A Summary of the Cat Motoneuron Project for ITCS 4140 Spring 2003

Matt T. Miller
Computer Science - UNC Charlotte
mtmiller@uncc.edu

Abstract:

The goal of this project is to clearly visualize the activity of motoneuron populations in a 3D model of a cat spinal cord in a temporal and spatial context. This is accomplished using data from biomedical publications provided by Dr. David Bashor (Biology, UNC Charlotte), which propose locations in the spinal cord for several motoneuron populations, along with the shape of the spinal cord, at 36 slices taken at specified points along the cord. Each motoneuron population is represented as a set of 100 small cubes, each cube representing a neuron, where each population is assigned a color. The neuron-cubes (referred to as "neurons" for the remainder of the text) are set to a very low opacity (but greater than 0.0). A given neuron will become completely opaque during a time step in which a spike occurs for that neuron. Our data (currently) uses 1500 time steps and 4 motoneuron populations. The 3D visualization was implemented using AC3D to model polylines over each of the 36 slices, which were imported into the C++ application using the Visualization Toolkit (which was used for building all of the 3D visualization)[5] where they were interpreted together as a structured grid. Colored cubes represent the individual neurons in the populations, each population having a unique color representing it. A graph on the lower-left shows, for each population, the sum of its active neurons over the last 100 time steps. A ball and stick diagram of a limb on the bottom-center shows the position of the limb at any time step. These three views show the relationship between the position of the limb and which neuron population is active, while showing the position in the spinal cord of the active neurons. The user interface was implemented with the Fast-Light Toolkit for C++ [4].

1. Introduction

I began this project knowing very little about 3D visualization, and during a semester of guidance by Dr. K.R. Subramanian, I was able to create this visualization program, and we (Dr. Subramanian, Dr. Bashor, Josh Foster, and I) were able to put out a paper at the end of March, 2003. This project extends the work documented in [2] by implementing a new interface that visualizes the spike-activity of motoneuron populations during locomotion over time in a 3D anatomical model of a cat spinal-cord with the neurons distributed inside. The interface in [2] visualizes more information than the new interface, but the information is not rapidly consumable, due to the lack of a spatial context. The information visualized in the new interface, albeit currently less information than the old interface is able to be consumed very rapidly, not only because of the spatial context, but also because two additional contexts were added to the interface. One is a 2D graph that plots, for each population, the sum of its neurons that spiked in the 100 time steps immediately preceding the current time step. The second is a ball-and-stick approximation of the movement of the limb. These three contexts shown on the same screen visualize the relationship between certain motoneuron populations' activity, where in the spinal cord this activity is occurring, and the limb movement.

2. The 3D Spinal Cord Model

The first step in the road towards completing this leg (no pun intended) of the project was to create a model of a cat spinal cord. Dr. David Bashor provided 36 2D images, each depicting the right-half (facing the cat) of a cat's spinal cord at a set point along the length of the cord (figure 1). This set of images was loaded into Adobe Photoshop. A blank, white rectangle was created with that has the width of the widest image in the set and the height of the tallest image in the set. The rectangle's width was then doubled. The following was done for each image in the set:

1. Copy the image and paste it onto the white rectangle's right side, so that the right edge of the image was lined up to the right edge of the rectangle.
2. Make an educated guess, based on having inspected all 36 images several times over, as to the location of the midpoint over the X axis of the hypothetical slice of the whole spinal cord (not just the right half) corresponding to this slice, and mark it on the image.
3. Copy the rectangular area that contains the image pasted in step 1 up to the midpoint-marker (this could be a smaller or a greater area than just the image from step 1).
4. Paste this area onto an arbitrary spot in the left-side of the white rectangle.
5. Transform the image from step 4 into its mirror image.
6. Position the image from step 5 so that its right-edge lines up with the left-edge of the image from step 1 - this combined image is a slice through the whole width of the spinal cord.
7. Position this combined image so it is centered in the white rectangle, and save the resulting image (figure 2).

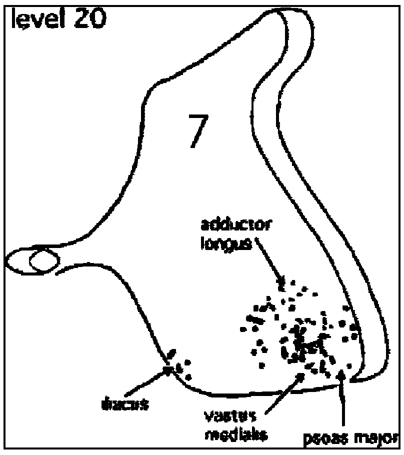


Figure 1: an Image of the the right-half of a spinal cord slice

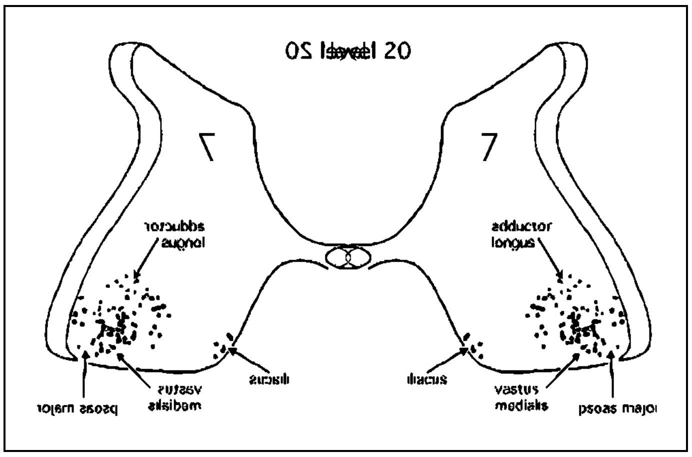


Figure 2: an Image of the whole slice – at a smaller scale

Next, AC3D [6] was used to create polylines around the edges of the combined spinal cord slices. In AC3D a rectangle of the same dimensions as the white rectangle created for the spinal cord slices. For all 36 slice images, the image was set as the texture for the new rectangle, and an 80-vertex polyline was manually created, approximating the curved shape of the cord (the fake-3D perspective in the image was ignored) (figure 4). After the 36 polylines were created and each saved to disk in ASCII format as a list of vertex coordinates and a list of texture coordinates, it was necessary to edit each file and erase the header, the footer, and the entire list of texture coordinates to get a file of only vertex coordinates (in a world coordinate system). It only took one appointment doing that before a filter program was written (filterAC, by Matt T. Miller) to take care of that menial task. The filtered file-set was loaded into the program, where it looks at each file, in order “z” location on the spinal cord (from front to back), reads in a set of three floating-point numbers, and inserts them onto a vtkPoints array [7], where all coordinate triages in all files are inserted onto the same vtkPoints array. To do this, the program must keep track of the index at which to insert a coordinate triage. This is accomplished simply by setting a counter variable to 0, and then for every 3 floating-point numbers it reads from a file, it increments itself. The counter does not reset when the next file is opened, but rather, it continues to increment on itself. Each time a new file is loaded in, though, the value of the “z” dimension is updated (we can’t have a 3D model if all vertices are on the same z coordinate!). We calculated that the spacing between each slice, from 1 to 31, to be 33% or 180.3406 “units” (where 100% (which equals 541.0254 units) refers to spacing over 3 slices (i.e. between slices 1 and 4) in the range from slice 1 to slice 31, and “units” are points in the world coordinate system). The last slices have a different, non-uniform spacing. Between each slice 31, 32, 33, and 34, the spacing is 40% (216.4102 units). Between slices 34 and 35, it is 20% (108.2081 units), and finally, between slices 35 and 36 the spacing is 30% (162.3076 units). After the last file is read in and closed , the vtkPoints array is inserted into an empty structured grid (vtkStructuredGrid). This structured grid is the model used in the program (well, the first model was generated this way -> figure 13 on the Models Page). The methods that accomplished this were in class MainWindow (MainWindow.h, MainWindow.cpp, written by Matt T. Miller), but were almost immediately thereafter moved to a stand-alone program (sgbuilder, written by Matt T. Miller), where they would, in addition, save the structured grid to disk as a single file with a ".sg" file extension (i.e. catSpinalCord.sg). A new, very small class called "SGridLoader" (SGridLoader.h, SGridLoader.cpp, written by Matt T. Miller) was created that does nothing more than load a saved vtkStructuredGrid from disk, and return a pointer to that structured grid to the object that called it. In order for class MainWindow to get the structured grid, it simply creates an instance of class SGridLoader. The generation of the first model was also done using this technique, on account that there was no second model at that time (even the newest models are still generated in this manner). The first model was very jagged and unnatural looking, so the polylines were reviewed, but tweaking these polylines resulted in little change. To compensate for this, all three dimensions of the model were smoothed using Kochanek-Bartels splines [3] (spc.o, written by Dr. K.R. Subramanian). The model was considerably better than the previous, but it still left much to be desired (figure 14 on the Models Page). After many other key things were implemented, the model was revisited, and attempts were made, once again, to smooth it out. The first "next" attempt saw the model as an approximation of a spinal cord, which meant deliberate tampering with the data to achieve a nicer-looking model was ok. For this model (not shown on the Models Page), a single polyline was selected (one that looked "nice and average"), and 35 copies of it were made, each saved to disk. Using "filterAC" to convert the AC3D files into a clean list of 3D vertices, the files were loaded into "sgbuilder" to produce a structured grid file as output. It was loaded into the visualizer and rendered to the screen. This model was extremely smooth, but it had one major problem: it was of uniform width and height everywhere along its length. This was much more visually appealing than the previous models, but it was just too inaccurate. The model we were after was nice and smooth, but still followed the general shape of the first splined-model (narrow at the front, then gradually widening and getting taller, then more-or-less maintaining this width and height, but still deviating from them slightly along the cord, and then towards the end, it narrows all the way to the last slice, which is smaller than all the rest. The next attempt met with a bit more success. For this model (figure 15 on the Models Page), I calculated the width and height of all 36 polylines (in world coordinate units), and saved them to a file for safe keeping. Then I loaded every fourth polyline, starting with number 1 (then 5, 9, etc) into AC3D. I would select a polyline (I started with polyline 1), and then manually shrink or grow it to match the width and height (listed in that file)of the next polyline (which wasn't loaded). I would then save that polyline to disk as if it were, in fact, the next polyline in the series (I preserved the old polylines in a tar file). For each polygon I (the original 9 polylines I loaded in (36/4 = 9)), I would shrink/grow it to generate i+1, i+2, and i+3. This model showed exceptional smoothness over the 4-slice "sets" of grown/shrunk polylines, but the boundaries between these still presented bumps and some jagged angles. We then ran this model through Dr. Subramanian's Kochanek-Bartels spline program (spc.o), and the resulting model (figure 16 on the Models Page) was pretty smooth, but it exhibited ripples all along the cord, and in a few places it smoothed out the sharp-angles into ledges that look like they were pulled on and stretched. The next model, however, was a winner (figure 17 on the Models Page). The techniques used in the previous 2 models were combined. A "nice and normal" polyline (I believe this was polyline 7) was and 35 copies of it were made. Then each of the 35 copies were grown or shrunk to correspond to the width and height of one of the other 35 original polylines. The resulting files were put through the filter (filterAC), and those resulting files were put through the structured grid builder (sgbuild). The resulting model was extremely smooth and showed no bumps, ripples, nor twists. This model did not need to be smoothed out with Kochanek-Bartels splines, which meant that it was significantly smaller in size than the "splined" models, having 40 times fewer vertices (the regular models contain 36 slices and 80 vertices per slice, which equals 2880 vertices, while the "splined" models contain 360 interpolated slices and 320 interpolated points per slice, which equals 115200 vertices). The smaller file allows the program to load up quicker, and the reduced geometry allows the visualization to run smoother on slower machines.

3. Motoneuron Populations

Work was begun to create the populations of neurons. The first attempt at this was to subclass the familiar vtkActor class from the Visualization Toolkit, but that met with failure due to the complicated, "factory" methods VTK uses to allocate objects. The second attempt met with success by creating a class called “Neuron” (in Neuron.h and Neuron.cpp, written by Matt T. Miller) that has a vtkActor as a data member. The initial layout of that class consisted of the vtkActor, a float[3] storing RGB color, a float to store an Alpha value, and a float[3] storing the X,Y, and Z values. For the sakes of convenience and robustness this class was soon fitted with its own vtkPolyDataMapper object, and its own vtkCubeSource object. The addition of these two objects meant that we no longer had to explicitly declare and allocate a cube source and a mapper for each Neuron, nor have to remember to delete all of these when done. As of this point, we could allocate a new neuron object and it would be immediately viewable on screen (after only adding its actor to the renderer) located at the default (0,0,0) position. The program was then modified to automatically create two whole populations of these neurons (100 neurons per population), one red (Sartorius, a hip flexor), one blue (Semimembranosus, a hip flexor). The default location was good to show that the neurons worked, but it is worthless for visualizing anything. A method for giving each neuron an explicit location inside the model was needed. The Dr. Bashor-supplied-images also show locations in the spinal cord for several different motoneuron populations. From these images the bounding-ranges along the length of the spinal cord were calculated for each population added to the program (Sartorius and Semimembranosus at this time). At any 2D slice taken along these ranges, the mean location of the neurons specific to that range is not in a static location, but rather the neuron group’s (neuron population’s) location shifts slightly from slice to slice. It was noticed that the shifting corresponded, on the whole, to the shifting from slice to slice of a linear interpolation between the mean location at the beginning of the range and the mean location at the end of the range. This allowed for a good approximation of a mean location on any point in a range on the model. The mean location was replaced with a rectangle that represents the bounds of a population along the axes orthogonal to the bounding-range. With a rectangle at the beginning of a bounding range, and one at the end of a range, a rectangle can be interpolated at any point along the range (figure 5). To generate 3D coordinates for a neuron, the following method was used.

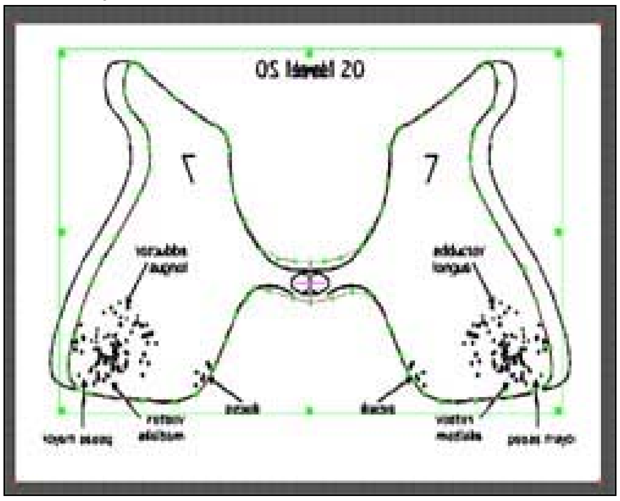
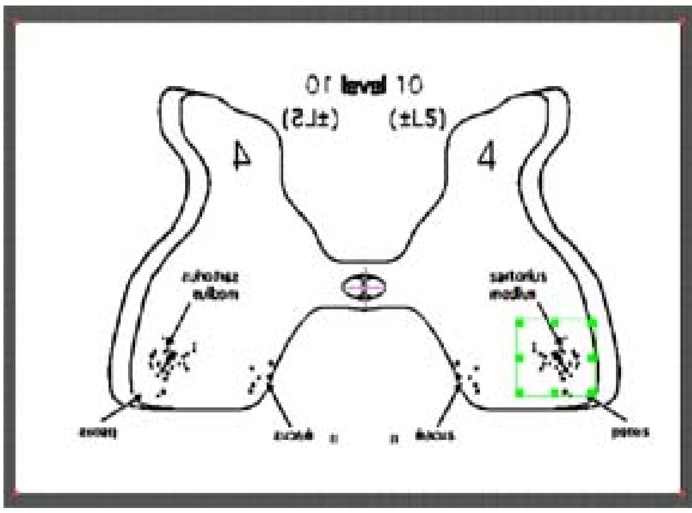


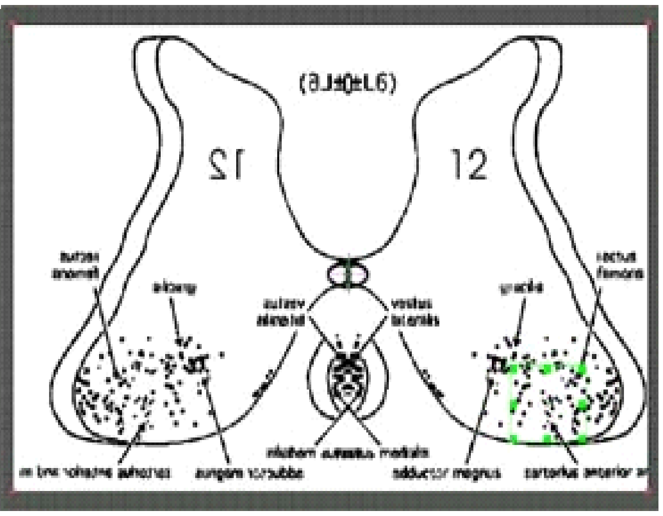
Figure 2: a Polyline approximating a cord slice

1. Determine which population the neuron belongs to.
2. Use the formula “f(x) = M + (m – M) * R” (where M is the maximum value in a range, m is the minimum value in the same range, and R is a random number between 0 and 1), to generate a coordinate along the length (“z” axis) of the spinal cord.
3. Use this location to linearly interpolate the bounding ranges orthogonal to the length of the spinal cord (the “x” and “y” axes).
4. Use the formula from step 2 to generate a coordinate inside the “x” range, and do the same for the “y” range.

The program generates 3D coordinates for each neuron in each population to be visualized and adds them to the same renderer used on the model. With the neurons sharing a renderer with the model, any transformations are applied to both the model and the populations of neurons. The final (for now) additions to the neuron class were an accessor method and a mutator method to get/set the opacity. Changing the opacity signifies whether a neuron is active or not in a given time step. On a range between zero and one, an opacity of 0.2 signifies inactivity, and an opacity of 1.0 signifies activity. Since at any given time step the number of active neurons is a small fraction of the total number of neurons on the screen, the default opacity is set to inactive (0.2). The color of the Semimembranosus population was changed from blue to green, per the liking of Dr. Subramanian (the green shows up better on paper print-outs of screenshots). After several weeks, the above techniques were shown to be stable, and two additional populations were added to the model: Rectus Femoris (a knee extensor) in a blue-green, and Posterior Biceps Semitendinosus (two knee flexors joined into one population) in orange.

When each Neuron object is allocated, a pointer to that neuron is stored on an array. The arrays hold all pointers-to-Neurons for a particular population (100 per population). Since one of the goals of the visualization is to visualize any number of populations, the number of arrays-to-Neurons must be set dynamically. To achieve this, a pointer-to-a-pointer-to-a-pointer-to-Neuron, called "***populations"





Figures 4 and 5: The rectangles at the beginning and end of population Sartorius's range along the length of the spinal

(Neuron ***populations), is declared. When the number of populations to be visualized is determined (this number is loaded in from a neural activity data file -> see next section), "***populations" is dynamically allocated some "pointer-to-a-pointer-to-a-Neuron"s, one for each population. Then each of these is dynamically allocated 100 "pointer-to-a-Neuron"s, which, in turn, are set to point to new allocations of Class Neuron. For each Neuron created, a 3D coordinate is given, as described earlier.

4. Visualizing the Neural Activity

With the completion of a functional class Neuron, attention was turned to visualizing the activity of the neurons. A simple ASCII data file was created for a two-population (Sartorius, and Semimembranosus) visualization, which consists of, on the first line, the number of time steps stored in the file and the number of populations data is stored for. Any subsequent line contains a time step ID, a population ID, and the number of active neurons for that population at that time step. If the number of active neurons is greater than 0, the next line contains the cell IDs for those active neurons. If the number of active neurons equals 0, the next line contains another "time step, popID, #active neurons" for either the next population in the same time step, or the first population in the next time step, depending on the previous population's ID. This file (and any other file like it) is opened by a new class called “NeuronActivity” (in NActivity.h and NActivity.cpp, written by Matt T. Miller). This class keeps a vector (a growable array) of objects of another class called “Timestep” (in Timestep.h and Timestep.cpp, written by Matt T. Miller). The number of Timesteps held in the vector is determined by the number of time steps stated in the data file. The Timestep class holds a vector-of-vector-of-integers, called “populations”. The number of neuron populations, as determined in the data file, sets the length of “populations” (vector<int> populations[numPopulations]). The vector-of-integers stores the Cell ID numbers (0...99) for the active neurons for a population in a given time step. If no neurons are active for a given population in a given time step, then the vector-of-integers is empty for that population in that Timestep object. This technique allows for the closing of the data file once its information has been read in, and also for the fast sharing of this data between the classes that use it.

The user-interface, defined in the MainWindow class (MainWindow.h, MainWindow.cpp), which orchestrates the loading of the activity, the creation of the Neurons, and the instantiation of the class (Canvas3D (Canvas3D.h, Canvas3D.cpp, written by Matt T. Miller)) that builds the 3D model and renders it, is fitted with VCR-style controls (figure 6), to control the visualization of the activity. When a button is pressed, the appropriate action ensues on the screen. While playing, the program determines the ID of the previous time step visualized, finds that time step in the activity object (instance of class NeuronActivity), and for each population’s vector of cell IDs, it “turns off” (sets opacity to 0.2) the neurons on screen that match the population and cell IDs. Then, using the current time step, it goes back to the activity object (from now on, referred to as *activity) does the same thing, but “turns on” (sets opacity to 1.0) these Neurons. Except for the “STOP” button, each VCR button works in the same way; it determines what the next time step to visualize will be. To accomplish this, the program determines which button was pressed, and if “PLAY” was pressed, it enters a loop where it: a) increments the current time step by 1, and then b) visualizes the new current time step. “Fast Forward” and “Rewind” (from now on referred to as “FF” and “RW”) do the exact same thing as “PLAY”, except they increment the current time step by CUESPEED (a defined constant) and –CUESPEED, respectively. “STEP FORWARD” and “STEP BACK” (from now on referred to as “SF” and “SB”) also act much in the way “PLAY” acts, however, instead of entering a loop, they only increment by one, and decrement by 1, respectively the current time step, and then visualize the new current time step. The “STOP” button stops any currently-running loop. It accomplishes this through the use of the FLTK (Fast-Light ToolKit) [4] for C++. The FLTK is an easy-to-use, quick-to-debug library of widgets for the **rapid** creation of graphical user interfaces. The FLTK makes possible the interruption of loops through multithreading. The visualization program, like most other regular C++, runs in a single thread given to it by the operating system. The FLTK widgets also run in that same thread, but FLTK runs its “action listeners” (to use the Java/C# name for it) in a separate thread. This is completely transparent to, not only the user, but the programmer too! FLTK creates this separate thread completely automatically. It is just easy to forget while programming that, while a “while loop” is executing, and, according to the exit-conditions of the loop, will continue until the very last, or very first time step has been visualized, the other buttons are still able to handle events. A positive (for me, the programmer) result of this feature is that it makes having the VCR buttons behave in the same way as on a real VCR easy. For example, if “PLAY” is pressed, causing the visualization to progress one time step at a time, and then “FF” is pressed, the visualization will, without delay, start progressing CUESPEED (usually 10) steps at a time, just like on a normal VCR. This is because the visualization is only running in a single thread, so for the “FF” loop to begin, FLTK has to suspend the “PLAY” loop. If “RW” was then pressed, the “FF” loop would be suspended, and the visualization would start progressing backwards CUESPEED steps at a time. When the end or beginning (when running backwards) is reached, the program implicitly sets the play-state to “STOP” (in order to prevent a segmentation fault caused by accessing time steps that do not exist), thus terminating the suspended loops. Despite that, the visualization program implicitly “STOP”s the visualization any time a button is pressed to accommodate for the “SF” and “SB” buttons. Since these buttons only visualize one step instead of enter a loop, FLTK naturally wants to resume any suspended loop interrupted by these buttons. For example, without the implicit “STOP”, if the visualization was playing in “PLAY” mode, and “SB” was pressed, the visualization would go back one step, visualize it, and then continue playing in "PLAY" mode without any pauses between any of the steps. Pressing "SB" or "SF" should leave the previous or next time step on the screen until another button is pressed, just as on a real VCR.

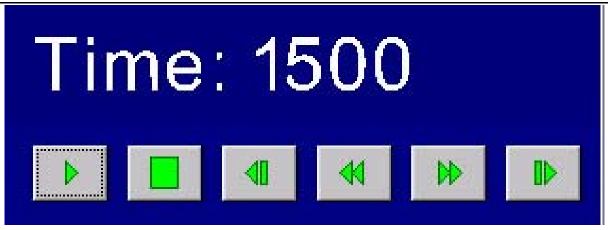


Figure 6: the VCR style controls

All of the VCR buttons (except "STOP") call the same method to visualize a time step. This method is located in the Canvas3D class (Canvas3D.h, Canvas3D.cpp), which contains all of the sources, mappers, actors, and renderers (aside from those in the Neurons, which it only uses), and puts the 3D graphics on the screen. The method that updates the visualization with that of the current time step, called "StepOnce", keeps track of the previous time step rendered, turns all the Neurons in that previous step "OFF", gets the population and cell IDs of the active Neurons in the current time step, and turns them "ON". The current time step is given to this method by the MainWindow. This is the method that the VCR buttons call over and over again when activated

There is also another way to change the rendered-time step, and that is using the "current-time-step slider bar" (from now on referred to as the "current-slider") (figure 7). The program offers two slider bars on the very bottom of the view-screen that allow the user to set a range of time steps to be played by the VCR buttons, instead of always playing from beginning to end and back-again. The slider on the left is the "current-slider", and the one on the right is the "end-slider". The end-slider has one purpose only: it sets the last time step to be visualized. Its default value is set to the total number of time steps available, but it can be set to any time step between there and one + the value of the current-slider. The visualization will never exceed this time step until the slider is repositioned. The current-slider has two jobs. It sets the starting time step for the visualization, and it is always positioned, along its track, at the current time step. It is by default set to the first time step the program wants to visualize (a programmer-defined constant), normally set to 1500. This slider can never precede this value, but it can be set forward to any value less than or equal to the end-slider. Dragging the position-marker on the current-slider will update the visualization to be that of its position (it will update while being dragged, not only after it has been set). To do this, it calls a method inside the MainWindow class each time the FLTK action-listener for this slider bar senses that it has been manually-moved. That method checks the new position of current-slider against the minimum possible position and against the position of end-slider. If the new position is a legal position, it updates the visualization (it calls the StepOnce method in the Canvas3D class). If the new position is an illegal value, it does not update the visualization, and it puts the position-marker of the current-slider to the last legal-position. The end-slider also uses this method to check its position, but it, of course, will not update the visualization, unless it is moved to a spot that precedes the current-slider (and even then, it does not update the visualization, but rather, it pushes the current-slider backwards, which updates the visualization).

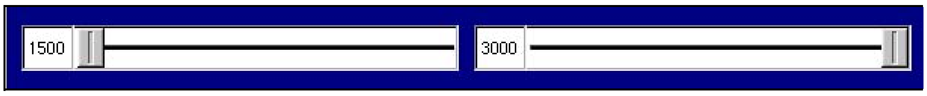


Figure 7: the current (left) and end (right) time step sliders

There are two additional views in the visualization program. These are a line graph (lower center-left of the screen and a ball-and-stick diagram of a limb (lower center-right of the screen). The line graph, defined in class GraphView (GraphView.h, GraphView.cpp written by Dr. K.R. Subramanian), shows the sum of active neurons, for each population, over the previous 100 time steps for each time step from the beginning to current (figure 8) The limb, defined in class LimbView (LimbView.h, LimbView.cpp written by Dr. K.R. Subramanian) loads a pre-calculated list of angles. This set stores the angle of the limb "stick" relative to the hip "ball" at all 1500 time steps (figure 9). Both of these classes contain their own sources, mappers, actors, renderer, and viewport, meaning they can be simply added to an existing render window and told the dimensions of their viewports, and that is it. To update them to the current time step, each of them has a “SetTime(int newTime)” method. All that is required is, when updating the Model and slider bars, simply add a call to these two’s SetTime methods, and they will take care of themselves. The GraphView class uses the same neuron activity data file that the model uses. Currently, due to time constraints, the GraphView class opens the data file, and parses it itself to get the number of spiking neurons per population per time step. Changing this so it gets this information from the already-in-memory *activity object is near the top of the list of modifications to be made.

The details of how the model and the Neurons go from bytes to pixels was not dealt with in detail up to now, in order to allow for an important higher-lever understanding of the system. The creation of the model and the Neurons, along with the placement of the Neurons has been explained in detail, but the explanation stopped there. There are not many steps between where the explanation stopped, and having the model on screen, and one who is familiar with VTK (Visualization ToolKit [5]) can easily guess them. The MainWindow class orchestrates (through the creation of objects of the classes that to so) the loading of the neuron activity (class NeuronActivity instantiated into an object referenced by “*activity”), the creation of the Neuron populations (using the number of populations determined by *activity’s reading of the activity-data-file, MainWindow itself instantiates 100 Neurons per population and stores pointers to these neurons in arrays, and pointers to those arrays (the populations) in another array, referenced by “Neuron ***populations”). Then MainWindow creates an object of class Canvas3D (Canvas3D.h, Canvas3D.cpp, written by Matt T. Miller), which takes in as parameters its viewport’s location on the main render window, the pointer to the vtkStructuredGrid that was loaded from disk, ***populations, the number of populations, *activity, a void pointer to this MainWindow instantiation, the beginning time step, and the ending time step. Canvas3D begins by instantiating a vtkRenderer object, which later takes actors and renders them to a viewport. It is created first because many of the following objects need a renderer to reference. This object starts out here having no actors assigned to it. Canvas3D then creates (from here on, “creates” is treated as a synonym for “instantiates <a/an> <class name> object”) a vtkVectorText, containing the title of the visualization (title text is hard-coded), and a mapper and actor for the text. It then makes the labels that hover above and below the model, and the “lines” (vtkCylinderSource) that “point” to the locations on the model the labels describe. Each label consists of a vtkVectorText, a vtkPolyDataMapper (to convert the text into polygonal primitives), and a vtkFollower, which is a special vtkActor that automatically orients itself so it is facing the active camera (facing the user). The text is fed to the mapper, and the mapper is then fed to the actor (in this case, the vtkFollower). Each “line” contains a vtkCylinderSource, a vtkPolyDataMapper (from now on referred to simply as a “mapper”), and a vtkActor (simply “actor” from here on). The cylinder source is fed to the mapper, which is fed to the actor, just like with the labels. This is part of the Visualization Pipeline, as defined in [5]. It is a hierarchy of objects that make for efficient visualization of data. It begins with a data source (can be point coordinates or a VTK-supplied geometric template, like vtkCubeSource), which is fed to a mapper (a mapper converts the geometry into polygonal primitives), which is fed to an actor (actors contain the polygonal data (in the mapper), and several modifiable properties for that data, such as position, orientation, origin (for rotating about), scale, visibility (“off” != opacity 0, but rather, “off” means the renderer won’t even try to render it), color, opacity, shading method, any many more). Actors are given to a renderer (vtkRenderer, which controls the lights, cameras, and actors, which together make up a 3D “scene”, and creates a 2D projection of it (like taking a picture with a camera)). Renderers are fed to render windows (vtkRenderWindow, which take 2D projections and put them on the screen inside a window). Interactors are the optional next-step in the pipeline. Interactors (vtkRenderWindowInteractor or a vtkFlRenderWindowInteractor) enter a loop that waits for user-input, and then handles the input accordingly (like rotating and moving the graphics). Moving on, Canvas3D then creates a vtkDataSetMapper to convert the model’s structured grid into polygons, and then creates a vtkActor to place the polygonal data in the 3D scene. Then a renderer for the title-actor is created, is given a small viewport in the window given to the Canvas3D. The title is of little importance and will no longer be richly discussed, nor will the labels and lines. The next step is to create that big “Timestep:XXXX” box that is positioned on top of the VCR buttons (figure 6), by creating vector text, a mapper, an actor, and its own renderer, which is given a viewport. Then the main renderer is given actors, starting with the Model's actor, then the Neurons' actors (self-contained inside of class Neuron), and finally, the actors of the lines and labels. This renderer is given a viewport that takes up most of Canvas3D's allotted window, so the Model will have plenty of room. The vtkRenderWindow and the interactor are located in class MainWindow, which gets references to the vtkRenderers in Canvas3D and feeds them to the render window. This program does not use a vtkRenderWindowInteractor, but instead uses a vtkFlRenderWindowInteractor [7]. This is a special interactor class that inherits from both vtkRenderWindowInteractor and Fl_Gl_Window (an openGL canvas class in FLTK), and contains methods that make them to work together, allowing a program (like this one) to use the full functionality of both FLTK and VTK without having two event handlers that would constantly fight and get confused over who the mouse and keyboard actions belong to. The event-loop is handled by FLTK, and thus, the event handling for all of the vtkRenderers (the Model, the Title, the Timestep text, the Graph, and the Hip Diagram) is performed by the vtkFlRenderWindowInteractor. This completes the journey from bytes to pixels.

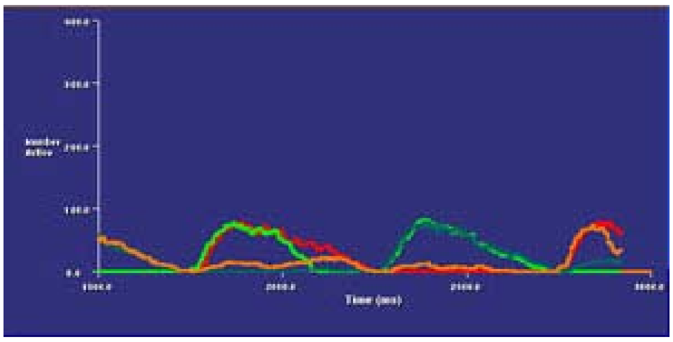


Figure 8: the Graph at time step 2922.

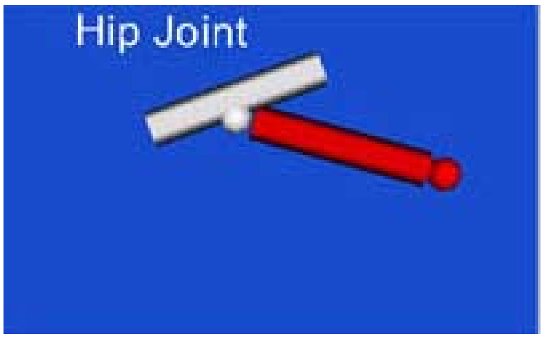


Figure 9: the Ball-and-Stick limb diagram

5. Other Features

This program is constantly being tweaked, updated, revamped, and (sometimes) overhauled. The other features mentioned here are not critical to the performance of the program, but offer some nice functionality. These features are accessible through the Settings Dialog Box (figure 10). This contains colored slider-bars for changing the RGBA values of the Model (only the cord, not the Neurons), and the RGB color of the background of that viewport (default set to light-brown). These set the values through a pointer to the running instance of class Canvas3D, that calls a public method that calls the SetColor and SetOpacity methods accessible via the Model's actor's GetProperties method. To change the background color, the pointer is used to call a method in Canvas3D that calls the renderer's SetBackground method. The next feature is a slider-bar that lets the user set the opacity of the Labels and Lines that surround the Model. It has 4 settings: full opacity (default), 66% opacity, 33% opacity, and Invisible. The first three are set through the actors' GetProperty()->SetOpacity() method, but the last setting is set through the actors' SetVisibility method, which when set to "false", tells the renderer to skip this actor. Next is the "Big3DView" feature. Pressing this button expands the Model's viewport to encompass almost the whole screen. Pressing it again returns the screen to normal. This was easily accomplished by removing the other views from the vtkRenderWindow, and then changing the viewport size of the Model's viewport. To change it back, simply return the Model's viewport to its original size and add the other view back to the vtkRenderWindow. The last feature is the Enable Record button. This button controls whether a record button will be visible among the VCR buttons. By default it is not, but when enabled, the record button saves an image to disk of each time step shown in the visualization (controlled using the other VCR buttons, or the "current-slider"). These images can later be assembled into a movie file. The record button has the enable/disable feature to prevent accidental recordings. A movie of 1500 time steps will consume almost a gigabyte of hard drive space!

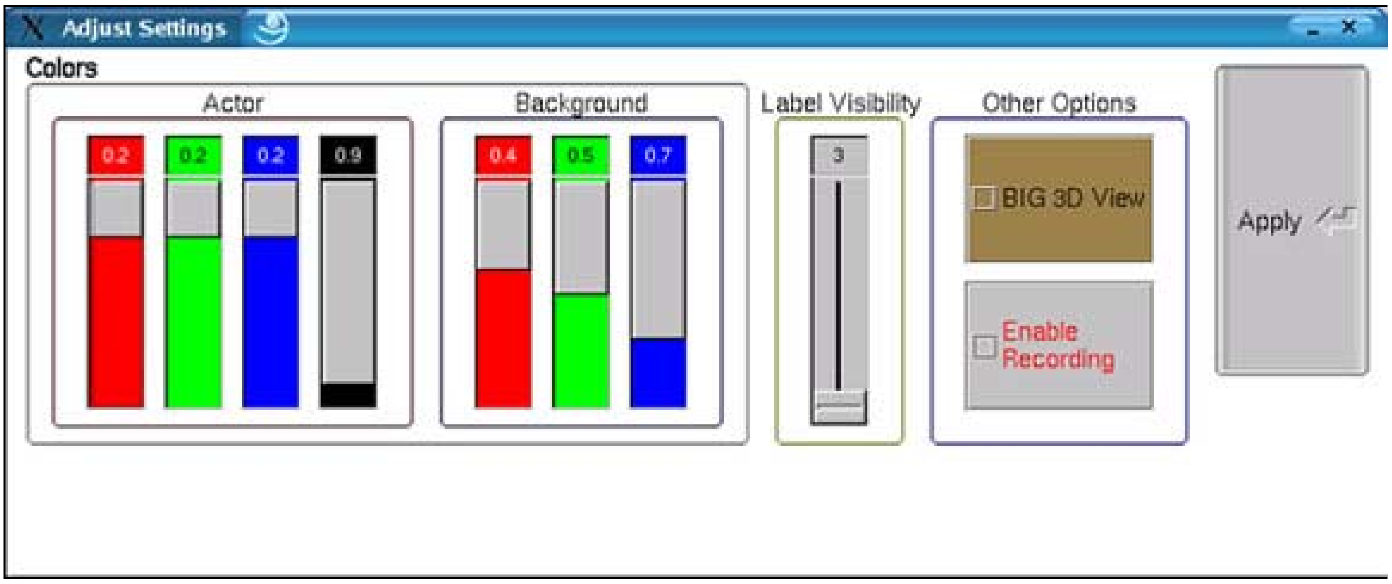
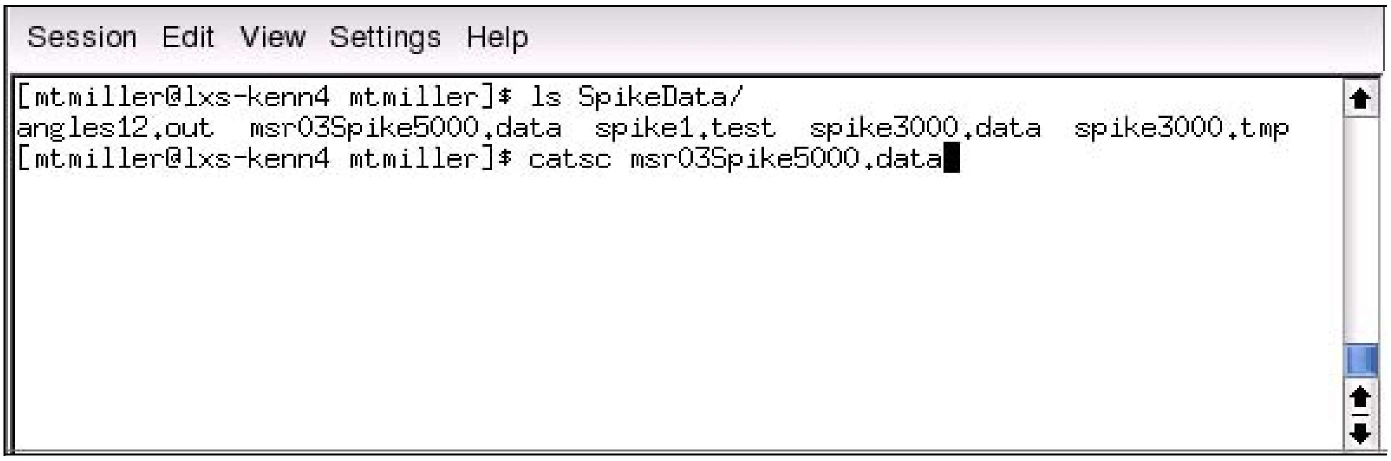


Figure 10: the Settings Dialog Box

6. Using the Program

The visualization program is quite easy to use (and will get even easier in future versions). To start, from the UNIX console, go to the directory containing the program and type "catsc <name of activity file>" (figure 10) and press enter. If no activity file is given, the visualization program will not run. When the program loads (figure 11), the screen will show the graph on the lower-left, the ball-and-stick diagram on the lower-middle, the VCR buttons (with the current time step number in big letters) on the lower-right, and the "current" and "end" time step sliders on the very bottom of the program's window. On top is a big light-brown box that appears to be empty. The top view is not really empty. It contains the 3D model of the spinal cord with the neurons inside. It is only invisible now because the program rotates and scales it to be in a user-friendly, side-view orientation (instead of looking down the thing through one end's opening, which is impractical). The vtkCamera that controls this view's orientation seems to require user-input before it will update, hence, leaving the view looking invisible. Simply zoom into the view by holding the right mouse button and dragging down (be careful to stay inside the brown viewport, otherwise you will zoom another viewport too (i.e. the ball-and-stick diagram). The model instantly appears on screen and gets bigger as you zoom. If at the bottom of the brown viewport and you still want to zoom in closer, release the right mouse button, move the mouse pointer to a position higher up in this viewport, hold the right-mouse button, and drag downwards again. Repeat as needed to get a nice view. To rotate the model, position the mouse-pointer over a spot on the model, hold the left mouse button and drag it in any direction. It behaves in a pulling or a pushing manner, depending on the direction of the dragging, and the orientation of the model. To move the model around in the viewport, position the mouse pointer somewhere on the viewport, hold the middle mouse button, and move the mouse. The model follows the mouse. This style is called Trackball Style Interaction. There is another, more difficult, Joystick Style Interaction, accessible by pressing the "J" key. With this interaction style, one positions the mouse-pointer somewhere in the viewport, holds a mouse button, and the model transforms (rotates, moves, or zooms) in the direction of the mouse pointer. The consensus over here is that Joystick Interaction offers less control over the transformations than does Trackball Interaction. To switch back to Trackball mode, press the "T" key.



```
[mtmiller@lxs-kenn4 mtmiller]$ ls SpikeData/
angles12.out  msr03Spike5000.data  spike1.test  spike3000.data  spike3000.tmp
[mtmiller@lxs-kenn4 mtmiller]$ catsc msr03Spike5000.data
```

Figure 11: starting the program

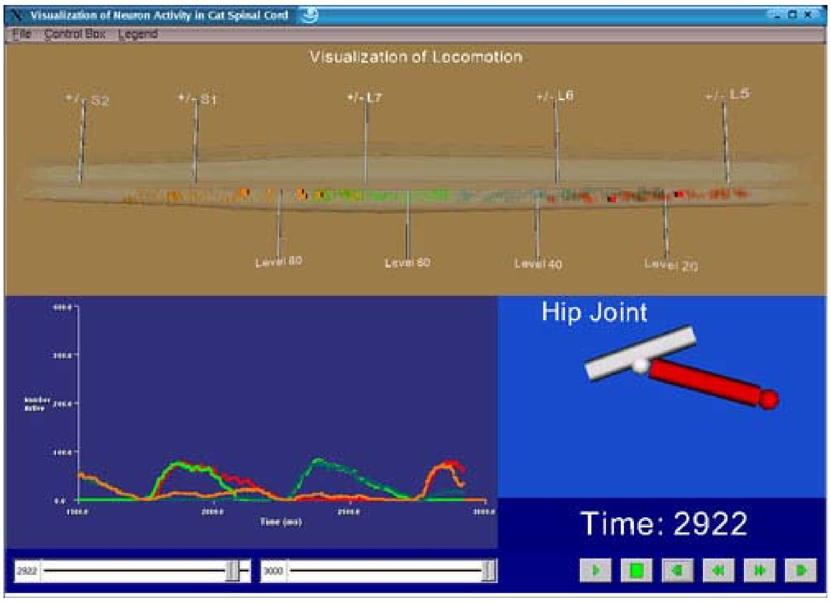


Figure 12: screen shot of the program

7. Up Next

There are still quite a number of things that need to be added, updated, or even overhauled. At the top of the list is modifying the GraphView class so it retrieves the active neuron counts from the NeuronActivity object "*activity" instead of reopening the activity file, and reparsing it. The addition of a graphical file-window for loading a model and/or activity file would be a welcomed change. After that, a revamping of the Settings Dialog window is in order. The colored-slider bars will be replaced with a single Fl_Color_Chooser object and a number of radial buttons, used to select which actor or background color the user wishes to change. The single color-chooser will connect to the selected radial-button.

8. Conclusions

It seems apparent that this visualization tool can greatly aid in the rapid-consumption of large amounts of data and quickly reveal patterns and correlations between activity in populations of motoneurons and the activity of a limb. Seeing the locations of the motoneuron populations, and seeing them spike may also bring forth some new ideas and theories about the interactions of these neuron groups, and may lead to a better understanding of how signals from the brain ultimately cause muscle action.

9. Acknowledgements

I would like to thank Dr. K.R. Subramanian for teaching me so much and pushing me to my utter limits this semester….. and for having patience with me and my overloaded schedule, I know I am not the friendliest, most open-minded person when I am sleep-deprived. I would also like to thank Dr. David Bashor, whose ideas were the mother of this project, and who supplied the raw data that made the Model possible. I would also like to thank Josh Foster, who helped get the program to a presentable point in time for submitting the paper, and for showing me how to use FLTK idles. I

would also like to thank Charl P. Botha for writing the vtkFlRenderWindowInteractor class. It really is a handy class to be able to use!

10. References

1. Bashor, David P. One Leg Walking: A Model of Rhyme and Pattern Generation in Cat Lumbar Spinal Cord. *Society Of Neuroscience Abstracts*, 26:747.5, 2000

2. Subramanian, K.R., and David Bashor. Nviz: An Integrated Environment for Simulation, Visualization, and Analysis of Spinal Reflex Circuits. In Preperation.

3. Kochanek, D.H.U., and R.H. Bartels. Interpolating Splines with Local Tension, Continuity, and Bias Control. *Computer Graphics*, 18(3), July 1984.

4. Spitzak, Bill, et. al. The Fast Light Toolkit. http://www.fltk.org

5. Lorensen, B., K. Martin, and W. Schroeder. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall Inc., 3rd edition, 2002

6. Colbourne, Andy. AC3D. http://www.ac3d.org

7. Botha, Charl P. vtkFlRenderWindowInteractor. http://cpbotha.net/vtkFlRenderWindowInteractor.html

8. Vanderhorst, V.G., and G. Holstege. Evidence for Monosynaptic Projections form the Nucleus Retroambiguous to Hindlimb Motoneurons in the Cat. *Journal of Computatopnal Neurology*, 17(3):1122-1136, February 1997.

9. McCrea, D.M. Spinal Cicuitry of Sensorimotor Control of Locmotion. *Journal of Physiology*, 533:41-50, 2001.

10. Edelman, G.M. *Neural Darwinism: The Theory of Neuronal Group Selection*. Basic Books, NY, 1987.

11. Bashor, David P. A Large Scale Model of some Spinal Reflex Circuits. *Biological Cybernetics*, 78:147-157, 1998.

12. MacGregor, R.J. *Neural and Brain Modeling*. Academic Press, NY, 1987.

13. Fetz, E.E. Are Movement Parameters Recognizably Coded in the Activity of Single Neuons? *Behavioral Brain Science,* 15:679-690, 1992.

14. Yakovnko, S., V. Mushahwar, V. Vanderhorst, and G. Holstege. Spatiotemporal Activation of Lumbosacral Motoneurons in the Locomotor Step Cycle. *Journal of Neurophysiology*, 87(3):1542-1553, 2001

The Models Page (1)

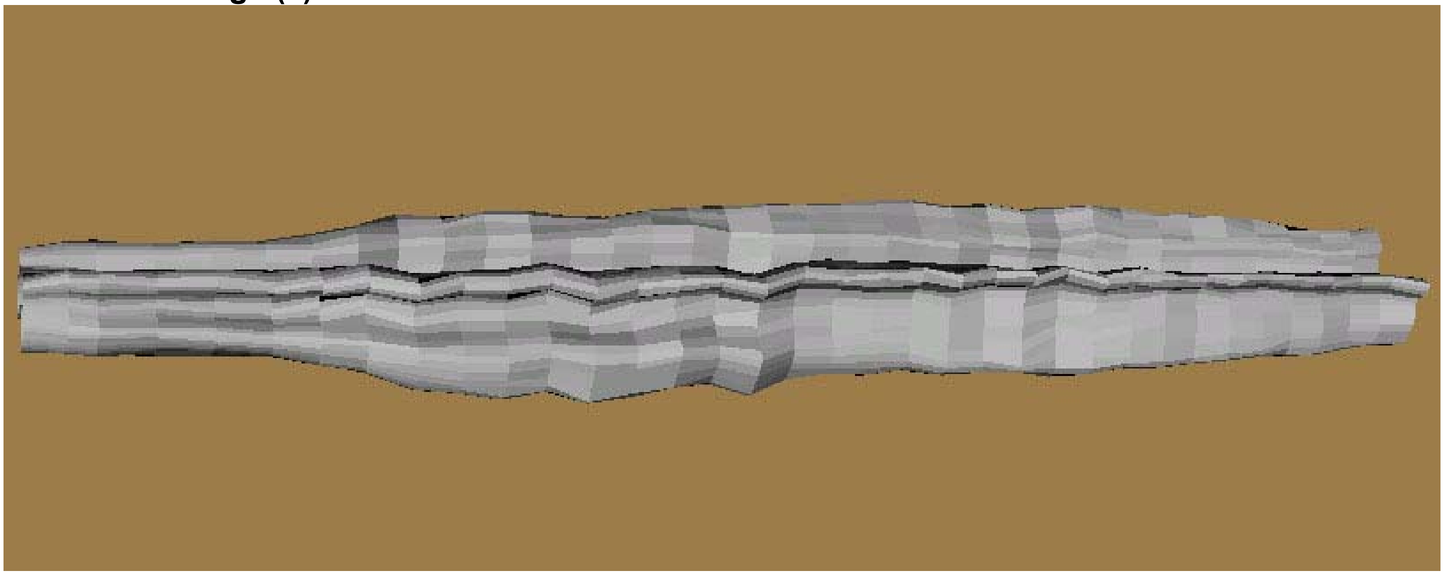


Figure 13: The Very First Model

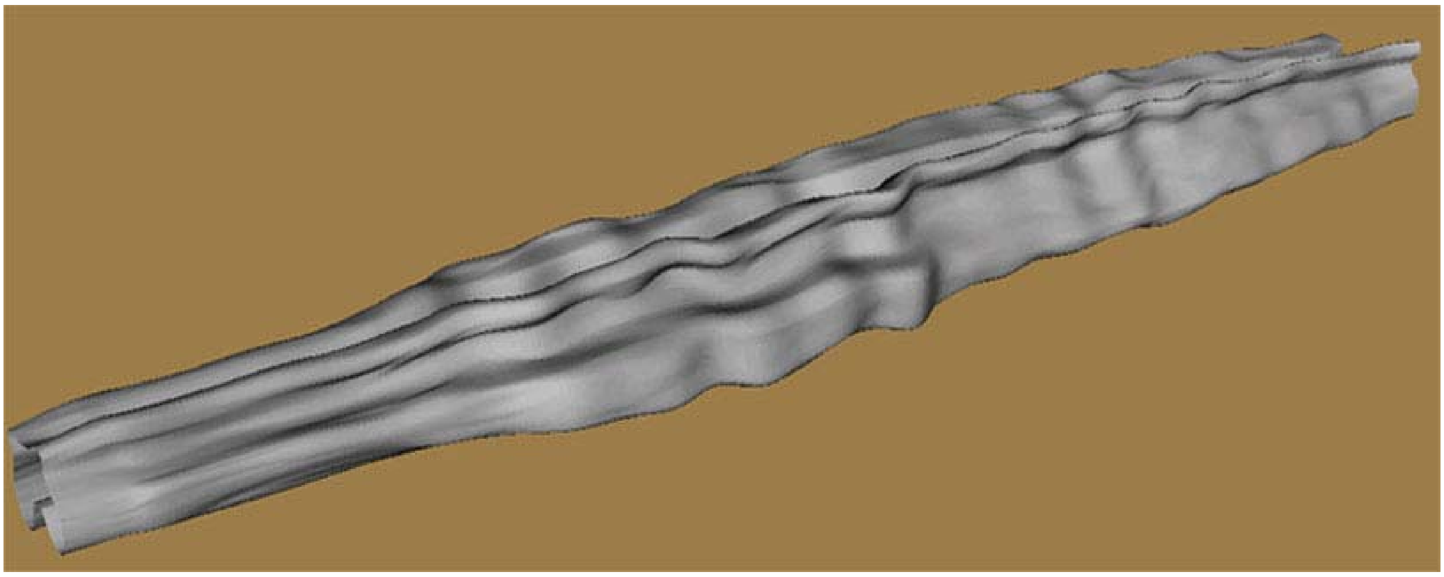


Figure 14: The First Model Applied with Kochanek-Bartels Splines in all 3 dimensions

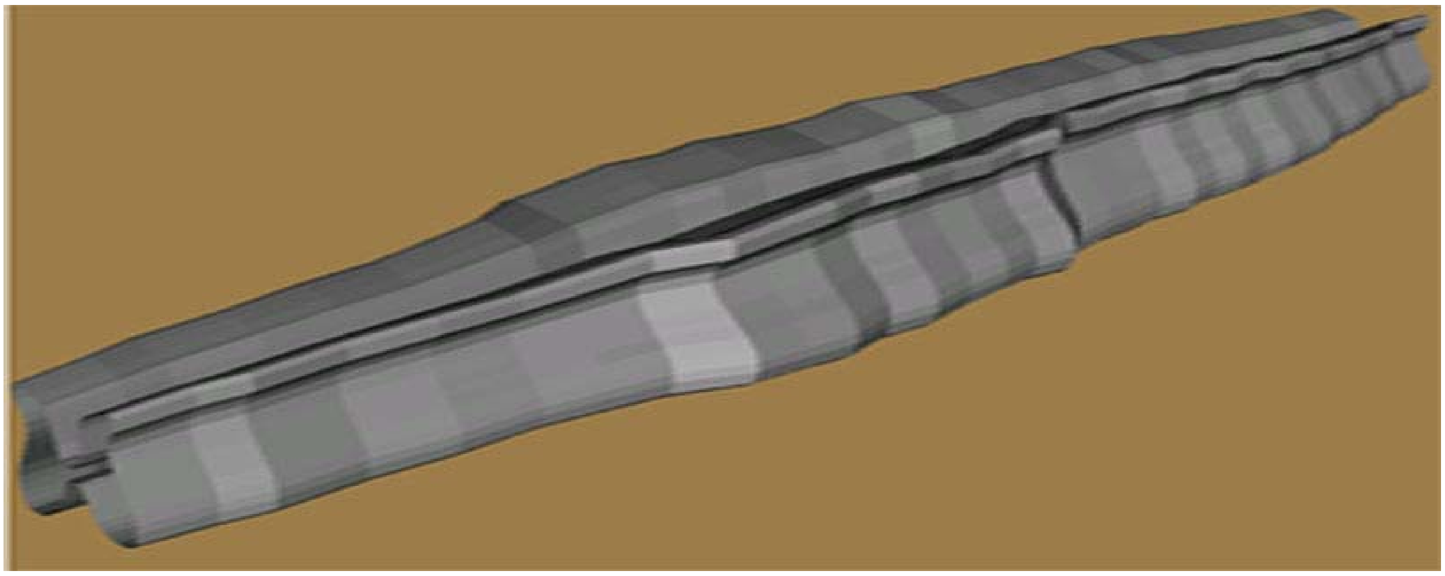


Figure15: The Model based on Interpolating from every 4th polyline

The Models Page (2)

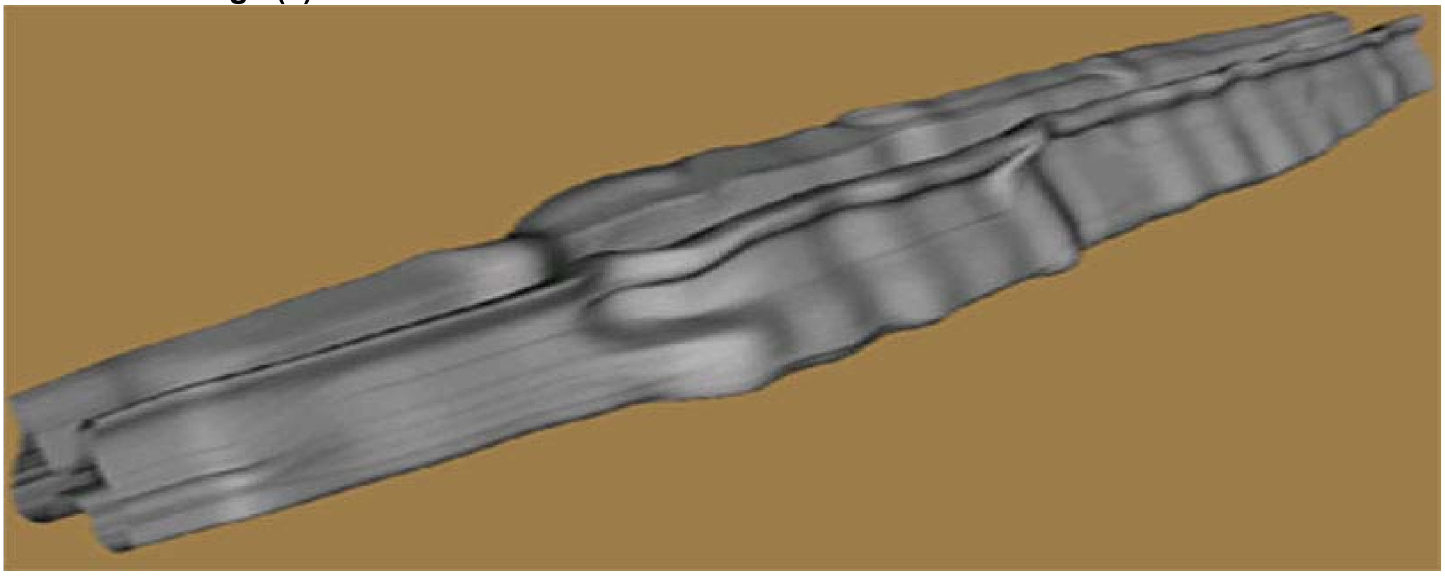


Figure16: The Model Above with Kochanek-Bartels splines applied

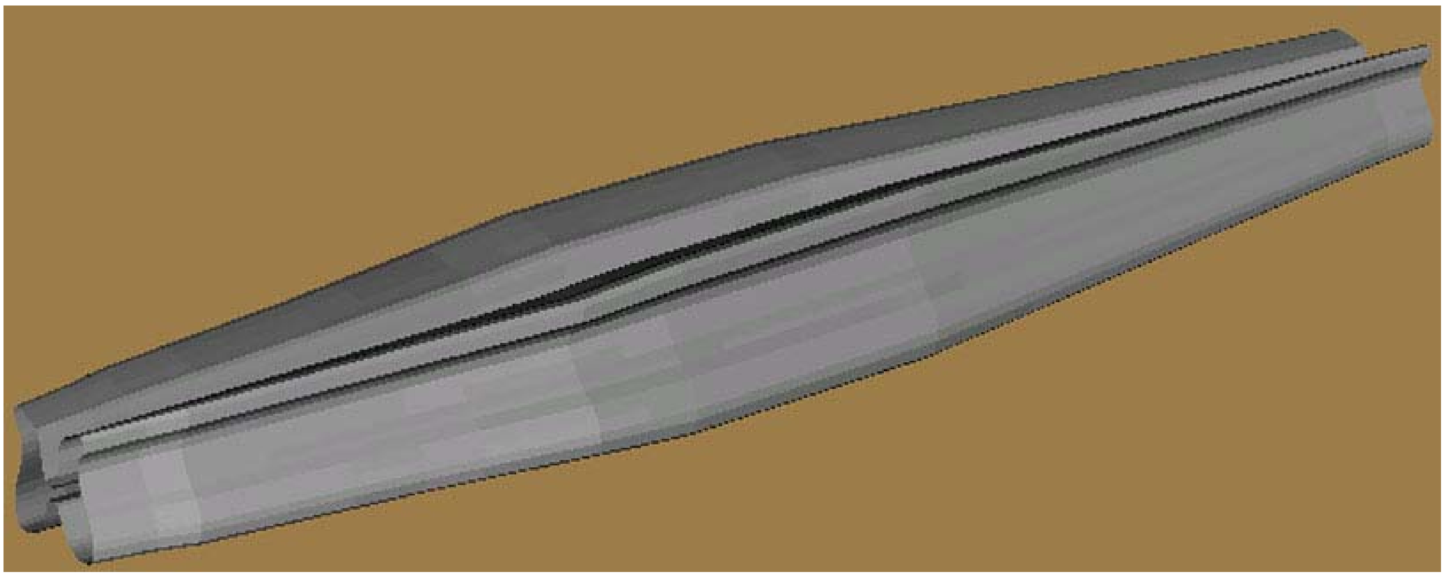


Figure 17: The Last(?) Model