Sensor Network Visualization

John Filippi

Advisors:

Dr. Kalpathi R. Subramanian

Dr. Asis Nasipuri

December 2005

Table of Contents

Chapter 1: Introduction

Municipal solid waste treatment is an important issue in almost all urban areas. Typically, management of solid waste involves direct burial into the subsurface, i.e. land filling. The main concerns with landfills include possible ground water contamination caused by escaping leachates and the generation of toxic gases such as methane that are generated as a byproduct of the biodegradation process [1]. To combat the problem of possible contamination, an enclosed structure is used to house solid waste, which prevents leachate from escaping. The ladder approach is known as the dry tomb concept. An alternative concept has also been conceived. The new concept, know as a bioreactor, entails speeding up the biodegradation process of the waste it contains by creating and maintaining favorable conditions for waste decomposition. Favorable conditions are throttled by controlled injections and extraction of air and moisture in a given bioreactor. With the help of computer based monitoring systems, these conditions can be logged to help determine the most favorable biodegradation settings.

With the advent of cheap and effective wireless networking technology, one can design a robust network infrastructure that performs the monitoring and control functions necessary to a given bioreactor. Numerous sensors are used to create an information subnet, linked together by adhering to the ad hoc based communication model. Information collected by the sensors is sent back to a centralized hub, where it is then forwarded to a given server for processing.

Visualization is used to visually describe the processed information originating from any given number of sensors. Since this research is relatively new, to our knowledge no software suite exists that performs the desired simulations and computations. In this project, we have developed three applications that enable the operator to visualize and manipulate the processed data originating from a given set of sensors. A number of API's have been used to develop the above applications. Such API's include the Fast and Light ToolKit (FLTK), the Visualization ToolKit (VTK), and Java 3D. VTK and Java 3D were used to perform visualization operations, where FLTK and Java Swing components were used to built the Graphical User Interface.

The underlying system has been successfully developed and tested. The visualization programs developed during this effort are leveraged by scientists to monitor conditions and perform error calculations. The remaining chapters in this report will go into more detail regarding the background, description, and specific implementation. Further more, conducted experiments are cited in conjunction with their results.

Chapter 2: Background

2.1 Bioreactor Landfill

A bioreactor in the context of this project can be thought of as a large apparatus containing a given amount of Municipal Solid Waste (MSW) (> 1600 kg). The waste is composed of steel, aluminum cans, glass, cardboard, newspaper, plastic bottles, and paperboard. The mix proportions were those of the average U.S. MSW composition reported by the U.S. EPA [1]. Such a structure has been conceived to provide a testbed for our experiments. A 10'x6'x5' bioreactor was designed within a geoenvironmental laboratory at UNC Charlotte with the primary objective of deriving models characterizing the effects of the flow of air and moisture through municipal waste [1].

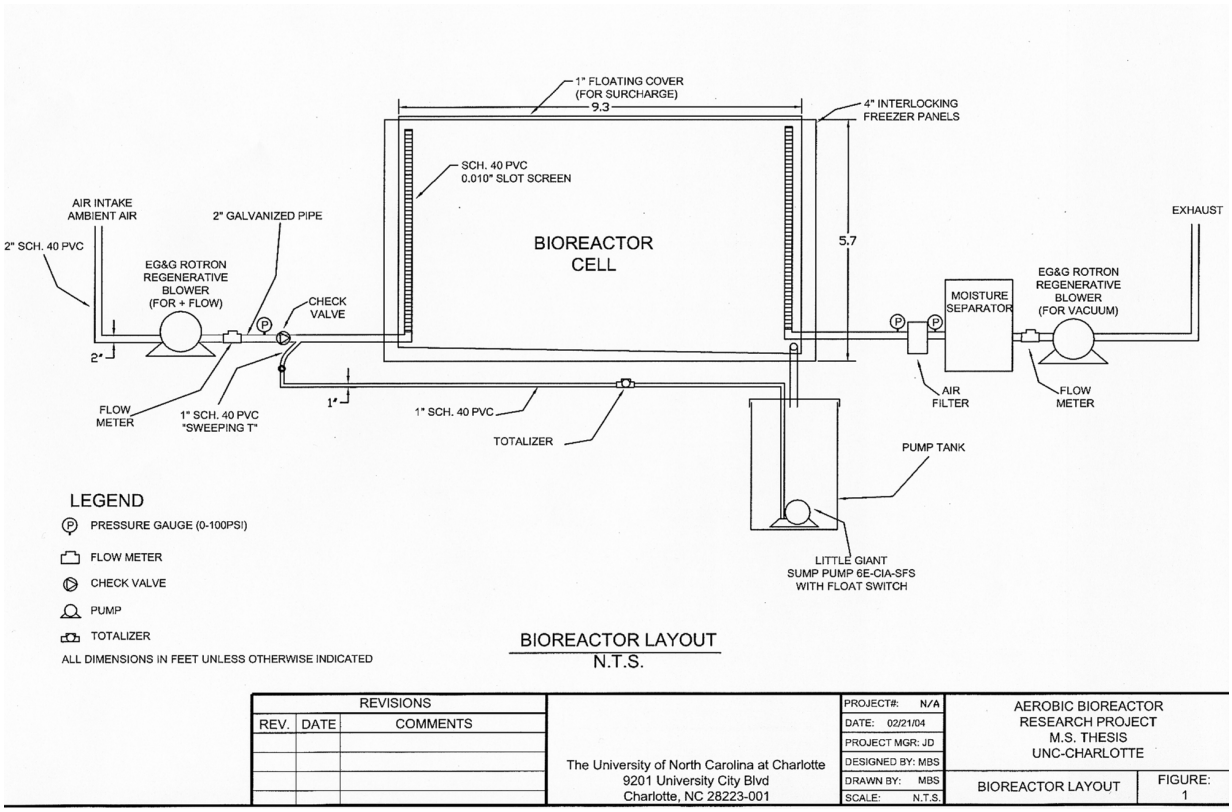


Figure 2.1: Overall process of the meso-scale bioreactor testbed

An elevation schematic of the system is shown in Figure 2.1. The vertical injection and extraction system is comprised of schedule 40 slotted PVC, and it is bolted through the interior of the bioreactor. Airflow is injected and extracted with 1.5 horsepower blowers that are connected to the PVC with quick-connect couplings. The leachate collection and re-circulation system also consists of schedule 40 slotted PVC and is located at the low point of the bioreactor floor. The leachate is collected and re-circulated into the waste through the injection well. An emergency leachate injection system was also constructed in the floating cover so that the leachate can be gravity fed if needed [1].

2.2 Wireless Sensor Network

In theory, the network of sensors that monitor a bioreactor will operate ad hoc. This means that the sensors discover other sensors within a given range. In order to transmit from point A to point C, point B may need to act as the middle man (see Figure 2.2). A sensor can search for target sensors by flooding the network with broadcasts, which in turn are forwarded by nearby sensors until the destination is reached. The network operates based on the IEEE 802.11 protocol. This protocol can further be broken down into a number of sub protocols. The 802.11b sub protocol is used in our experiment.

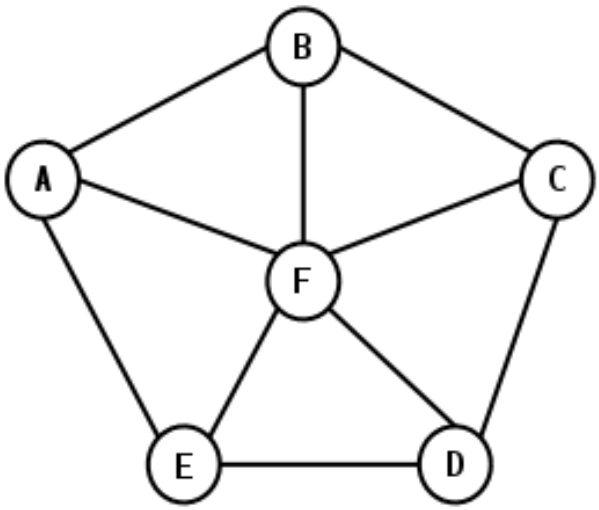


Figure 2.2: Sensor Network

2.3 Visualization

Visualization is the use of interactive, sensory representations, typically visual, of abstract data to reinforce cognition [4]. In the case of our research, visualization consists of parsing information collected from sensors encapsulated within the bioreactor, and displaying a visual end product in a virtual three dimensional environment.

As mentioned in the introduction, a number of API's were leveraged in conjunction with our research. To start, the Fast Light Toolkit (FLTK) was used to develop an intuitive Graphical User Interface. FLTK is an open source, cross-platform toolkit for C++ that provides a quick means to building robust user interfaces. The Visualization Toolkit (VTK) is an open source, cross platform for a number of languages, including C++, Java, TCL/TK, Python, and so on. VTK provides a means for the most commonly used visualization techniques, as well as support for 3D computer graphics and image

processing. Java 3D is an open source Java API that provides a means for 3D computer graphics and user interaction, and was used in place of VTK for one of the applications that we developed.

2.4 Marching Contour Algorithm

The marching contour algorithm can be thought of as a superset to the marching cubes algorithm. This technique is used in one of the projects that is presented later in this paper, to construct an isosurface based on interpolated temperature values. Unlike marching cubes, the marching contour algorithm does not require a given volume of data. Instead, contour data is used. The only stipulation is that the contour must be dense, and the cross-section distance should be close to the intra-section resolution [2]. The algorithm is described as follows: First, define the voxel value of contour point as 1, and otherwise 0. Second, for slice I after interpolating, we apply the marching cube algorithm only on those points in the contour [2].

The marching cubes algorithm is a two-step discretisation of a given isosurface. The algorithm can be described as follows:

(1) Subdivision of the display volume:

   a. The displayed volume is discretized into cube shaped elements which consist of six sides and eight nodes. The scalar field quantities are assigned to the vertices of the cube. The node densities can be classified

as either less or equal to the chosen isosurface density, or greater than the isosurface density.

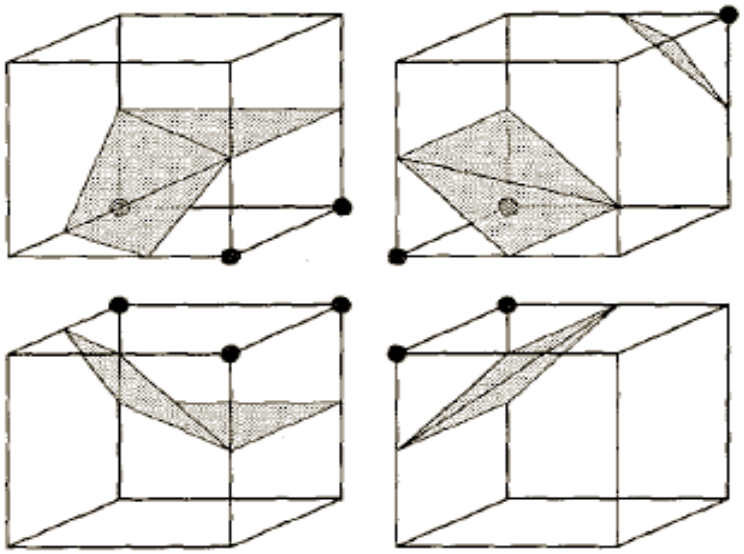


Figure 2.3: Triangulated isosurface of four neighboring cubes

(2) Triangulation of intersecting surfaces:

a. The intersection of the isosurface with the edges of the cube defines points of intersection. These crosspoints on the cube edges are calculated by linear interpolation between the node densities, and are the basis for the triangulation of the isosurface (see figure 2.3) [3].

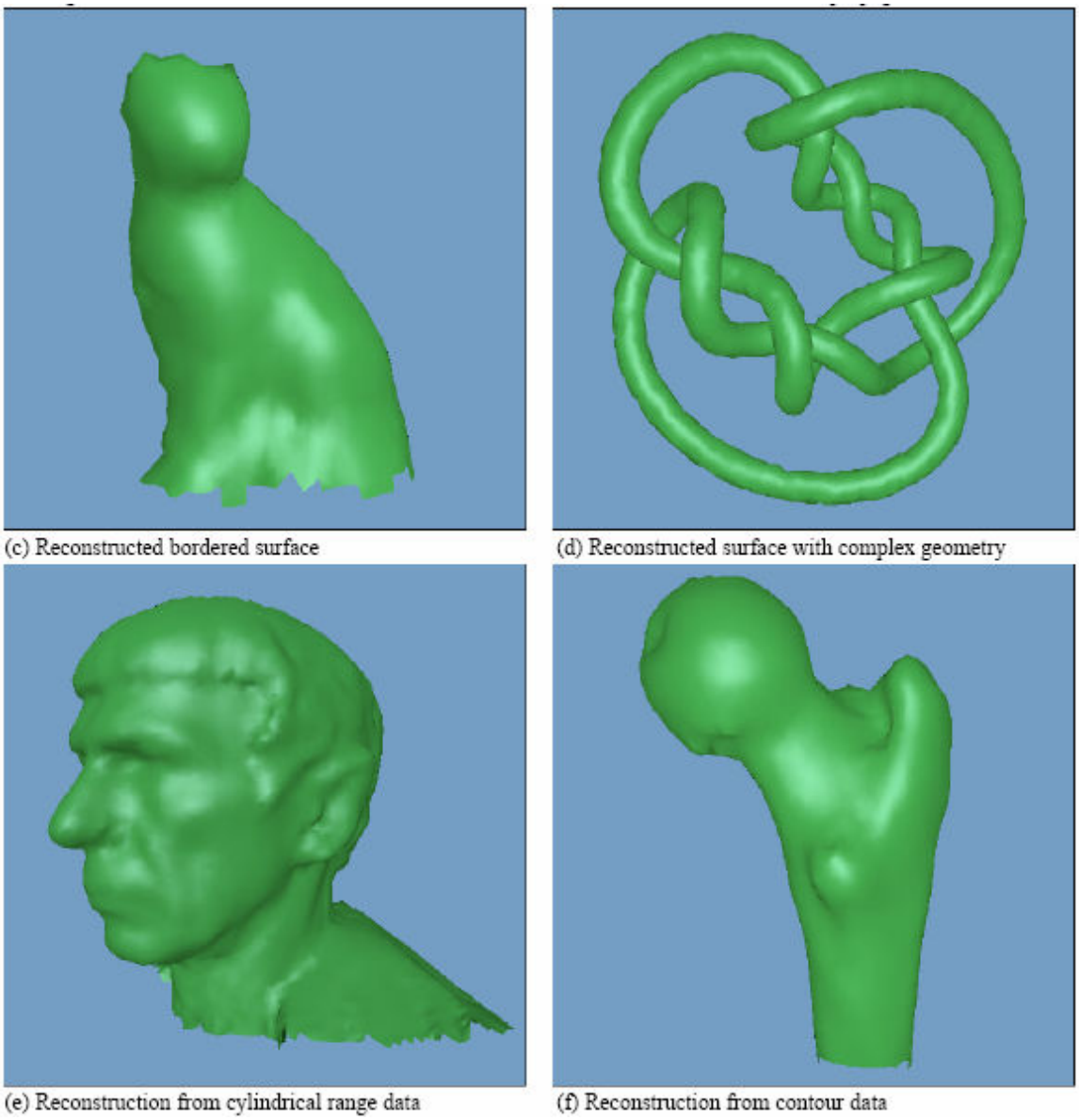
2.5: Surface Reconstruction from Unorganized Points

The algorithm described in this section takes as input an unorganized set of points $\{x_1,\ldots,x_n\}$ on or near an unknown manifold M, and produces as output a simplicial

surface that approximates M [5]. The algorithm has proven to be useful in the context of our work, and is used as a basis for Gaussian based mean squared error calculation (see Chapter 3). Given a set of contours, the application uses the Surface Reconstruction technique to reconstruct an isosurface resembling that of the given contours. Essential what this algorithm does is construct a surface based on given partial information of an unknown surface. One of the primary uses of the algorithm is to construct surfaces based on a set of contours, as is the case for our use of the algorithm.

Without going into detail, the algorithm consists of two stages, as follows. First, a function f is defined such that it estimates the distance to the unknown surface M. The estimate for M is defined by the zero set $Z(f)$. This estimation is defined in the second stage of the algorithm, where $Z(f)$ is approximated by a simplicial surface.

The end product of the algorithm is a reconstruction of a surface in 3D space with or without boundaries from a set of unorganized points scattered on or near the surface [5]. For some surface reconstruction examples, please refer to Figure 2.4.



(c) Reconstructed bordered surface

(d) Reconstructed surface with complex geometry

(e) Reconstruction from cylindrical range data

(f) Reconstruction from contour data

Figure 2.4: Reconstruction Examples

Chapter 3: System Description

3.1 Overview

The system described in this paper is comprised of a multi-tier set of components, including hardware systems, network architecture, and a software suite (See Figure 3.1). The hardware system serves as a means to monitor and collect data about a given environment. The network architecture serves as the gateway between the hardware systems and the software suite. The software suite consists of three separate applications, all of which perform some sort of visualization based on output from the hardware systems.

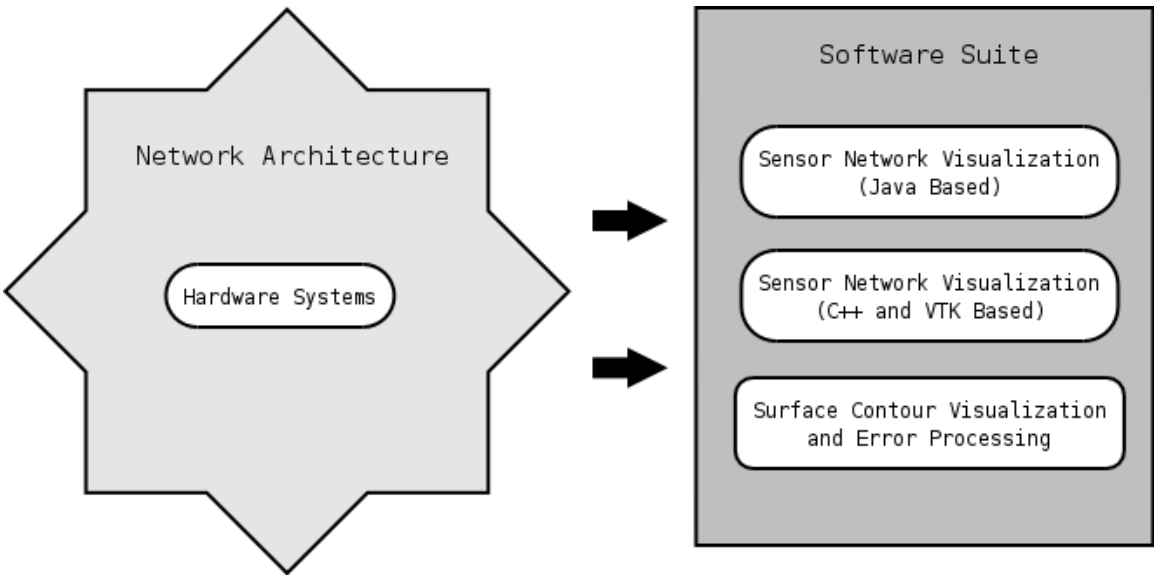


Figure 3.1: System Overview

3.2 Network Architecture and Hardware Systems

The network architecture and hardware systems schematic are depicted in Figure 3.2. The exact locations of the sensors seen in Figure 3.2 are given in Table 3.1. At each sensor location, there exists one temperature sensor and one humidity probe. The system is comprised of 12 temperature sensors and 12 moisture probes, located on three geogrids. All sensors are wired to a nearby wireless node (mote) that is equipped with a data acquisition board. The motes are programmed to periodically sample the sensor observations and wirelessly transmit the data to a wireless-to-Ethernet interface board located in the laboratory. The interface board provides connectivity to the campus local area network (LAN). A networked computer serves as the central monitoring station and database, which runs an application program to receive all data from the interface board and save them in appropriate data files. Since multiple sensors may be connected to each mote, the motes multiplex the data from each input (sensor) and transmit the collective information together to the interface boards. Appropriate demultiplexing is performed at the monitoring station to identify the sources of various data streams [1].

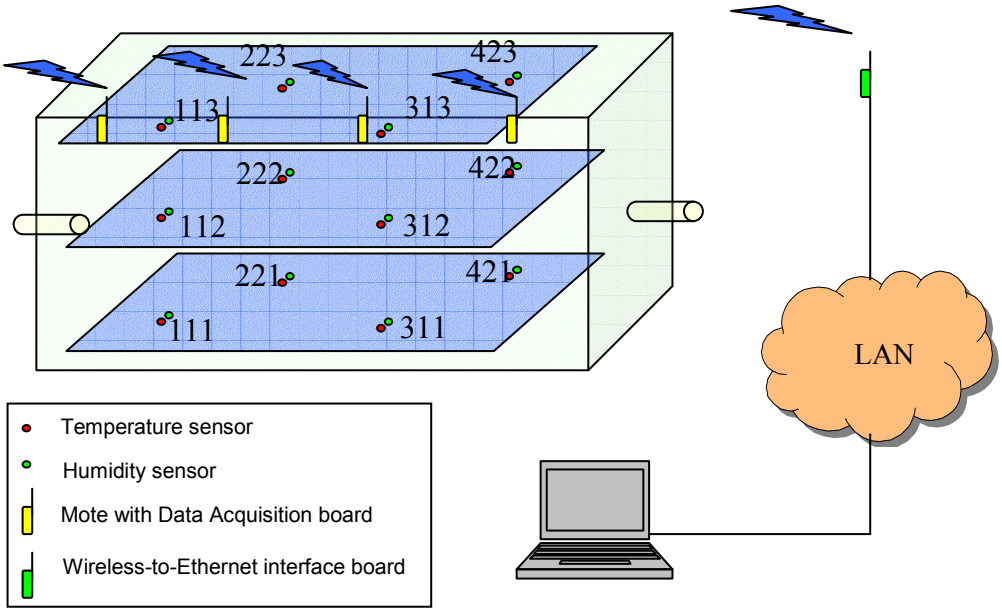


Figure 3.2: Schematic diagram of the network architecture and hardware systems

Bottom geogrid				Middle geogrid				Top geogrid			
ID	x	y	z	ID	x	y	z	ID	x	y	z
111	1'2"	1'3"	9"	111	1'2"	1'3"	9"	111	1'2"	1'3"	9"
221	3'10"	2'4"	9"	221	3'10"	2'4"	9"	221	3'10"	2'4"	9"
311	6'7"	1'6"	9"	311	6'7"	1'6"	9"	311	6'7"	1'6"	9"
421	9'1"	2'1"	9"	421	9'1"	2'1"	9"	421	9'1"	2'1"	9"

Table 3.1: Sensor Locations

The wireless components used are second generation experimental embedded devices manufactured by Crossbow with technical collaboration with UC Berkeley. Each mote is equipped with an AtMega128L micro-controller, program memory, 10-bit analog to

digital converter (ADC), and 433 MHz ISM band radio. The transmission range of the radio transceiver at the motes is between 100 – 200 feet using a 6.8 inch whip antenna. The motes use a specially designed open-source software platform called TinyOS that provides a convenient platform for developing applications for performing various sensor signal processing and wireless communications/networking. A complete list of hardware components used in the system is given in Table 3.2 [1].

Description	Hardware	Manufacturer
Temperature probe	108 probe	Campbell Scientific
Humidity probe	Echo20	Decagon
Wireless processor/radio node (mote)	MPR410	Crossbow
Data acquisition board	MDA300	Crossbow
Wireless-to-Ethernet Interface	MIB510	Crossbow

Table 3.2: List of hardware components use

3.3: Bioreactor Monitoring and Visualization

3.3.1: Overview

Both the Java based version and the standalone C++/VTK version of the sensor network visualization application perform visualizations based on sampled sensor temperature

distributions over time (4D). The standalone C++/VTK version allows for more visualization options, where the Java version allows for better portability. In fact, the Java application can be embedded into any webpage.

Both versions of the application render 12 spheres, representing the 12 sensors located within the bioreactor. A large rectangular cube representing the bioreactor is also drawn (Refer to Figure 3.2). Three 3D transparent planes are also drawn in order to represent the three geogrids. Over time, the spheres are shaded based on the given sampled temperature of a sensor at the given time offset. The color is clamped to a user defined range, given by the following formula: $(\partial - Color_{min})/(Color_{max} - Color_{min})$, where ∂ is the temperature at a given time offset and $Color_{min}$ and $Color_{max}$ are defined through the GUI (See Figures 6 and 7).

The use of these applications has proven helpful in monitoring how the conditions in the bioreactor change with respect to time. Instead of looking at raw data, one can simply load a given dataset into one of the applications and observe the actual temperature distributions changing over time.

3.3.2: File Format: Sensor Position File

The file format for denoting the location of a given sensor in 3D space is straightforward. The sensor ID and the x, y, and z components are separated by tab delimitations, where each line represents a different sensor.

e.g.

iID_0 fX_0 fY_0 fZ_0
iID_1 fX_1 fY_1 fZ_1

3.3.3: File Format: Sensor Temperature Over Time File

Each sensor has a corresponding file defining the temperatures sampled over time. There are a total of 12 files, corresponding to the 12 sensors in the bioreactor, in any given dataset. The file itself contains two entries per line, the time of sampling and the temperature sample at the given time. Entries are delimited by tabs.

e.g.

$fTimeOffset_0$ $fTemperature_0$
$fTimeOffset_1$ $fTemperature_1$

3.3.4: Sensor Network Visualization (Java Implementation)

By using Java technology as a basis for implementation, one can easily run this application from any operating system with an available JVM. In the case of our implementation, we have embedded the application into a webpage, using Java Applet technology.

Because Java Applets execute on the client side, and the data files are located on the server side, we had to implement a means to break out of the Java security sandbox. To do so, we signed out Applet with a 128-bit key. The end user is presented with a dialog box to confirm lifting the security restrictions. Once access has been granted to access external resources, the client pulls the data from the defined data sets located on the server.

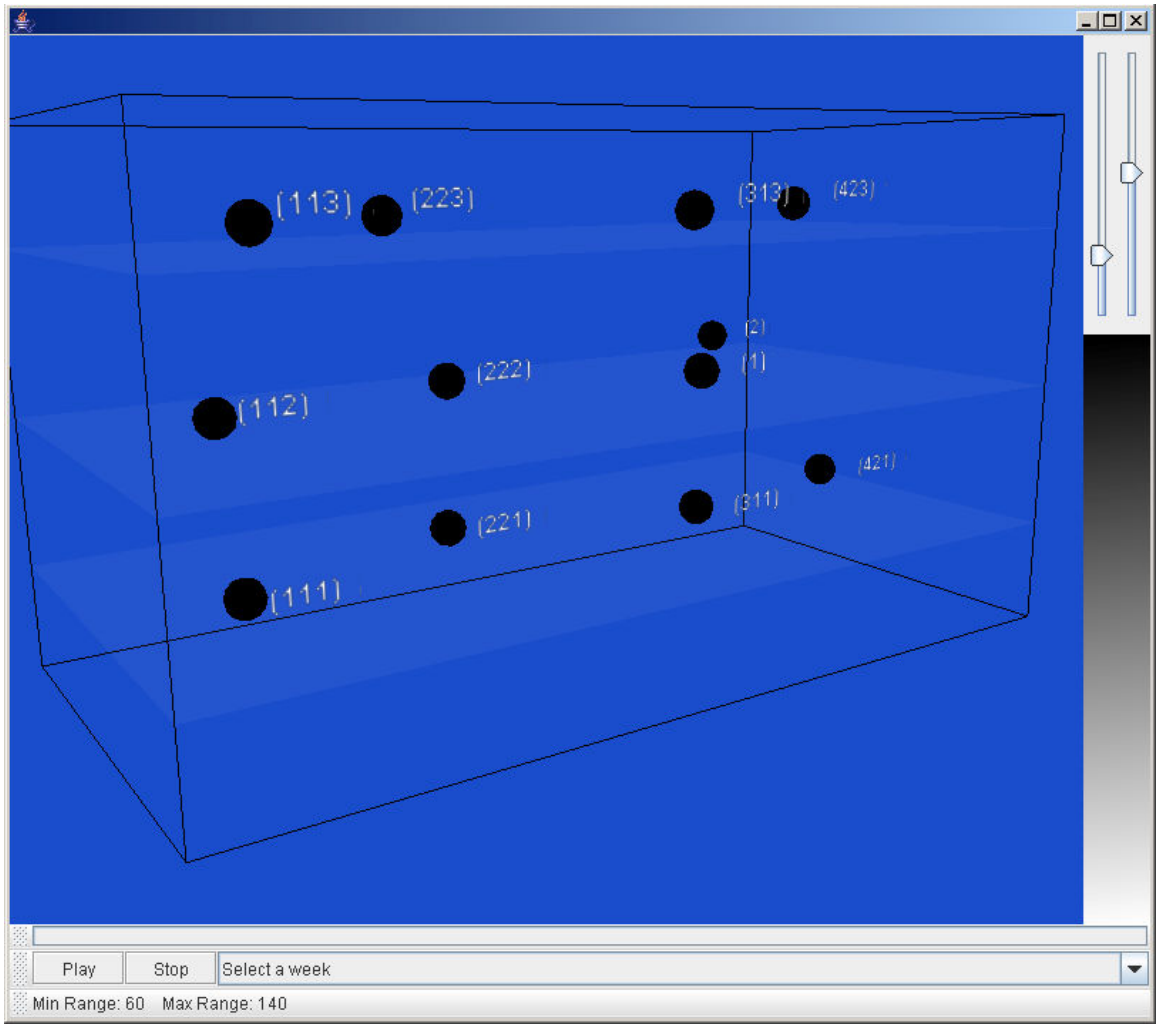


Figure 3.3: Sensor Network Visualization (Java Implementation)

The user interface features an intuitive VCR like control system that allows for the end user to control the state of the time offset. A progress bar at the bottom of the Applet displays the overall progress of the animation with respect to the current time offset. A gradient shaded pane is also rendered on the side of the application, to visually depict the temperature range. Slider bars provide the means to control the desired temperature range.

The Java 3D API was used as a means to manage and render our 3D environment. Key features of this API include scene graph based actor management and a comprehensive class library. The API provided most of the classes necessary for this project; however subclasses had to be implemented in some cases to perform a desired functionality (see Chapter 4).

3.3.5: Sensor Network Visualization (C++/VTK Implementation)

This version of the application contains all of the features implemented in the Java version, along with a number of enhancements. VTK is utilized to visualize temperature contours, using the Marching Contours Algorithm (see Chapter 2). A separate control window allows for precise control over a number of variable features. Such features include: min color range, max color range, contour threshold, time interval, vcr controls, and actor visibility toggles (Refer to Figure 3.4).

Because C++ lacks a standard 3D API as complete as Java 3D, VTK is employed. Similar to Java 3D, VTK boasts a comprehensive class library. As mentioned earlier, the FLTK library was used to build the GUI. Widgets such as menu bars, progress bars, value bars, and buttons are used to capture and display input and output.

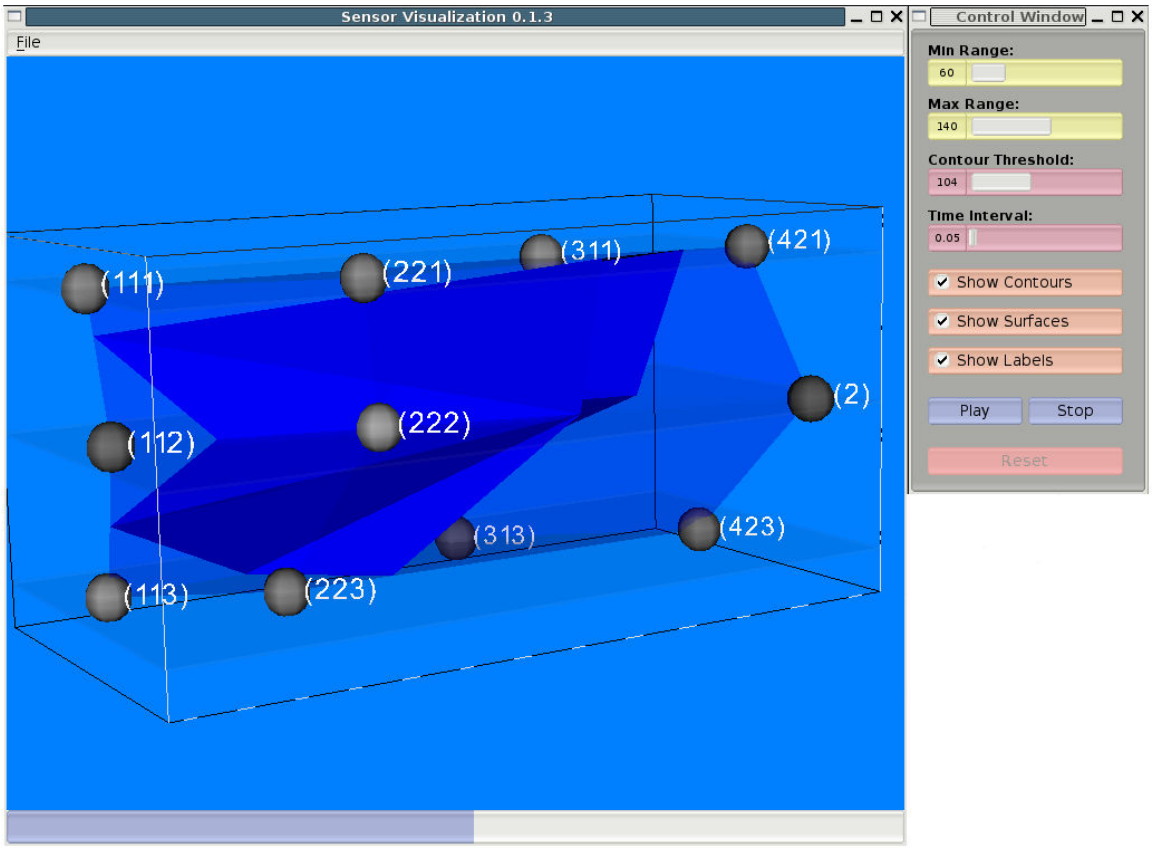


Figure 3.4: Sensor Network Visualization (C++/VTK Implementation) GUI

Initially, the operator loads the desired dataset into the application. After doing so, he plays the animation by clicking the Play button located on the control window GUI. This triggers a new thread to be spawned, which controls the operations relating to the animation. During each frame of the animation, the spheres (sensors) are shaded corresponding to their sampled temperatures at the current time offset. Also occurring during each frame, the marching contour surface is re-rendered based on the new temperature distribution. This process continues until the last frame in the animation is reached.

3.4: Surface Contour Visualization and Error Processing

3.4.1: Overview

The Surface Contour Visualization and Error Processing application was designed to calculate the mean squared error based on a given set of contours. A Gaussian density function is used for error evaluation, though any smoothly varying function could be used. The application features an FLTK based GUI, with VTK based rendering.

When the application is first loaded, the operator is presented with a Gaussian based generated surface. The Gaussian surface can be adjusted to any resolution, and is defined by the following algorithm:

```
float fSpacing = 100.0f/iRows;
float fX=0.0, fY=0.0;

float* pPoint;

for(int i=0; i<iRows; i++) {
  fY = 0.0f;
  for(int j=0; j<iCols; j++) {
    pPoint = new float[3];
    pPoint[0] = fX;
    pPoint[1] = fY;
    pPoint[2] = fGetDensity(fX, fY);

    //Add point to vector
    m_vAllPoints.push_back(pPoint);

    fY += fSpacing;
  }

  fX += fSpacing;
}
```

Note that `iRows` and `iCols` are equal to each other, and can be controlled with a value slider located on the GUI. The Gaussian density value is obtained by calling `fGetDensity()`, and is defined as follows: $100 * e^{\frac{-2|(x_0-50)^2+(y_0-50)^2|}{2500}}$

Edge sensor locations are loaded from a given dataset (see Section 3.4.2). Multiple edge sensors make up any given contour. Each contour has a file defining the edge sensors that belong to it. Once the contours are loaded, graphical representations of each sensor are rendered. A small sphere is rendered at each of these locations. Next, a volume is reconstructed based on the dataset of contours loaded in the previous step. The reconstruction process is directly based on the research of Hugues Hoppe [5] (see Chapter 2).

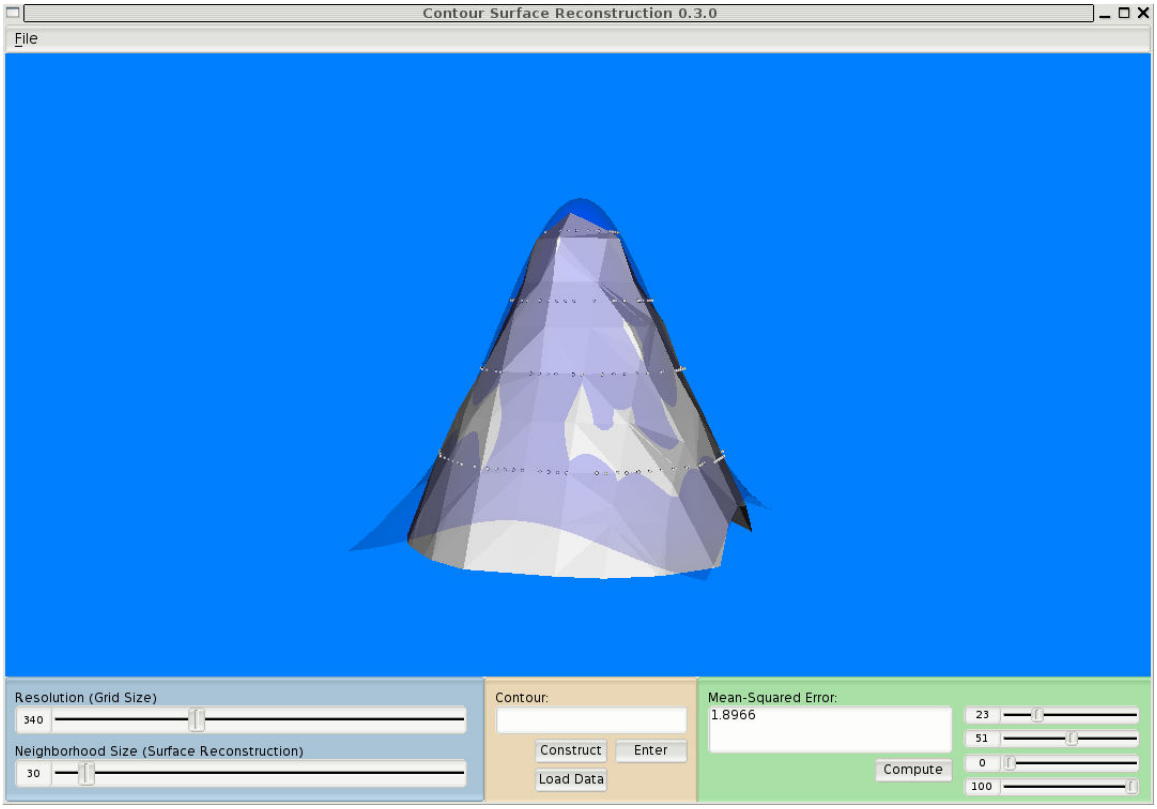


Figure 3.5: Surface Contour Visualization and Error Processing

Once the surface has been reconstructed, a set of points defining the surface is extracted. Using these points, the mean squared error is calculated. The error for any particular point P is defined as the difference between the P_z and `fGetDensity(`P_x`, `P_y`)`. The equation is defined as $\sqrt{\frac{\sum(\omega-\Phi)}{n}}$, where w is P_z, Φ is `fGetDensity(`P_x`, `P_y`)`, and n is the number of points sampled. Remember that `fGetDensity()` returns the Gaussian density value at point P. The algorithm is defined as follows:

```
int iCounter=0;
float fDifference, fAverage=0.0f, fX, fY, fZ;
for(int i=0; i<m_pSurfacePoints->GetNumberOfPoints(); i++) {
  fX = m_pSurfacePoints->GetPoint(i)[0];
  fY = m_pSurfacePoints->GetPoint(i)[1];
  fZ = m_pSurfacePoints->GetPoint(i)[2];

  if((fZ >= fMinX) && (fZ <= fMaxX) &&
     (fY >= fMinY) && (fY <= fMaxY)) {
    fDifference = m_pSurfacePoints->GetPoint(i)[2]-
                  (fGetDensity(fX, fY));
    fDifference *= fDifference;

    fAverage += fDifference;
    ++iCounter;
  }
}

fAverage /= iCounter;

return sqrt(fAverage);
```

The `GetNumberOfPoints()` function returns the number of points extracted from the reconstructed surface. The conditionals located within the for loop allow the operator to clamp the error calculation to a defined subset of the volume. The minima and maxima for multiple axes can be controlled via value sliders on the GUI. The end result of this

entire process is the calculated mean squared error, which is displayed in a text box located on the GUI.

3.4.2: File Format: Surface Contour File

Each contour has a corresponding file containing the edge sensors that define it. Note the camera is initially points down the negative y axis, which means the z axis is running perpendicular, in the up direction. File names contain the z coordinate for a given contour. For example, a contour C_0 with z=50 is named cont50. Each line in the file represents an edge sensor on C_0. Each line has two tokens, x and y, which are tab delimited. Note that a contour can have any given number of edge sensors, and the more contours that exist in a particular dataset the more accurate the reconstructed surface will be.

e.g.

fX_0 fY_0
fX_1 fY_1

Chapter 4: Implementation

4.1: Class Hierarchy and Visualization Pipeline

4.1.1: Sensor Network Visualization (Java Implementation)

The class structure illustrated above is fairly straight forward. The GUI is defined and constructed in CMain, which is a sub class of JApplet. Event handling takes place in CMain as well. Because Java doesn't offer a gradient panel by default, the custom component CGradientPanel was created by sub classing the JComponent class.

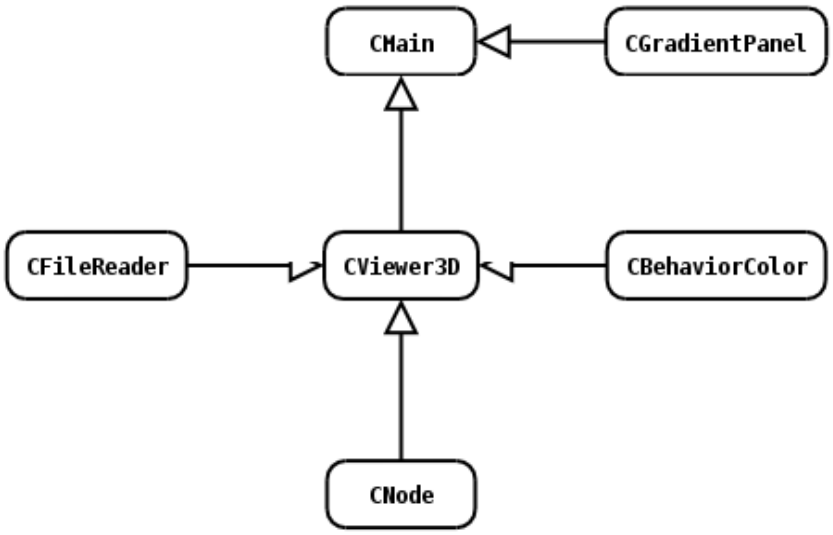


Figure 4.1: Class Hierarchy, Sensor Network Visualization (Java Implementation)

CFileReader is used to parse a number of files, including the positions file and the sensor temperature over time file. CNode is a representation of a sensor, and encapsulates the operations a sensor must perform. Finally, a thread based approach was conceived to process the sensor animations every .15 second. This was accomplished by sub classing

the Behavior class, which the Java 3D API defines. Essentially, a stimulus is processed when .15 seconds has elapsed. The stimulus is defined such that a given sensor is shaded corresponding to the temperature at the given time offset.

CViewer3D contains all the code necessary for rendering the scene. Actor management, rendering logic, and user based transformations are defined. The Java3D pipeline as implemented is CViewer3D is illustrated in Figure 4.2. Notice all components eventually link to a BranchGroup. The BranchGroup component is an implementation of a scene graph based actor management scheme.

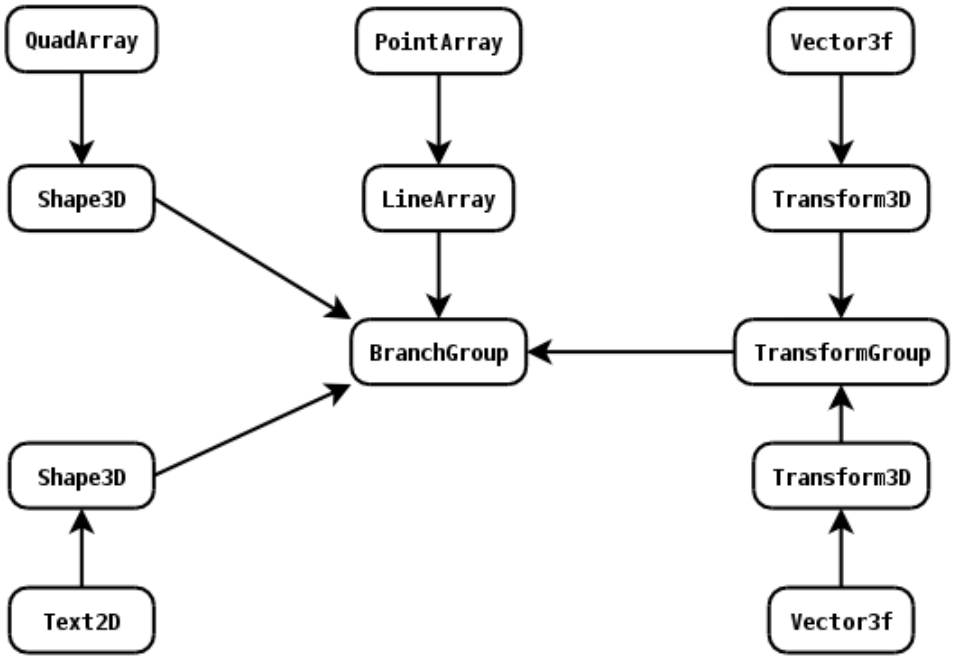


Figure 4.2: Java3D Pipeline, Sensor Network Visualization (Java Implementation)

4.1.2: Sensor Network Visualization (C++/VTK Implementation)

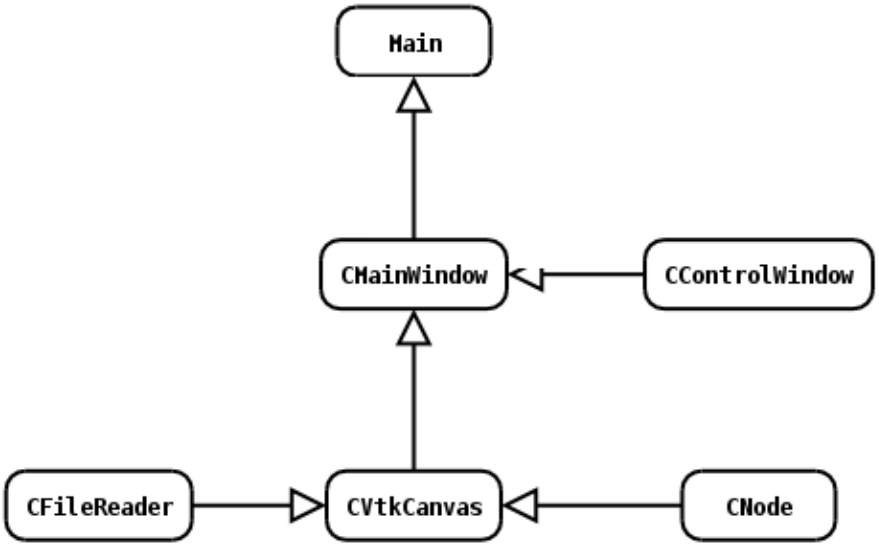


Figure 4.3: Class Hierarchy, Sensor Network Visualization (C++/VTK Implementation)

The structure illustrated in 4.3 is very similar to that of Figure 4.1, with the exception of implementation based specifics. Main acts as the executable insertion point, where it then calls CMainWindow to load the GUI. CMainWindow defines the main GUI and is responsible for managing the events generated by the widgets it encapsulates. Similar, CControlWindow is responsible for creating and managing the widgets associated with the control window. Like CViewer3D in the previous implementation, CVtkCanvas contains all the code necessary for rendering the 3D scene. The VTK pipeline defined in CVtkCanvas is illustrated in Figure 4.4.

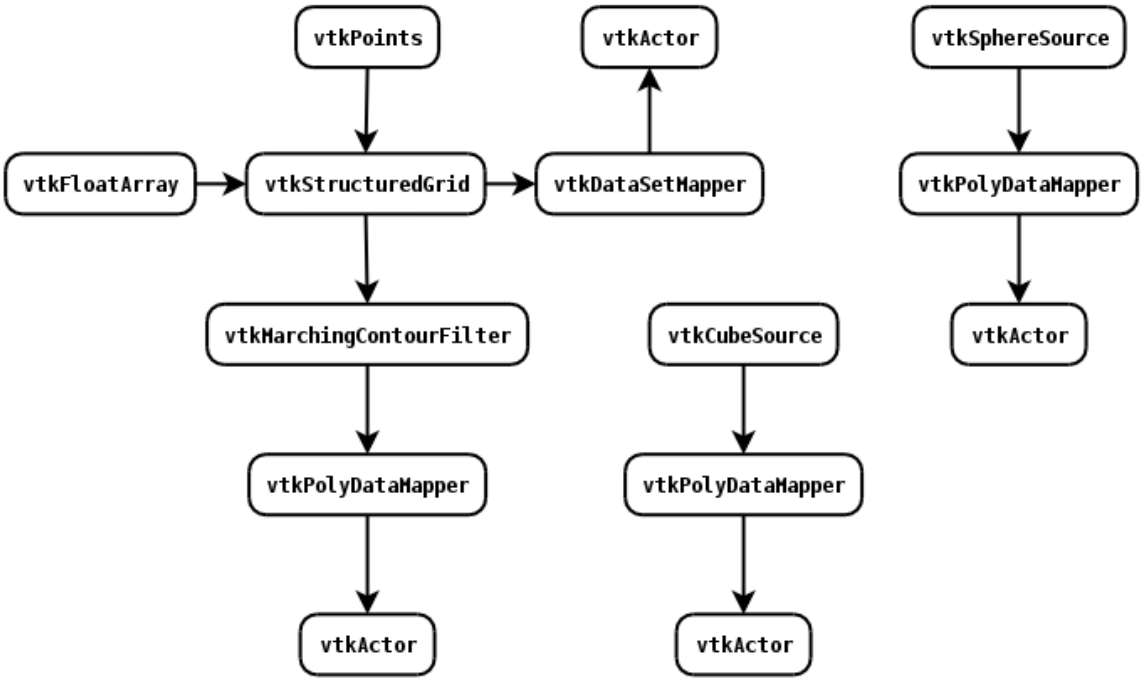


Figure 4.4: VTK Pipeline, Sensor Network Visualization (C++/VTK implementation)

4.1.3: Surface Contour Visualization and Error Processing

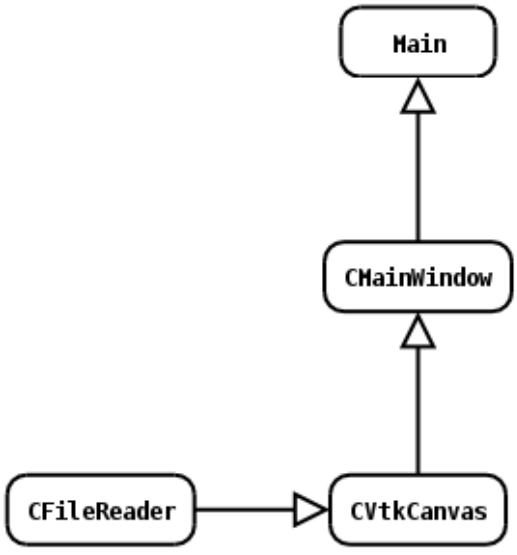


Figure 4.5: Class Hierarchy, Surface Contour Visualization and Error Processing

The class hierarchy is virtually identical to that presented is section 1.2, with the exception of a few missing classes. However, CVtkCanvas in this application implements a completely different VTK pipeline, as shown in Figure 4.6.

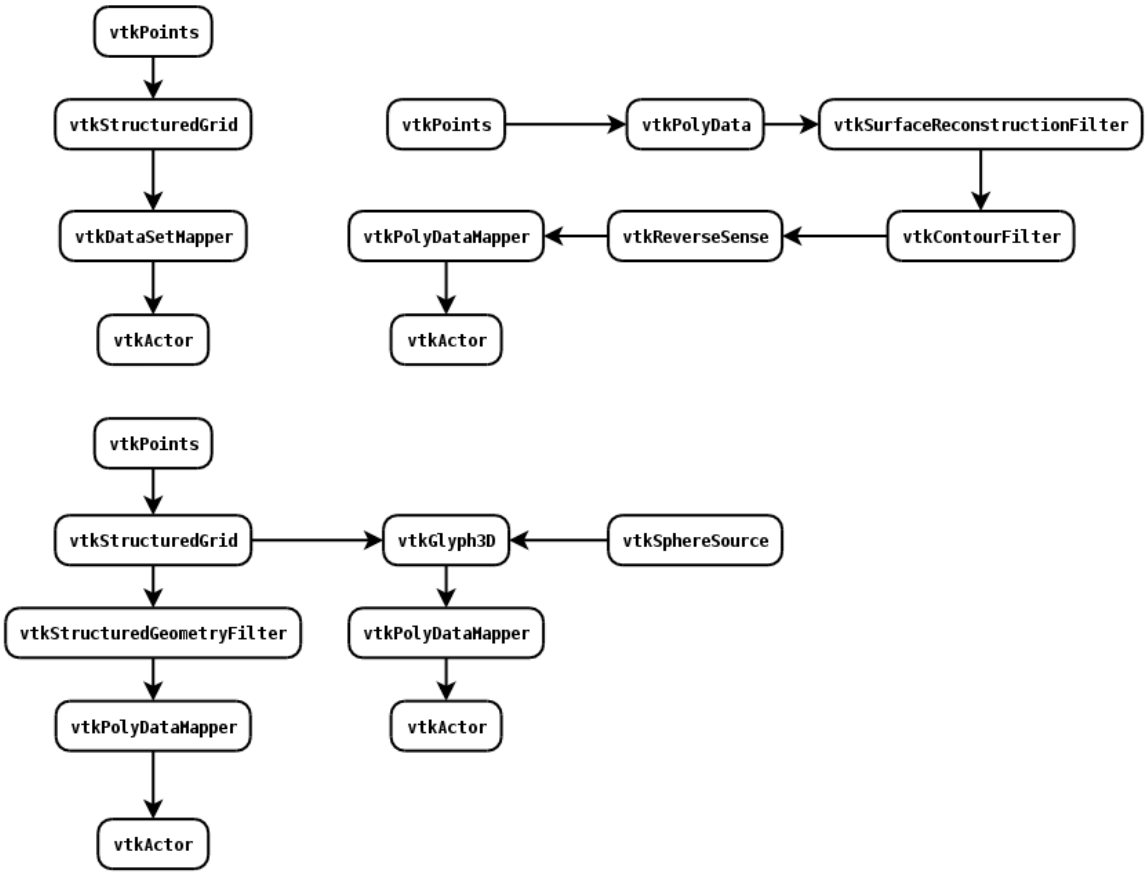


Figure 4.6: VTK Pipeline, Contour Surface Visualization and Error Processing

Chapter 5: Experiments

5.1: Experiment 1: Sensor Network Visualization

Temperature readings obtained at sensors 221, 222, and 223 for weeks starting 10/20/2004 and 11/05/2004 are plotted in Figure 5.1. The rapid fluctuations of temperature are due to finite precision error caused by the ADC. The plots indicate the difficulty in assessing the condition inside the bioreactor by observing variations of signals over time from a few sensors. In Figure 5.2 we show snapshots taken from the 3D visualization tool that uses volume rendering to construct and update the surface depicting the boundary where the temperature crosses 100°F. The snapshots show how the three-dimensional region where the temperature exceeds 100°F varies with time [1].

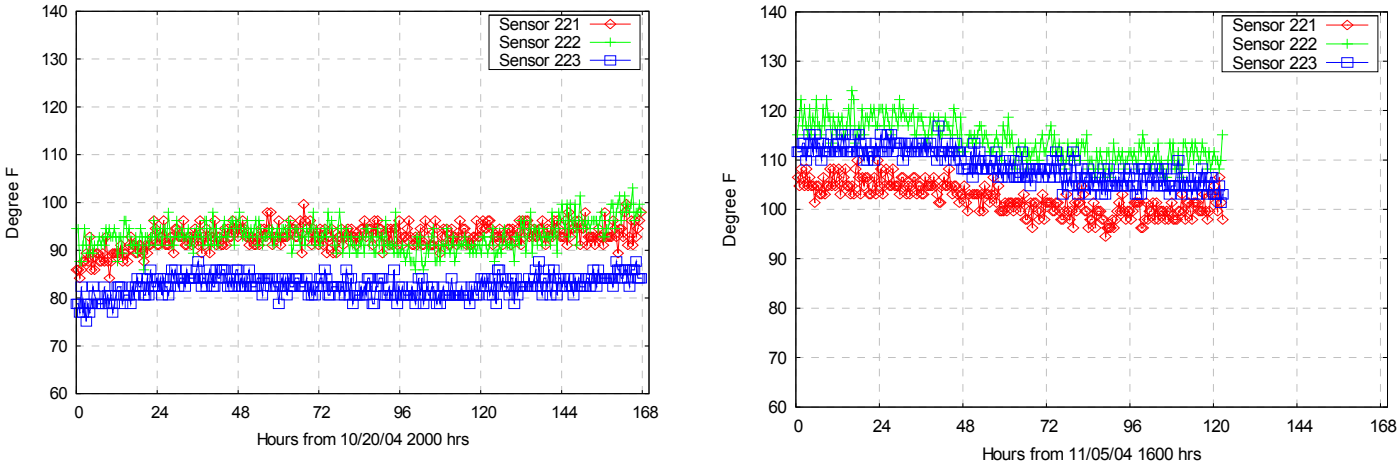
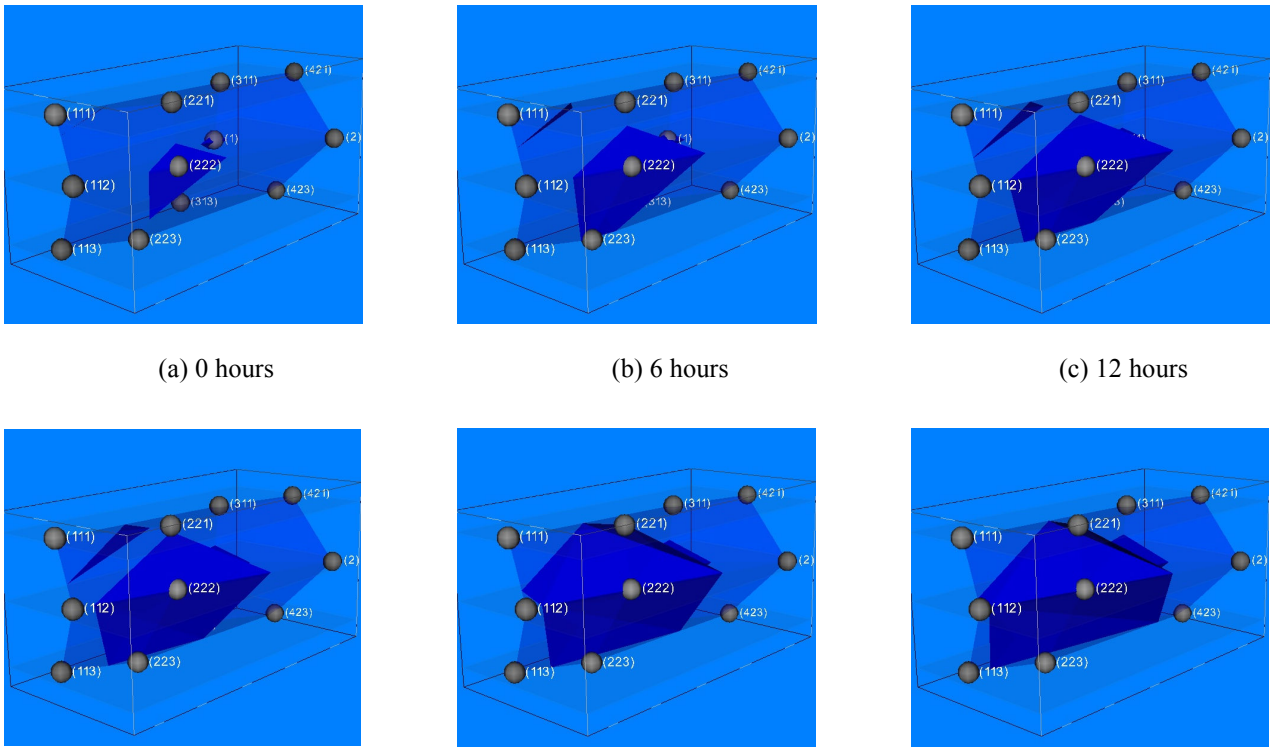


Figure 5.1: Temperature observed at three centralized sensors over one week periods.



(a) 0 hours (b) 6 hours (c) 12 hours

(d) 18 hours (e) 24 hours (f) 30 hours

Figure 5.2: Snapshots from an online video that shows the variation of a surface enclosing the region within which temperature>102°F

5.2: Experiment 2: Surface Contour Error

The objective of this experiment was to calculate the mean squared error for a number of datasets, using the same set of parameters. Such parameters include Gaussian resolution, reconstruction neighborhood size, and z based range clamping. The most ideal settings were discovered by means of trial and error. The results are presented in Figure 5.3, where noise variance is plotted versus the mean squared error.

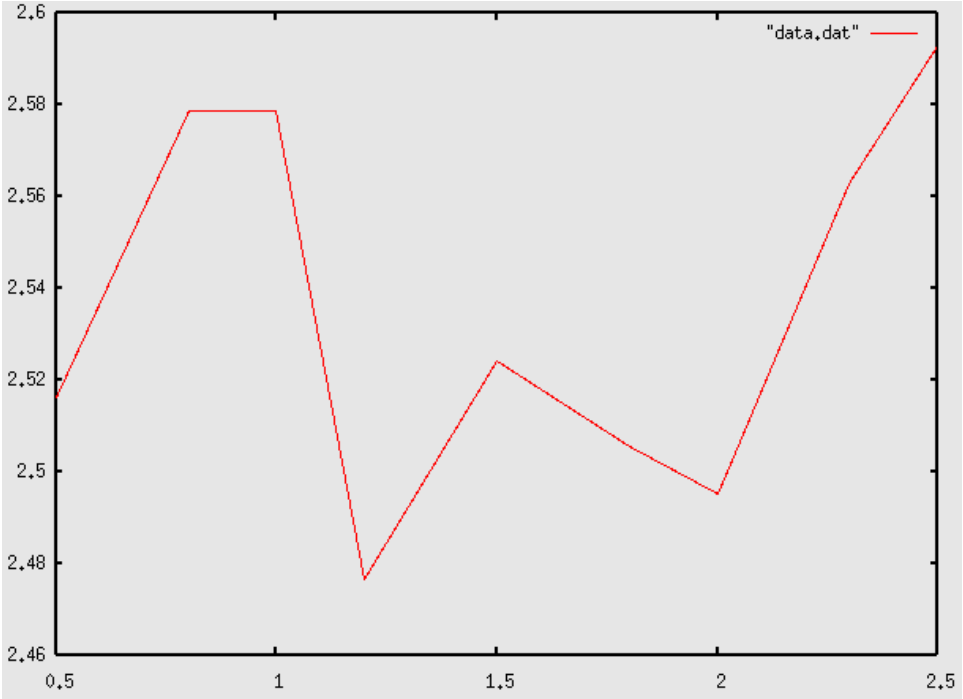


Figure 5.3: Noise Variance (y axis) vs. Means Squared Error (x axis)

Chapter 6: Conclusion

The overall goal of this project was to build sensor network visualization tools that parse time varying data being generated from the UNC Charlotte bioreactor, and in turn generate visual output that allows for further testing. The suite of software applications presented in this paper does just that. Using these applications, one can monitor the temperature of the bioreactor and perform error calculations based on edge sensor locations. Being able to visually observe changes in a given set a sensors over a period of time is infinitely better or more advantageous than looking at raw data output.

The next step that needs to be taken is enhancements to the software applications described in this paper. Though useful, all of the applications need to be subjected to further GUI enhancements to improve usability. It also may be useful to port the Contour Surface application to Java, where it could be easily accessible through any java supported web browser. For the Surface Contour Visualization and Error Processing application, the implementation of time varying rendering would be useful. The framework is already in place for such an implementation.

References

[1] Nasipuri, A., and Subramanian, K. “Development of a Wireless Sensor Network for Monitoring a Bioreactor Landfill.” <http://www.ece.uncc.edu/~anasipur/pubs/geo06.pdf>

[2] Guoyan Zheng, Shuxiang Li, and Jingdong Yan. “Engineering in Medicine and Biology Society.” 1998. Proceedings of the 20th Annual International Conference of the IEEE Volume 2, 29 Oct.-1 Nov. 1998 Page(s):594 - 597 vol.2

[3] Bartsch, M., Weiland, T., and Witting, M. " Generation of 3D isosurfaces by means of the marching cube algorithm.” 1996. Magnetics, IEEE Transactions on Volume 32, Issue 3, Part 1, May 1996 Page(s):1469 - 1472

[4] “Visualization – Wikipedia, the free encyclopedia.” Wikipedia. 2005. 19 November 2005 < http://en.wikipedia.org/wiki/Visualization_%28graphic%29>

[5] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. “Surface reconstruction from unorganized points.” In Computer Graphics (SIGGRAPH ’92 Proceedings), volume 26, pages 71–78, July 1992.