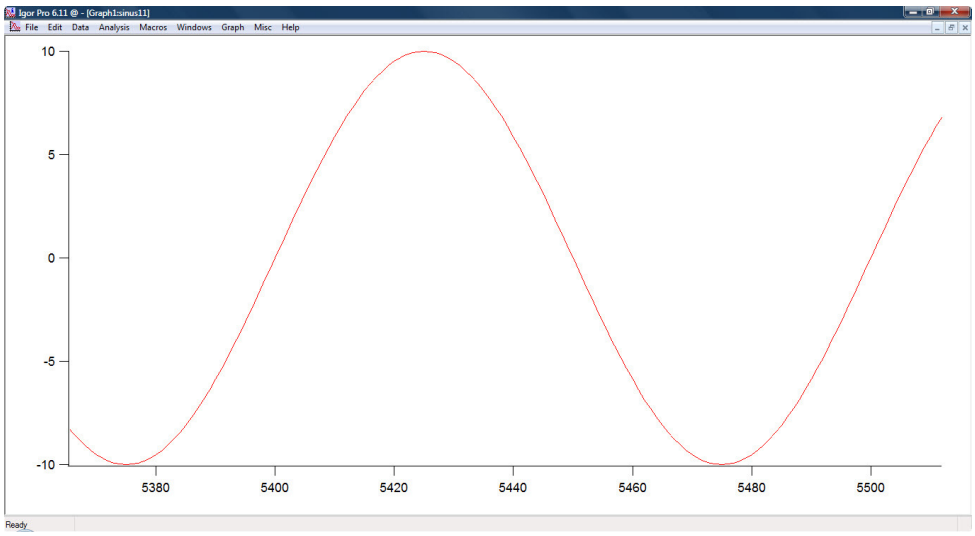


Figure 5-2 Signal with 200 Hz frequency and amplitude of 10

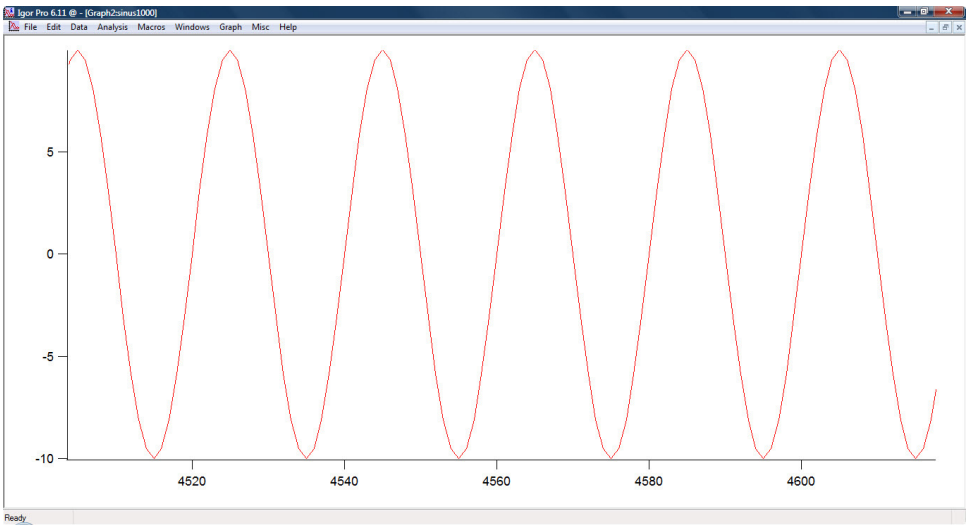


Figure 5-3 Signal with 1000 Hz frequency and amplitude of 10

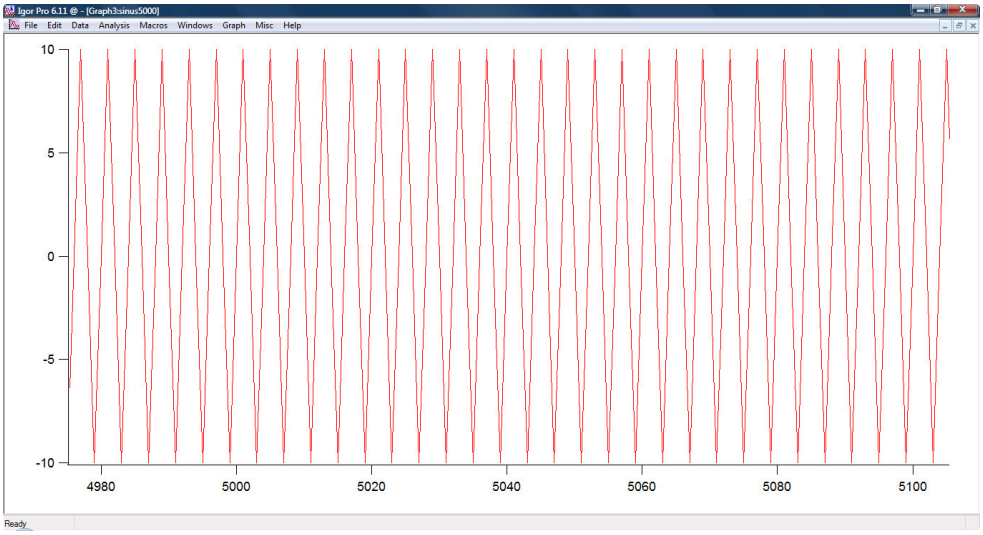


Figure 5-4 Signal with 5000 Hz frequency and amplitude of 10

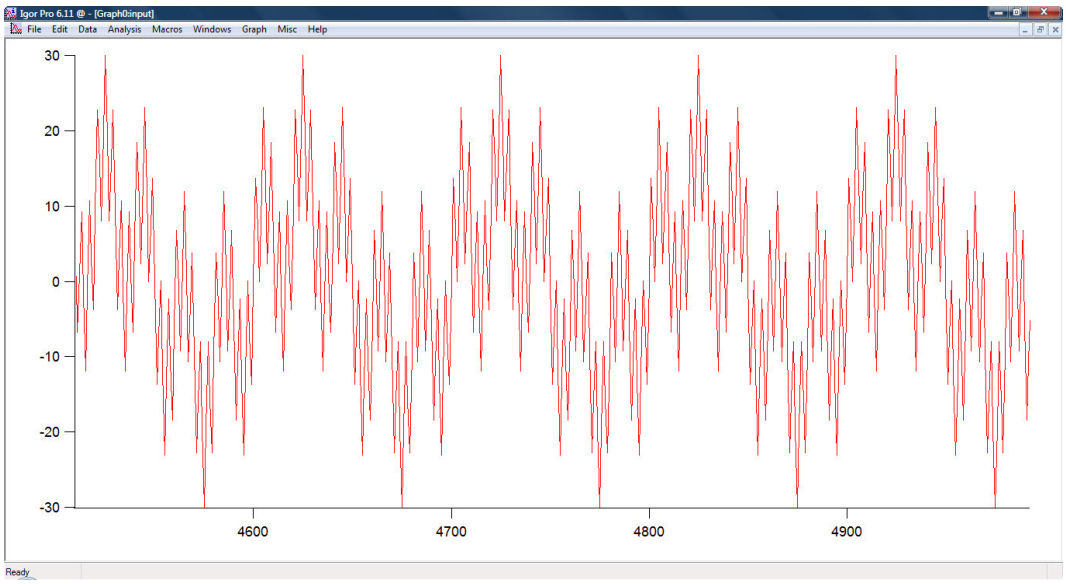


Figure 5-5 Input signal. The sum of the three signals with the frequencies of 200 Hz, 1000Hz and 5000Hz

After the three signals were generated and summed up, using the *FFTReal* library, the frequency spectrum was extracted (**Figure 5-6**). Now, we can easily see which the main frequency components of the input signal are. Therefore, all we need to do is to apply some filtering, using the convolution between the input signal and the kernel of the filter.

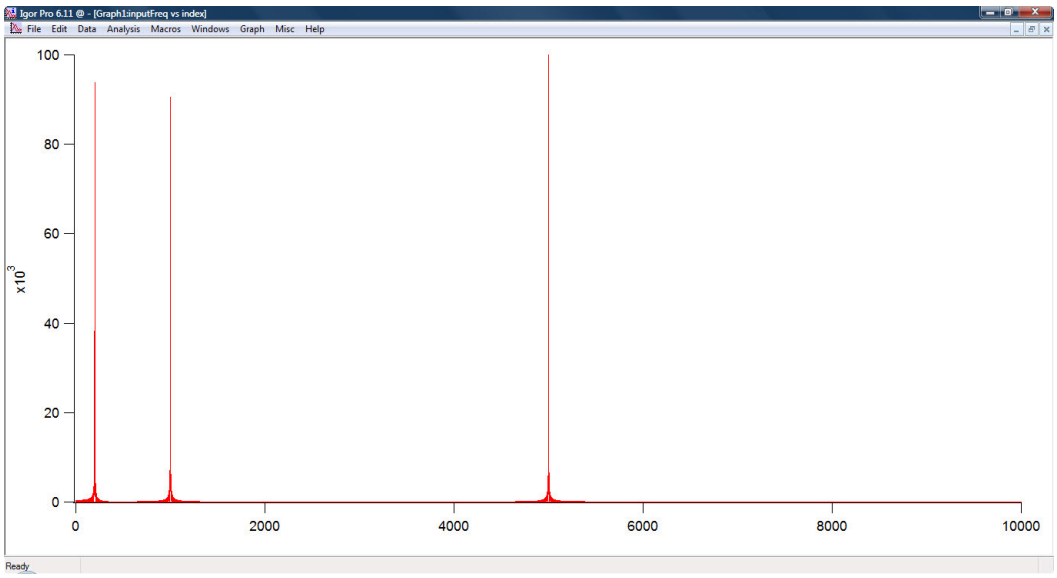


Figure 5-6 Input signal frequency spectrum

In the following figures, we will visualize the output signal and its corresponding frequency spectrum, after the filtering process. In the **Figure 5-7** it is displayed the signal obtained after a low-pass filtering was applied to the input signal. Because the cutoff frequency of the filter was 500 Hz, we can easily see, if we are considering the **Figure 5-8,** that only the 200 Hz frequency signal is kept. If we compare the result from **Figure 5-7** with the first component of the input signal from **Figure 5-2** we can see that we have obtained the same signal.

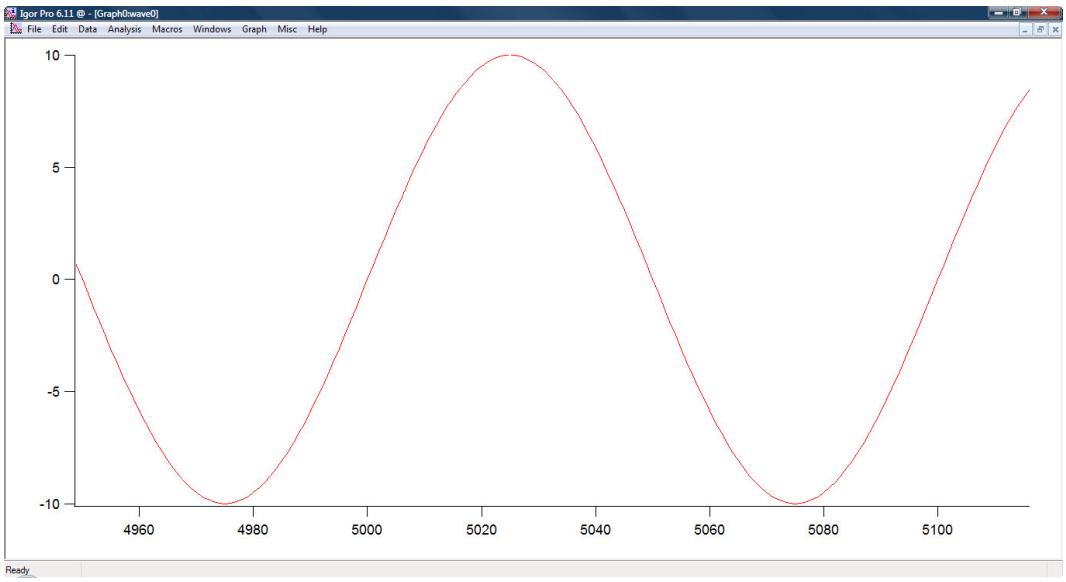


Figure 5-7 Low-pass filtered signal, with the cutoff frequency of 500Hz

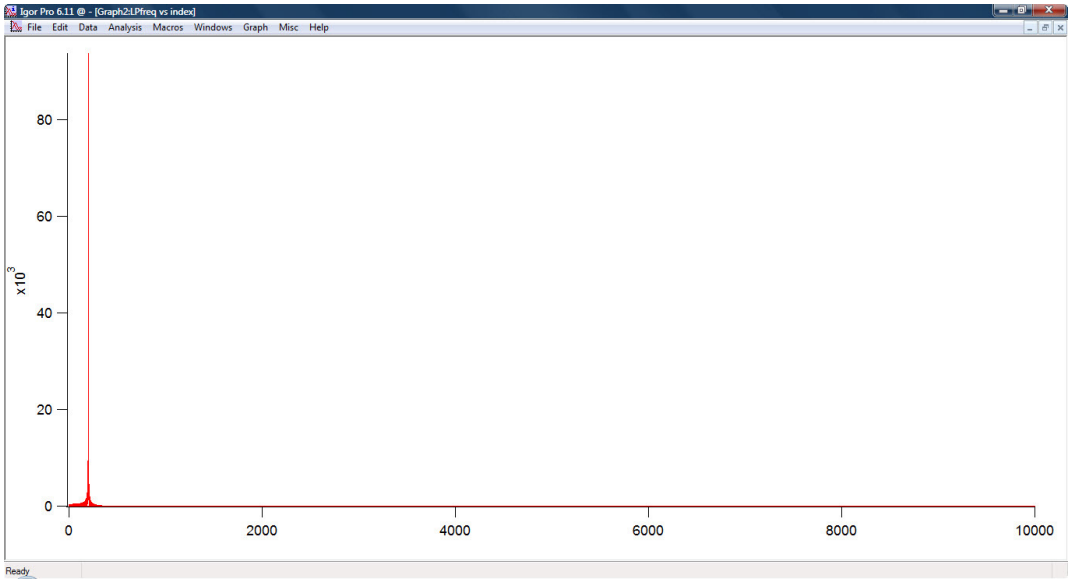


Figure 5-8 The spectrum of the low-pass filtered signal

As we know, a high-pass filtered signal is obtained by convolving the high-pass filter kernel with the input signal. The high-pass filter kernel can be obtained by applying the spectral inversion method, described in the **Chapter 3**, to low-pass filter kernel already designed. In the **Figure 5-9** we can visualize the obtained signal if we are applying such a filter with the same cutoff frequency of 500 Hz, to the input signal. We can easily observe in the **Figure 5-10** that only the 1000 Hz and 5000 Hz signals are kept.

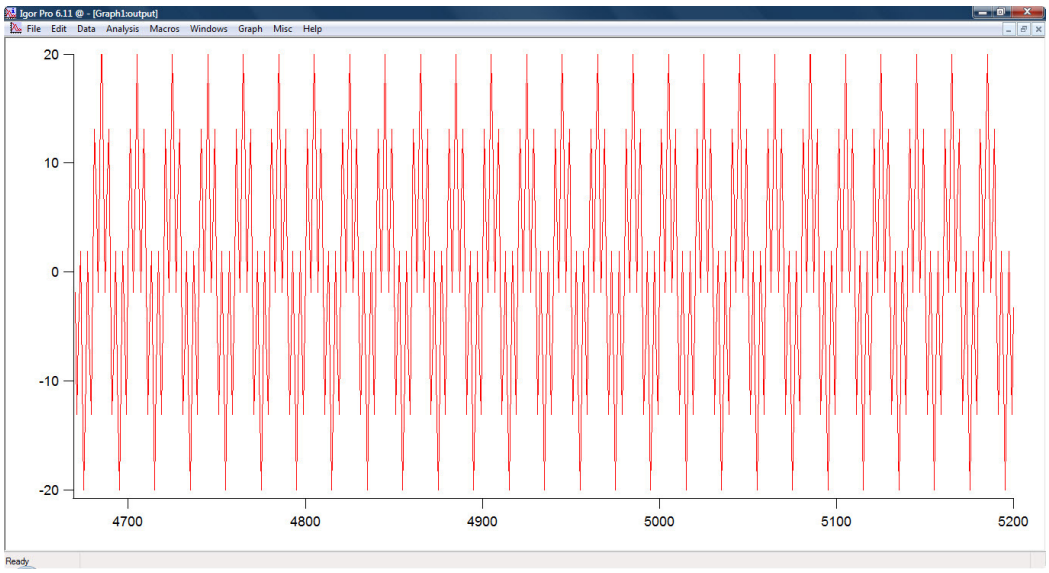


Figure 5-9 High-pass filtered signal, with the cutoff frequency of 500Hz

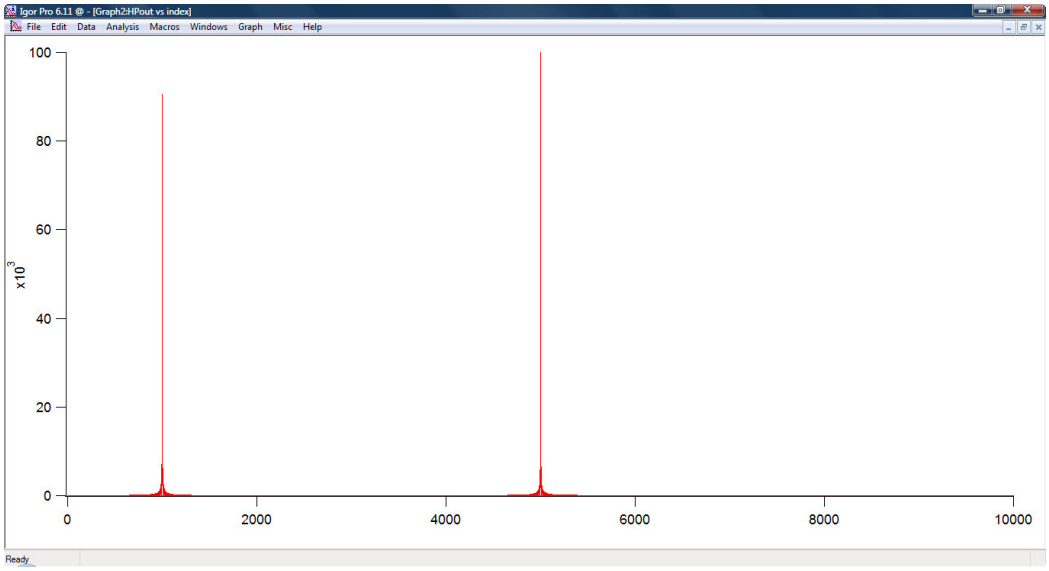


Figure 5-10 The spectrum of the high-pass filtered signal

In a similar way as we described above, the input signal can be passed through a band-pass filter with the cutoff frequencies of 500 Hz and 1500 Hz, results in the signal from the **Figure 5-11**, with its corresponding spectrum in the **Figure 5-12**. In this case we can observe that only the 1000 Hz signal is selected, which is similar with the one from the **Figure 5-3**.
If we are applying a band-stop filter to the input signal, the result can be visualized in the **Figure 5-13**, with its corresponding frequency spectrum from the **Figure 5-14**.

Using the simplest signals, the sinusoidal ones, have test the implemented solution meets the desired requirements. Now, a future work will be to read the local field potential signals, which were addressed in this paper as the most useful neural signals. More about this type of signals is presented in the **Chapter 3**.

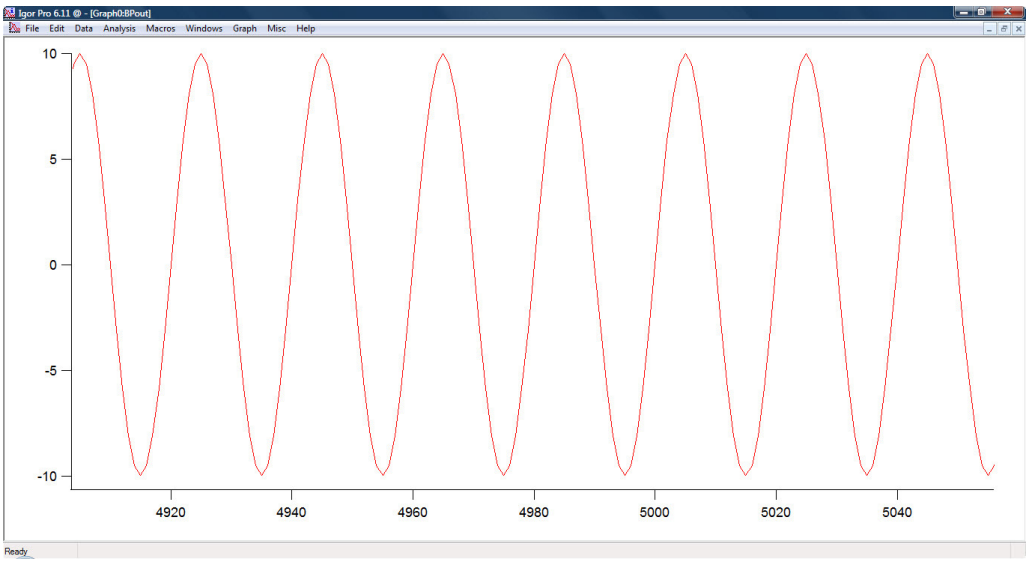


Figure 5-11 Band-pass filtered signal, with the cutoff frequencies of 500Hz and 1500 Hz

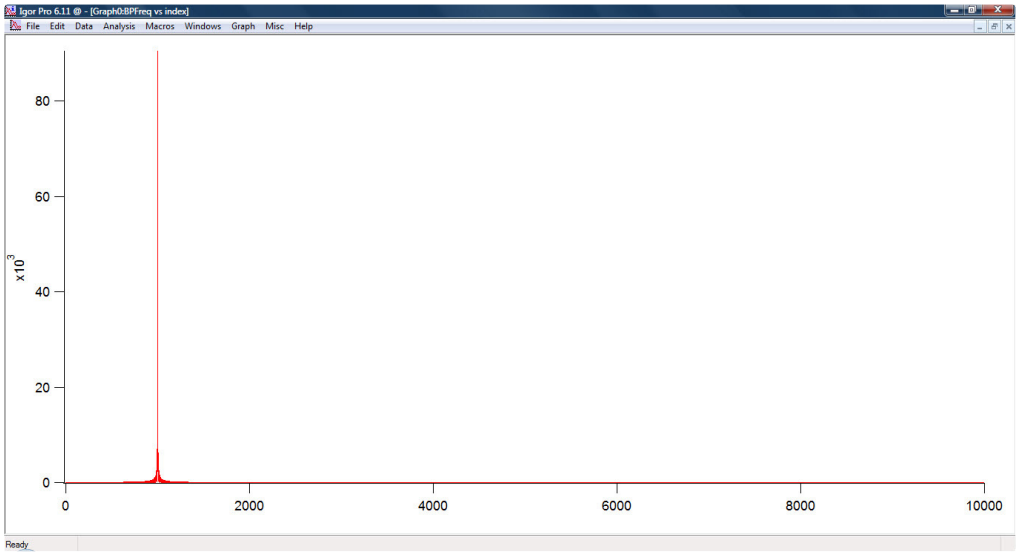


Figure 5-12 The spectrum of the band-pass filtered signal

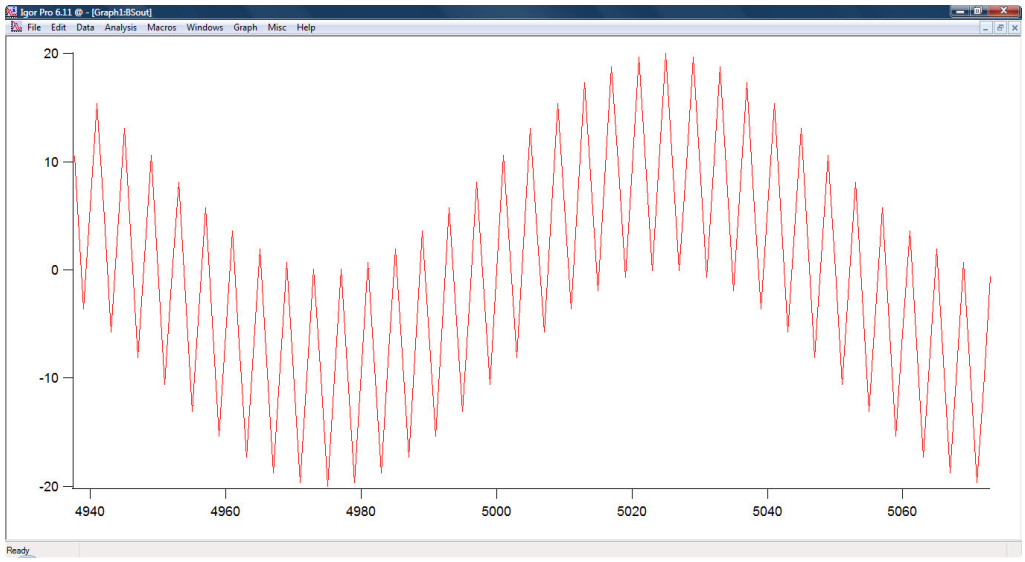


Figure 5-13 Band-reject filtered signal, with the cutoff frequencies of 500Hz and 1500Hz

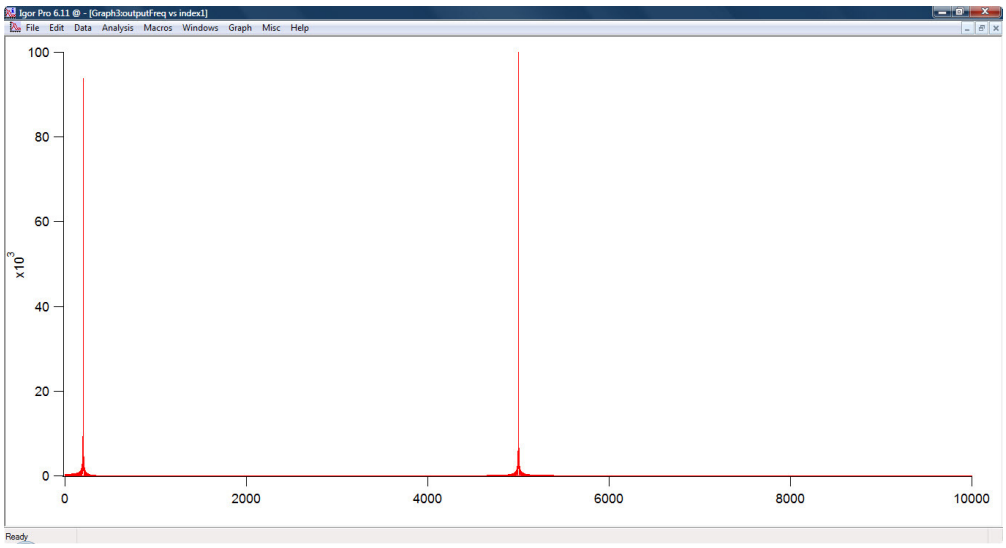


Figure 5-14 The spectrum of the band-reject filtered signal

5.2. Visualization Techniques

For the experimental results of this section, we create a case study. We consider a dataset of Dicom images, containing CT data recorded from a patient during the surgery, which were already transformed in *.raw* files or integrated in a *.vtk* file (as was described in the **Chapter 4**). For comparison, MRI data were processed in the same way as the CT data, and was recorded from the same patient. The data was provided by the *Department of General Neurology, University of Tübingen,* Tübingen, Germany.

As we know from the previous chapters, we need to know some aspects about how the data acquisition was done. The header files included in the Dicom images contain such information, and are given in the following (the information is exactly in the form that we read from the header file, and is contained in the squared brackets):

- For CT data: *Rows (0028,0010) 1 US [512]*
  Columns (0028,0011) 1 US [512]
  Pixel Spacing (0028,0030) 2 DS [0.603515625\0.603515625]
  Slice Thickness (0018,0050) 1 DS [2]
  Pixel Spacing (0028,0030) 2 DS [0.603515625\0.603515625]
  Bits Allocated (0028,0100) 1 US [16]
  Bits Stored (0028,0101) 1 US [12]
  High Bit (0028,0102) 1 US [11]

From the CT data information provided in the header file, we can say that the CT study contains 73 slices (this information is not contained in the header file), spaced 2 mm apart (*Slice Thickness…DS [2]*). Each slice has 512^2 pixels (*Rows…US [512]; Columns…US [512]*) spaced 0.603515625 mm apart (*Pixel Spacing…DS [0.603515625\0.603515625]*) with 12 bits of gray level.

- For MR data: *Rows (0028,0010) 1 US [256]*
  Columns (0028,0011) 1 US [256]
  Pixel Spacing (0028,0030) 2 DS [1\1]
  Slice Thickness (0018,0050) 1 DS [1]
  Bits Allocated (0028,0100) 1 US [16]
  Bits Stored (0028,0101) 1 US [12]
  High Bit (0028,0102) 1 US [11]

Similar, the MR study contains 144 slices, spaced 1mm apart. Each slice has 256^2 pixels spaced 1 mm apart with 12 bits of gray level.

Our challenge is to take this gray scale data (over 36 megabytes) and convert it into information that will aid the surgeon. We will see how we can visualize the data, the way in which the isocontouring techniques are used to extract the skin and bone surfaces and display orthogonal cross-sections to put the isosurface context. Also, we are interested in direct visualization of the volume in the most appropriate ways as possible.

5.2.1. 2D Visualization

The first thing when we record a set of CT data is to visualize it. The steps for visualize the data were described in the **Chapter 4**, our task being now to analyze the data, to see what useful information it can bring.

In the **Figure 5-15** we can observe a CT cross section through the head loaded in the visualization screen; it consists of levels of gray that vary from black (for air), to gray (for soft tissue), to white (for bone). We can easily predict that we have an axial slice, taken perpendicular to the spine through the ears. The gray boundary around the head shows the ears and the nose. The dark regions on the interior of the slice are the nasal passages and ear canals. We can also observe a part of the stereotactic frame that was used in to fix the head. Usually, this frame is attached during the entire surgery period, because it helps to take the decision about the nickel-sized (14 mm) burr hole that needs to be made in the patient's skull.

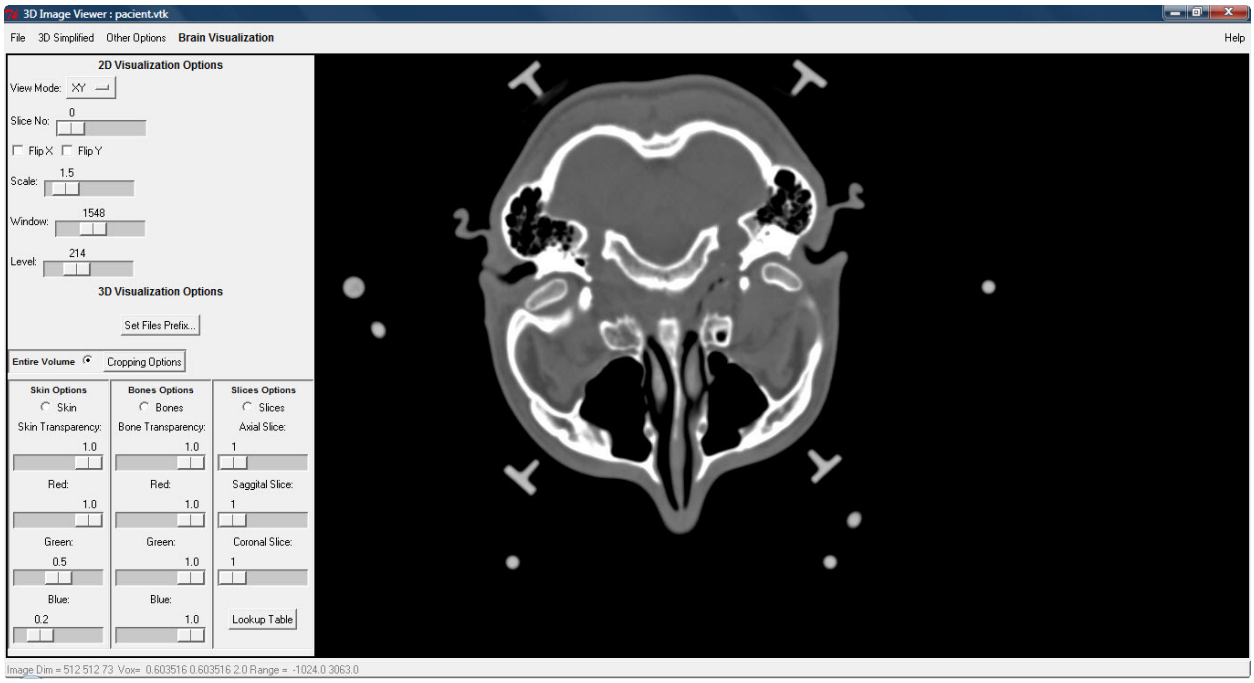


Figure 5-15 The axial slice number 0 of the CT dataset, no flipping, scaled at 1.5, with a window value of 1424 and a level value of 214.

In the header file of each Dicom image there exist information about the Window and the Level values of the image and this information, and the range of these values is obtained using the *GetPointData*, *GetScalars* and *GetRange* methods of the *vtkImageData* class. In the **Figure 5-16** we can observe that for a Window value of 557 and a Level value of 400, only the bright part, corresponding to bone, is visible. In this way we can analyze the bone structure of this slice.

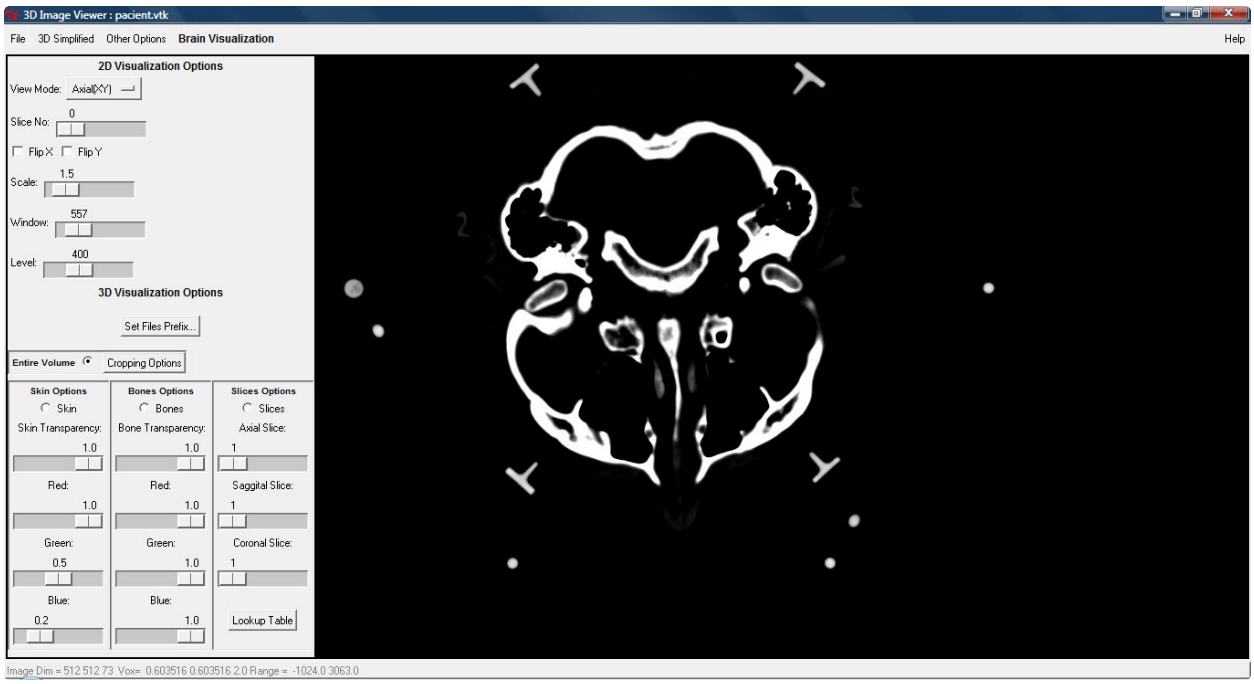


Figure 5-16 The axial slice number 0 of a CT dataset, no flipping, scaled at 1.5, with a window value of 557 and a level value of 400.

The Window/Level values determine the brightness and contrast of the displayed slice. If we increase the Window value, as it is done in the **Figure 5-17**, we can observe a flat distribution of the grey values and we even detect the boundaries of the slice. In this case it is very hard to separate the bone from tissue or air.

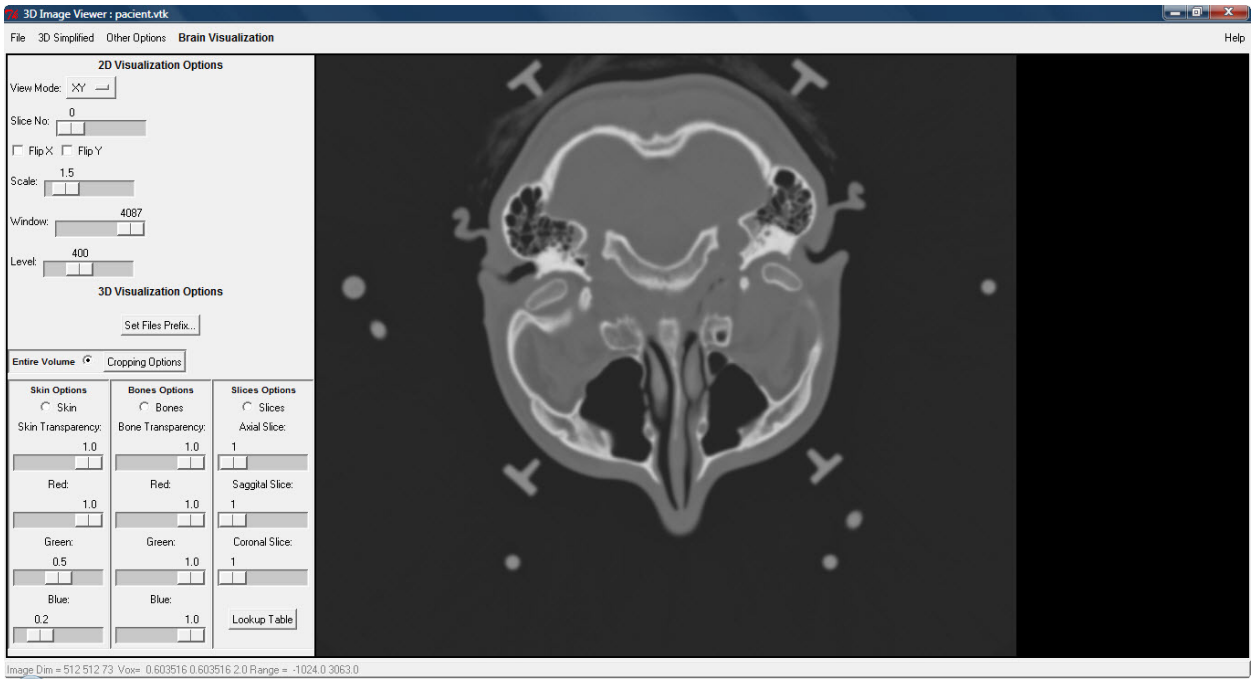


Figure 5-17 The axial slice number 0 of a CT dataset, no flipping, scaled at 1.5, with a window value of 4087 and a level value of 400.

If the Level value is increased, we lose in contrast. In the **Figure 5-18** is presented this situation. We can observe that is very hard to distinguish the bone, like there were just parts of the bone. So, it is very important to set the Level and Window values of a slice in the most

appropriate way, depending on the region of interest, otherwise, wrong interpretations may arise, this being critical in analyzing a medical image.

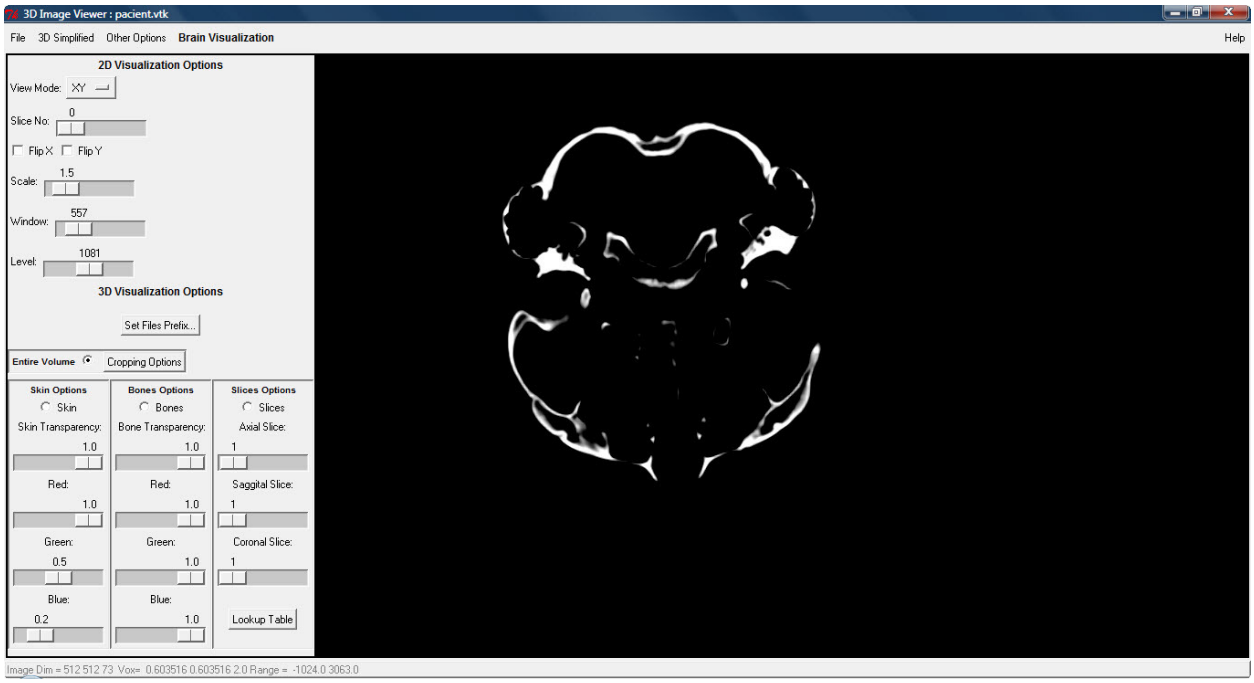


Figure 5-18 The axial slice number 0 of a CT dataset, no flipping, scaled at 1.5, with a window value of 557 and a level value of 1081.

The Dicom images are tridimensional images, meaning that the acquisition is done in the axial, sagittal and coronal view modes. When we want to visualize such images it is important to be able to see all the three view modes. As was mentioned in the **Chapter 4**, the three visualization modes can be obtained if we just make a permutation of the X, Y and Z axes, and was obtained using the *vtkImagePermute* class. The coronal and the sagittal view modes of the slice number 256 are illustrated in the **Figure 5-19** (coronal) and **Figure 5-20** (sagittal).

Figure 5-19 The coronal slice number 256 of a CT dataset, no flipping, scaled at 2.5, with a window of 1548 and a level of 400.

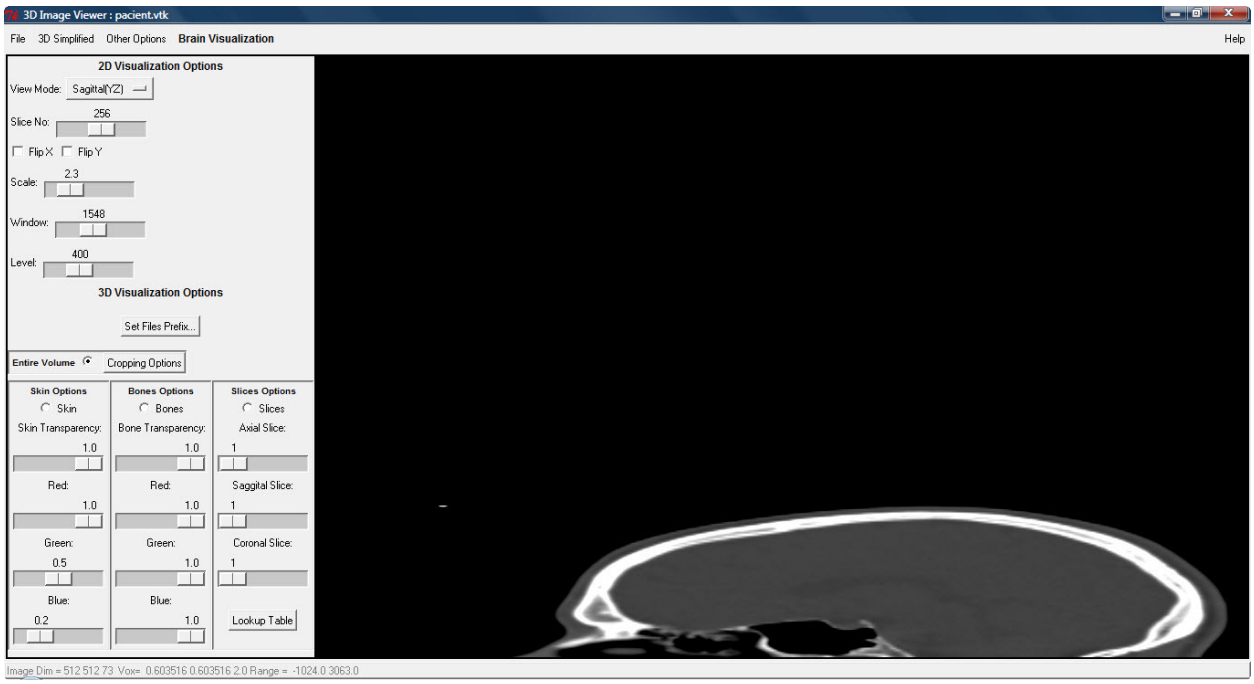


Figure 5-20 The sagittal slice number 256 of a CT dataset, no flipping, scaled at 2.3, with a window of 1548 and a level of 400.

If we are considering the MR data study case, the 2D visualization is quite different. In the **Figure 5-21** the Window and Level values were chosen in such a way to have the best visualization possible of the slice number 69 (Window value of 451 and a Level value of 326). All the other combinations of these parameters will result in a darker and low contrasting image.
Maybe, now is the moment when the following question arises: Why such a difference? Why we are not able to view only the bones? Where are the bones in this slice?

Well, when a MRI is taken, the radiologists need some information regarding the tendons and ligaments. If we are comparing this figure with the previous ones, we can observe that here we can easily observe some parts of the brain, tendons, and ligaments. What are the parts that are best seen depends on the density of the tissues that compose the tendons and ligaments.
CT is the preferred modality for cancer, and pneumonia. Bleeding in the brain, especially from injury, is better seen on CT than MRI. But a tumor in the brain is better seen on MRI. On the other hand, CT is far superior at visualizing the lungs and organs in the chest cavity. MRI is not a good tool for visualizing the chest or lungs at all.

Then, the difference in the way the images are produced in MRI and CT is the physics involved, and was described in the **Chapter 3**.

Another important thing that may be observed in this figure is that even if in the left side of the interface, the "Axial(XY)" is selected, in the visualization screen we see a sagittal mode. This was intentionally left in this way, because in the **Chapter 4** the importance of the data acquisition was discussed. It is important to know the order in which the data were recorded in order to analyze it. In this case, the dataset were just processed into a *.vtk* file and the slice order was not considered, so the view modes are interchanged (sagittal with axial).

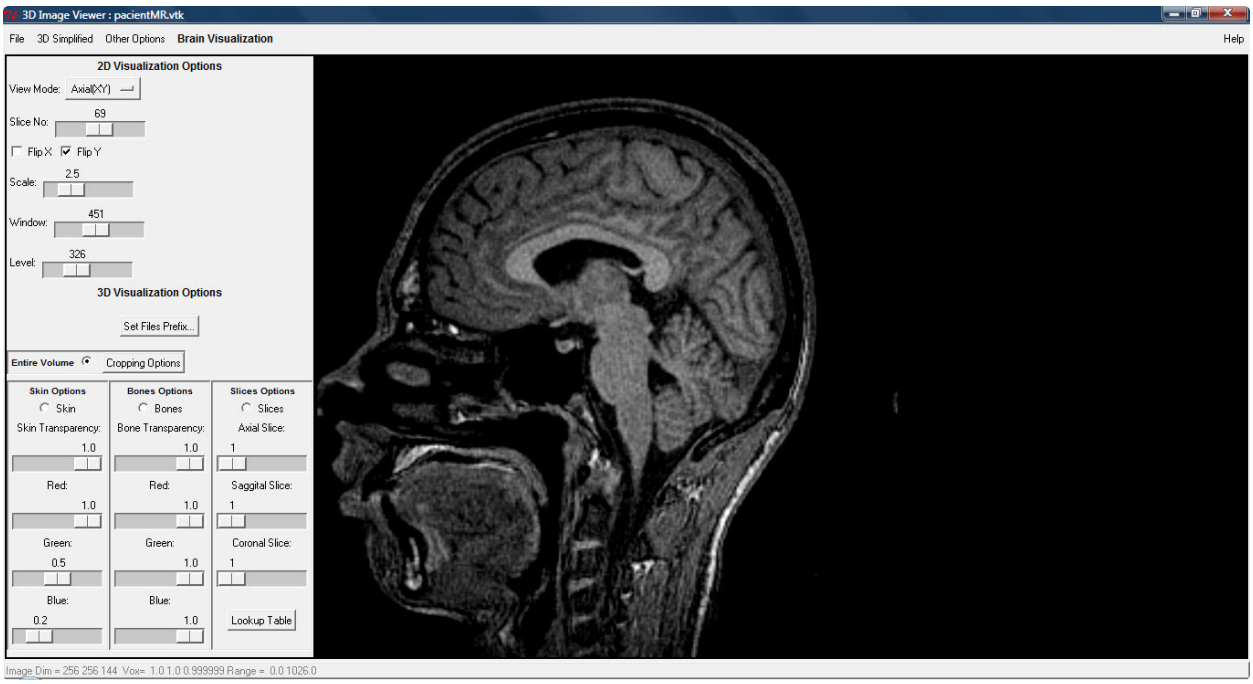


Figure 5-21 The sagittal slice (but axial, due to the acquisition mode) number 69 of a MRI dataset, flipped around Y, scaled at 2.5, with a window of 451 and a level of 326.

5.2.2. Isosurface Extraction

Now that we were able to visualize all the slices from the CT and MRI datasets, we are interested in converting datasets in more appropriate information. The research in isocontour extraction arose in the medical imaging field, with the use of sampled medical images to extract boundaries of organs. In [W.E.Lorensen et al., 1987] was firstly introduced the Marching Cubes algorithm also described in the **Chapter 3**, and it is still used and implemented in VTK, adding some techniques to enhance the visualization if the isosurface.

One application of the isosurface extraction is the neurosurgery, the main domain analyzed in this thesis. Neurosurgery requires high geometrical accuracy since severe functional impairment may result if a structure is injured due to a minor positioning error. The use of a stereotactic frame to establish an accurate and stable coordinate system for the surgical field it is a common practice. In order to define a plan of action surgical planning, the target must be defined and the safest possible approach to the lesion that causes the least possible damage to healthy anatomic structures. The planning has to be based on precise data of different medical imaging modalities that deliver complementary information and that have to be fused: CT for bone structures, and MRI for normal gray and white matter anatomy. In addition, computer based brain atlases are used.

Finally, to utilize the precise quantitative information that imaging databases and surgery planning provide, sophisticated techniques have to be applied during surgery: stereotactic systems, ultrasound and Doppler imaging for accurate localization during neurosurgical operations, robotic instruments for precise microsurgery, and several methods such as electroencephalography (EEG) for intraoperative monitoring.

In the **Figure 5-22** the skin isosurface is extracted, using the Marching Cubes approach. We can also observe the outline around the data, which provides context for the isosurface.

In [W.Schroeder, et al., 2002] is described that a density value of 500 will define the air/skin boundary, and a value of 1150 will define the soft tissue/bone boundary. As we have discussed in the **Chapter 4**, two different ways for reading the images were took into account. As we can observe in the figure, there exists a visible difference between the two methods, even

if the same value of the density, and the same properties for the isosurface were added. This difference is due to the header files that are included in the Dicom files. Even if we are using *.dcm* files or *.vtk* files, the result will be the same like in the left side of the figure. This arises because the 16-bit pixel of the image is used to mark connectivity between voxels, and the isosurface extraction is not very accurate.

But, if we are using *.raw* files, meaning that we are "cleaning" the header, padding those bits with zeroes, the resulting isocontour will be the one illustrated in the right side of the figure.

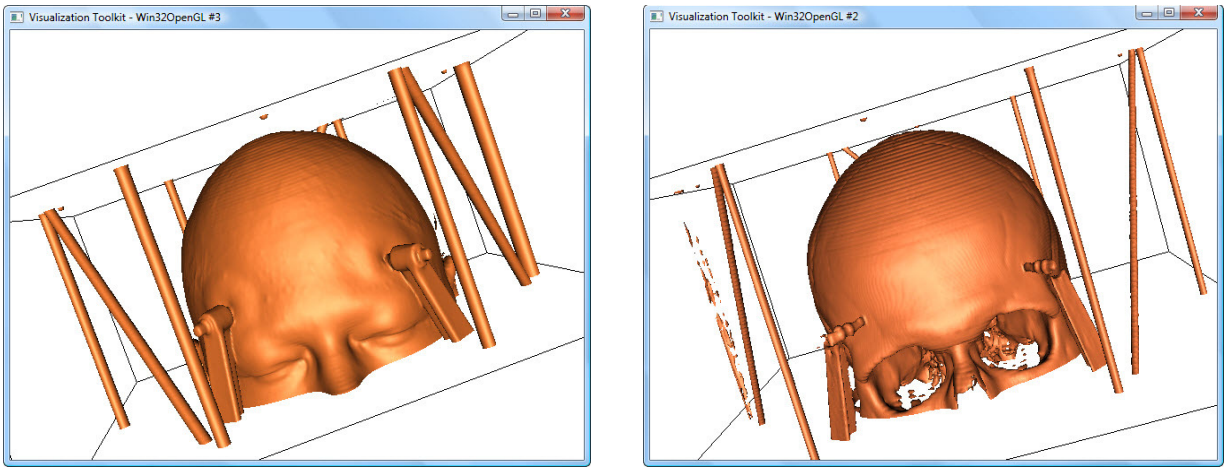


Figure 5-22 The skin isocontour extraction from a CT dataset (left: using *.raw* images; right: using .vtk or Dicom images).

In the **Figure 5-23** the density values for the isosurface extraction was 100, instead of 500. We can observe that the isosurface is comparable with the one from the Figure 5-22, but some artifacts also appear, visible especially on the eyelids. So, even if we adjust the density value, the isosurface extraction is not accurate, the best way for extracting the isocontour from a 16-bit data image remaining when we are using *.raw* files.

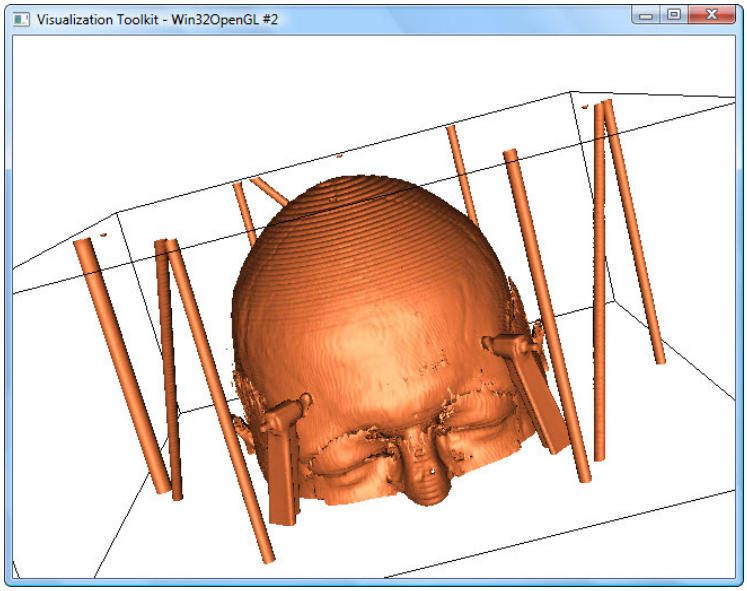


Figure 5-23 The skin isocontour extraction from a CT dataset using *.vtk* or Dicom files. In this case the value for the contour is 100.

If we are setting the density value at 1150, the bones are extracted from the dataset. The two reading methods were considered, and the difference between them is presented in the **Figure 5-24.** We can observe that in the picture from the right side of the figure, very small bone

portions are detected. Any value we choose for the bone isosurface density, the result will remain the same; we cannot obtain, like in the **Figure 5-23**, a better result if we change this value. In the left side of the figure the bone appears clearly, without any confusion.

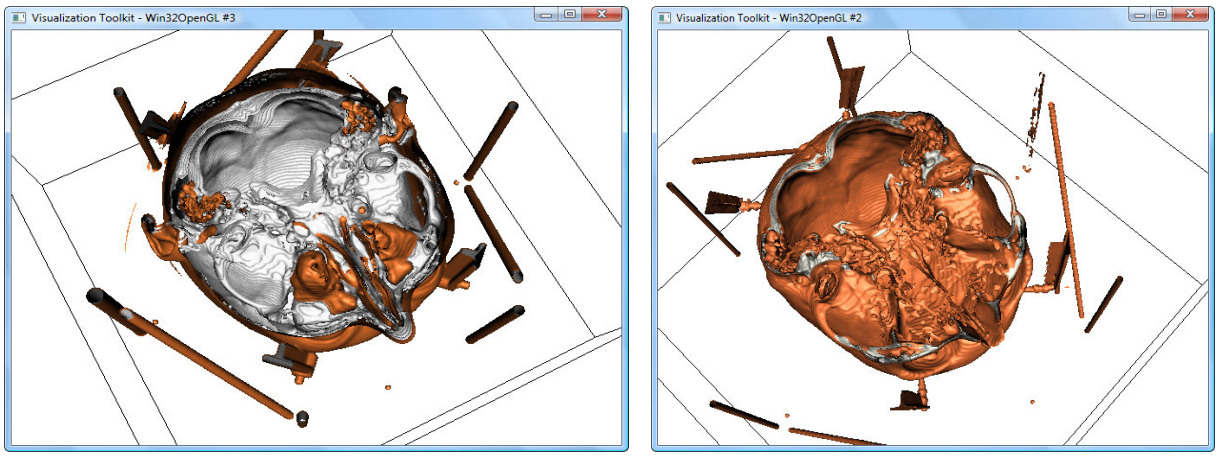


Figure 5-24 The skin and bones isocontours extraction from a CT dataset (left: using *.raw* images; right: using .vtk or Dicom images).

The transparency of the isosurface can be manipulated using the *SetOpacity* method from the *vtkActor* class. This property is useful when we want to see the skin and bones in the same time, to correlate them. In neuroscience it is useful if we can correlate the skin with the brain structure, but it is difficult to record images exactly for this purpose, because some special techniques are needed in order to observe the skin and the brain in the same time (we need a technique that is between CT and MRI, or we must process the images specially for this purpose). For now, we visualize only the skin and the bones simultaneously in the **Figure 5-24** (left). The bone isosurface is the most visible in this case, the skin having the opacity of 0.3 from a scale from 0 to 1. In the right side of the figure, the bone's opacity is 0, so we cannot see it and the skin opacity has the value 0.7, so we can slightly see through it.

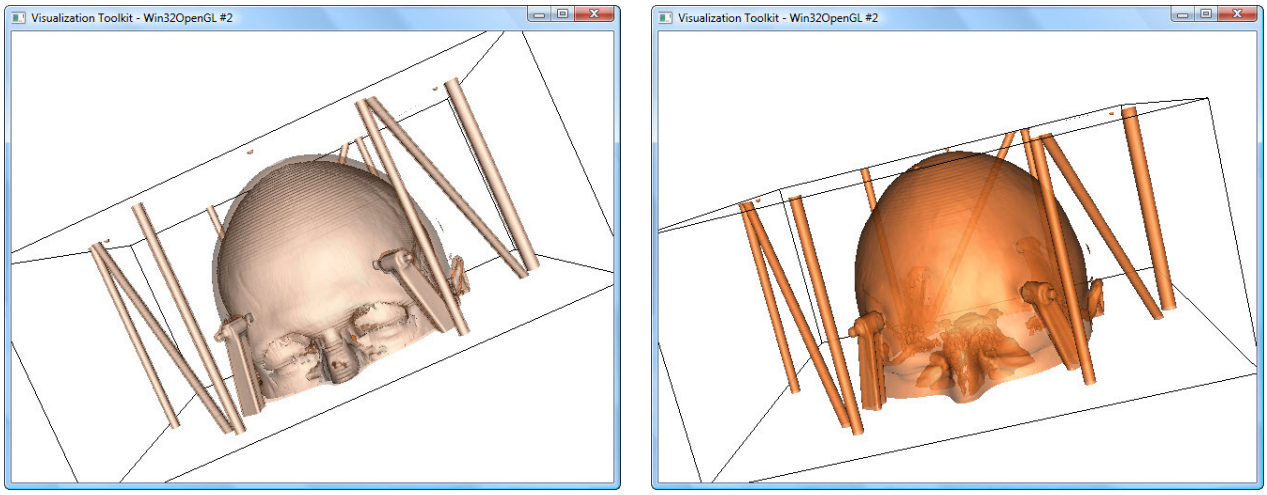


Figure 5-25 The skin and bones isocontours extraction from a CT dataset (left: bones and skin, skin opacity of 0.3; right: skin, skin transparency of 0.7).

In the **Figure 5-26** the axial slice number 2 can be observed in parallel with the skin and bones isosurfaces. We can observe how the bones pierce the slice. If we want to see another slice, for example the axial slice number 40, the tissue will cover the slice and we will not be able to see the orthogonal plane. In this situation, the skin and bone isosurfaces are modified, as

we can see in the **Figure 5-27**. Moreover, the saturation, the hue, and the value ranges of the orthogonal planes can be adjusted using the interactive look-up table provided in the interface. In the **Figure 5-26** the hue is 0, the saturation is 0, and the value is 1, so a grey scale slice will be obtained.

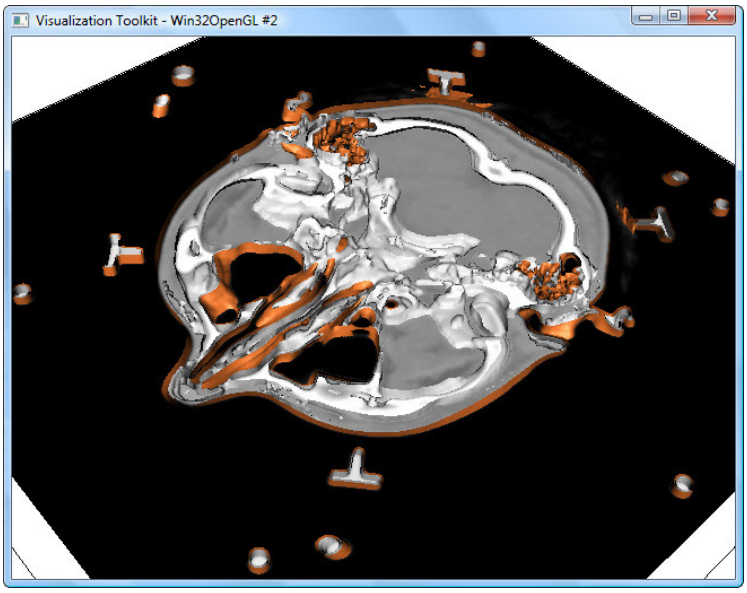


Figure 5-26 The axial orthogonal plane visualization of slice 3 in the volume

The situation presented in the **Figure 5-27** is useful if we want to exactly localize the slices in the volume (in this case the axial slice number 46 is visualized in different ways). In the picture from the left side we can see the bones and the skin, with adjusted opacity, and in the picture from the right side only the skin (with a certain transparency) is visualized, but the values provided in the lookup table are changed.

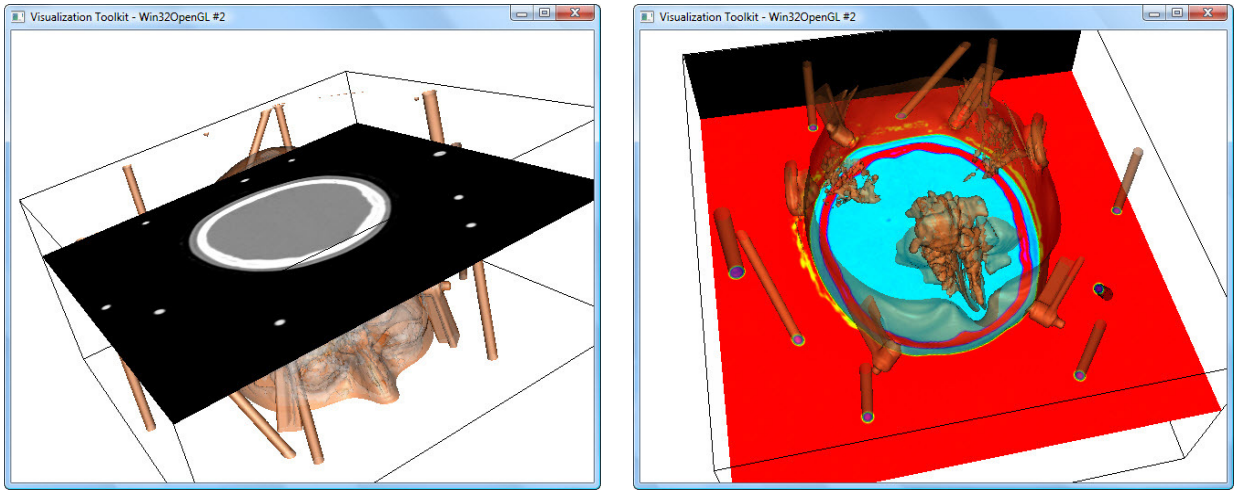


Figure 5-27 Different modes for volume visualization including the orthogonal planes

We can provide more context of the volume if we are able to see all the orthogonal planes (axial, coronal and sagittal), and to visualize the isosurface between certain slices. For example, in the **Figure 5-28** the axial (slice number 0), the coronal (slice number 215), and the sagittal (slice number 114) orthogonal planes are visualized together with the skin isosurface of 0.5 value opacity. This visualization mode helps in visualizing just a part the volume, the one generated between the selected slices.

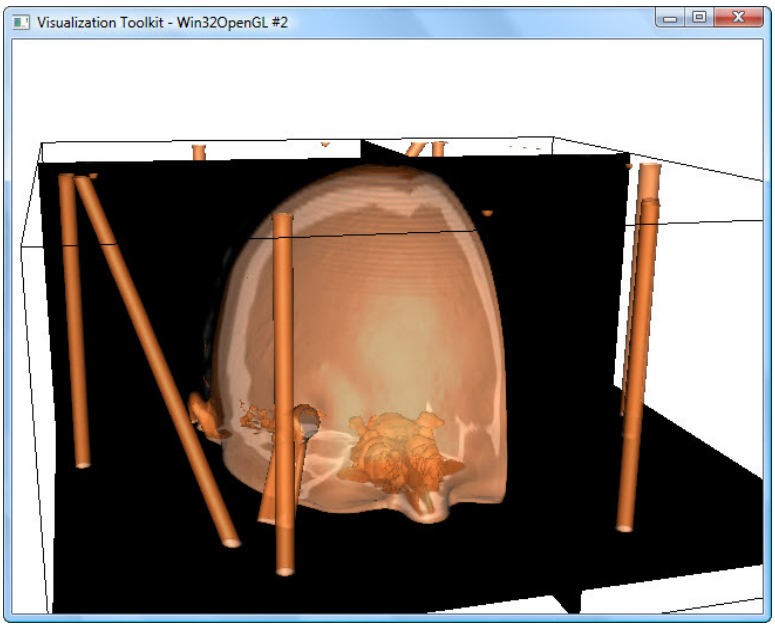


Figure 5-28 Visualization of the orthogonal planes and the skin isosurface

In some situations we are interested only in how the slices are distributed in the volume, such the cases presented in the **Figure-29**. Here we can see the importance of setting different colors for the orthogonal planes, because we can easily distinguish between them. In the left side of the figure, the grey scale images are displayed using the three planes, without any adjustment. We can hardly distinguish between them, but in some certain situations, this can be useful. On the other hand, in the right side of the figure different values for hue, saturation and value were selected for each orthogonal plane in such a way we can easily distinguish between them. The importance of such a view mode increase when we want to visualize certain brain structures, because we can observe the continuity between them.

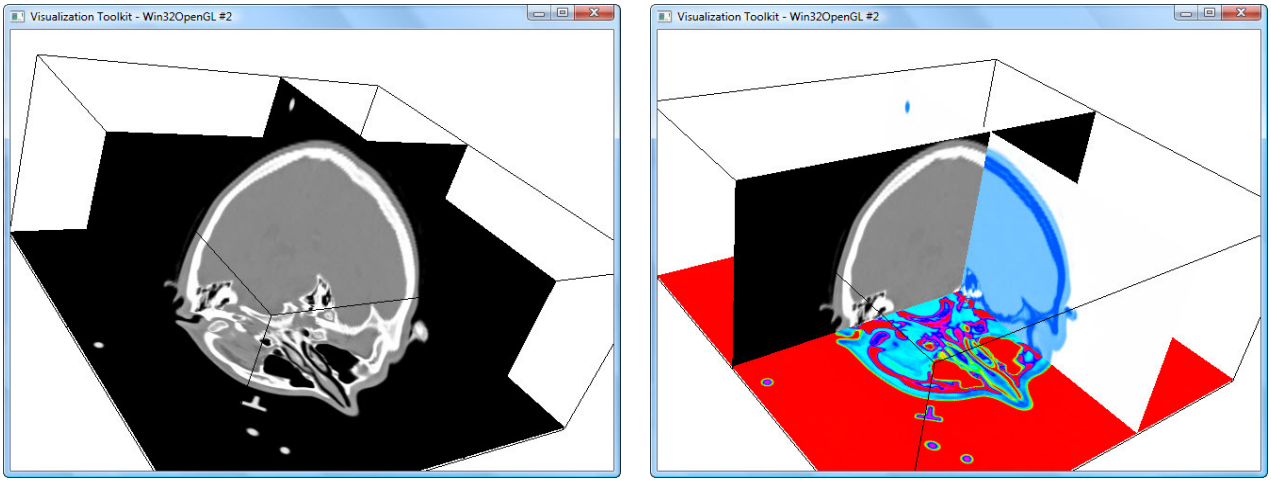


Figure 5-29 Axial, coronal and sagittal planes visualization

5.2.3. Direct Volume Visualization

Volume visualization is a method of extracting meaningful information from volumetric data using interactive graphics and imaging. The volume is visualized for diagnostic purposes or for planning of treatment or surgery. Volume rendering conveys more information than surface-rendered images, but at the cost of increased algorithm complexity and consequently increased rendering times. In the **Chapter 3**, the Ray Casting technique for direct volume visualization was described as the best solution for our situation, and in the **Chapter 4**, was presented the mode in which this technique can be implemented. Different properties were added to the rendered volume in order to make it more natural.

The normal visualization mode of the head visualization is presented in the **Figure 5-30**. Additionally from the basic visualization using Ray Casting technique, the color transfer function provided by VTK was used. This transfer function has the role to map the voxel intensities to colors. In this way, three colors can be observed: one color for skin (setting a color transfer function value between 0 and 500), one color for flesh (with the color transfer function between 500 and 1000), and another color for bone (the value of 1150 and over). We can also observe some differences between tissue types. This is done by adding a scalar opacity to the entire volume. Besides scalar opacity, the gradient opacity function was used to decrease the opacity in the "flat" regions of the volume while maintaining the opacity at the boundaries between tissue types. The gradient is seen as the amount by which the intensity changes over unit distance. In this case, the tri-linear interpolation mode was used for volume rendering. Considering this additional properties, the rendered volume can be seen in the **Figure 5-30**.

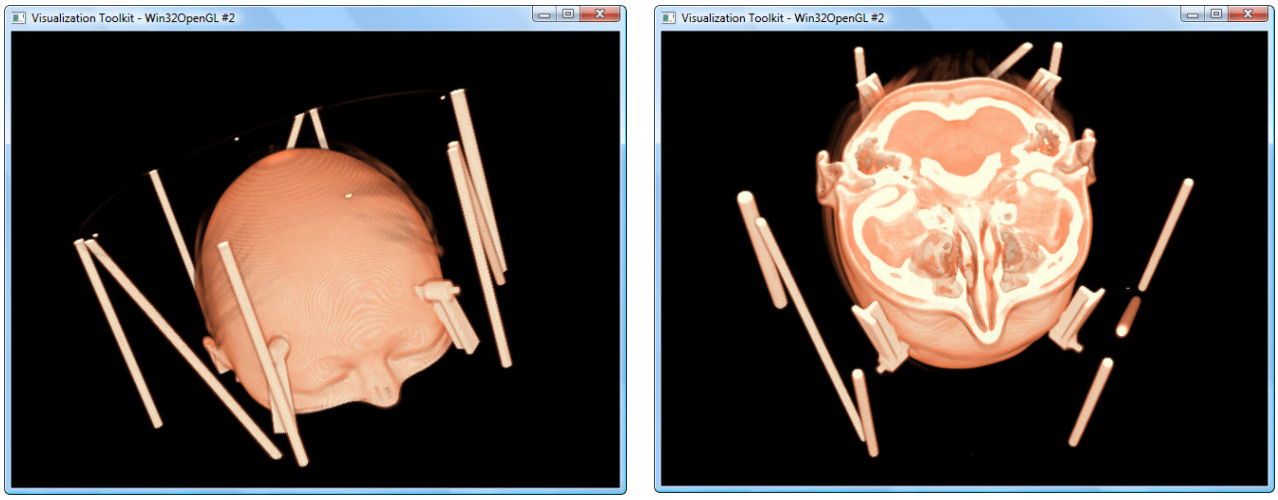


Figure 5-30 Direct volume visualization using scalar opacity and gradient opacity

A more natural visualization mode can be reached if we use the methods included in the *vtkVolumeProperty* class. If we use the *ShadeOn* option, we can turn on the directional lighting, which will enhance the appearance of the volume, and make it look more "3D". If we additionally set just this option, the volume will look something like in the right image of the **Figure 5-30**. Now, using an Ambient coefficient of 0.4, a Specular coefficient of 0.2 and a Diffuse coefficient of 0.6, we can decrease the impact of the shading. The final volume is obtained in the **Figure 5-31** (the left screenshot). This is the most natural way of visualizing such a volume.

One thing must be clarified regarding the right part of the **Figure 5-31**. This image was obtained using exactly the same properties of the volume obtained in the left part, but using another version of VTK, a newer one. For the image from the left side, the VTK 5.4 was used and for the image from the right side, the VTK 5.6 was used. We can easily observe that there is a huge difference between them. For now, we cannot make anything regarding this problem,

maybe just predicting that the VTK 5.6 version has some problems with the methods implemented in the *vtkVolumeProperty* class, and signaling the authors of this class about this problem (they were already announced about the problem, in the VTK users list).

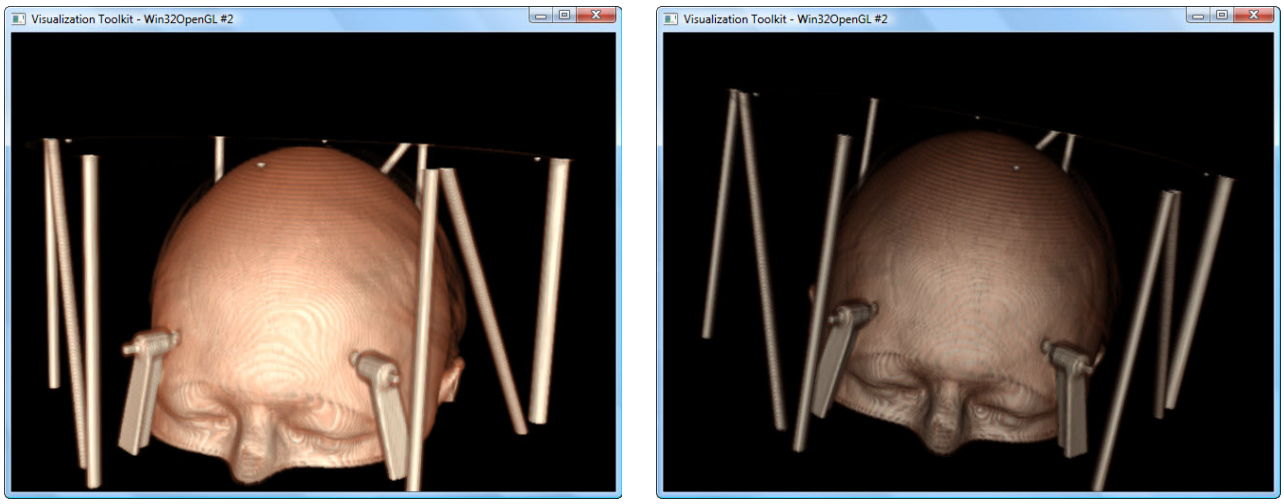


Figure 5-31 Direct volume visualization (left: the CT dataset rendered using VTK 5.6; right: the same CT dataset rendered using VTK 5.4)

And now, back to the beginnings. The **Figure 5-32** illustrates the simplest view that can be obtained using the Ray Casting technique. In the screenshot from the left side, there was used the same dataset as the one used for the previous visualization techniques, but without any additional property added. As we observe, it is very hard to distinguish that there is the head of the patient. Moreover, we can easily distinguish the stereotactic frame that was attached during the surgery, because no color transfer function was added. Actually, this screenshot has the role to illustrate once again, the drawbacks of using .vtk files for large data sets. Not only the fact that it has not any form, but if we try to add some transfer functions, we cannot interact with the volume anymore, because the rendering process is really slow. Consequently, using *.vtk* files including large amount of data, is the worst idea ever. Anyway, this aspect is eliminated if the data is below 5 Megabytes. In this case, it can work perfectly. As an example, if we segment a MR dataset considering the existing brain structures, and every structure of the brain is the saved in a *.vtk* file, the rendering process will be faster. Also, we can decrease the resolution of the dataset, if we consider that the volume extraction will not be affected, and we if we process the files into a *.vtk* file or we simply read them as Dicom files we may not have problems with the rendering process.

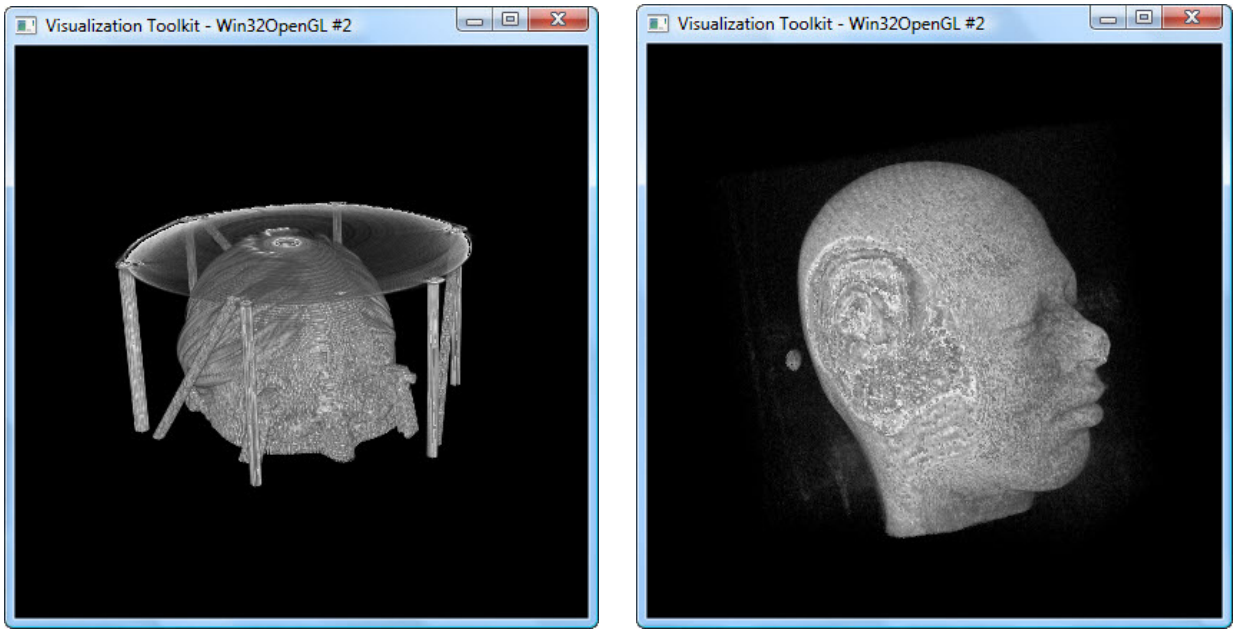


Figure 5-32 The basic direct volume visualization without any property added (left: volume obtained from a CT dataset; right: volume obtained from a MR data set).

As we can see in the **Figure 5-33**, other features can be added once we obtained the volume. The cropping option is one of the most used techniques in medical volume processing, because the surgeons often need to explore into the volume, to extract some information that can be used in taking the decisions regarding the treatments or the affected area. This is the basic way to cut the volume, because is done using orthogonal planes, but in the most situations is enough to have only this option. For example, in the Deep Brain Stimulation (DBS) technique, briefly described in the **Chapter 2,** only a very small part of the skull must be eliminated. Of course, the cutting using the orthogonal planes is not useful in this situation, because we must have the flexibility to adjust the cutting angle, but starting from this technique, we may add the cutting at a different angle feature. Moreover, if we consider this example, it is not necessary to have the possibility to cut using the sagittal and coronal orthogonal planes, because in this case the patient will lose the cerebral fluid, so the brain will be affected.

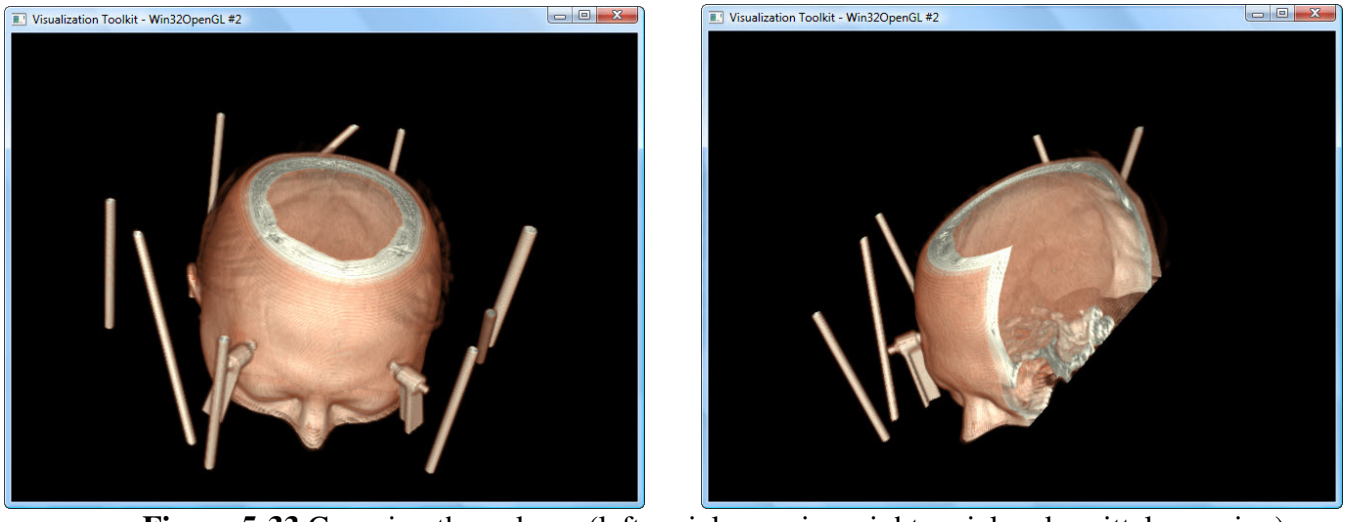


Figure 5-33 Cropping the volume (left: axial cropping; right: axial and sagittal cropping)

The **Figure 5-34** was selected in order to compare the nearest-neighbor rendering method with the tri-linear one. Because of the high resolution of the dataset, the differences between them are not so visible, but for a low-resolution dataset, the differences must be taken into account. If we are looking at the eyebrows, we can easily observe the differences between them. Since nearest-neighbor interpolation is simply voxel copying, and not really interpolation at all, we can see discontinuities between different regions on the subject face. Also, we can observe that if we are using tri-linear interpolation, these discontinuities disappear. In this situation, the nearest-neighbor method does not produce such large discontinuities, and can be considered because it will considerably increase the rendering speed and the interaction response.

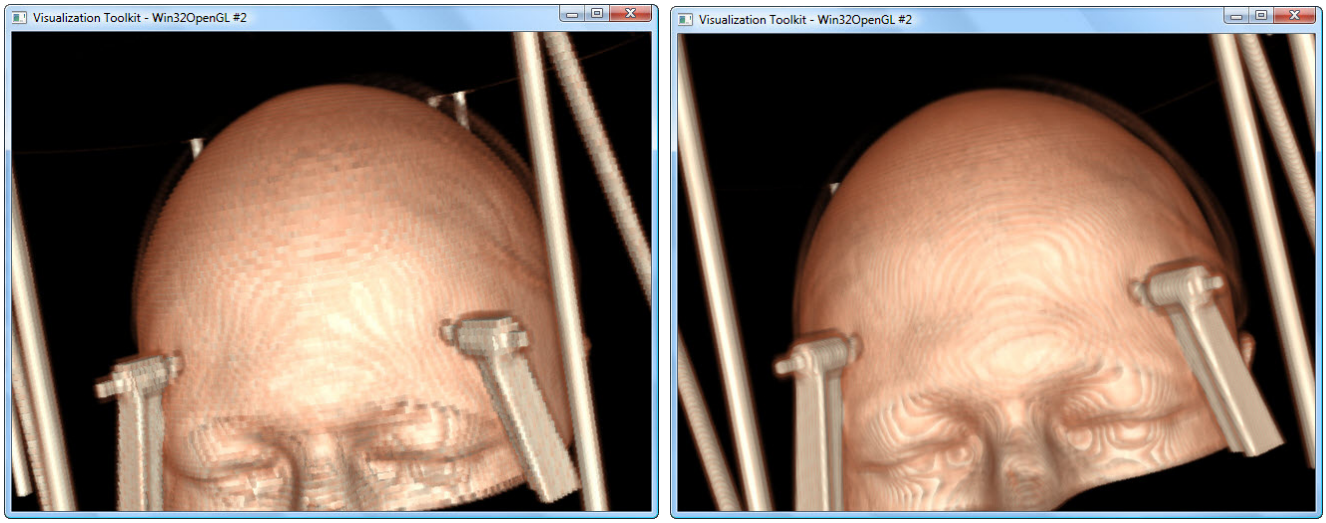


Figure 5-34 Comparison between nearest-neighbor (left) rendering and tri-linear (right) rendering.

Another way, in which the rendering process can be speeded up, is to change the distance between the slices. For the CT study case, the distance between slices was 2 mm. If we double this distance, the volume from the **Figure 5-35** will result. Of course, this example was considered in order to see the effect of changing the distance between slices, but if we consider a value of 2.5 mm, for example, this will also help in obtaining a faster volume rendering process.

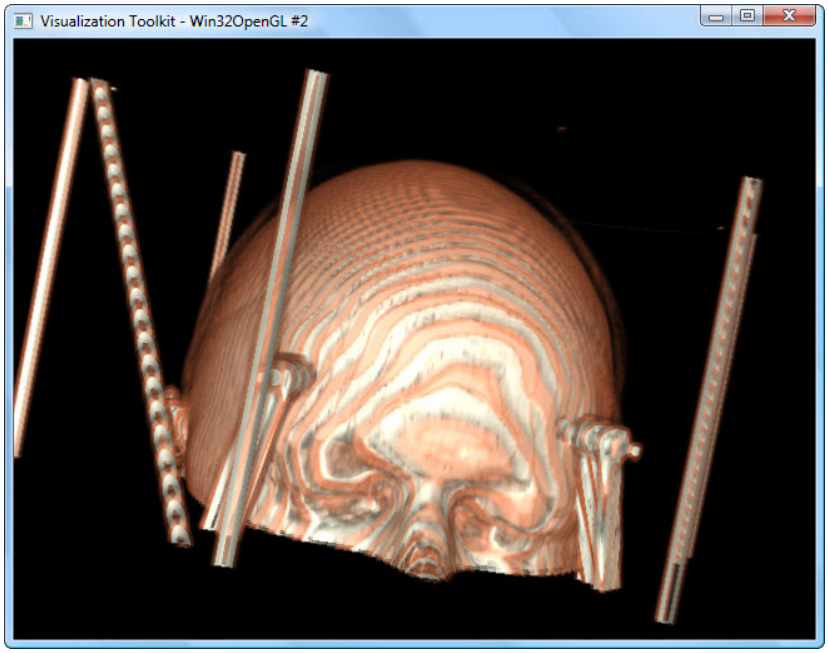


Figure 5-35 Changing the distance between the slices to 4 mm.

In the section dedicated for 2D visualization the main differences between CT and MR slices were presented, and we also observed that the MR is not capable in extracting the bone or the skin regions. The theoretical explanation of this fact is described in the **Chapter 3**. In the **Figure 5-36** we can visualize what actually happens when we try to extract the skin and the bones from such a dataset. In the image from the right side, was intended a direct volume visualization of the MR dataset considered previously. As we can observe, it is very hard to distinguish the volume. Unexpected, a better view we can see in the **Figure 5-32** (right), where the simple Ray Casting method was considered. The volume considered in the left side of the figure can be explained if we are looking at the right side of the figure, where a skin extraction was intended to be done, without success. We can hardly detect where the eyes are situated in this image, but nothing more. The situation presented in this image can also be explained if we are looking at the 2D visualization of a MR slice **(Figure 5-21)** where even if we tried to achieve the best quality of the slice (setting the window and level parameters), we were not able to distinguish the bones (the brightest sections in the slice). So, the result from the right side of the **Figure 5-36** was predictable.

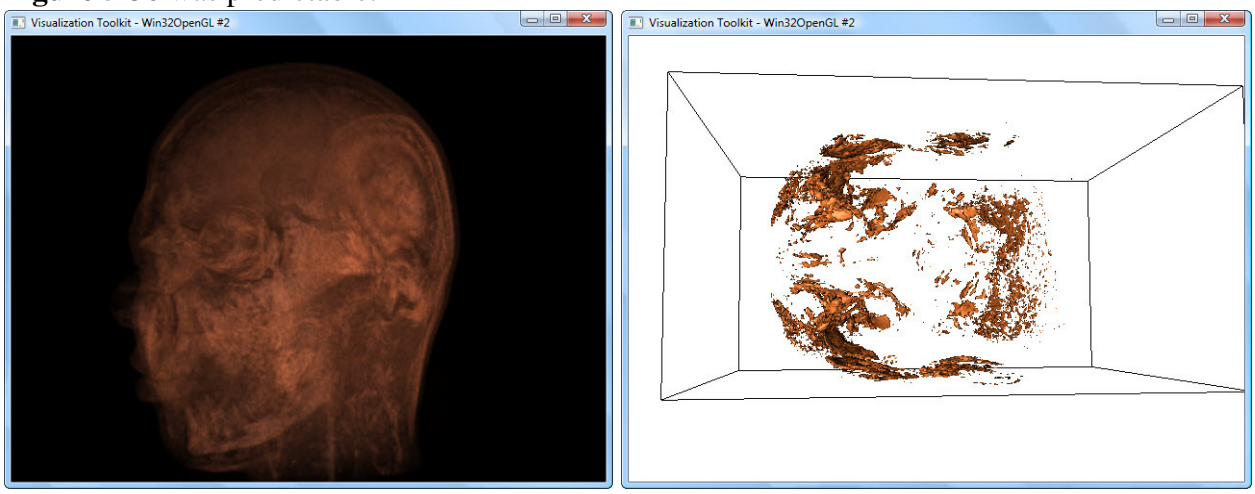


Figure 5-36 Left: Direct volume visualization using a MRI dataset. Right: Skin isosurface extraction from a MRI dataset.

Chapter 6. Conclusions and Perspectives

The scope of this paper was orientated in two main directions. The first direction had the objective to develop algorithms for modelling neural signals, while the second direction followed the understanding of the basic visualization algorithms, and the techniques used for medical imaging visualization. However, the paper attempts to highlight the dependence between the neural signal processing and visualization of medical images when a treatment has to be established, or the affected brain structure has to be localized within the entire brain.

Based on the existing knowledge in the field of digital signal processing, one of the paper's aims was to create fundamental real time signal processing algorithms that can work with the neural signals. For this, different types of neural signals were analyzed, but the local field potential (LFP) was the one most considered in this paper. This signal is a summation of the activity of several brain cells, and it is considered by neurologists the most useful one for taking decisions in different fields of neuroscience. How the useful information can be extracted from these signals, is still an active research area, because until now the research focused more on how the LFP signals can be extracted using different electronic circuits. So, all the signal processing techniques were electronic based, more that software based. This paper analyzed this aspect and the signal processing instruments already used for LFP processing. It also provides a solution for creating common filtering techniques, like: low pass, high pass, band-pass, band-stop filtering, which can be further be applied to the real-time recorded LFP signals. In the implementation of filters, the Fast Fourier Transform was used to extract the frequency spectrum of the input signals, and the convolution between the input signal and the kernel of the filtered has been done.

In the second section of the research different 2D and 3D visualization algorithms were tested in order to see which is the best solution for medical image visualization, and how we can extract the useful information from these images. Because the paper considered multiples ways for neural signal processing and for the visualization of medical images, it should be noticed that not all directions investigated for the final implementation, lead to viable and visible results. Maybe the best example in this way is the one in which the segmented data from a 3D brain atlas was read. The scope of this task was to see what VTK class need to be used in such a situation, and to understand how ParaView software tool loads 3D segmented brain atlases in the renderer window.

The study regarding the neural signals can be deepened by considering more neural signals, and by researching how they can provide useful information about the activity in the brain. In order to extract this useful information, new signal processing techniques may be considered. Also, a GUI must be developed to include both signal processing and visualization techniques aspects. The synchronization between these two main sections is the hardest part that must be done if we want to develop such an instrument, because we must very accurately align all the recorded images with the signals recorded within the brain.

In the medical imaging visualization field, faster ways to render the generated isosurface or volume can be researched, one solution being to use the VTK library in its original programming medium (C++), because the wrapping process may slow down the rendering and visualization processes if we are working with huge amount of data. Furthermore, we can explore some image segmentation techniques (some algorithms are already developed in the ITK library), especially if we decide to work with two-dimensional brain atlases. There already exist some software tools that allow manually segmentation and volume reconstruction using two-dimensional images, but most of them do not include options for filtering and processing the volumes.

Considering the thesis as a start, a 3D visualization and database software for stereotaxic neurosurgical navigation in non-human primate research can be done. Such systems already exist

and are in continuous competition, because of the features they offer. The role of stereotaxic navigation is to make recording experiments on non-human primates in order to evaluate the neural activity present in a certain region of the brain. This is a way of doing research in the neuroscience domain, and it is very helpful especially when new treatment methods need to be tried, especially in the treatment of other disorders treated with DBS, such as dystonia, depression and Tourette syndrome. Usually, several days are consumed in neurophysiological mapping of the target brain region, in order to find the exactly location of a certain structure within the brain. If we are considering these types of neurosurgical experiments, it is important to be able to understand how the activity of the neural cells can be measured on the non-human primates' brain and to be able to understand how this information will be useful for developing a certain treatment in an injured human brain. For designing such a visualization system, we need to find out which are the main aspects considered by the neurologists in developing such an instrument. Also, we may meet complement and contrasting methods considering this approach. Anyway, the goal will be to integrate all the complementing and contrasting results from these methods to develop a comprehensive understanding of DBS mechanisms.

Another interesting and very actual research area that may be taken into account starting from the fundamental bases that were achieved by now is the *neuromodelling* domain. It is dependent on what we have discussed so far and its role is to understand the fundamental relationship between neuron structure and function, to analyze their geometric shapes and spatial interrelationships. Currently, there exists a lot of modeling and simulations of the electrical properties of neurons, reflecting numerous simplifying assumptions about the structure of neurons. In the most studies that were taken in this direction, the main problem is that is very hard to analyze the small structures of neurons, such as the dendrites that are an important component of a neuron. Anyway, it is a challenge to bring new reconstruction and computational tools for neurons models, and to create a multiscale and spatially realistic electrical simulations of neuronal activity. The most analyzed component of a neuron are considered the dendrites, so if the next perspective will be the neuromodelling field, we must understand how the dendrites are able to convey information from one neural cell to the other, and also different standard models of neurons must be considered. Some aspects regarding the neuromodelling field were discussed in the theoretical part of this paper, and they must be taken into account when we deal with a certain type of neural signal.

Concluding, this work may be extended for a master thesis which may have the objective to create a software tool that serves in DBS surgery planning, including both neural signal analysis and medical imaging. Starting with DBS surgery technique, it can be extended then for working with adjacent techniques. Also, the experience in the neurology domain obtained from this research can be used to develop some reconstruction and computational tools for neurons. Furthermore, we can go deeper and analyze some aspects regarding artificial neural networks in order to gain an understanding of the biological neural networks (made up of real biological neurons). Finally, in [C.R. Johnson, C.D. Hansen, 2005] the visualization was considered an intelligence amplifier (IA), providing support to human intelligence, and was complementary presented with the artificial intelligence (AI), which was seen as a "human replacement" technique. We can try to find a way in creating a software instrument which will either contain AI and IA aspects. This will be a great support in the visualization of different structures and neurons within brain and also taking some decisions regarding the signals that can be recorded from those sections or neurons. For sure, this aspect is deeply researched from mathematicians, programmers and biochemists from the entire world.

References

[C.R. Johnson, C.D. Hansen, 2005]
C.R. Johnson, C.D. Hansen – *The Visualization Handbook*. Elsevier Inc., 2005

[R. Sherman, 1993]
R. Sherman-Gold – *The Axon Guide for Electrophysiology & Biophysics Laboratory Techniques*. Axon Instruments, Inc, 1993.

[S.Barre, 2003]
S. Barre. *dicom2 application*. Download webpage: http://www.barre.nom.fr/medical/dicom2/

[M.Gordan, 2011]
M. Gordan – *Digital Image Processing*- courses, *Chapter6. Image enhancement*. Technical University of Cluj-Napoca, 2011.

[Halle M. et al., 2011]
Halle M. et al – *Multi-modality MRI-based Atlas of the Brain*. NIH's National Center for Research Resource (NCRR), SPR, Apr-2011.

[InfoVis, 1995]
The First Information Visualization Symposium. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[E.R.Kandel et al., 2000]
E. R. Kandel, J. H. Schwartz, T. M. Jessel. *Principles of Neural Science*. McGraw-Hill, New York, 2000.

[W.E.Lorensen et al., 1987]
W.E. Lorensen and H. E. Cline. *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*. Computer Graphics. 21(3):163-169, July 1987.

[B.H.McCormick et al., 1987]
B. H. McCormick, T.A. DeFanti, and M.D. Brown – *Visualization in Scientific Computing*. Report of the NSF Advisory Panel on Graphics, Image Processing and Workstations, 1987.

[B.Preim, 2007]
B. Preim, D. Bartz – *Visualization in Medicine. Theory, Algorithms, and Applications*. Elsevier Inc., 2007.

[K.P. Purpura, 2008]
K. P. Purpura et al. *Neural Signal Processing: Tutorial 1*. Cold Spring Harbor Laboratory, New York.

[L. de Soras, 2005]
L. de Soras – *FFTReal C++ and Delphi library*.

[P. Brown et al., 2004]
P. Brown et al. – *Brain State-Dependency of Coherent Oscillatory Activity in the Cerebral Cortex and Basal Ganglia of the Rat*. J. Neurophysiol 92: 2122-2136, 2004.

[L.Rosenblum et al., 1994]
L. Rosenblum et al. *Scientific Visualization Advances and Challenges*. Harcourt Brace & Company, London, 1994.

[N. Wisniewski, 2006]
N. Wisniewski. *Local Field Potential. A Modeling Study.* Computation and Neural Systems, California Institute of Technology, Pasadena, CA 90026.

[P. Andrews, et al.]
P.Andrews et al. *The Chronux Manual*. August, 16, 2008, www.chronux.org.

[S. Zambal]
S. Zambal. *Implementation of the Shear-Warp Algorithm*.
(http://www.cg.tuwien.ac.at/courses/projekte/vis/finished/SZambal)

[S.W.Smith, 1997]
S.W.Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing. ISBN 0-0660176-3-3 (1997).

[W.Schroeder, et al., 2002]
W. Schroeder, K. Martin, B. Lorensen– *The Visualization Toolkit (3rd edition): An onject-Oriented Approach to 3D Graphics.* Kitware, Inc., New Jersey, Pearson Education Inc., 2003, ISBN: 1-930934-07-6.

[S.Svensson]
S. Svensson, K.Vrotsou – *vtkGUI. Development of an open source graphical user interface for the Visualization Toolkit.* http://vtkgui.cineca.it.

[W. Schroeder, 1998]
W. Schroeder. *The VTK User's Guide*. Kitware Inc., 1998, ISBN: 1-930934-06-8.

[Y. Perelman, R.Ginosar, 2007]
Y. Perelman, R. Ginosar – *An Integrated System for Multichannel Neuronal Recording With Spike/LFP Separation, Integrated A/D Conversion and Threshold Detection*. IEEE Transaction on Biomedical Engineering, vol.54, no.1, January 2007.

[B.B. Welch,1999] - *Practical Programming in Tcl & Tk, Third Edition*. Prentice Hall PTR, 1999.

[M.F.Vaida et al. 2007]
L. Chiorean, Mircea F. Vaida, Petre G. Pop, C. Striletchi - *Elemente de baza si obiectuale privind dezvoltarea aplicatiilor in limbajul de programare C/C++*, Editura U.T. Press, 2007, pp. 380, ISBN 978-973-662-406-3

Appendix A

Appendix A: The C++ classes developed for creating the filtering algorithms

A.1 Source code of the Band-Pass Windowed-Sinc Filter Class

```
//Description:
//The constructor for the BPFilter1(BAND-PASS WINDOWED-SINC FILTER) class.
//Parameters:
//     KernelLen - The length of the kernel , it must be an ODD NUMBER!!!
//     SamplingRate - The sampling rate of the input signal that will pass
//                                        through this filter
//     PassBandFreq - the first frequency corner, under this level the frequencies are cut
//                    and the higher frequencies pass
//     StopBandFreq - the second frequency corner, below this level frequencies pass
//                    and above this level the frequencies are cut
//     Window       - The window function used with this type of filters.
//

CBPFilter1::CBPFilter1(const int nSamplingRate,
                       const int nKernelLen,
                       const int nPassBandFreq,
                       const int nStopBandFreq,
                       CSWindow *Window,
                       CONVOLUTION_TYPE ConvolutionType /* = FREQUENCY_BASED*/ ,
                       int   nFFTLen   /*=   0   */):CSFilter(nSamplingRate,   nKernelLen,
             ConvolutionType, nFFTLen)

{

CSFilter  *LowPassFilter  =  new  CLPFilter1(nSamplingRate,  nKernelLen,  nStopBandFreq,
Window, TIME_BASED, nFFTLen);
CSFilter  *HighPassFilter  =  new  CHPFilter1(nSamplingRate,  nKernelLen,  nPassBandFreq,
Window, TIME_BASED, nFFTLen);

DataType* lpLowPassFilter = LowPassFilter->getKernel();
DataType* lpHighPassFilter= HighPassFilter->getKernel();

m_nPassBandFreq = nPassBandFreq;
m_nStopBandFreq = nStopBandFreq;

//the two kernels are added and spectral inversion is applied
for( int i = nKernelLen - 1;  i >= 0 ; i-- )
       m_vdtKernel[ i ] = 0 - ( lpHighPassFilter[ i ] + lpLowPassFilter[ i ] );

       m_vdtKernel[ nKernelLen / 2 ] = m_vdtKernel[ nKernelLen / 2 ] + 1;

       if (ConvolutionType == FREQUENCY_BASED)
       {
              for (int i= m_nKernelLen; i<m_nFFTLen; i++ )
                     m_vdtKernel[ i ] = 0.0;
                     m_RealFFTObj->do_fft( m_vdtFreqResp, m_vdtKernel );
       }

       delete LowPassFilter;
       delete HighPassFilter;
}
```

A.2 Source code of the Band-Stop Windowed-Sinc Filter Class

```
/*
Description:
	The constructor for the BSFilter1(BAND-STOP WINDOWED-SINC FILTER) class.
	Parameters:
		KernelLen    - The length of the kernel , it must be an ODD NUMBER!!!
		SamplingRate - The sampling rate of the input signal that will pass
		               through this filter
		StopBandFreq  -  The  first  frequency  corner,  under  this  level  the
	frequencies are cut and the higher frequencies pass
		PassBandFreq  - The second frequency corner, below this level frequencies
pass and above this level the frequencies are cut
		Window        - The window function used with this type of filters.
*/

CBSFilter1::CBSFilter1    (const int nSamplingRate,
                           const int nKernelLen,
                           const int nStopBandFreq,
                           const int nPassBandFreq,
                           CSWindow *Window,
                           CONVOLUTION_TYPE ConvolutionType /* = FREQUENCY_BASED*/ ,
                                        int nFFTLen /*= 0 */)
                           :CSFilter(nSamplingRate,nKernelLen,ConvolutionType,nFFTLen)
{

CSFilter *LowPassFilter   =   new   CLPFilter1(nSamplingRate,nKernelLen,nStopBandFreq,
Window, TIME_BASED, nFFTLen);

CSFilter *HighPassFilter  =   new   CHPFilter1(nSamplingRate,nKernelLen,nPassBandFreq,
Window, TIME_BASED, nFFTLen);

	DataType* lpLowPassFilter = LowPassFilter->getKernel();
	DataType* lpHighPassFilter= HighPassFilter->getKernel();

	m_nPassBandFreq = nPassBandFreq;
	m_nStopBandFreq = nStopBandFreq;

	//the two kernels are added and spectral inversion is applied
	for( int i = nKernelLen - 1;  i >= 0 ; i-- )
		m_vdtKernel[ i ] = lpHighPassFilter[ i ] + lpLowPassFilter[ i ] ;

		if (ConvolutionType == FREQUENCY_BASED)
		{
			for (int i= m_nKernelLen; i<m_nFFTLen; i++ )
				m_vdtKernel[ i ] = 0.0;

		m_RealFFTObj->do_fft( m_vdtFreqResp, m_vdtKernel );
		}

	delete LowPassFilter;
	delete HighPassFilter;
}
```

A.3 Source code of the Low-Pass Windowed-Sinc Filter Class

```
/*
	Description:
		The constructor for the LPFilter1(LOW-PASS WINDOWED-SINC FILTER) class.
	Parameters:
		nSamplingRate - The sampling rate of the input signal that will pass
                                through this filter
		nKernelLen    - The length of the kernel , it must be an ODD NUMBER!!!
		nCutoffFreq   - The cutoff frequency for the filter
		Window        - The window function used with this type of filters.
*/

CLPFilter1::CLPFilter1               ( const int nSamplingRate ,
                                       const int nKernelLen      ,
                                       const int nCutoffFreq     ,
                                       CSWindow *Window,
                                       CONVOLUTION_TYPE ConvolutionType /* = NONE*/ ,
                                       int nFFTLen /* = 0 */)
                                       :CSFilter(nSamplingRate,nKernelLen,
ConvolutionType, nFFTLen),
                                       m_nCutoffFreq(nCutoffFreq)
{

	CSFilter::DataType sum = 0.0 ;

	if( nCutoffFreq < 0 || nCutoffFreq > nSamplingRate / 2 )
			AThrowMsg("Invalid sampling rate");

	CSFilter::DataType fracCutoffFreq =((DataType)m_nCutoffFreq)/ nSamplingRate ;
	//the cutoff frequency as a fraction of the sampling rate

	for(int i = 0; i <= ( m_nKernelLen - 1 ); i++)
	{
		if (i- (( m_nKernelLen - 1 ) / 2) == 0 )
			m_vdtKernel[ i ] = 2 * M_PI * fracCutoffFreq;
		else
			m_vdtKernel[ i ] = sin( 2.0 * M_PI * fracCutoffFreq * (i - ((
nKernelLen - 1 ) / 2))) / ( i - ( m_nKernelLen - 1 )/2);
		m_vdtKernel[ i ] = m_vdtKernel[ i ] * ( Window->GetElement( i, nKernelLen
) );
	}

	//normalization of the low pass filter for unity gain at DC
	for(int i = 0; i < m_nKernelLen; i++ )
		sum = sum + m_vdtKernel[ i ];
	for(int i = 0; i < m_nKernelLen; i++ )
		m_vdtKernel[ i ] = m_vdtKernel[ i ] / sum;

	if (ConvolutionType == FREQUENCY_BASED)
	{
		for (int i= m_nKernelLen; i<m_nFFTLen; i++ )
			m_vdtKernel[ i ] = 0.0;

		m_RealFFTObj->do_fft( m_vdtFreqResp, m_vdtKernel );
	}
}
```

A.4 Source code of the High-Pass Windowed-Sinc Filter Class

```
/*
	Description:
		The constructor for the HPFilter1(HIGH-PASS WINDOWED-SINC FILTER) class.
	Parameters:
		KernelLen     - The length of the kernel , it must be an ODD NUMBER!!!
		SamplingRate - The sampling rate of the input signal that will pass
                                    through this filter
		CutoffFreq    - The cutoff frequency for the filter
		Window        - The window function used with this type of filters.

*/
CHPFilter1::CHPFilter1           (  const int nSamplingRate ,
                                    const int nKernelLen    ,
                                    const int nCutoffFreq   ,
                                    CSWindow *Window,
                                    CONVOLUTION_TYPE ConvolutionType,
                                    const int nFFTLen)
                                   :CSFilter(nSamplingRate, nKernelLen,
ConvolutionType, nFFTLen),
                                    m_nCutoffFreq(nCutoffFreq)
{
	CSFilter::DataType sum = 0.0 ;

	if(nCutoffFreq < 0 || nCutoffFreq > nSamplingRate / 2 )
		AThrowMsg("Invalid sampling rate");


	CSFilter::DataType fracCutoffFreq =( CSFilter::DataType ) nCutoffFreq /
nSamplingRate;            //the cutoff frequency as a fraction of the sampling rate

	for(int i = 0; i <= ( m_nKernelLen - 1 ); i++)
	{
		if (i- (( m_nKernelLen - 1 ) / 2) == 0 )
			m_vdtKernel[ i ] = 2 * M_PI * fracCutoffFreq;
		else
			m_vdtKernel[ i ] = sin( 2.0 * M_PI * fracCutoffFreq * (i - ((
m_nKernelLen - 1 ) / 2))) / ( i - ( m_nKernelLen - 1 )/2);

		m_vdtKernel[ i ] = m_vdtKernel[ i ] * ( Window->GetElement( i,
m_nKernelLen ) );
	}

	//normalization of the Low pass filter for unity gain at DC
	for(int i = 0; i < m_nKernelLen; i++ )
		sum = sum + m_vdtKernel[ i ];

	for(int i = 0; i < m_nKernelLen; i++ )
		m_vdtKernel[ i ] = m_vdtKernel[ i ] / sum;

	//spectral inversion of the low pass filter kernel
	//into a high pass filter kernel
	for(int i=0; i < m_nKernelLen; i++)
		m_vdtKernel[ i ] = -1.0 * m_vdtKernel[ i ];
	m_vdtKernel[ m_nKernelLen / 2 ] = m_vdtKernel[ m_nKernelLen / 2 ] + 1 ;

	if (ConvolutionType == FREQUENCY_BASED)
	{
		for (int i= m_nKernelLen; i<m_nFFTLen; i++ )
			m_vdtKernel[ i ] = 0.0;

		m_RealFFTObj->do_fft( m_vdtFreqResp, m_vdtKernel );
	}

}
```

A.5 Source code of the SFilter Class

```
//the empty constructor
CSFilter::CSFilter():
	m_nSamplingRate(0),
	m_nKernelLen(0),
	m_nFFTLen(0),
	m_vdtFreqResp(NULL),
	m_vdtKernel(NULL),
	m_vdtOABuffer(NULL),
	m_RealFFTObj(NULL),
	m_eConvolutionType(NONE),
	m_bValidOAContent(false)
	{
	}

CSFilter::CSFilter( const int nSamplingRate,
				const int nKernelLen ,
				CONVOLUTION_TYPE	ConvolutionType	/*	=
FREQUENCY_BASED*/ ,
				const int nFFTLen /* = 0 */ ):
	m_bValidOAContent(false),
	m_vdtFreqResp(NULL),
	m_vdtKernel(NULL),
	m_vdtOABuffer(NULL),
	m_RealFFTObj(NULL)
{
	//std::cout<<"Kernel len="<<nKernelLen<<std::endl;

	if ( ConvolutionType ==TIME_BASED )
		if (InitTimeComponents( nKernelLen, nSamplingRate ) == false )
			throw new CSFilterException(_T("The time components could not be
initialized.") );


	if ( ConvolutionType == FREQUENCY_BASED )
	{
		//	The length of the Fast Fourier Transform must be a power of two
and
		//half of it must be able to contain the length of the kernel + 1.

		if(InitTimeComponents( nKernelLen, nSamplingRate, nFFTLen ) == false )
			throw new CSFilterException( _T("The time components could not be
initialized." ));

		try
		{
			if ( CSFilter::InitFreqComponents( nFFTLen ) == false )
				throw new CSFilterException( _T("The freq components could
not be initialized."));
		}

		catch( CSFilterException* e)
		{
			e->Delete();

			ClearTimeComponents();

			m_eConvolutionType = NONE;

			throw;
		}
	}
}

CSFilter::~CSFilter()
{
	delete[] m_vdtFreqResp;
	delete[] m_vdtKernel;
```

```
	delete[] m_vdtOABuffer;
	delete m_RealFFTObj;
}
/*****************************************************************************/
/*
	Description:
			Initializes the components that are needed in a frequency-based
		operating mode(). Prior to this the components needed in a time-based
		convolution must be initialized otherwise an exception will be thrown.
		It sets the convolution mode to a frequency-based convolution.
Input:
		Type		Name			Description

		int		nKernelLen		-the length of the kernel,
must be an odd positive integer

		int		nSamplingRate		-the sampling rate, must be a
positive integer

		int		nFFTLen			-the length of the Fast
Fourier Transform(FFT),must be a power of two and also half of it must be
long enough contain the length of the kernel plus 1 samples.

	Return:
		Type					Description

		bool				-if the components can not be
initialized due to insufficient resources(memory) or incorrect parameters,
false will be returned ,otherwise true.

	Throws:
		CSFilterException			- in case the parameters are
incorrect,not in case a memory exception occurs.
*****************************************************************************/
const bool CSFilter::InitFreqComponents(const int nFFTLen )
{
	assert( nFFTLen > 0 );

	if (!ValidTimeComp())
		throw new CSFilterException( _T("The time components are not
initialized." ));

	if ( ValidFFTLen( m_nKernelLen, nFFTLen ) == false )
		throw new CSFilterException( _T("The Fast Fourier Transform length is
invalid."));

	ClearFreqComponents();

	try
	{
		m_vdtFreqResp= new DataType[ nFFTLen ];
	}
	catch( CMemoryException* e )
	{
		e->Delete();

		return false;
	}

	m_nFFTLen		= nFFTLen;

	try
	{
		m_vdtOABuffer=	new DataType[ m_nKernelLen - 1 ];
	}
	catch( CMemoryException* e )
	{
		e->Delete();
```

```
		m_nFFTLen	= 0;

		delete[] m_vdtFreqResp;
		m_vdtFreqResp = NULL;

		return false;
	}

	try
	{
		m_RealFFTObj = new ffft::FFTReal<DataType>( nFFTLen );
	}
	catch( CMemoryException* e )
	{
		e->Delete();

		m_nFFTLen	= 0;

		delete[] m_vdtFreqResp;
		m_vdtFreqResp = NULL;

		delete[] m_vdtOABuffer;
		m_vdtOABuffer = NULL;

		return false;
	}

	m_eConvolutionType = FREQUENCY_BASED;

	return true;
}
/*****************************************************************************
	Description:
		Checks if a given length of a Fast Fourier Transform is enough
	to contain the length of the kernel and also if it is a power of 2.
		In the current implementation the length of the kernel must be
	at most half plus one samples of the length of the Fast Fourier
	Transform.

	Throws:
		Nothing.
*****************************************************************************/


const bool	CSFilter::ValidFFTLen	( const int nKernelLen,
                                          const int nFFTLen )
{
	assert( nKernelLen > 0 );
	assert( nFFTLen > 0 );

	if (nFFTLen > 0 && SMath::IsPowOf2( nFFTLen ) == true && ( nFFTLen/2 )+ 1 >=
nKernelLen )
		return true;

	else
		return false;
}
/*****************************************************************************
Description:
		Clears the frequency response buffer, the overlap-add buffer
	and the Fast Fourier Transform object and sets the convolution type
	to time-based.
Throws:
		Nothing.
*****************************************************************************/
void CSFilter::ClearFreqComponents()
{
	delete[]	m_vdtFreqResp;
	m_vdtFreqResp	=	NULL;
```

```
	delete[]	m_vdtOABuffer;
	m_vdtOABuffer		=	NULL;

	delete		m_RealFFTObj ;
	m_RealFFTObj		=	NULL;

	m_nFFTLen			=	0;

	m_eConvolutionType	=	TIME_BASED;
}

bool CSFilter::FFTConvolve(	DataType* vdtInput,
					DataType* vdtOutput )
{
	DataType* vdtOutputAsFreq;

	try
	{
		vdtOutputAsFreq = new DataType[ m_nFFTLen ];
	}
	catch(CMemoryException *e)
	{
		e->Delete();

		return false;
	}

	m_RealFFTObj->do_fft( vdtOutput, vdtInput );

	double temp;
	for( int i = 1; i < m_nFFTLen/2; i++ )
	{
		temp = vdtOutput[ i ]* m_vdtFreqResp[ i ] - vdtOutput[ i + m_nFFTLen/2 ]
* m_vdtFreqResp[ i + m_nFFTLen/2 ];

		vdtOutputAsFreq[ m_nFFTLen/2 + i ] = vdtOutput[i] * m_vdtFreqResp[i +
m_nFFTLen/2 ] + vdtOutput[ i + m_nFFTLen/2 ] * m_vdtFreqResp[ i ];

		vdtOutputAsFreq[ i ] = temp;
	}

	vdtOutputAsFreq[ 0 ]				=	vdtOutput[ 0 ] *
m_vdtFreqResp[ 0 ];
	vdtOutputAsFreq[ m_nFFTLen / 2 ] =	vdtOutput[ m_nFFTLen / 2 ] *
m_vdtFreqResp[ m_nFFTLen / 2 ];

	//we transform the convoluted signal back to the time domain
	m_RealFFTObj->do_ifft(vdtOutputAsFreq , vdtOutput);

	//we rescale(divide each element with the length of the FFT) the output signal
//because it is scaled when we use
	//forward Fourier transform + inverse Fourier transform
	m_RealFFTObj->rescale( vdtOutput );

	return true;
}

CSFilter::DataType* CSFilter::getKernel()
{
	return m_vdtKernel;
}

int CSFilter::getFFTLength()
{
	return this->m_nFFTLen;
}
int CSFilter::getKernelLength()
{
	return this->m_nKernelLen;
}
```

Appendix B

Appendix B: Interfacing the Tcl with VTK

B.1 Tcl source code for implementing the isocontour extraction

```
proc ExtractBone2 { } {

       global params
       global aCamera2

       vtkRenderer aRenderer
              renWin2 AddRenderer aRenderer
vtkMarchingCubes skinExtractor2
  skinExtractor2 SetInputConnection [v16 GetOutputPort]
  skinExtractor2 SetValue 0 500
vtkPolyDataNormals skinNormals2
  skinNormals2 SetInputConnection [skinExtractor2 GetOutputPort]
  skinNormals2 SetFeatureAngle 60.0
vtkShrinkPolyData shrinkData2
shrinkData2 SetInputConnection [skinNormals2 GetOutputPort]
shrinkData2 SetShrinkFactor 0.9
vtkStripper skinStripper2
  skinStripper2 SetInputConnection [shrinkData2 GetOutputPort]
vtkPolyDataMapper skinMapper2
  skinMapper2 SetInputConnection [skinExtractor2 GetOutputPort]
  skinMapper2 ScalarVisibilityOff
  if { $params(actnorm) == "lod" } {
       vtkLODActor skin2
       } else {

       vtkActor skin2
       }
  skin2 SetMapper skinMapper2
  [skin2 GetProperty]  SetDiffuseColor $params(colorR) $params(colorG) $params(colorB)
  [skin2 GetProperty] SetSpecular .3
  [skin2 GetProperty] SetSpecularPower 20
vtkContourFilter boneExtractor2
  boneExtractor2 SetInputConnection [v16 GetOutputPort]
  boneExtractor2 SetValue 0 1150
vtkPolyDataNormals boneNormals2
  boneNormals2 SetInputConnection [boneExtractor2 GetOutputPort]
  boneNormals2 SetFeatureAngle 60.0
vtkStripper boneStripper2
  boneStripper2 SetInputConnection [boneNormals2 GetOutputPort]
vtkPolyDataMapper boneMapper2
  boneMapper2 SetInputConnection [boneStripper2 GetOutputPort]
  boneMapper2 ScalarVisibilityOff
  if { $params(actnorm) == "lod" } {
       vtkLODActor bone2
       } else {

       vtkActor bone2
       }

  bone2 SetMapper boneMapper2
  [bone2 GetProperty] SetDiffuseColor $params(bcolorR) $params(bcolorG)
$params(bcolorB)
vtkOutlineFilter outlineData2
  outlineData2 SetInputConnection [v16 GetOutputPort]
vtkPolyDataMapper mapOutline2
  mapOutline2 SetInputConnection [outlineData2 GetOutputPort]
vtkActor outline2
  outline2 SetMapper mapOutline2
  [outline2 GetProperty] SetColor 0 0 0
```

```
if { $params(SkinExtr) == "skin" } {
        bone2 VisibilityOff
         }
vtkLookupTable bwLut
  bwLut SetTableRange  0 2000
  bwLut SetSaturationRange  $params(sat11) $params(sat22)
  bwLut SetHueRange  $params(hue11) $params(hue22)
  bwLut SetValueRange  $params(val11) $params(val22)
vtkLookupTable hueLut
  hueLut SetTableRange  0 2000
  hueLut SetHueRange  $params(hue1) $params(hue2)
  hueLut SetSaturationRange $params(sat1) $params(sat2)
  hueLut SetValueRange $params(val1) $params(val2)
vtkLookupTable satLut
  satLut SetTableRange  0 2000
  satLut SetHueRange  $params(hue111) $params(hue222)
  satLut SetSaturationRange  $params(sat111) $params(sat222)
  satLut SetValueRange  $params(val111) $params(val222)
vtkImageMapToColors saggitalColors
  saggitalColors SetInputConnection [v16 GetOutputPort]
  saggitalColors SetLookupTable bwLut
vtkImageActor saggital
  saggital SetInput [saggitalColors GetOutput]
  saggital SetDisplayExtent $params(saggital3D) $params(saggital3D)  0 [expr
$params(row) -1]  0 [expr $params(esli)-1]
vtkImageMapToColors axialColors
  axialColors SetInputConnection [v16 GetOutputPort]
  axialColors SetLookupTable hueLut
vtkImageActor axial
  axial SetInput [axialColors GetOutput]
  axial SetDisplayExtent 0 [expr $params(row)-1] 0 [expr $params(col)-1]
$params(axial3D) $params(axial3D)
coronalColors SetInputConnection [v16 GetOutputPort]
  coronalColors SetLookupTable satLut
vtkImageActor coronal
  coronal SetInput [coronalColors GetOutput]
  coronal SetDisplayExtent 0 [expr $params(col)-1] $params(coronal3D)
$params(coronal3D)  0 [expr $params(esli)-1]
if { $params(SkinExtr) == "slices" } {
       aRenderer AddActor saggital
       aRenderer AddActor axial
       aRenderer AddActor coronal
       }
aRenderer AddActor outline2
aRenderer AddActor skin2
aRenderer AddActor bone2
aRenderer SetActiveCamera aCamera2
aRenderer ResetCamera
[skin2 GetProperty] SetOpacity $params(skin3D)
[bone2 GetProperty] SetOpacity $params(bone3D)
aRenderer SetBackground 1 1 1
renWin2 SetSize 640 480
```

B.2 Tcl source code for implementing the direct volume visualization

```
if { $params(SkinExtr) == "all" } {

       aRenderer RemoveAllViewProps
       vtkVolumeRayCastCompositeFunction rayCastFunction2

       vtkVolumeRayCastMapper volumeMapper2
       volumeMapper2 SetInput [v16 GetOutput]
       volumeMapper2 SetVolumeRayCastFunction rayCastFunction2
       volumeMapper2 CroppingOn
       volumeMapper2 SetCroppingRegionPlanes $params(left1) $params(right1)
$params(back1) $params(front1) $params(bot1) $params(top1)

vtkColorTransferFunction volumeColor2
  volumeColor2 AddRGBPoint 0    0.0 0.0 0.0
  volumeColor2 AddRGBPoint 500  1.0 0.5 0.3
  volumeColor2 AddRGBPoint 1000 1.0 0.5 0.3
  volumeColor2 AddRGBPoint 1150 1.0 1.0 0.9
vtkPiecewiseFunction volumeScalarOpacity2
  volumeScalarOpacity2 AddPoint 0    0.00
  volumeScalarOpacity2 AddPoint 500  0.15
  volumeScalarOpacity2 AddPoint 1000 0.15
  volumeScalarOpacity2 AddPoint 1150 0.85
vtkPiecewiseFunction volumeGradientOpacity2
  volumeGradientOpacity2 AddPoint 0   0.0
  volumeGradientOpacity2 AddPoint 90  0.5
  volumeGradientOpacity2 AddPoint 100 1.0

vtkVolumeProperty volumeProperty2
  volumeProperty2 SetColor volumeColor2
  volumeProperty2 SetScalarOpacity volumeScalarOpacity2
  volumeProperty2 SetGradientOpacity volumeGradientOpacity2
  if { $params(linear) == "nearn" } {
              volumeProperty2 SetInterpolationTypeToNearest
              } else {
              volumeProperty2 SetInterpolationTypeToLinear }
  volumeProperty2 ShadeOn
  volumeProperty2 SetAmbient 0.4
  volumeProperty2 SetDiffuse 0.6
  volumeProperty2 SetSpecular 0.2
vtkVolume volume2
  volume2 SetMapper volumeMapper2
  volume2 SetProperty volumeProperty2
aRenderer AddViewProp volume2

wm withdraw .
aRenderer AddViewProp volume2
aRenderer SetBackground 0 0 0

rayCastFunction2 Delete
volumeProperty2 Delete
volumeMapper2 Delete
volumeColor2 Delete
volumeScalarOpacity2 Delete
volumeGradientOpacity2 Delete
volume2 Delete
```