Skeletonization and Centerline Extraction and Bioinformatics Lab Notebook

Christopher Tart

May 2006

Advisors:
Dr. Kalpathi R. Subramanian
Dr. Anthony Fodor

Table of Contents

Preface 3

Skeletonization and Centerline Extraction 4
Chapter 1 Introduction 5
Chapter 2 Centerline Algorithms 6
2.1: Previous Algorithms 6
2.2: New Algorithm 9
Chapter 3 Implementation 11
3.1: Components 11
3.1.1: Visualization Tool Kit 11
3.1.2 Fast Light Toolkit 11
3.1.3: Existing Application Classes 12
3.2: New Classes 12
3.2.1: MainWindow 13
3.2.2: VtkCanvas 14
3.3: Algorithm Processing 16
3.3.1: VolumeBSCoding 16
3.3.2: VolumeSSCoding 18
3.3.3: VolumePartition 19
3.3.4: Skeleton Extraction 20
3.4: BOX Algorithm 22
Chapter 4 Conclusion 26

Bioinformatics Lab Notebook 27
Chapter 1 Introduction 28
Chapter 2 The Prototype 30
Chapter 3 Design Requirements 32
Chapter 4 Implementation 33
4.1: Components 33
4.1.1: Fast Light Toolkit 33
4.1.2: OpenGL 33
4.1.3: MySQL and MySQL++ 33
4.2: Design 34
4.2.1: Class Structure for Data Input 34
4.2.2: Class Structure for Data Display 38
4.2.3: Class Structure for User Interaction 42
Chapter 5 Conclusion 44

References 45

Preface

For my senior project I was involved in two distinct projects, skeletonization of medical volumes, and a bioinformatics lab notebook. I will break each project into two sections and discuss each separately.

The first project was a conversion of an application originally created by Jianfei Liu, a Phd student studying here at UNCC. His application was a proof-of-concept implementation of a centerline extraction algorithm that he created. The second project involved a software redesign and improvement of a prototype bioinformatics viewer created by Dr. Anthony Fodor.

Skeletonization and Centerline Extraction

Chapter 1

Introduction

One of the tasks of computer visualization is to determine algorithms to analyze data, and construct, store, and display meaningful results from this analysis. Structural skeletonization is one such process. As more high-resolution volume datasets are becoming available, there is more demand to compact the structure for these datasets for more efficient memory usage. Given a set of a tree like volume from biological data, it is useful to determine the centerline skeleton to provide useful cues for highlighting structural features [16]. Virtual endoscopy would be one important use. Virtual endoscopy is an application which deals with the exploration of hollow organs and anatomical cavities using volume data [1]. There are many uses for virtual endoscopy in the medical field, such as using data from a patient to project the best path for an endoscopic tool within the body.

There are several existing algorithms for this problem; however they are less then ideal for different cases. Jianfei Liu, a PhD student attending UNCC, created a new algorithm which attempts to address several of the problems with the existing algorithms. He developed a Windows application as a proof-of-concept for his algorithm using OpenGL. My task was to port this application to the Linux platform, and use the Visualization Toolkit (VTK) for much of the processing and display. In addition the Fast Light Toolkit (FLTK) was used for the interface. Both of these toolkits are cross-platform compatible and ultimately the program developed did run on both Linux and Windows.

Chapter 2

Centerline Algorithms

2.1: Previous Algorithms

There exist several algorithms that attempt to skeletonize a volume. They succeed to a varying degree depending on the structure of the data they are trying to skeletonize.

- Voronoi methods

  Voronoi methods [2, 3, 4] take advantage of Voronoi diagrams to generate skeletons with Voronoi polygons in 2D and Voronoi polyhedra in 3D. It is mainly used for polygon defined objects and not appropriate for volumetric models. In addition, noisy boundaries tend to generate dense Vorioni diagrams and cause many spurious branches.

- Skeleton replacement approaches

  Paik [5] proposed a practicable thinning approach to extract a reliable path for virtual endoscopy. It repeatedly determines the shortest path along the outmost layer of the volumetric objects generated by a parallel unrestricted thinning algorithm until only one central path remains. Extending from [5], Mahnaz [6] modifies the computation of the central path in each exposed outmost layer by distance transformation. These methods guarantee the connectivity of the centerline, but the manual detection of branch tips as well as computation efficiency needs to be improved.

- Template based thinning

  Ma [7] proposed a fully parallel, connectivity-preserving thinning algorithm to reduce computational costs. It avoids expensive testing of simple points [8] by

matching the 26-neighborhood of the points with specific templates obtained by rotating and reflecting 4 so-called template cores. However, it may disconnect the centerline when a point belonging to the class D templates is deleted in parallel with some of its adjacent points, breaking the simple point condition. In addition, it often generates a skeleton with many spurious branches and leaves unorganized discrete points. Vilanova [9] treats the templates of class D in a separate iteration, but it still can't guarantee the connectivity when the adjacent points in class D templates are deleted in parallel. The connectivity of the skeleton is kept in [10] by adjusting a point deleted with its neighbor points in different iterations. A length parameter is also applied to remove the influence of spurious branches in [10]. But these methods are invalid if the length of a spurious branch is more than that of a real one.

- Integrating distance based skeletonization

  Zhou [11] proposed an efficient solution to settle each skeleton point to the maximum DFB-distance (Distance From Boundary) within the voxel cluster of the same DFS-distance (Distance From a Starting point). However, this algorithm is based on local heuristics and may disconnect the candidate skeleton and stitch it back together. Bitter [12] provided a global minimum cost path-searching algorithm with the penalty distance, which is defined as a heuristic combination of the DFS-distance and the DFB-distance. Sato [13] extended this algorithm to tree-structure skeletons by an adaptive sphere. Although Bitter [14] further improved the penalty distance algorithm to correct two small mistakes in [12] and [13], these techniques still could not completely prevent the "hugging-corner"

problem. Wan [15] proposed a more efficient skeletonization algorithm to solve this problem by delivering a centered path rather than a shortest one using the exact Euclidean distance. But the skeleton still does not follow the medial axis when several candidate points occupy the same distance value.

Almost all these algorithms are not effective in eliminating the influence of a branch on other branches, and spurious branches [3].

2.2: New Algorithm

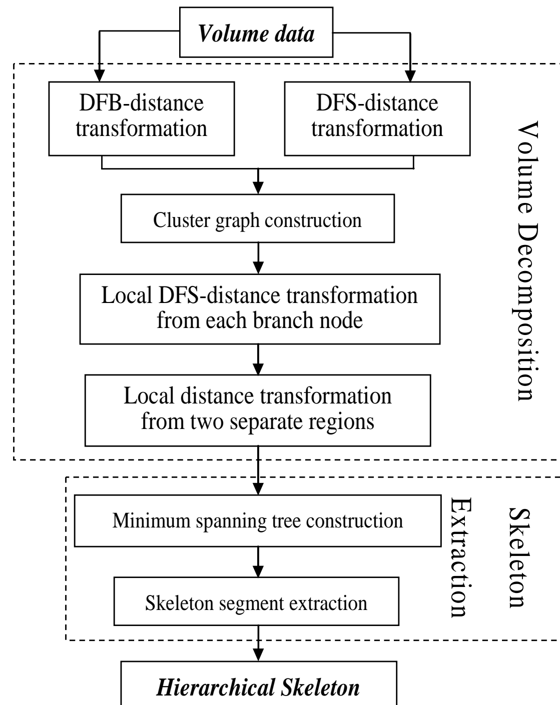

Figure 2.1: Algorithm Structure

As shown in Figure 2.1, this algorithm has two main parts: volume decomposition, and skeleton extraction. The key idea of this algorithm is first to decompose the whole input object into single-tube components, find the centerline of each sub volume, and finally create a hierarchical skeleton with high centeredness.

The first main part of the algorithm is volume decomposition. The first step to achieve this is the distance from boundary computation (DFB). This step computes how far from the boundary each voxel is, and marks which voxels we want to use based on this distance. The second step is to compute the distance from the starting point (DFS) for each voxel. The starting point in this case is defined as the center of the trunk of the volume. The third step is to break the volume up into clusters of voxels with the same DFS, construct a cluster graph, and identify two specific types of clusters: node tips and

branch points. The fourth step is to compute a local DFS from each of the branch points computed in the previous step. For the last step of volume decomposition, the algorithm looks for two clusters with the same local DFS values and performs a distance transform sequentially backwards until the two layers meet, which marks the boundary of the branch.

The second main part is the construct the final centerline skeleton. It begins by first constructing a minimum spanning tree (MST) using the DFB and DFS calculations. Once constructed, the MST is used to compute the centerline for each sub volume, and the centerlines are hierarchically connected to form the final skeleton. Figure 2.2 shows visually the results from each of the two main parts of the algorithm.

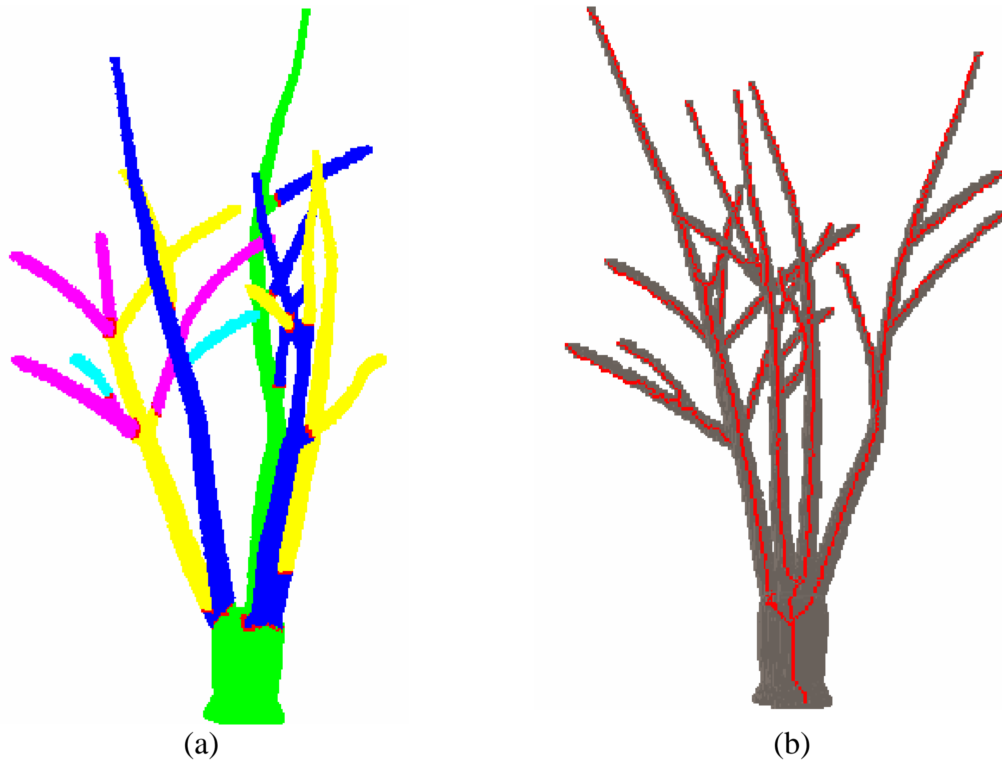

Figure 2.2: Visualization from Volume Decomposition (a) and Centerline Skeleton (b)

Chapter 3

Implementation

3.1: Components

3.1.1: Visualization Toolkit

The Visualization Toolkit (VTK) is an open source C++ class library for 3D computer graphics, image processing, and visualization. It is freely available and used by thousands of researchers and developers around the world [4]. VTK includes many useful tools for this application. The basic concept of VTK is to setup a visualization pipeline by connecting various VTK objects together to read data, modify it, apply filters, and place it inside a renderer.

3.1.2 Fast Light Toolkit

The Fast Light Toolkit (FLTK) is a cross-platform C++ GUI toolkit which provides a high level of GUI functionality and support for 3D graphics via OpenGL [5]. FLTK provides a simple way to create the interface of our application. The basic object in FLTK is a widget. Widgets can also contain other widgets so you can create a hierarchy of objects. For our application we use a single window widget which contains all our other widgets.

3.1.3: Existing Application Classes

The existing application was programmed in C++ as a pure Windows application using OpenGL for the display. The data used by this program consisted of volume data stored in 8 bit Windows BMP files, where each file was a single slice of the data.

In the existing application there are four main class files: VolumeBSCoding, VolumeSSCoding, VolumePartition, and CenterlineExtraction. These do all of the computation work for the algorithm. There are also several support classes: Algorithm, DistanceTransform, Obj, SortHeap, Tree, and VoxelAlgorithm, as well as a header file for program wide definitions.

3.2: New Classes

The preceding three components combine to recreate the interface and functionality that was created through the Windows specific MFC in the existing application. Two classes were created: MainWindow to handle the interface, and VtkCanvas to handle data processing and the visualization display. The VtkCanvas class takes advantage of a support class, vtkFlRenderWindowInteractor, which makes it easy to place a VTK render window within a FLTK window. The VtkCanvas class also contains the classes from the existing program. Figure 3.1 shows the general structure for the application, and the classes used in MainWindow and VtkCanvas.

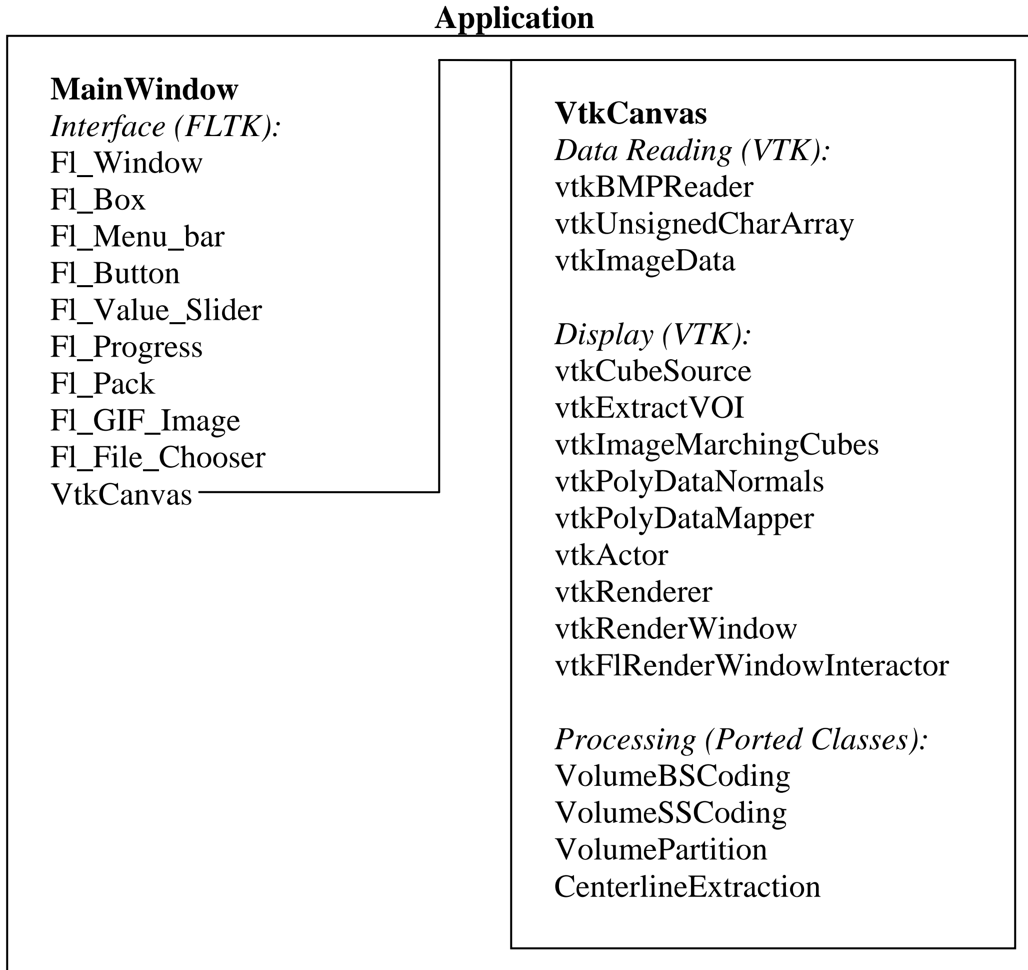

Figure 3.1: Program Structure

3.2.1: MainWindow

MainWindow contains all the interface elements for the program. It creates a new Fl_Window and adds several widgets to it. The Fl_Menu_bar contains several commands and options. The Fl_Button and Fl_Value_Slider are used during the algorithm processing steps, and the Fl_Box is used to display a text message. Within MainWindow are also the callback functions that are called when you interact with these widgets. Finally MainWindow creates a new VtkCanvas which is where the majority of the work is done.

3.2.2: VtkCanvas

VtkCanvas contains the data reading, processing and output functions, as well as the classes implementing the algorithm, as shown in Figure 3.1. It uses vtkBMPReader for data reading to input a collection of BMP files, which are stored as vtkImageData. It then pulls the raw data from the vtkImagaData and stores it as a vtkUnsignedCharArray which is converted and stored within a structure that the classes implementing the algorithm use.

When creating the new VtkCanvas, the visualization pipeline is instantiated in the VtkCanvas constructor as seen in Figure 3.2.

```
bmpinput = vtkBMPReader::New();

subsamp = vtkExtractVOI::New();

mcubes = vtkImageMarchingCubes::New();
mcubes->SetInput(subsamp->GetOutput());
mcubes->SetValue(0, 1);

norm = vtkPolyDataNormals::New();
norm->SetInput(mcubes->GetOutput());
norm->SetFeatureAngle(60.0);

mapper = vtkPolyDataMapper::New();
mapper->SetInput( norm->GetOutput() );
mapper->ScalarVisibilityOff();

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

ren = vtkRenderer::New();
ren->SetBackground( 0.0f, 0.5f, 1.0f );

renwin = vtkRenderWindow::New();
renwin->AddRenderer( ren );

iren = new vtkFlRenderWindowInteractor( x, y, w, h, NULL );
iren->SetRenderWindow( renwin );

iren->Initialize();
```

Figure 3.2: Visualization Pipeline from VtkCanvas Constructor

Initially bmpinput is empty. Selecting a menu option runs the openBMP function which does two things. First it loads the bmpinput class with the data. The marching cubes algorithm [23] is applied to obtain a polygon representation of the volume. The output from marching cubes is sent through a polygon normal filter and a polygon data mapper before being set as a vtkActor object and added to the renderer. This creates an outline for the volume which we use as a reference as we display more data, as seen in Figure 3.3. Secondly it pulls the raw data out of the bmpinput class, which is then used to populate an array within a volume data structure.

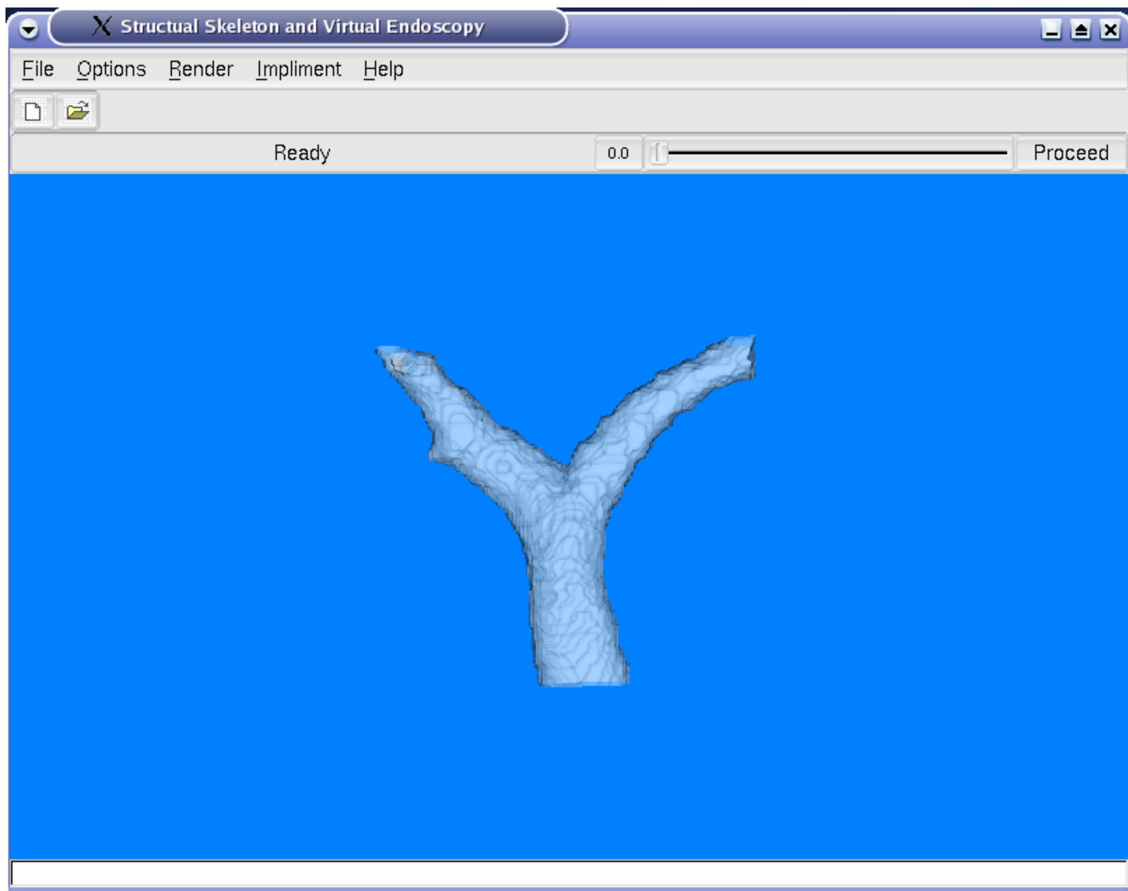

Figure 3.3: Completed Loading Data

3.3: Algorithm Processing

Once loaded, the data needs to be sent through each step of the algorithm. Each step is dependent on the previous, and the first two steps require the input of a parameter specific for each step, which needs to be adjusted for the specific data to ensure the process is successful. The two primary interface items used are the slider, and the Proceed button as seen in Figure 3.4. The slider varies the parameter for the current step. The Proceed button sends the program to the next step of the algorithm. After each step of the algorithm, visualization output is displayed to show the result of the computation. Thus each step has two functions, one for the computation, and one for the rendering.

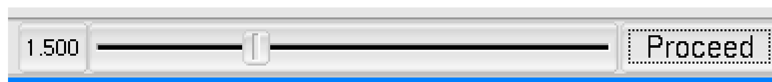

Figure 3.4: Slider and Button Interface Elements

3.3.1: VolumeBSCoding

This class implements the distance from boundary computations for the algorithm. For this step, the parameter determines the distance cutoff for the voxels to include in the next steps. The smaller the slider number, the more voxels are included.

For this step, the display shows all the voxels which are within the cutoff set by the slider. The output is first converted to a custom structure called BOX. We will discuss the BOX algorithm later. Each BOX is then rendered using a vtkCube. Moving the slider causes the rendering to update so you are able to visually see the effects of changing the parameter. Figure 3.5 shows the result of this step with the default value for the parameter, and Figure 3.6 shows it with lower value for the parameter.

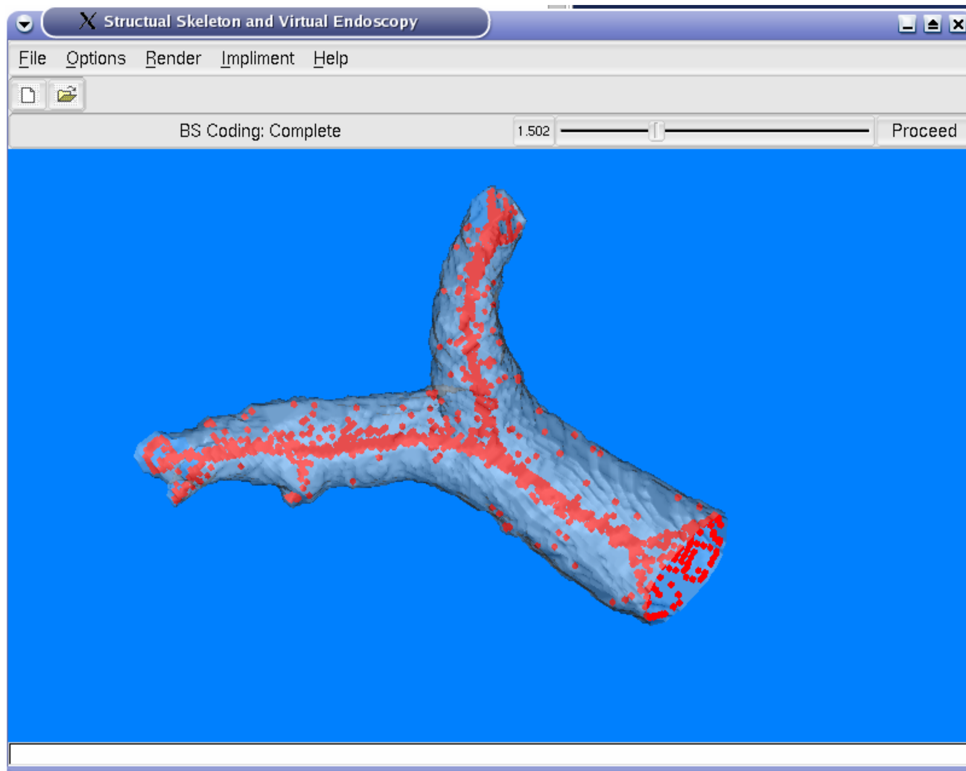

Figure 3.5: BS Coding with default parameter

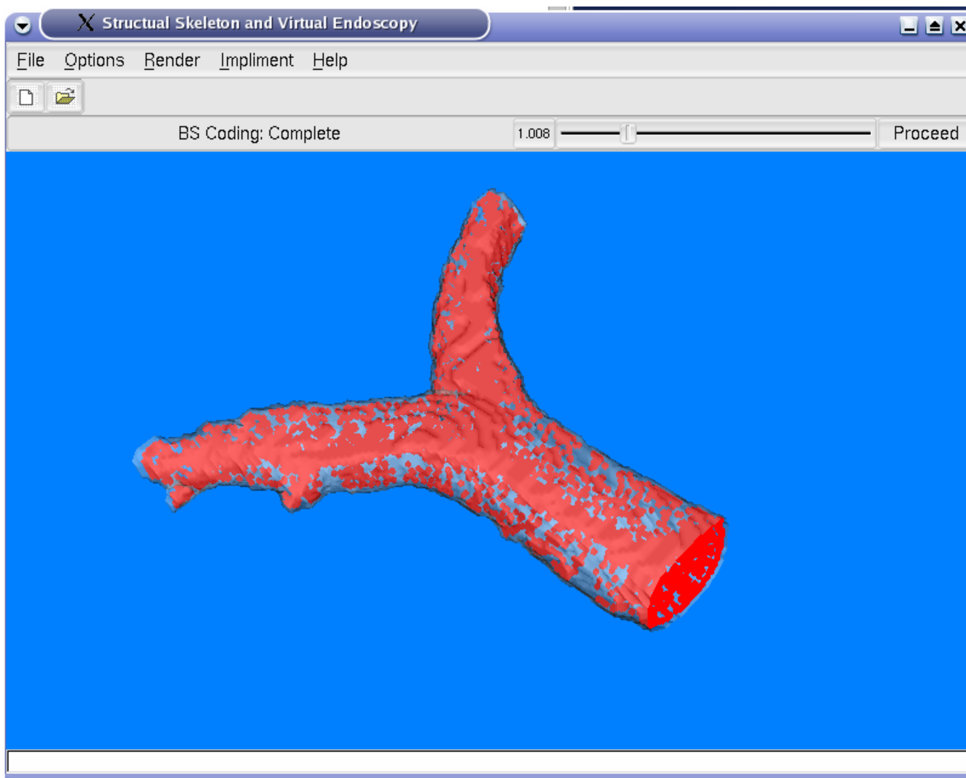

Figure 3.6: BS Coding with a lower parameter

3.3.2: VolumeSSCoding

This class implements the distance from starting point computations for the algorithm. In this step the parameter determines the minimum distance for a valid branch. Increasing the slider value causes fewer branches to be valid. For output we display the trace back line from the tip of each branch to the starting point. The branch points are displayed as large green boxes. Figure 3.7 shows the result of this step with a value of zero for the parameter, and Figure 3.8 shows the result with a slightly higher value, which has removed two of the branches.

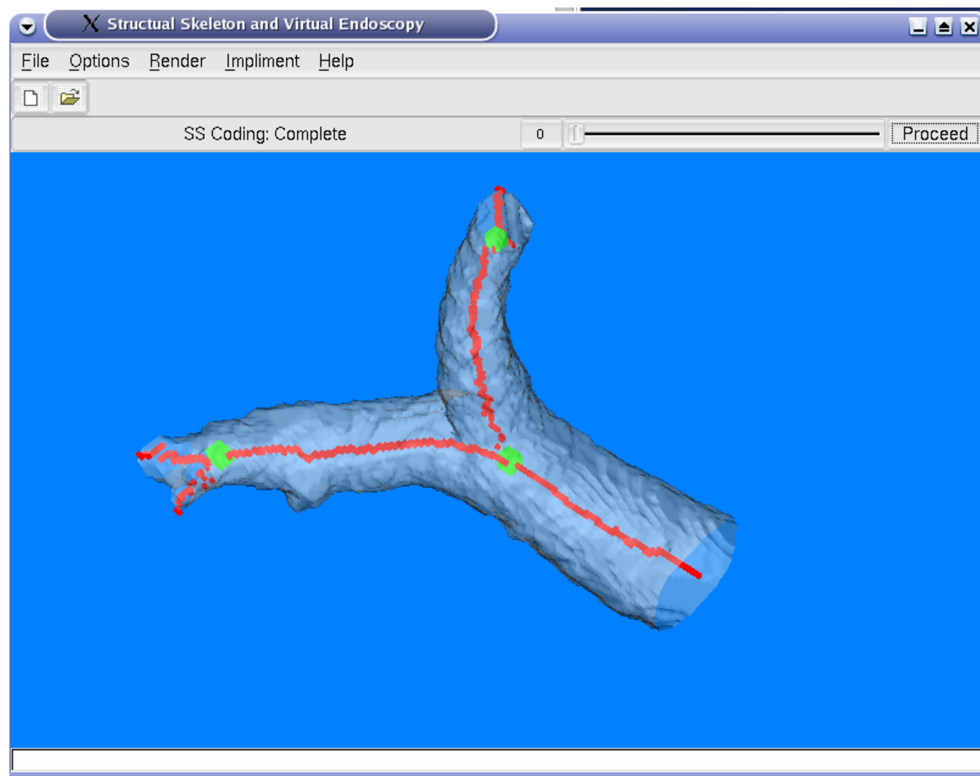

Figure 3.7: SS Coding with lowest parameter

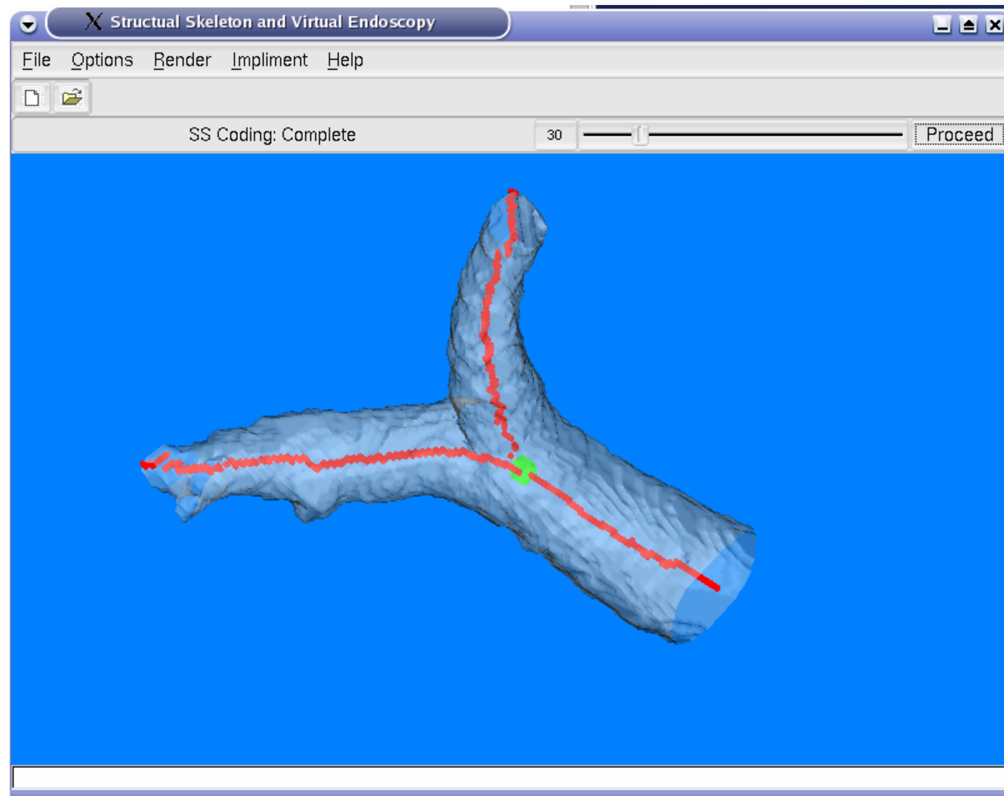

Figure 3.8: SS Coding with higher parameter

3.3.3: VolumePartition

This class implements the cluster graph construction, local DFS transformation and local distance transformation steps of the algorithm. It does not require the input of any parameters; it simply takes the results of the previous two classes to generate the result. For output, all voxels within the volume are rendered, but their colors are determined by which branch they are in. In addition the branch border is rendered in red. This step relies heavily on the Box algorithm to achieve acceptable performance. Figure 3.9 shows the result of this step.

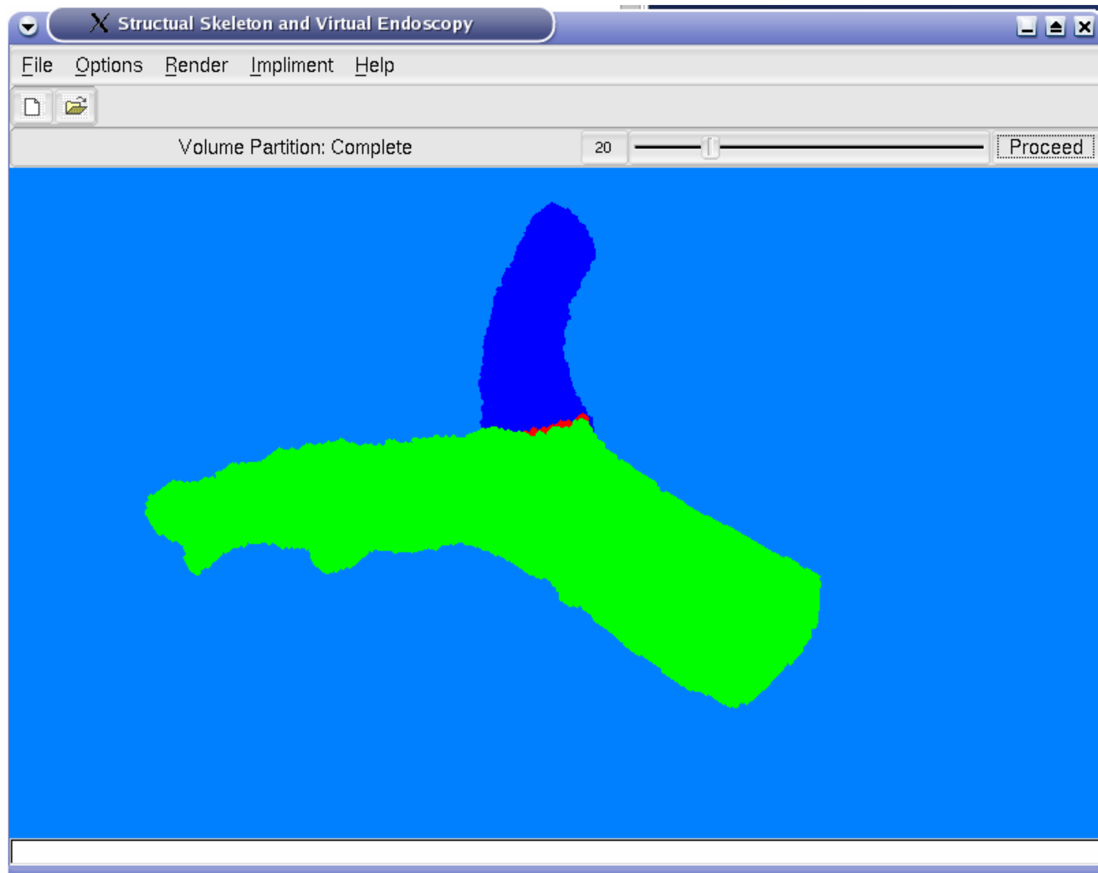

Figure 3.9: Volume Partition

3.3.4: Skeleton Extraction

This class implements the minimum spanning tree construction and skeleton segment extraction steps of the algorithm. This step also requires no input parameters, it only uses the results from the previous classes to generate the final skeleton. The output is simply the final centerline skeleton. Figure 3.10 shows the full view of the result of this step. Figure 3.11 shows the same image zoomed inside looking at the branch point.

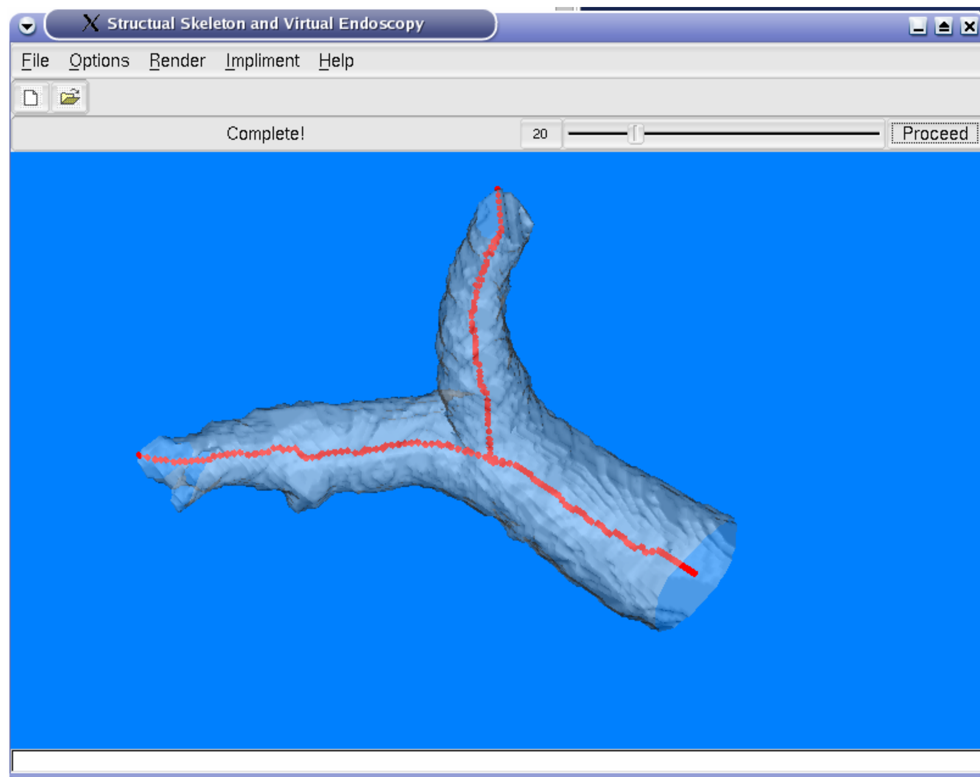

Figure 3.10: Centerline Extraction full view

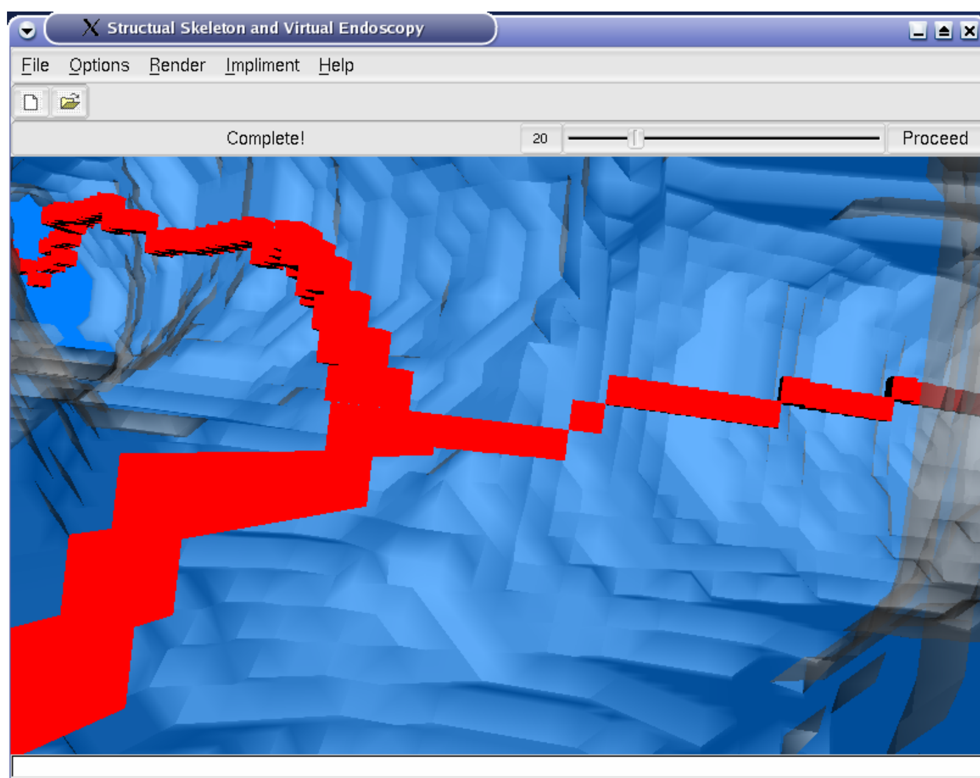

Figure 3.11: Centerline Extraction zoomed view

3.4: BOX Algorithm

A large portion of this project was simply converting the existing code. The steps of the algorithm were already in functional classes so only the proper input needed to be given and the output properly interpreted. However for display, the existing program used pure OpenGL to render its results, and it did so by rendering each voxel as a separate six face polygon cube. This worked well, and ran quickly.

For our code, however, we are rendering through VTK, which is completely object oriented. The first attempt was implementing the same process used in the existing program, except rendering each voxel as a vtkCube. This however soon showed to be nearly unmanageable, since each vtkCube needed to be instantiated and this overhead was slowing down the display substantially when you had over a few 1000 voxels.

The only solution to this was to reduce the number of vtkCube objects created. Fortunately, a large number of the voxels we are trying to display could be connected, thus allowing us to render many voxels using one vtkCube object. A three voxel by three voxel cube, for example, could be rendered using a single vtkCube object with the dimensions set to three. So during the rendering steps of BSCoding and VolumePartition, the output data is run through VoxelToBox function.

```
struct BOX
{
	double x, y, z;
	double w, h, l;
	int label;
};
```

Figure 3.12: BOX Struct

As seen in Figure 3.12, the BOX struct contains a starting point, and the width, height, and length of the box, as well as a label, which is used to set the color in the volume partition display.

The volume data from the algorithm is stored in a two dimensional integer array, with one dimension actually storing the data for two dimensions. It is simplest to look at it as a three dimensional array. In this array “-1” defines a background voxel, while any other number defines it as part of the volume. The algorithm starts by walking through the array until it finds the first voxel that is part of the volume. From that point it then walks along one dimension as far as there are voxels that are also part of the volume. It then walks along another dimension as far as it can go while each line of voxels is within the volume. Finally it walks along the last dimension as far as it can go while each square of voxels is within the volume. Once this is done, it now has the width, height, and length of the new box, each which corresponds to the distance walked along each dimension. It creates the new BOX and adds it to the list. All voxels that were handled are changed to “-1” so that the algorithm will not attempt to handle them again. Finally it proceeds to the next unhandled voxel in the volume and repeats this procedure again, until every voxel in the volume has been handled.

The reduction in the number of cubes to create can be quite large. Figure 3.12 shows the number of voxel needed to be rendered compared to the number of boxes that were rendered at the BSCoding step with various parameter values (refer to section 3.3.1 for a description of the parameter value), and the VolumePartition step.

Step	Parameter	Voxels	Boxes	Savings
BSCoding	1.008	4909	2797	43.02%
BSCoding	1.500	1217	766	37.06%
BSCoding	2.016	257	168	34.63%
VolumePartition	N/A	69203	1753	97.47%

Figure 3.12: Box Algorithm Test Data

The greatest savings is achieved when the object is solid, because this allows the most voxels to be combined into a single box, and alternatively, the more sporadic and random the position of the voxels, the smaller the savings. With BSCoding, different voxels make the cut off so the arrangement can be somewhat scattered. VolumePartition on the other hand is rendering all voxels, so it is one solid group. Because of that, volume partition can start with roughly fourteen times as many voxels, but end up with 1,000 fewer boxes than the first example of BS Coding rendering.

This algorithm could be further refined by finding the largest box in each iteration, thereby keeping the number of boxes to an absolute minimum. However this process would require traversing the whole array once per iteration to locate the largest box. For very large arrays this could become computationally slower then the overhead saved by reducing the number of boxes. It would take some experimenting to determine when or if this change would provide a speed improvement.

Another potential change would be allowing already handled voxels to still be crossed for another box. As the algorithm is designed now, when a voxel has been included in a box, it is flagged and will not be used in any other box. Allowing voxels to be included in more then one box could provide savings in certain cases, for example, when rendering a

plus sign shaped volume. If you break it up according to how the algorithm is designed currently, it would contain three boxes. However, by overlapping in the center the volume could be rendered with only two boxes.

Chapter 4

Conclusion

The purpose of this project was to port an existing application written specifically for Microsoft Windows to a cross platform program that used VTK for its display. The application presented here does that with fair success. Due to time limitations not all of the extra features from the Windows application were implemented in the new application, however, the core functionality was achieved.

Functionality in the old application left to implement includes the ability easily open different datasets, the ability to save images from the view, ability to change the colors used for the output, ability to animate the view, and the ability to select different rendering types.

Bioinformatics Lab Notebook

Chapter 1

Introduction

"Bioinformatics and computational biology involve the use of techniques from applied mathematics, informatics, statistics, and computer science to solve biological problems" [20]. It is a large and growing field with many areas of research.

One task in bioinformatics is the mapping and study of the genome of life on Earth and the related protein structures. The genome, or DNA, of species is made up of a sequence of thousands or millions of nucleotides. There are four types of nucleotides, given the labels "A", "C", "G", "T". These nucleotides form what are called base pairs: "A" will only pair with "T", and "C" will only pair with "G". This causes each strand of DNA to contain two copes of the information, one which is the reverse compliment of the other. This structure allows DNA to be resilient and simple to duplicate.

The central dogma of molecular biology is that DNA is transcribed into RNA which is then translated into proteins. Within the genome of each species, there are many different genes which control the different aspects of the organism. The genes are rarely in a single sequence, however. Within the sequence there are sections called introns which are not part of the coding region of the gene, and exons which are part of the coding region. The introns get spliced out before the RNA gets translated into protein. Because of this there are also cases where genes can get spliced differently causing them to do slightly different things. This adds a layer of complexity to studying this topic.

Studying bioinformatics is often done through associating an unknown sequence with a similar one that you do know. The tool BLAST [22] (Basic Local Alignment Search Tool) is widely used because it is able to quickly search and align a sequence with a large database of known sequences. BLAST will also assign what is called an E-score to each result, which tells you what confidence there is that the match is legitimate.

Because of the complexity and large amount of data involved in the research, having a good tool to store, organize, access, and visualize the data would be very helpful to researchers. Dr. Anthony Fodor created a prototype application in Java that he used to assist him in his research. This tool was only used to better visualize certain aspects of the data, but he wanted to see it expanded to include more features to allow it to become a lab notebook type application for bioinformatics researchers.

For this project we used Dr. Fodor's application as a template for a new application. The new application was programmed in C++ and used OpenGL for the visual display. The Fast Light Toolkit was used for the interface, and MySQL was used for the database storage.

Chapter 2

The Prototype

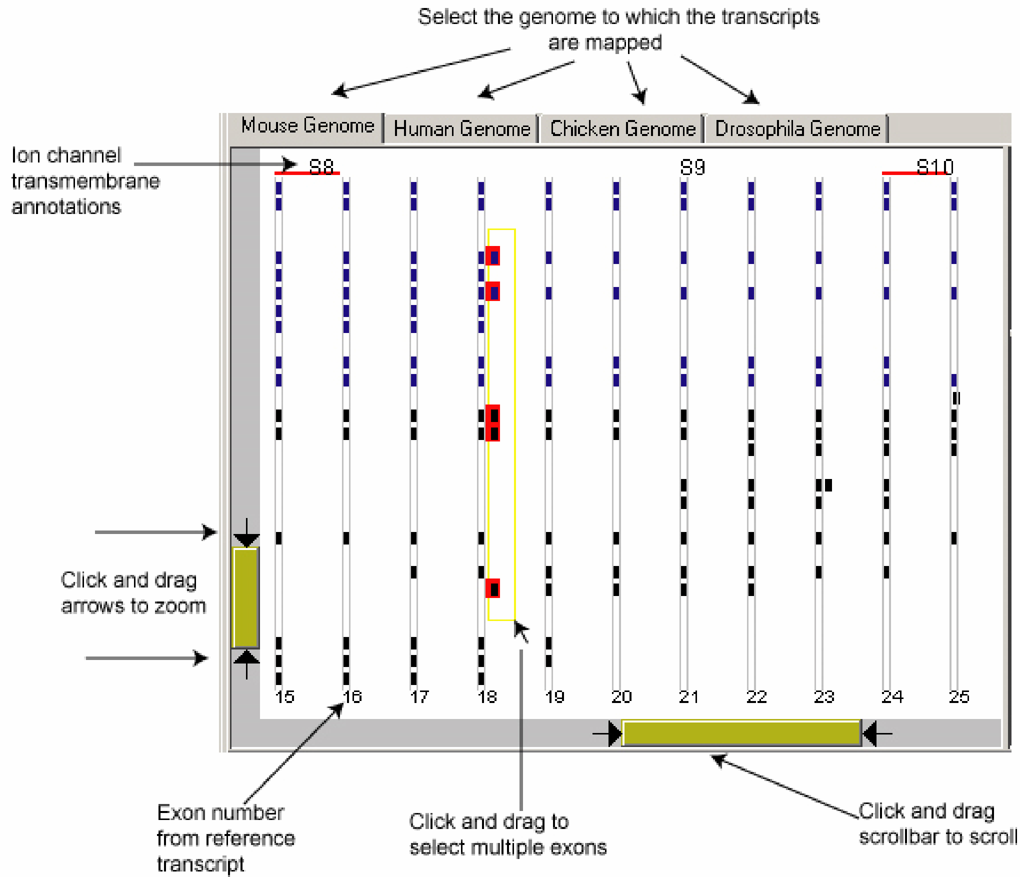

Figure 2.1: Prototype Java Application

The application Dr. Fodor created was programmed in Java. It used a set of text flat files for the data, which was created from a database. It has a set list of so called target species to which the transcripts are mapped. The transcripts, which are sequences from different query species, were BLAST against the targets giving us a list of high scoring pairs which are places where there was good alignment. As seen in Figure 2.1, the target species appear as tabs along the top of the application, allowing the user to change from one target to the next. The query species displayed can be changed through a checkbox list.

Each line in the display is a different sequence. Displayed are the various high scoring pairs from the query sequence being BLAST against the target species. Whether a pair is displayed or not is determined by an e-score cutoff. The e-score is user changeable within a certain range. The color for the sequence lines is varied for each species. The user has the ability to zoom and move about the display via two scroll bars. They can also select an area within the view to see specific information about the selected pairs.

Chapter 3

Design Requirements

The new application would be much more complicated than the prototype. There were several core requirements that we set out to include in this new version:

1. Dynamic display similar to the prototype allowing zooming, scrolling, user selection.
2. High level of performance. The Java application was sluggish at times, so efficient and speedy performance was desirable.
3. Database back end. Instead of using static flat files, an easy to update and extensible database that was accessible from different locations would be needed.
4. Extension of features to take advantage of database and enable the application to be more useful then a simple viewer.
    a. User based with individual profiles and sets of preferences
    b. Ability to update and add data easily
    c. Storage of different aspects of a bioinformatics experiment in one place
    d. Automatic generation of primers for PCR and tiling microarray
    e. Ability to BLAST a new sequence and incorporate data directly into the database automatically

Chapter 4

Implementation

4.1: Components

4.1.1: Fast Light Toolkit

Once again we used FLTK for the interface of our application. In this program we used two windows, one for the view of the data, and one for the display of the details of the selected data.

4.1.2: OpenGL

"OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API) [21]…" FLTK provides an easy way to incorporate an OpenGL view into its windows making it the ideal way to render our data.

4.1.3: MySQL and MySQL++

For the database backend we used MySQL and a library of C++ classes called MySQL++ to allow the application to interface with the database. MySQL is the world's most popular open source database application and it has the proper support for our C++ based applications to access it, so it was a logical choice to use with our program.

4.2: Design

The first step we took was to recreate the basic functions of the prototype application. For this we decided to use the same text flat files used in the prototype to get the new version up and running. The functionality of the application can be broken down into three main sections: data input, data display, and user interaction.

4.2.1: Class Structure for Data Input

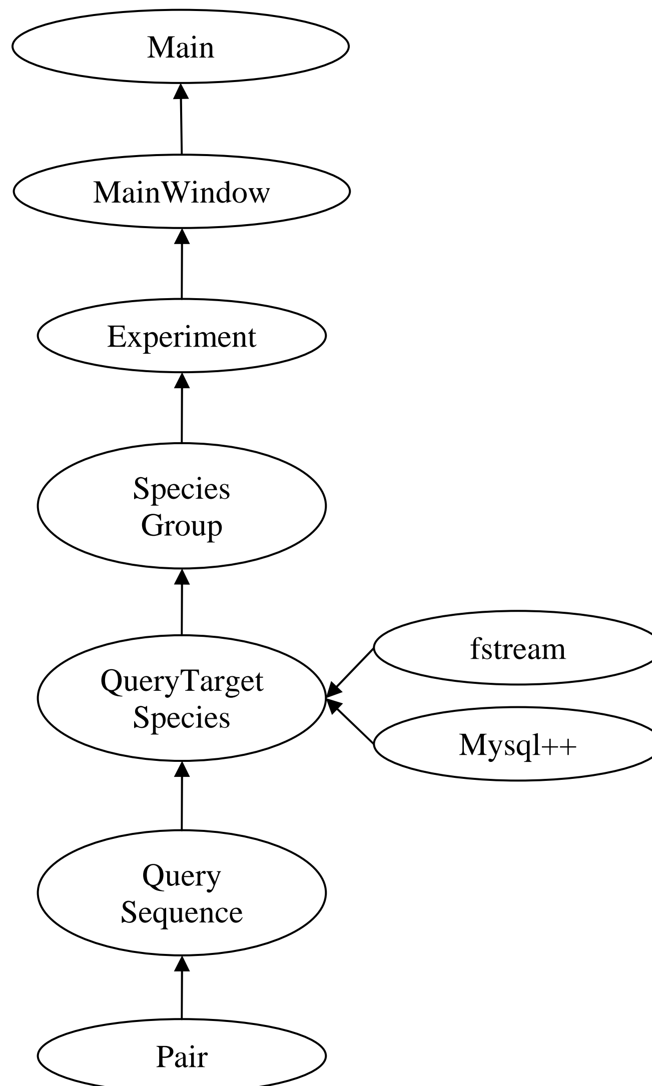

Figure 4.1: Data Input Class Structure

Figure 4.1 shows the structure for data input. Within the constructor for MainWindow, a vector with the list of query species is created, as seen in Figure 4.2. Then the Experiment class is created, which represents the entirety of data used by the program. After creating the Experiment, each target species along with the list of query species is passed to it, as seen in Figure 4.2. This list is the same list used in the prototype application since we are using the same flat files at first.

```
mQueryList.push_back("turtle");
mQueryList.push_back("orphan");
mQueryList.push_back("C_elegans");
mQueryList.push_back("drosophila");
mQueryList.push_back("chicken");
mQueryList.push_back("human");
mQueryList.push_back("cow");
mQueryList.push_back("rat");
mQueryList.push_back("mouse");

mpData = new Experiment();
mpData->AddSpeciesGroup("human", "unigene", mQueryList);
```

Figure 4.2: Portion of Experiment Constructor

Adding a new species to the Experiment causes it to create a new SpeciesGroup class as seen in Figure 4.3. A SpeciesGroup represents the target species which the sequences were BLAST against. Passed to the species group constructor is the target species name and data source name. Then each species in the queries list is added to the new SpeciesGroup class.

```
void Experiment::AddSpeciesGroup(string target, string source,
vector<string>queries)
{
        SpeciesGroup* tmp = new SpeciesGroup(target, source);
        for(int j = 0; j < queries.size(); j++)
        {
                tmp->AddQuery(queries[j]);
        }
        mList.push_back(tmp);
}
```

Figure 4.3: AddSpeciesGroup Function within Experiment Class

When adding a new query, the SpeciesGroup class creates a new QueryTargetSpecies, which represents all the sequences related to a single species BLAST against the target species. Passed to the QueryTargetSpecies constructor is the id, the target species name, the query species name, and the source of the data, as seen in Figure 4.4.

```
void SpeciesGroup::AddQuery(string query)
{
        QueryTargetSpecies* tmp = new
        QueryTargetSpecies(mList.size(), mTarget, query,
        mSource);
        mList.push_back(tmp);
}
```

Figure 4.4: AddQuery Function within SpeciesGroup Class

Upon being created, the QueryTargetSpecies class uses the information passed to it to load the data, as seen in Figure 4.5. For loading the flat files, a file name is generated using a specific format. For the database the target and query name strings are used in the query to return the correct data.

```
QueryTargetSpecies::QueryTargetSpecies(int id, string target,
string query, string source)
{
        mId = id;
        mTargetSpecies = target;
        mQuerySpecies = query;
        mSource = source;
        mFullPath = FilesPath + mTargetSpecies + "_" +
        mQuerySpecies + "_" + mSource + "_hsps.txt";
        LoadData(1);
}
```

Figure 4.5: QueryTargetSpecies Class Constructor

During parsing of the file or database input, new QuerySequence classes are created as needed, and the Pair classes are added to them. A QuerySequence represents a single

sequence that was BLAST against the target genome. The set of high scoring pairs returned are what is stored in the files or database.

The Pair class is the key class which stores all the data for a particular high scoring pair, as seen in Figure 4.6. On the display, each square represents atleast one Pair. Contained in Pair is the id, QuerySequence id, query start and end points, target start and end points, the pertinent portion of the query and target sequences, the relation of the two sequences, the e-score, and the proposed protien translation.

```
class Pair
{
  private:
      int mId;
      int mQuerySequenceId;
      int mQueryStart, mQueryEnd;
      int mTargetStart, mTargetEnd;
      string mQuerySequence;
      string mTargetSequence;
      string mRelation;
      float mEScore;
      string mProposedTranslation;
      ............................
      ............................

}
```

Figure 4.6: Portion of Pair Class Constructor

The structure of this dataset is set up for quick access by sorting all the data into their own classes which can be searched and accessed quickly. If the data was simply kept in a single list, then each time it is accessed for data for a single species it would need to traverse the whole list. Using this structure, it can simply locate the correct class for that species, and work its way down from there knowing all the data needed is contained there.

4.2.2: Class Structure for Data Display

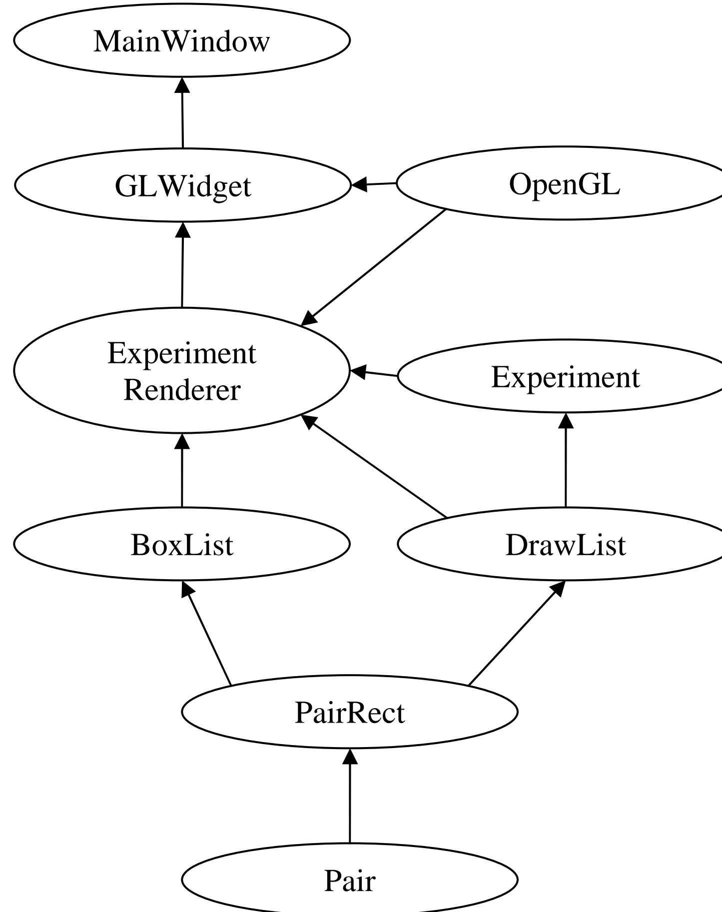

Figure 4.7: Data Display Class Structure

Figure 4.7 shows the class structure for the data display classes. ExperimentRender is the class which handles all the rendering for a specific Experiment class. The Experiment class produces a DrawList, which contains a list of PairRects. Each PairRect represents a single Pair which is to be rendered. The ExperimentRenderer then converts the DrawList into a two dimensional array of BoxList. This array represents breaking up the display area into different boxes, where any box that contains a PairRect gets colored. In the constructor of the ExperimentRenderer, we pass the source Experiment class, and the width and height for the display.

For the horizontal display there are several numbers to keep track of. First is maximum target, which represents the highest target end value of any pair in the data. For this program, 0 is assumed to be minimum target, however this could be made variable for circumstances where the minimum is not set at 0. The next is the current target position which represents where on the genome horizontally we are viewing. The current target position starts at 0, and is controlled by the horizontal scrollbar. Finally there is the target range which represents the current width of data. The target range is a multiplier less than or equal to one that is multiplied against the maximum target. The value slider controls this variable, which starts at one. Thus in terms of genome space, your current horizontal view range is (current_target, current_target + (max_target * target_range)).

For the vertical display there is simply a current line variable which controls what sequence appears as the first line. This starts at 0 and is controlled by the vertical scrollbar.

There are several constants defined, such as the box size, border distances, and line spacing. Using these constants and the size of the render window, the number of horizontal boxes is computed. The array of BoxList is then defined as one dimension having the size equal to the number of horizontal boxes, and the second dimension having the size equal to the number of sequences. Then the DrawList is iterated through, and each PairRect is copied to one or more BoxList based on their position and range. For example, if we have fifty boxes across, and we are viewing base pairs 1000-2000 on the genome, then each box would span twenty base pairs. If a Pair has a start of 1041,

and an end of 1055, then it would be placed in the third box. If another pair spans 1070-1090, then it would be placed in boxes four and five. This process is repeated for each DrawRect in the DrawList until all DrawRects that are within our view range have been sorted into the BoxList array.

Once the BoxList array is populated, figuring out where to draw is a simple task of going through each element in the array and drawing a square when a BoxList is not zero, as seen in Figure 4.8.

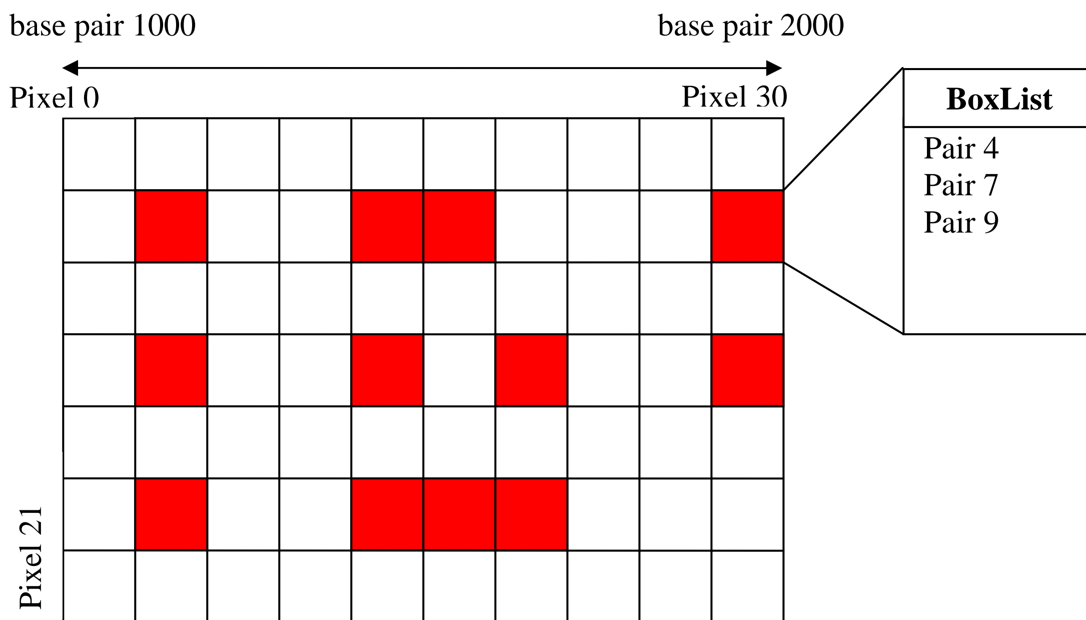

Figure 4.8: Small Portion of a Display Example

```
for(int i = 0; i < lines; i++)
{
        for( int j = 0; j < mBoxCount; j++)
        {
                int bl = LEFTBORDER + (j) * BOXSIZE;
                int bt = TOPBORDER + i * (BOXSIZE + SPACING);
                boxList* boxlist = mpBoxList[mCurrLine + i][j];

                if(boxlist->size() > 0)

                {
                        glBegin( GL_POLYGON );
                        glVertex2f( bl, bt);
                        glVertex2f( bl, bt + BOXSIZE );
                        glVertex2f( bl + BOXSIZE, bt + BOXSIZE );
                        glVertex2f( bl + BOXSIZE, bt );
                        glEnd();
                }
         }
}
```

Figure 4.9: Rendering Code

Figure 4.9 shows the algorithm contained in the Draw function within the ExperimentRenderer, which is called from the draw function of the GLWidget each time the screen needs to be redrawn. It iterates through each element in the BoxList array, and draws a square polygon if the box size is greater then zero, as seen in Figure 4.10.

Figure 4.10: Viewer Display

4.2.3: Class Structure for User Interaction

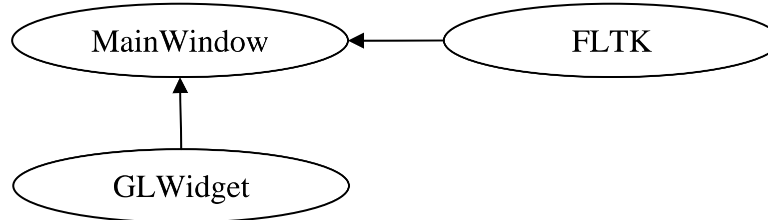

Figure 4.11: Class Structure for User Interaction

Figure 4.11 shows the class structure for the user interaction. As previously stated, there are three controls used to control the view of the data: a horizontal and vertical scrollbar, and a value slider. The scroll bars move about the data while the value slider controls the zoom, as seen in Figure 4.12. In addition, it is possible to select boxes within the view, which displays the list of Pairs contained in the selected boxes and allows you to get more information about a particular Pair.

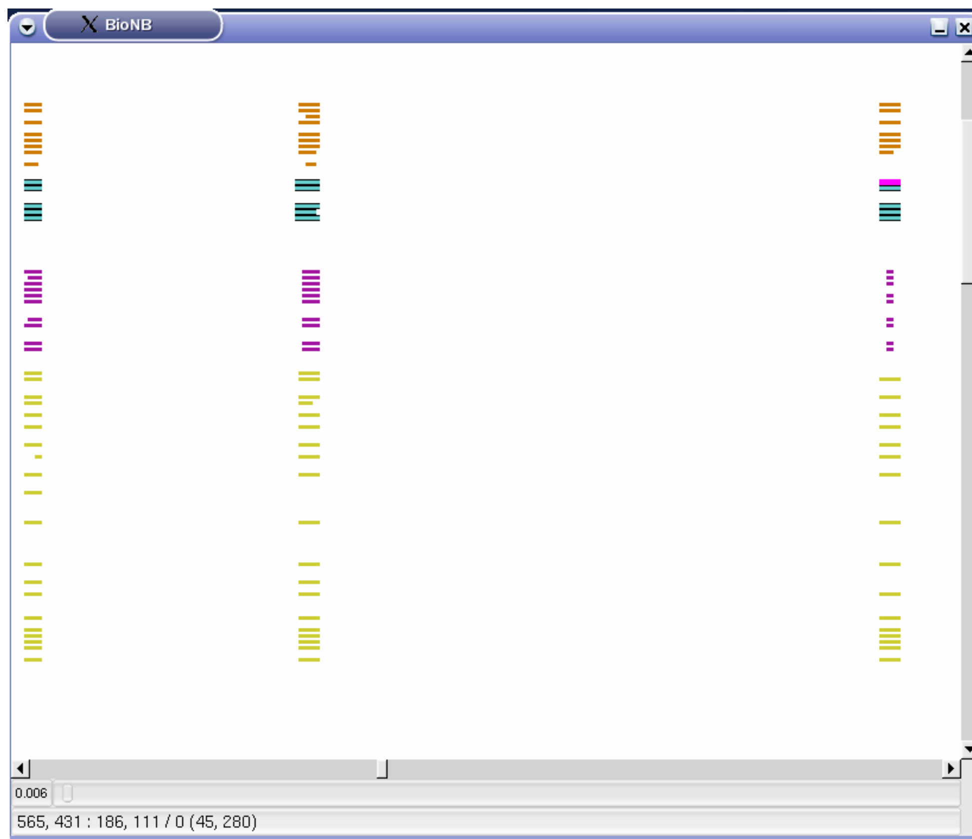

Figure 4.12: Zoomed Display

When clicking and dragging within the view area, the mouse clicks are converted to box coordinates, and then the PairRects within the appropriate BoxList have a boolean mSelected set to true. Then during drawing, any PairRect with mSelected as true gets a highlight drawn around it indicating that it is selected.

Right clicking the mouse will bring up a popup menu with some options. The user can change the color of the group currently under the mouse, zoom in on the currently selected area, or zoom fully out. The color changing feature was added to demonstrate the user customizable features that could be implemented in the program. The colors are stored within the database, so exiting and reopening the program will keep your new color preferences.

In a separate window, there is a list which contains all of the currently selected Pairs, as seen in Figure 4.13. Clicking on one will display more detailed information about that particular Pair in the textbox below the list. In addition, whichever Pair is selected in the list gets mHighlighted set to true in its PairRect, which allows it to get a unique highlight on the display showing where that particular Pair is located, as seen in Figure 4.13.

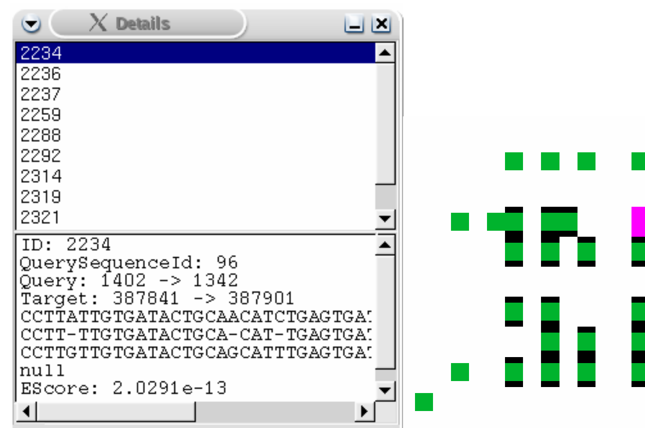

Figure 4.13: Details Window and Selection Highlight

Chapter 5

Conclusion

The purpose of this project was to convert the functionality of the prototype Java application into a C++ based application, using OpenGL for the display and extending the functionality using MySQL. The application currently replicates much of the basic functionality of the prototype, and it does so using OpenGL for the display. It just barely scratches the surface of added functionality through the use of the database, however. We were able to show that the use of a database would work, and provide benefits such as portable user profiles that would work from any workstation. Overall the project was successful in getting this program started. Much additional work is left to be done, though.

First the database design needs to be extended to be more flexible and store all the data that needs to be stored. Second, portions of the program need to be redesigned to better support the database. The first implementation was done using flat files and much of the program is designed to handle those more efficiently, rather then the database. After this, adding more and more features would be easier.

References

[1] König, Andreas, and Gröller, Eduard. 3D Medical Visualization: Breaking the Limits of Diagnostics and Treatment. 02 May 2006. <http://www.ercim.org/publication/Ercim_News/enw44/koenig.html>.

[2] R. Ogniewicz, M. Ilg. Voronoi Skeletons: Theory and Applications. In Proc. Computer Vision and Pattern Recognition, 1992, pp. 63-69.

[3] E. C. Sherbrooke, N. M. Patrikalakis, E. Brisson. An Algorithm for the Medial Surface Transform of 3D Polyhedral Solids. *IEEE Transactions on Visualization and Computer Graphics*, 1996, 2(1): 44-61.

[4] W. C. Ma, F. C. Wu, M. Ouhyoung. Skeleton Extraction of 3D Objects with Radial Basis Functions. In Proc. Shape Modeling International, Seoul, Korea, 2003, pp. 207-215.

[5] D. Paik, C. Beaulieu, R. Jeffery, G. Rubin, S. Napel. Automatic Flight Path Planning for Virtual Endoscopy. *Medical Physics*, 1998, 25(5): 629-637.

[6] M. Maddah, A. Afzali Kushaa, H. Soltanian-Zadeh. Fast Centerline Extraction for Quantification of Vessels in Confocal Microscopy Images. In Proc. IEEE International Symp. Biomedical Imaging Macro to Nano (ISBI)}, Washington, D.C., 2002, pp. 461-464.

[7] C. Min Ma, M. Sonka. A fully parallel 3D thinning algorithm and its application. *Computer Vision and Image Understanding*, 1996, 64(3): 420-433.

[8] G. Betrand, G. Malandain. A new characterization of three dimensional simple points. *Pattern Recognition Letters*, 1994, 15: 169-175.

[9] A. Vilanova, A. Konig, E. Groller. VirEn: A Virtual Endoscopy System. *Machines Graphics and Vision*, 1999, 8(3): 469-487.

[10] Jianfei Liu, Xiaopeng Zhang, Frédéric Blaise. Distance Contained Skeleton for Virtual Endoscopy. In Proc. IEEE International Symp. Biomedical Imaging Macro to Nano (ISBI)}, Arlington, Virginia, 2004, pp. 261-264.

[11] Y. Zhou, A. W. Toga. Efficient Skeletonization of Volumetric Objects. *IEEE Transactions on Visualization and Computer Graphics*, 1999, 5(3): 196-209.

[12] I. Bitter, M. Sato, M. Bender, K. McDonnell, A. Kaufman, M. Wan. CEASAR: A Smooth, Accurate and Robust Centerline Extraction Algorithm. In Proc. Proc. Visualization 2000, pp. 45-52.

[13] M. Sato, I. Bitter, M. Bender, A. Kaufman. TEASAR: Tree-structure Extraction Algorithm for Accurate and Robust Skeletons. In Proc. 8th Pacific conf. Computer Graphics and Applications, 2000, pp. 281-289.

[14] I. Bitter, A. Kaufman, M. Sato. Penalized-distance Volumetric Skeleton Algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 2001, 7(3): 195-206.

[15] M. Wan, Z. Liang, Qi Ke, L. Hong, I. Bitter, A. Kaufman. Automatic Centerline Extraction for Virtual Colonoscopy. *IEEE Transactions on Medical Imaging*, 2002, 21(12): 1450-1460.

[16] Scientific visualization. *Wikipedia*. 02 May 2006. <http://en.wikipedia.org/wiki/ Scientific_visualization>.

[17] Liu, Jianfei. Volume Decomposition and Hierarchical Skeletonization for Tree-like Objects. Unpublished report.

[18] VTK Homepage. 02 May 2006. < http://www.vtk.org/>.

[19] Fast Light Toolkit. 02 May 2006. <http://www.fltk.org/>.

[20] Bioinformatics. *Wikipedia*. 04 May 2006. <http://en.wikipedia.org/wiki/Bioinformatics>.

[21] OpenGL Overview. 04 May 2006. <http://www.opengl.org/about/overview/>.

[22] NCBI HomePage. 12 May 2006. <http://www.ncbi.nih.gov/>.

[23] W.E. Lorensen and H.E. Cline. Marching Cubes: A High Resolution 3D Surface Reconstruction Algorithm. *Computer Graphics.* Vol 21 no 4. July 1987.