

April 1, 2005

Dr. Sid Fels, Professor  
Department of Electrical and Computer Engineering  
University of British Columbia  
#421-2356 Main Mall  
Vancouver, BC V6T 1Z4

Dear Dr. Fels,

Please accept the report titled "2D Articulatory Vocal Tract Model" that was prepared as part of the project initiated on January 17, 2005.

This report outlines the work completed in building the vocal tract model. The report discusses the objectives of the project, the methodology that was used, the designs that were implemented, and the results that were obtained after the completion of the vocal tract model.

In addition, I would like to thank Dr. John Lloyd, for his guidance in interpreting project requirements, and for his tutoring on computational geometry.

Sincerely,

Eric Lok

# 2D Articulatory Vocal Tract Model

Eric Lok  
Student # 56456965  
April 1, 2005



**Supervisors:** Dr. S. Fels  
Dr. J. Lloyd  
Dr. K. van den Doel

**Project Code:** SSF-496-05-6

# I. ABSTRACT

This paper investigates the construction of a computational model for the vocal tract. This system will allow users to draw a vocal tract outline using a vector graphics drawing tool, and animate the model using an animator tool. The vocal tract model will be useful for users, such as linguists, in the investigation of how the shape of the vocal tract affects sound output.

The system was created using Java and JOGL (Java for OpenGL). The vocal model was to be coupled with an aero-acoustic renderer to create the vocalized sounds. Additionally, it was to be integrated into the Artisynt framework, which is a system for speech synthesis, facial modeling, and vocal tract modeling. Through the process of developing this system, we developed solutions to creating Bezier curves, implemented a system for object picking and selection, created a tool to detect errors in OpenGL calls at run-time, created a monitor object to allow multi-thread safe access to the OpenGL state-machine, and improved on Mermelstein's parameterization of the vocal tract model.

We used a subdivision algorithm to generate the points that trace the Bezier curve. Selection mode rendering using OpenGL was used to implement object picking and selection. An error sniffer tool that could throw a stack trace was used to detect errors in OpenGL calls. The OpenGL state-machine was wrapped in a monitor object to ensure multi-thread safe access. Mermelstein's vocal tract model, with 10 degrees of freedom, was extended to 14 degrees of freedom by adding translational movement to the tongue center and the jaw.

Two objectives were not achieved in this project, which were the coupling of the vocal tract model with the aero-acoustic renderer, and the integration of the system into Artisynt. The coupling of the model to the aero-acoustic renderer was not implemented because of time constraints, and the complexity of finding the cross-sectional dimensions

of the vocal tract. Integration of the system into Artisynt was also not accomplished because of time restrictions, and because the Artisynt system was being reworked.

The creation of the computational model for the vocal tract, the modeling tool, and the animator tool was a large task. We hope that this system will be useful for users, such as linguists, in the study of how the shape of the vocal tract affects rendered speech.

## II. TABLE OF CONTENTS

I.	ABSTRACT.....	ii
II.	TABLE OF CONTENTS.....	iv
III.	LIST OF FIGURES .....	vi
IV.	GLOSSARY .....	vii
1.	INTRODUCTION .....	1
2.	METHODOLOGY .....	3
	i. Technology Used.....	3
	ii. Possible Approaches.....	4
	1. <i>System Implementation</i> .....	4
	2. <i>Animation</i> .....	5
	3. <i>Modeling of Vocal Tract</i> .....	6
	4. <i>Data Persistence</i> .....	7
	iii. System Description.....	8
	iv. Design Concepts .....	8
3.	DESIGNS AND IMPLEMENTATIONS.....	10
	i. Use Case Diagram for the System.....	10
	ii. Use Case Descriptions .....	11
	iii. Sequence Diagrams .....	14
	1. <i>MoveSlider Sequence Diagram</i> .....	14
	2. <i>SaveModel Sequence Diagram</i> .....	15
	iv. Class Diagrams .....	15
	1. <i>Geometry Class Diagram</i> .....	15
	2. <i>Overall Class Diagram of System</i> .....	16
	v. Package to Class Mappings .....	17
	1. <i>Animator Package</i> .....	17
	2. <i>Utilities Package</i> .....	18
	3. <i>Math Package</i> .....	19
	4. <i>Geometry Package</i> .....	20
	5. <i>Config Package</i> .....	21
	6. <i>Entrypoint Package</i> .....	22
	7. <i>GUI Package</i> .....	22
	iii. Vocal Tract Modeling Tool.....	23
	1. <i>OpenGL Monitor Object</i> .....	23
	2. <i>OpenGL State Sniffer</i> .....	27
	3. <i>Object Picking and Selection</i> .....	28
	4. <i>Bezier Curves</i> .....	29
	5. <i>Surfaces: Piece-wise Bezier Curves</i> .....	32
	6. <i>Interface Screenshot</i> .....	33
	iv. Vocal Tract Animation Tool.....	34
	1. <i>Parameterization of the Vocal Tract Model</i> .....	34
	2. <i>Degrees of Freedom</i> .....	35
	3. <i>Interface Screenshots</i> .....	40
4.	DISCUSSION.....	42
	i. Vocal Tract Modeler Tool .....	42

ii. Vocal Tract Animator Tool .....	43
iii. Two Dimensional Vocal Tract Model .....	44
iv. Coupling with the Aero-Acoustic Renderer .....	45
v. Integration of the Model into Artisynth .....	46
5. CONCLUSIONS.....	47
6. References.....	48

### III. LIST OF FIGURES

Figure 1 Model View Controller (MVC) Pattern .....	9
Figure 2 MVC Pattern Used in System.....	9
Figure 3 Use Case Diagram .....	10
Figure 4 Move Slider Sequence Diagram.....	14
Figure 5 SaveModel Sequence Diagram.....	15
Figure 6 Relationships between Geometry Classes .....	15
Figure 7 Relationships between Major Classes of the System .....	16
Figure 8 Animator Package .....	17
Figure 9 Utilities Package.....	18
Figure 10 Math Package .....	19
Figure 11 Geometry Package.....	20
Figure 12 Config Package.....	21
Figure 13 Entrypoint Package.....	22
Figure 14 GUI Package.....	22
Figure 15 Two Code Fragments Called By Separate Threads.....	23
Figure 16 Code Fragments Executed Sequentially.....	24
Figure 17 Code Fragments Executed Concurrently.....	25
Figure 18 State machine Solution to Race Condition.....	26
Figure 19 DrawableFactory Monitor Object.....	27
Figure 20 Valid and Invalid OpenGL API Call.....	28
Figure 21 Stack Trace of Error Detected at Run-time by OpenGL State Sniffer .....	28
Figure 22 Subdivision of Bezier Control Points .....	29
Figure 23 Bezier Curve Created with One Level of Subdivision .....	30
Figure 24 Bezier Curve with Two Levels of Subdivisions.....	30
Figure 25 Bezier Curve Created with Three Levels of Subdivision.....	31
Figure 26 Bezier Curve Created with Four Levels of Subdivision.....	31
Figure 27 Two Bezier Curves Smoothly Joined.....	32
Figure 28 Two Bezier Curves Disjointly Joined.....	32
Figure 29 Screenshot of the User Interface.....	33
Figure 30 Parameterization of the Vocal Tract Model .....	35
Figure 31 Opening of the Jaw .....	36
Figure 32 Opening of the Jaw with Lower Jaw Protruded Forward.....	37
Figure 33 Movement of the Tongue Tip.....	37
Figure 34 Movement of the Lips .....	38
Figure 35 Movement of the Hyoid.....	38
Figure 36 Movement of the Tongue Center.....	39
Figure 37 Movement of the Velum.....	39
Figure 38 Screenshot of Vocal Model Loaded onto the Graphical Modeler.....	40
Figure 39 Screenshot of the Slider Controls used to Animate the Model .....	41

## IV. GLOSSARY

1. **API:** Application Program Interface. It defines the services available for a program.
2. **Artisynth:** A modular component based system for performing simulations on the vocal tract and face
3. **ASY:** A two dimensional vocal tract model implemented by Haskins Laboratory
4. **Scene Graph:** A hierarchical structure to describe objects in a scene, and the relationships between objects

# 1. INTRODUCTION

The purpose of this project is to create a two dimensional articulatory vocal tract model for Artisynt.

This system allows users to rapidly construct a vocal tract model, and to test how varying configurations of the vocal tract can affect rendered speech output. The system is designed such that no programming or scripting is required. Therefore, this system is accessible to a wider range of users.

A computational model of the vocal tract is useful for studying the linguistics, and for examining how the configuration of the vocal tract affects the vocalization of sounds. The vocal tract model that was created is highly configurable, has a higher degree of articulatory freedom than past models such as ASY (Articulatory Synthesis), and has built in support for adding more articulatory movement to the vocal tract model if needed [1].

This project successfully created a computational model of the vocal tract from the ground up. It includes building a vector graphics drawing tool for creating vocal tract models, and a GUI (Graphical User Interface) based model animator tool for dynamically configuring the vocal tract model in real time. Additionally, there are software packages that were created to support rendering, computation of model transformations, and for saving or loading models. However, this system has not yet been integrated with the aero-acoustic renderer, which is used to generate speech based on the configuration of the vocal tract.

This report examines the project in the following sections: methodology, designs, discussion, and conclusion. The methodology section outlines the technology used for implementing the design, and the rationale for using the technology. The design section analyzes what subsystems were constructed, how they were constructed, and why they

were constructed. The discussion section examines what objectives were met, what objectives were not met, and suggestions for future improvements. Finally, the conclusion section summarizes the scope of the processes undertaken in the creation of this project's system.

## 2. METHODOLOGY

This section discusses the technology that was used to build the system, the possible approaches for building the system, an overall description of the entire system, and the general design concepts that were used.

### i. Technology Used

This system was constructed using Java and JOGL. Java was the language in which the system was constructed, and JOGL was the technology that allowed for efficient rendering of objects onto a display. Both Java and JOGL were chosen because they allowed for rapid development, easy Artisynt framework integration, and good performance.

Creating the vocal tract modeling system in Java allowed for easier integration into Artisynt, because Artisynt itself was constructed using the Java programming language. We also chose to use Java because system development time using Java is usually much faster than using older languages such as C++. One of the reasons is because Java is a managed-code programming language, in which the user does not have to handle memory allocation and deallocations. Therefore, we saved time in building this system by not having to write code to explicitly manage memory for the program. The Java garbage collection system would automatically cleanup objects residing in the memory that were no longer in use. In contrast, a C++ programmer must explicitly handle when to allocate or deallocate memory, which is time consuming and prone with errors. Furthermore, Java has built-in language support for multi-threading, in which languages such as C++ do not possess. Therefore, we chose Java as our programming language because it would allow us to build the system quickly, and would facilitate the system's integration into the Artisynt framework.

A key requirement for the system was the ability to render graphical objects onto the display system. This would allow the user to quickly view the current configuration of the vocal tract model and be able to view changes to the model. Of the rendering packages available, Java3D and JOGL were the ones considered.

JOGL was chosen as the technology for rendering graphics, because it is generally faster than Java3D, and it is the main renderer used in Artisynth. However, JOGL does have several weaknesses that Java3D does not have. Firstly, because JOGL is based on OpenGL, and OpenGL is intended for use in a procedural language such as C, it does not fit the object-oriented architecture and multi-threaded approach of typical Java programs. JOGL is not multi-thread safe, and care must be taken to ensure that JOGL API calls do not occur concurrently from separate threads. In contrast, Java3D was designed to be an object-oriented API that supports a multi-thread safe approach to rendering. Therefore, Java3D is easier to incorporate into Java programs. In terms of performance, JOGL is faster than Java3D, because JOGL can be viewed as a thin wrapper around the OpenGL API. In contrast, Java3D is not a thin wrapper around a rendering API, but is actually a full featured graphics library that is built on top of a drawing API such as DirectX or OpenGL. Another advantage of using JOGL for rendering the vocal tract modeling system is that Artisynth also used JOGL for rendering. Therefore, integration of this system into Artisynth's framework would be more efficient, because JOGL rendering calls would not have to be extensively modified. Therefore, JOGL was chosen because of its good performance, and because it would allow for easier integration of the system into Artisynth.

## **ii. Possible Approaches**

### ***1. System Implementation***

There were three approaches considered for implementing the system. The first approach would be to analyze the source code for the ASY system, and to port the model into the

Artisynth framework. The second option would be to construct a rigid-body or particle-system model of the vocal tract using the existing modeling tools in Artisynth. The final option would be to create a completely new system for modeling the vocal tract.

The advantage of porting the ASY system's vocal tract model is that development time is shortened. This is because algorithms and designs can be copied from the ASY system. However, copying the ASY system directly involves no innovation, and does not provide much of a challenge for a student project. The second option was to implement the vocal tract as a rigid-body or particle-system model using Artisynth. The advantage is that we could use the existing Artisynth modeling framework, instead of having to create a new one. However, we wanted to create a 2D model with curved surfaces, and Artisynth at that time did not contain any tools for such a function. An alternative would be to use multiple line segments to approximate a curve, but creating such a model would be tedious and complicated because of the large number of line segments required. The final option was to create a new system for modeling the vocal tract. The advantage is that we could quickly prototype custom modeling tools in the new system, which is not easily achievable in a large and established system such as Artisynth. We can therefore quickly add and test tools, which can model objects such as curved surfaces. The disadvantages to creating a new system are increased development time, and the requirement that the system must be able to merge into Artisynth once the model has been shown to work properly. Nevertheless, we chose to create a new system, because it would be more innovative than porting the ASY system, and allows for quicker testing of new modeling tools since they do not have to be immediately a part of Artisynth.

## ***2. Animation***

A vocal tract model consists of a set of control points, which determines its shape. To animate the model, we must determine which control points to move, and how to move the control points.

One option is to create a non-constrained model, in which all the control points have two translational degrees of freedom. This will create a model in which it is possible to move the control points independently to any location. The advantage of this method is flexibility, since the model can be configured without restriction. However, this method does not allow for complex controls, such as rotating a control point, or moving a set of control points in unison. The second option is to create a constrained model with a complex set of controls. This option would constrain some control points on the model to only certain kinds of movements, such as rotational movement. Additionally, this option allows for grouping of control points for more complex movements, such as rotating all the control points that define the jaw around the TMJ (temporomandibular joint). However, a constrained model is less flexible than an unconstrained model, since not all control points have two degrees of freedom.

For our system, we chose to use both the non-constrained and the constrained approach. To custom fit the model to a vocal tract outline from an MRI (magnetic resonance imaging) image, it would be best to allow the user to interact with a non-constrained view of the model. When the user wants to animate the model, it would be best to present to the user a constrained view of the model. The constrained view allows complex movements, which is better for simulating vocalizations.

### ***3. Modeling of Vocal Tract***

The two options considered for creating the model, was to construct it out of line segments, or to construct it using Bezier curves. Constructing the model out of line segments is the easiest method, because the equation for straight lines is a simple linear equation, and JOGL supports the rendering straight lines. The downside is that straight lines do not closely model the curved surfaces of a vocal tract. The second option would be to use Bezier curves, which allows us to create a model that closely fits the curved surfaces of the vocal tract. However, Bezier curves are more mathematically complex than straight lines, and implementing an algorithm to create a Bezier curve is non-trivial.

For our vocal tract model, we chose to use Bezier curves, because we believed the additional development time required to implement a Bezier curve algorithm is worth the benefit of having a model that closely fits an actual vocal tract profile.

#### ***4. Data Persistence***

Data persistence allows users to store the model they designed, and the configuration of the model. This is important, because without data persistence, the user would be required to create and configure a vocal tract model each time the system is executed. The two options considered for data persistence were Java Serialization, and simple text files.

Java Serialization is a technology that allows objects to be converted into a byte stream at run-time. If the model of the vocal tract is represented as a set of objects in the program, we can use Java Serialization at run-time to convert the object into a byte stream, and save this stream into a file on a disk. To load the model, we can read the byte stream from the file, and convert it back into the set of objects that describes the vocal tract model using Java Serialization. This approach to data persistence is very simple and fast to implement. It allows complex objects to be easily stored and retrieved without the need for any file parsing. However, this method has a significant flaw. We shall illustrate this flaw with the following example. Let us have the user save the model into a file using serialization. Afterwards, the developer decides to modify the programming code that was used to describe the model. If the user tries to load up the model using the new version of the program, Java Serialization will not work. The reason is that the object that was saved into a file is an older version than the object that was coded by the developer. Therefore, Java Serialization is not a good method for data persistence if we know that our code will be frequently modified.

The other option is to save the model in a text file format. The advantages of this approach are that the data is stored in a human readable format, and modifying the source

code would not cause the saved model file to become useless. However, creating tools to save and load data to a text file is very time consuming. We must develop a file format to save the data with, and we must develop a method to correctly read and write the model data into a text file. Nevertheless, we chose to save the model into a text file, because we are constantly modifying the source code, which would make the files saved using Java Serialization useless.

### **iii. System Description**

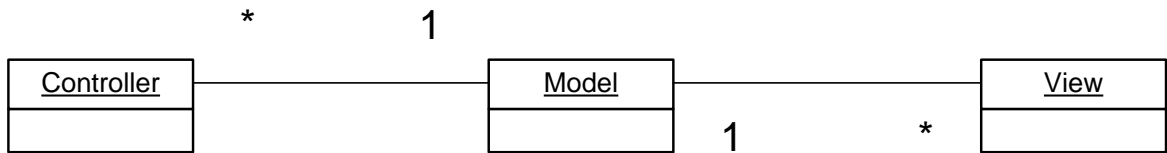
The vocal tract modeling system consists of two main components. The first component is a vector graphics drawing tool, which allows a user to rapidly construct a vocal tract outline using Bezier curves. The second component is a model animator, which allows a user to control a set of sliders to animate the vocal tract.

The vector graphics drawing tool is a more flexible component than the model animator. A user can construct any shape desired using the drawing tool. In contrast, the current implementation of the model animator is not flexible, because it does not allow the user to customize a set of controls for a custom model. This is because the current model animator system is fixed at compile-time. Fortunately, the architecture of the system can allow for the addition of the feature in which a user can specify a set of controls for a custom model. However, we have chosen to develop a fixed model animator because it takes less time to implement than a flexible model animator does.

### **iv. Design Concepts**

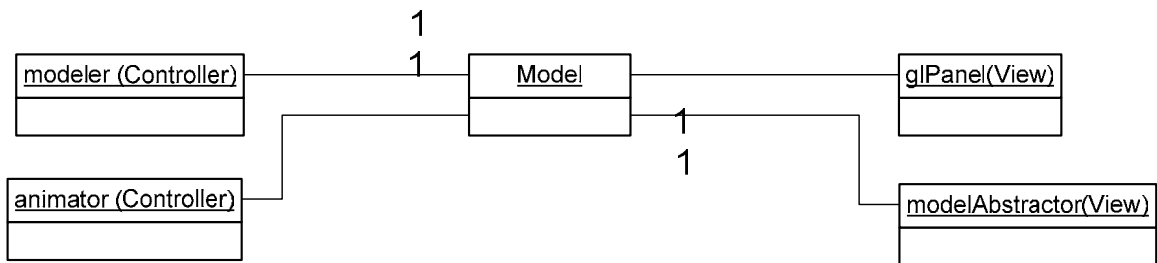
The design of the vocal tract modeling system follows the MVC (model view controller) design pattern (Figure 1). In this design pattern, only the Model is responsible for maintaining domain knowledge. The Controller is a set of mutator functions that modifies the domain knowledge within the Model and is the interface that interacts with the user. The View is responsible for presenting the domain knowledge to the user. The MVC

pattern was chosen, because it is an elegant method of separating the problem of maintaining, manipulating, and presenting a model into three smaller problem domains.



**Figure 1 Model View Controller (MVC) Pattern**

The MVC pattern that is used in our system is presented in Figure 2. It can be seen that there are two Controllers and two Views connected to one Model. The glPanel View is responsible for presenting a rendered image of the Model to the user. The modelAbstractor View presents a view of the Model that is abstracted with Mermelstein parameters [1]. The modeler Controller is responsible for creating and manipulating shapes that form the Model. Finally, the animator Controller is responsible only for moving control points on the model. By using the MVC pattern, we were able to create a system that is easily maintainable because tasks such as manipulation, maintenance, and view are encapsulated in their own subsystems.



**Figure 2 MVC Pattern Used in System**

### 3. DESIGNS AND IMPLEMENTATIONS

This section examines the use cases for the system, and the major subunits that were designed and implemented for the system.

#### i. Use Case Diagram for the System

The following use case diagram illustrates the behavior of the system as seen from the point of view of the user (Figure 3).

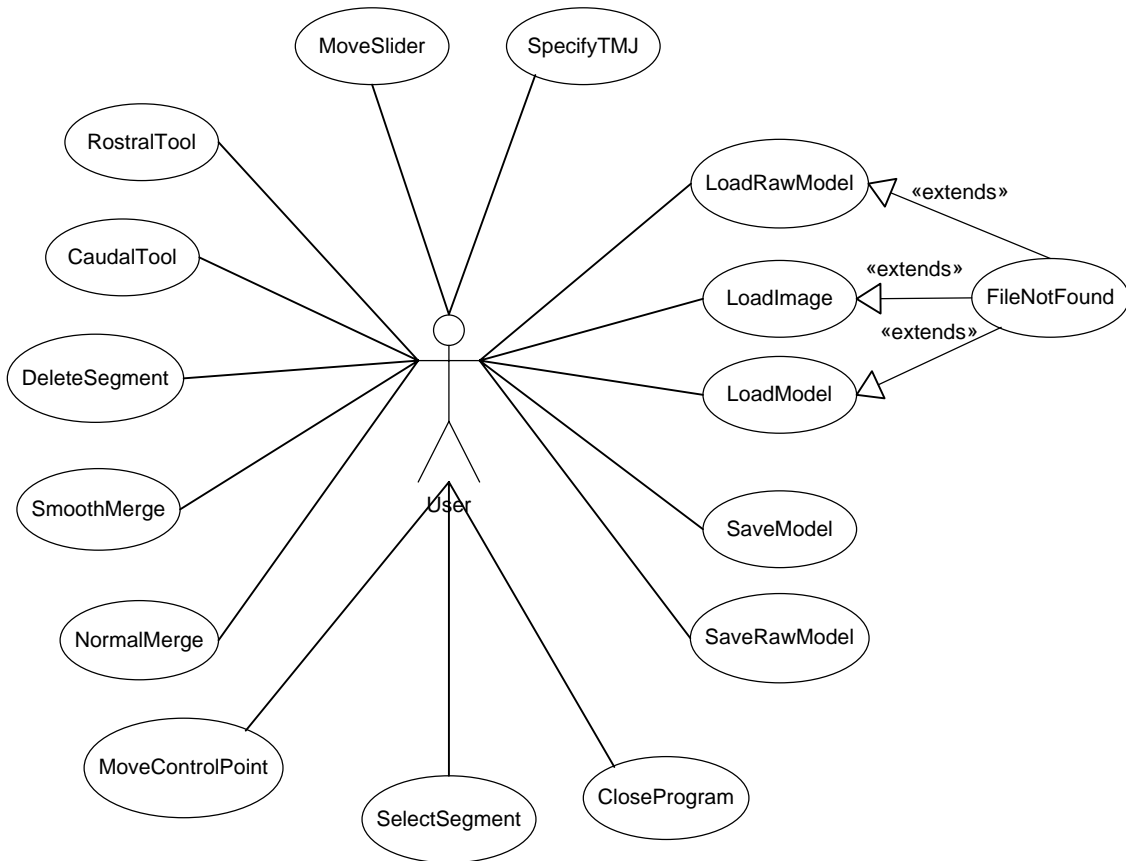


Figure 3 Use Case Diagram

## ii. Use Case Descriptions

<b>Use-Case Name</b>	LoadRawModel
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User activates the LoadRawModel function
<b>Use Case</b>	<ol style="list-style-type: none"> <li>1. GUI presents an open file dialog</li> <li>2. User selects file to open</li> </ol>
<b>Exit Condition</b>	The raw model is presented to the user via the GUI

<b>Use-Case Name</b>	LoadModel
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User activates the LoadModel function
<b>Use Case</b>	<ol style="list-style-type: none"> <li>1. GUI presents an open file dialog</li> <li>2. User selects file to open</li> </ol>
<b>Exit Condition</b>	The model is presented to the user via the GUI. The sliders for the animator tools are presented in a separate window via the GUI.

<b>Use-Case Name</b>	LoadImage
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User activates the LoadImage function
<b>Use Case</b>	<ol style="list-style-type: none"> <li>1. GUI presents an open file dialog</li> <li>2. User selects file to open</li> </ol>
<b>Exit Condition</b>	Image is presented to the user via the GUI.

<b>Use-Case Name</b>	SaveModel
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User activates the SaveModel function
<b>Use Case</b>	<ol style="list-style-type: none"> <li>1. GUI presents a save file dialog</li> <li>2. User enters a filename to save</li> </ol>
<b>Exit Condition</b>	Model is saved

<b>Use-Case Name</b>	SaveRawModel
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User activates the SaveRawModel function
<b>Use Case</b>	<ol style="list-style-type: none"> <li>1. GUI presents a save file dialog</li> <li>2. User enters a filename to save</li> </ol>
<b>Exit Condition</b>	Raw model is saved

<b>Use-Case Name</b>	CloseProgram
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User activates the CloseProgram function
<b>Use Case</b>	None
<b>Exit Condition</b>	System is shutdown

<b>Use-Case Name</b>	SelectSegment
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User clicks on a curve segment using a mouse
<b>Use Case</b>	None
<b>Exit Condition</b>	Curve segment is selected

<b>Use-Case Name</b>	MoveControlPoint
<b>Participating Actors</b>	User
<b>Entry Condition</b>	Mouse cursor is hovering above a control point
<b>Use Case</b>	1. Control point moves according to the position of the dragged mouse
<b>Exit Condition</b>	Control point is moved to the location where the user releases the mouse button

<b>Use-Case Name</b>	NormalMerge
<b>Participating Actors</b>	User
<b>Entry Condition</b>	Mouse cursor is hovering above two endpoints of two separate Bezier curves
<b>Use Case</b>	None
<b>Exit Condition</b>	The two Bezier curves are merged together non-smoothly

<b>Use-Case Name</b>	SmoothMerge
<b>Participating Actors</b>	User
<b>Entry Condition</b>	Mouse cursor is hovering above two endpoints of two separate Bezier curves
<b>Use Case</b>	None
<b>Exit Condition</b>	The two Bezier curves are merged together smoothly

<b>Use-Case Name</b>	DeleteSegment
<b>Participating Actors</b>	User
<b>Entry Condition</b>	SelectSegment use case was valid, and user activates DeleteSegment function
<b>Use Case</b>	1. Dialog window prompts user to confirm whether or not to delete segment
<b>Exit Condition</b>	Segment is deleted

<b>Use-Case Name</b>	CaudalTool
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User activates the CaudalTool function
<b>Use Case</b>	None
<b>Exit Condition</b>	User's drawing tool is now configured for drawing the Caudal surface of the vocal tract

<b>Use-Case Name</b>	RostralTool
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User activates the RostralTool function
<b>Use Case</b>	None
<b>Exit Condition</b>	User's drawing tool is now configured for drawing the Rostral surface of the vocal tract

<b>Use-Case Name</b>	MoveSlider
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User activates the MoveSlider function
<b>Use Case</b>	<ol style="list-style-type: none"> <li>1. Slider value changes according to the position in which the user sets the slider</li> <li>2. The new slider value changes the configuration of the vocal tract model</li> </ol>
<b>Exit Condition</b>	The GUI presents to the user a new configuration of the vocal tract model according to the new value of the slider

<b>Use-Case Name</b>	SpecifyTMJ
<b>Participating Actors</b>	User
<b>Entry Condition</b>	User activates the SpecifyTMJ function
<b>Use Case</b>	<ol style="list-style-type: none"> <li>1. Dialog box opens to prompt user to enter an x-coordinate value</li> <li>2. User enters an x-coordinate value</li> <li>3. Dialog box opens to prompt user to enter an y-coordinate value</li> <li>4. User enters a y-coordinate value</li> </ol>
<b>Exit Condition</b>	The TMJ position for the model is updated

### iii. Sequence Diagrams

#### 1. MoveSlider Sequence Diagram

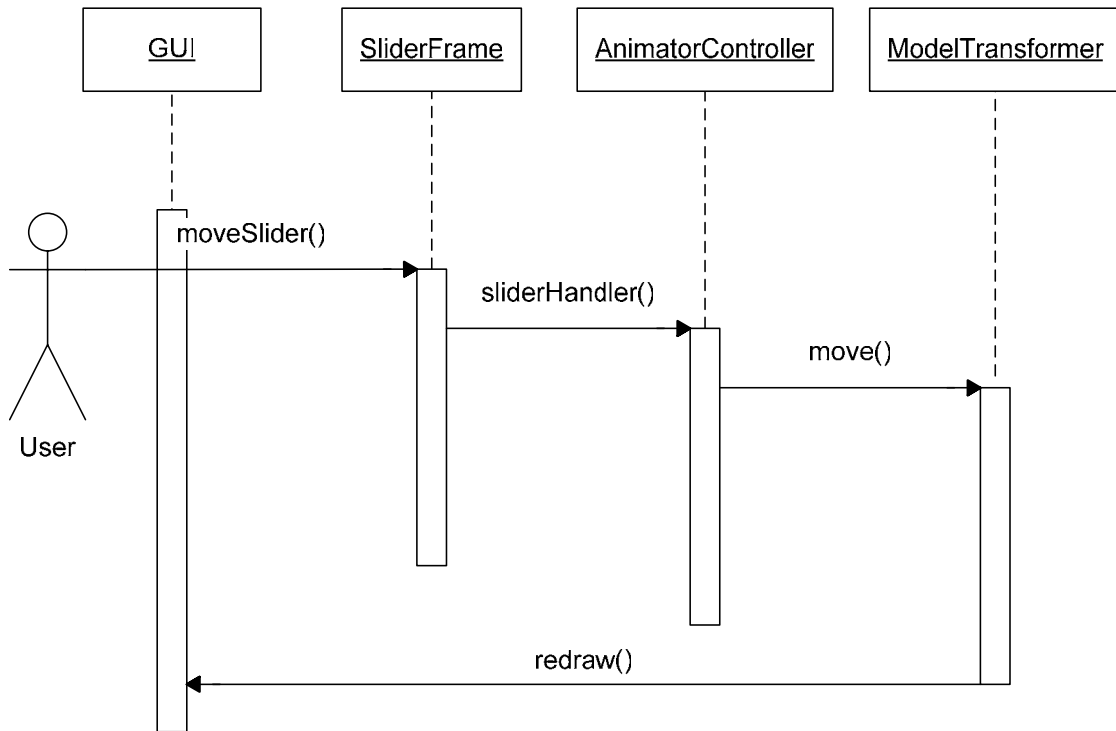


Figure 4 Move Slider Sequence Diagram

## 2. SaveModel Sequence Diagram

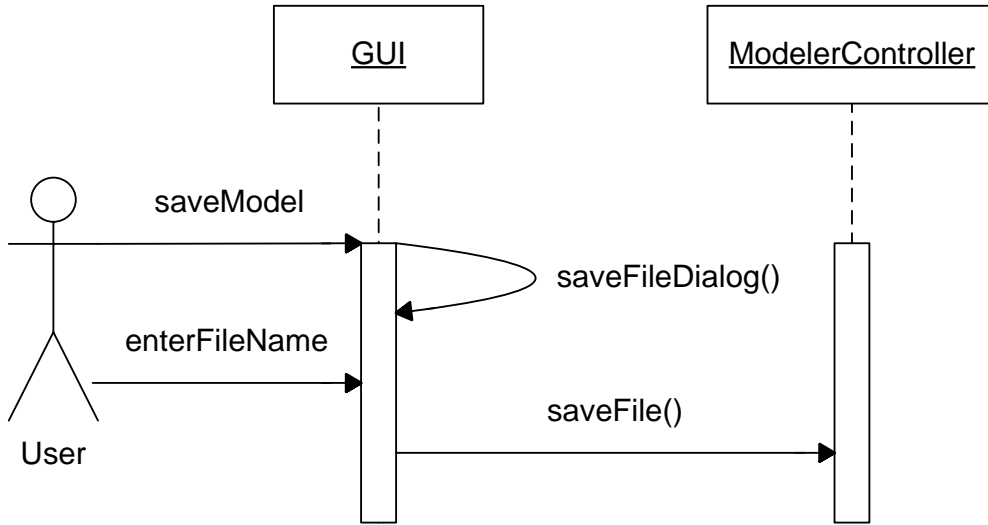


Figure 5 SaveModel Sequence Diagram

## iv. Class Diagrams

### 1. Geometry Class Diagram

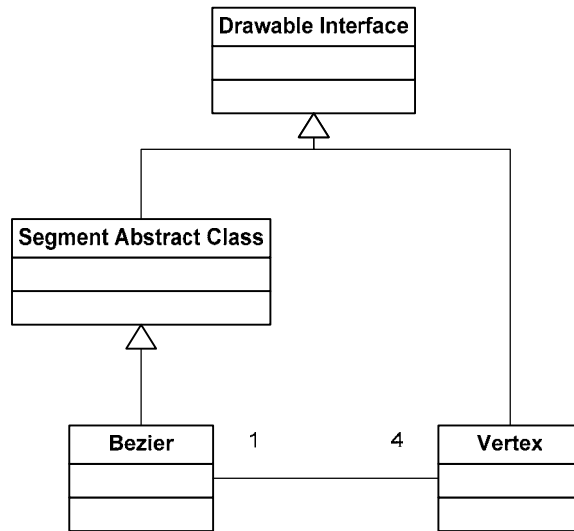
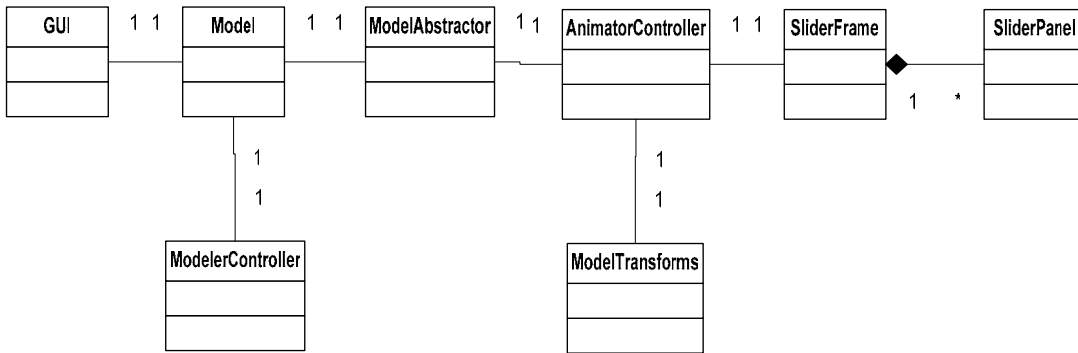


Figure 6 Relationships between Geometry Classes

It can be seen in Figure 6, that Bezier curves and vertices both inherit the Drawable interface. All objects that can be drawn on the GUI must implement this interface. The Segment Abstract Class is a generalization of segments, such as line segments and curve segments. Therefore, this design allows us to add a line segment object, which will inherit the Segment Abstract Class.

## 2. Overall Class Diagram of System



**Figure 7 Relationships between Major Classes of the System**

The major classes of the system, consists of the GUI, Model, ModelController, ModelAbstractor, AnimatorController, ModelTransforms, SliderFrame, and the SliderPanel (Figure 7). The GUI is the only class that has the ability to present a view of the Model to the user. The ModelerController is the interface with which the user can manipulate the model. The ModelAbstractor presents a Mermelstein parameterized view to the AnimatorController [1]. The AnimatorController is responsible for animating the Model. ModelTransforms encapsulates the transformations required to actuate the Mermelstein parameters. The SliderFrame consists of zero or more SliderPanels, which function to allow the user to animate the Model.

## v. Package to Class Mappings

### 1. Animator Package



Figure 8 Animator Package

The Animator Package consists of classes, which are responsible for animating the model (Figure 8). The SliderFrame and SliderPanel are the interfaces for manipulating the model. The AnimatorController receives updates from the sliders, and uses the ModelTransforms class to manipulate the model via the ModelAbstractor.

## 2. Utilities Package



**Figure 9 Utilities Package**

The Utilities Package consists of tools for loading images, filtering files, detecting errors, and generating unique keys. The TextureLoader class is a JOGL texture loader, which permits the loading of PNG, JPG, and BMP files. The file filters are used to allow users to open only certain file types. The ModelFileFilter restricts the system to loading only files with a .model extension. The ImageFileFilter restricts the system to loading only .png, .jpg, and .bmp files. The RawModelFileFilter restricts the system to loading only .raw extension files. GLErrorStateSniffer is a tool for detecting error codes in the JOGL state machine. IDFactory and KeySet work together to create a subsystem that generates unique keys.

### 3. Math Package

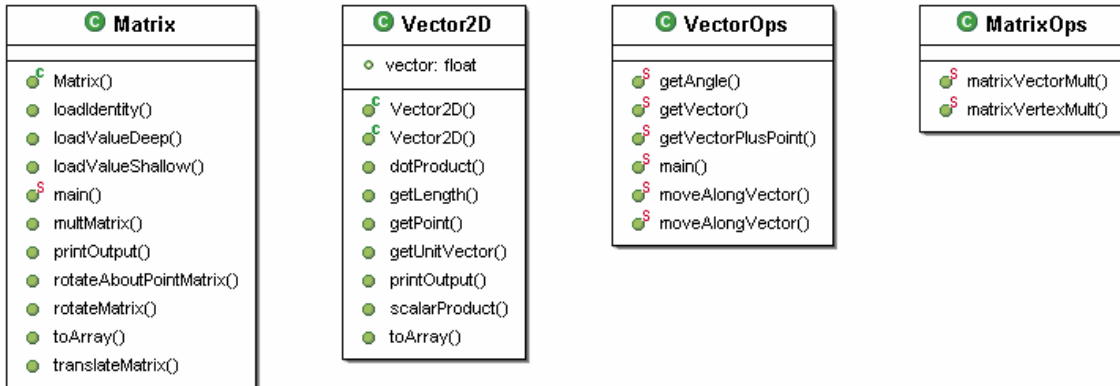


Figure 10 Math Package

The Math Package consists of classes used for affine transformations. Matrix and Vector2D are classes that represent a 3 by 3 matrix, and a 3 by 1 column vector. Vector2D is a 3 by 1 column vector because transformations are done using homogeneous coordinates. VectorOps and MatrixOps are utility classes used for calculating common vector and matrix operations.

#### 4. Geometry Package

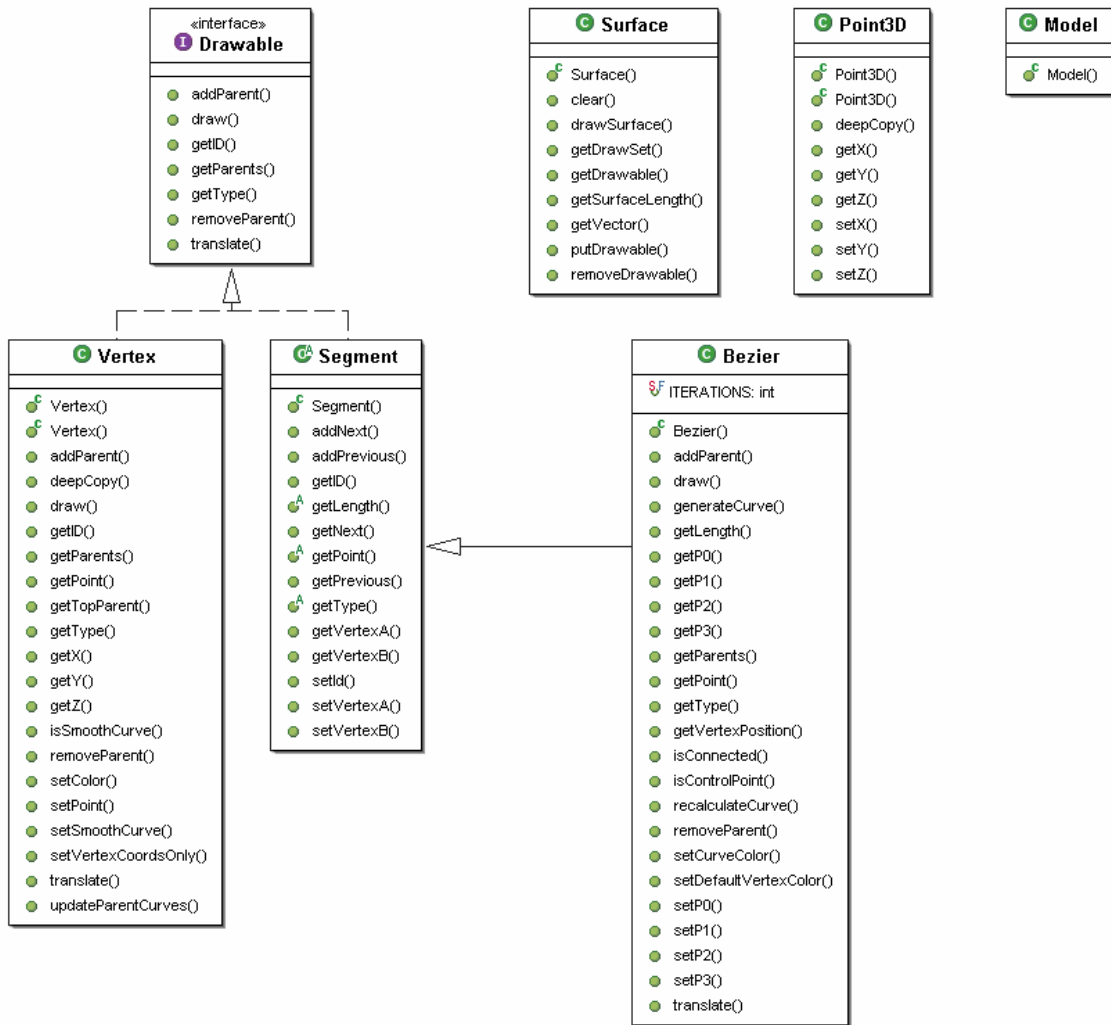
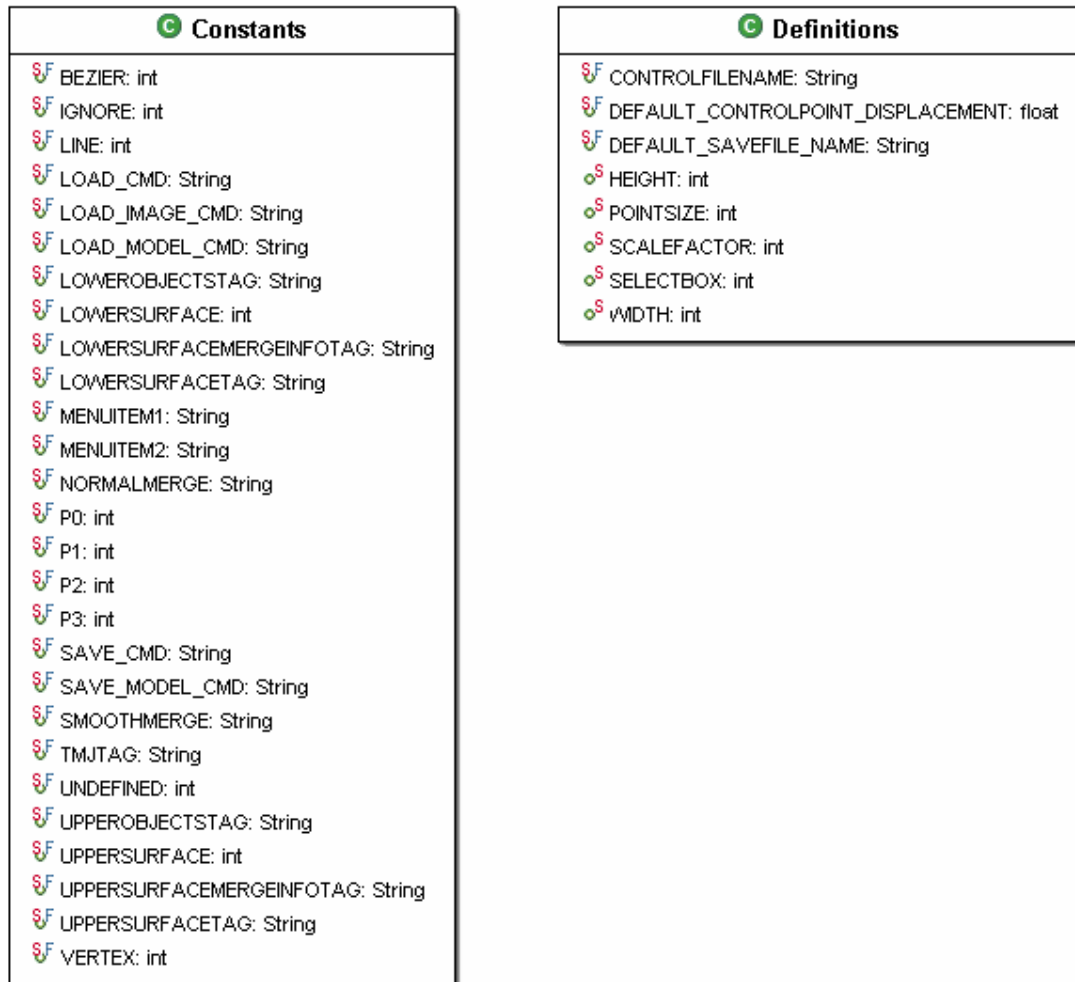


Figure 11 Geometry Package

The geometry package consists of classes used for defining the layout of the model. A model consists of two surfaces, with each surface consisting of one or more segments.

## 5. Config Package



**Figure 12 Config Package**

The Config Package consists of classes which contain definitions and constants that are used in the system. The Definitions class contains data that affect the appearance and the function of the GUI. The Constants class contains parameters that are global to the entire system.

## 6. Entrypoint Package

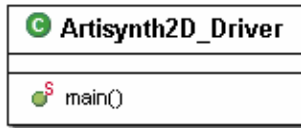


Figure 13 Entrypoint Package

The Entrypoint Package contains the class that initializes the entire system.

## 7. GUI Package

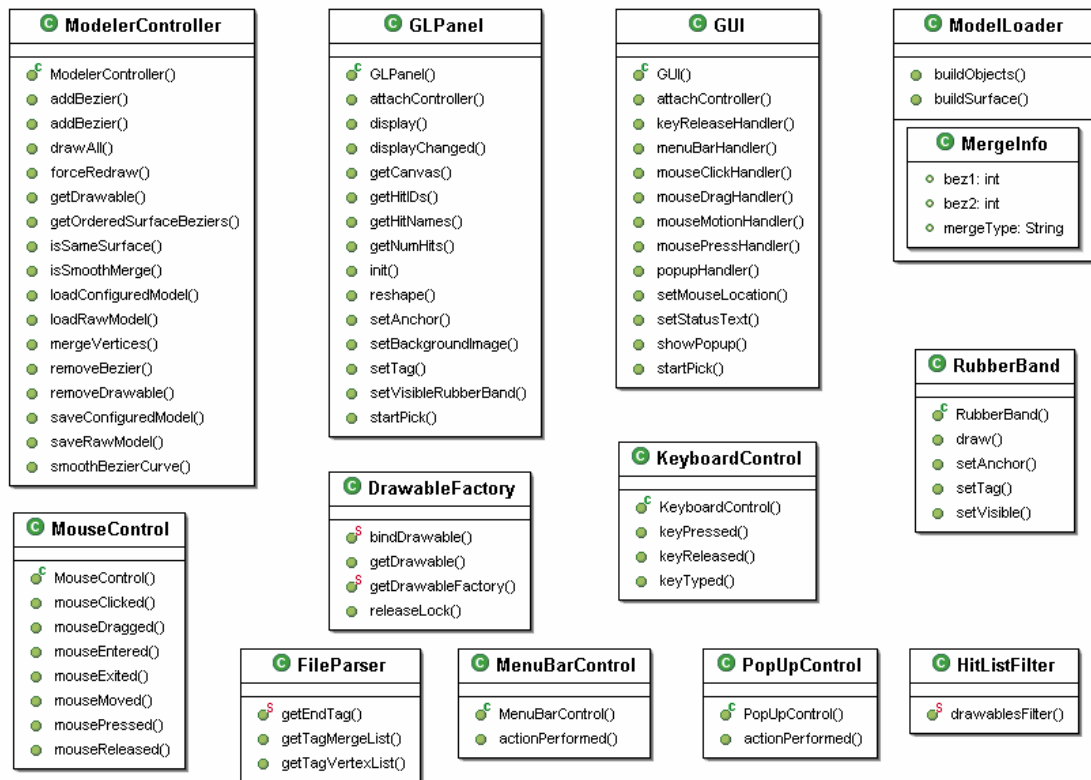


Figure 14 GUI Package

The GUI package consists of classes which implements the user interface.

### iii. Vocal Tract Modeling Tool

This section discusses the notable subunits within the modeler system. The subunits are not subsystems, but are portions of the implementation that is innovative or technically challenging.

#### 1. *OpenGL Monitor Object*

A difficulty arises when using JOGL, because it does not support safe multi-thread access to the OpenGL state-machine. Java programs often are multithreaded, especially programs that utilize Swing, because there is a separate thread used for handling Swing components. Oftentimes, we would want to allow the user to modify a rendered scene using the GUI. For example, the user may click a button to change the color of the background. Unfortunately, the thread that handles the GUI actions is not the same thread that handles the OpenGL calls. Therefore, we may have two threads attempting to access the OpenGL state-machine concurrently. JOGL places no restriction on whether or not multiple threads can access the OpenGL state machine. Without multi-thread safe access to OpenGL, we must find a method that prevents problems with concurrent modifications to the OpenGL state machine. We shall first illustrate the multi-thread problem as follows.

<pre>//Draw Blue Line glColor3f(0,0,1); glBegin(glLines); glVertex3f(0,0,0); glVertex3f(1,1,0); glEnd();</pre>	<pre>//Draw Red Line glColor3f(1,0,0); glBegin(glLines); glVertex3f(0,0,0); glVertex3f(10,10,0); glEnd();</pre>
--	---

Figure 15 Two Code Fragments Called By Separate Threads

Figure 15 lists two code fragments that are called by separate threads. If the threads execute the code fragments sequentially, then the program will have no error, and we will be able to draw one blue line and one red line (Figure 16).

	<b>Thread 1</b>	<b>Thread 2</b>
Thread 1 is running	<pre>//Draw Blue Line glColor3f(0,0,1); glBegin(GL_LINES); glVertex3f(0,0,0); glVertex3f(1,1,0); glEnd();</pre>	
Thread 2 is running		<pre>//Draw Red Line glColor3f(1,0,0); glBegin(GL_LINES); glVertex3f(0,0,0); glVertex3f(10,10,0); glEnd();</pre>

**Figure 16 Code Fragments Executed Sequentially**

However, if the two threads executed the code fragments concurrently, we can get a situation in which the blue line is drawn as a red line (Figure 17). The situation that we have here is a race condition, because the outcome of the program depends on which thread can execute its code fragment first.

	<b>Thread 1</b>	<b>Thread 2</b>
Thread 1 is running	//Draw Blue Line glColor3f(0,0,1);	//Draw Red Line
Thread 2 is running		glColor3f(1,0,0);
Thread 1 is running	glBegin(glLines); glVertex3f(0,0,0); glVertex3f(1,1,0); glEnd();	
Thread 2 is running		glBegin(glLines); glVertex3f(0,0,0); glVertex3f(10,10,0); glEnd();

**Figure 17 Code Fragments Executed Concurrently**

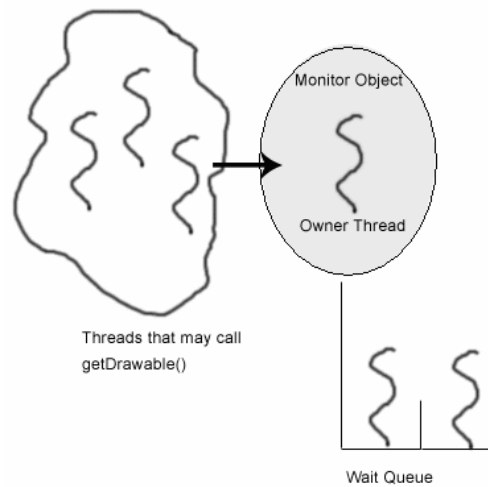
One possible solution to handling race conditions is to create a state machine. In Figure 18, we prevented the race condition by using the state variable **drawRedLine**. Thread 2 no longer competes with Thread 1 to execute the code fragments, therefore there is no race condition. Most programs written with JOGL use the state-machine solution for handling concurrency problems. However, this method is tedious and error-prone because the developer must create and manage a state-machine.

Thread 1	Thread 2
<pre> Class Renderer{   boolean drawRedLine=false;    public void display(){     while(true){       glColor3f(0,0,1);       glBegin(GL_LINES);        glVertex3f(0,0,0);       glVertex3f(1,1,0)        glEnd();        if(drawRedLine==true){         glColor3f(1,0,0);         glBegin(GL_LINES);         glVertex3f(0,0,0);         glVertex3f(10,10,0);         glEnd();       }     }   } } </pre>	<pre> //draw red line renderer.drawRedLine=true; </pre>

**Figure 18 State machine Solution to Race Condition**

Instead of using a state-machine to solve the concurrency problems, we created a monitor object to wrap the OpenGL state machine to make it multi-thread safe. The monitor object was implemented using the Singleton pattern, such that only one object of its kind can exist in the system. This is intuitive because we should only have reference to one OpenGL state machine in the system. In our system, we call the monitor object the DrawableFactory (Figure 19). There is no public constructor for the monitor, because we do not want users to be able to create multiple instances of the object. Instead, a static function returns a reference to the monitor object. The interesting part of the DrawableFactory, is the method `getDrawable()`. This method returns a Drawable object, which contains the reference to the OpenGL state machine. If no thread possesses the DrawableFactory monitor, then a thread that calls `getDrawable()` will get the Drawable object. However, if there is a thread that currently possesses the DrawableFactory

monitor, then a thread that calls `getDrawable()` will be placed into a wait queue (Figure 19). The thread that is in the wait queue is then placed into a sleep state. When the thread that has possession of the `DrawableFactory` monitor gives up control of the monitor, it wakes up a thread in the wait queue, and that thread is given possession of the monitor.



**Figure 19 DrawableFactory Monitor Object**

The advantage of using a monitor object is to provide multi-thread safe access to the OpenGL without the need to create a state machine. With the `DrawableFactory` monitor, any thread can safely access OpenGL without worrying that it may interfere with OpenGL calls from other threads. Although creating the monitor object is time consuming, we saved a lot of time in the implementation because we did not have to create and maintain a state-machine.

## 2. *OpenGL State Sniffer*

Detecting errors in OpenGL API calls is difficult because they usually do not cause any compile-time or run-time errors. For example, the incorrect code in Figure 20 generates no compile-time or run-time errors, but is an invalid OpenGL API call.

Incorrect API Call	Correct API Call
glBegin(GL_LINE)	glBegin(GL_LINES)

**Figure 20 Valid and Invalid OpenGL API Call**

Since no errors are generated, incorrect API calls are usually undetected. Therefore, there may be errors in the system that may occur unnoticed. One possible solution would be to use the `glGetError()` API call to detect if there had been any invalid API calls. Unfortunately, this method only detects the error, and does not provide a stack trace that will aid in debugging. We have developed a utility tool called the OpenGL state sniffer. The advantage of using this tool is that it generates a stack trace when an incorrect API call occurs. This tool greatly helps in detecting and finding the location of incorrect API calls. In Figure 21, a stack trace is thrown when an incorrect OpenGL API call is used. We can see that the error is located on line 165 of the Bezier class.

```

java.lang.Exception: Non-Fatal GLError Code 1280
    at utilities.GLErrorStateSniffer.getGLErrorState(GLErrorStateSniffer.java:33)
    at geometry.Bezier.draw(Bezier.java:165)
    at geometry.Surface.drawSurface(Surface.java:53)
    at gui.ModelerController.drawAll(ModelerController.java:252)
    at gui.GLPanel.drawObjects(GLPanel.java:201)
    at gui.GLPanel.display(GLPanel.java:183)

```

**Figure 21 Stack Trace of Error Detected at Run-time by OpenGL State Sniffer**

### ***3. Object Picking and Selection***

Object picking systems are essential for graphical editing programs where users may want to manipulate objects on the screen by selecting them using a mouse. Our system detects objects around the vicinity of the mouse cursor by rendering objects in a small viewing volume around the cursor. Objects that are rendered inside this viewing volume will register as being selected by the mouse. Objects that are completely clipped by the viewing volume will not be registered as being selected. The advantage of using this method is that OpenGL has a rendering mode built specifically for such a task. The selection rendering mode does not render objects for displaying, but is used to test which objects will not be completely clipped by the viewing volume.

#### 4. Bezier Curves

Bezier curves are arcs that are specified using four control points. We shall designate the four control points as  $P_0$ ,  $P_1$ ,  $P_2$ , and  $P_3$  (Figure 22). The first control point  $P_0$ , and the last control point  $P_3$  lie along the curve. However, the control points  $P_1$  and  $P_2$  do not lie along the curve. Instead, the control points of  $P_1$  and  $P_2$  are used to designate the slope of the line at  $P_0$  and  $P_3$  respectively.

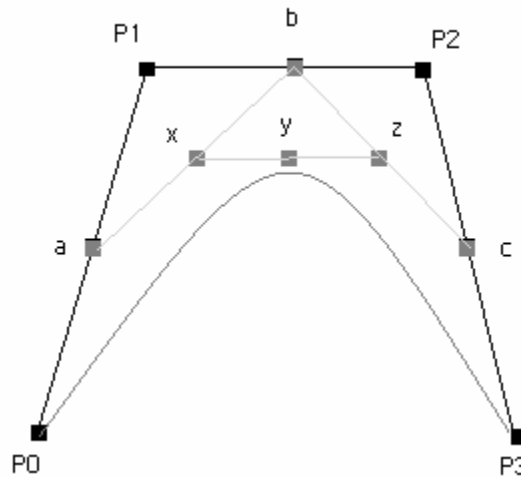
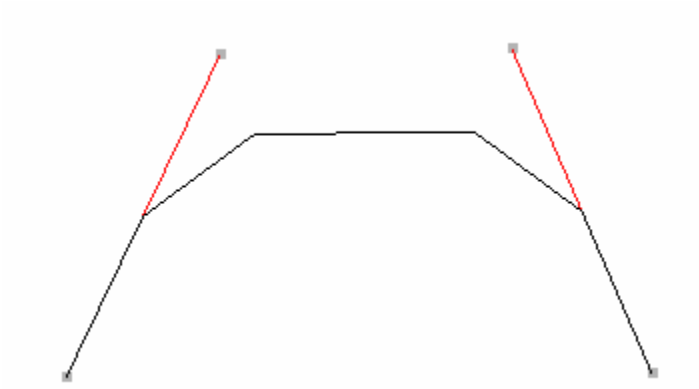


Figure 22 Subdivision of Bezier Control Points

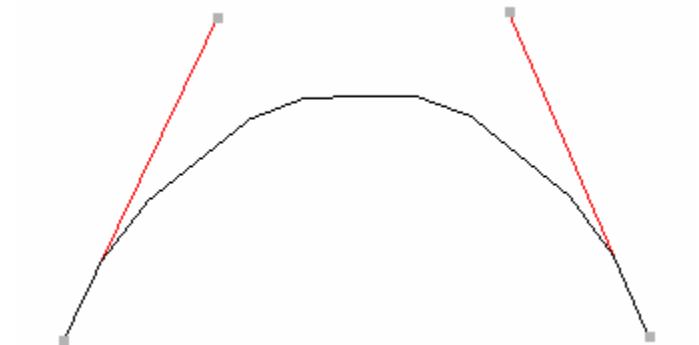
In this system, Bezier curves were constructed using a subdivision algorithm [2]. This method divides a set of four control points into two sets of four control points. The four control points in each of the two sets are then further subdivided. As the number of subdivisions increases, the control points generated approaches the curvature of a Bezier curve.

The algorithm for the subdivision of a set of four control points is as follows. Point 'a' is the midpoint of  $P_0$  and  $P_1$ . Point 'b' is the midpoint of  $P_1$  and  $P_2$ . Finally, Point 'c' is the midpoint of  $P_2$  and  $P_3$ . Similarly, point 'x' is the midpoint of points 'a' and 'b'. Point 'z' is the midpoint of points 'b' and 'c'. Finally, point 'y' is the midpoint of 'x' and 'z'. The

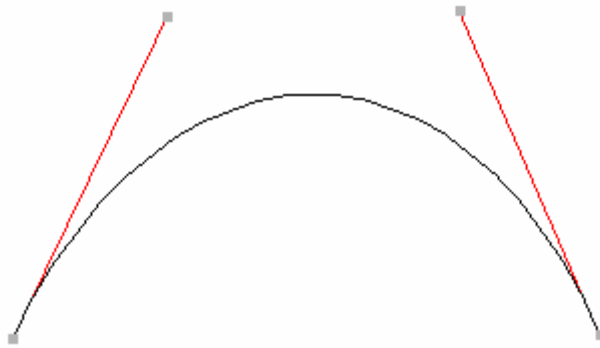
points  $P_0$ , 'a', 'b', 'x', and 'y' are then the four control points of the subdivided Bezier. Points 'y', 'z', 'c', and  $P_3$  are the four control points of the other half of the subdivided Bezier. These two sets of points can then be further subdivided by repeating the algorithm.



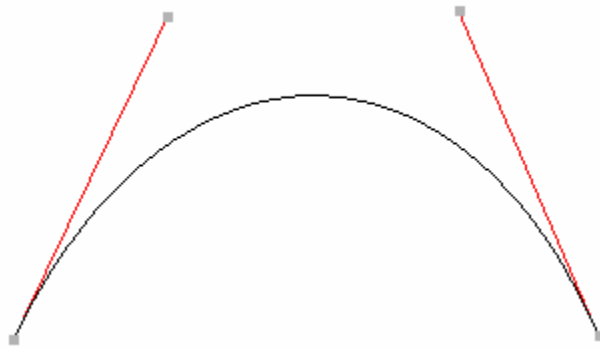
**Figure 23 Bezier Curve Created with One Level of Subdivision**



**Figure 24 Bezier Curve with Two Levels of Subdivisions**



**Figure 25 Bezier Curve Created with Three Levels of Subdivision**



**Figure 26 Bezier Curve Created with Four Levels of Subdivision**

In Figure 23, we can see that one level of subdivision does not yield a smooth looking curve. Two levels of subdivision, as shown in Figure 24, shows that the curve is also not smooth. At three levels of subdivision, the curve that is generated is reasonably smooth. Finally, at four levels of subdivision, the curve that is generated is very smooth. However, using four levels of subdivisions does not always create a smooth curve. In general, the further apart the control points are, the greater the number of subdivisions that is needed to create a smooth looking curve.

## 5. Surfaces: Piece-wise Bezier Curves

Surfaces of the vocal tract are modeled by joining Bezier curves in a piece-wise fashion. This system supports joining Bezier curves using smooth join and disjoint join methods. For the smooth join method, the two Bezier curves are joined such that the curve around the joining point maintains its smoothness. This is accomplished by having the two control points that neighbor the joining point to be collinear to each other (Figure 27).

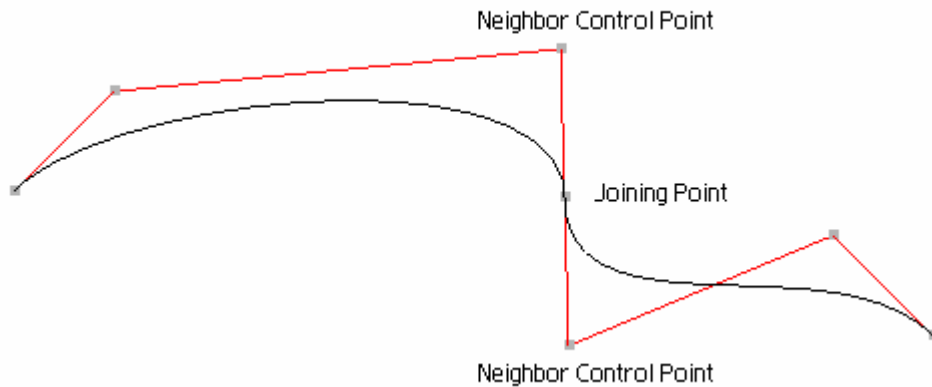


Figure 27 Two Bezier Curves Smoothly Joined

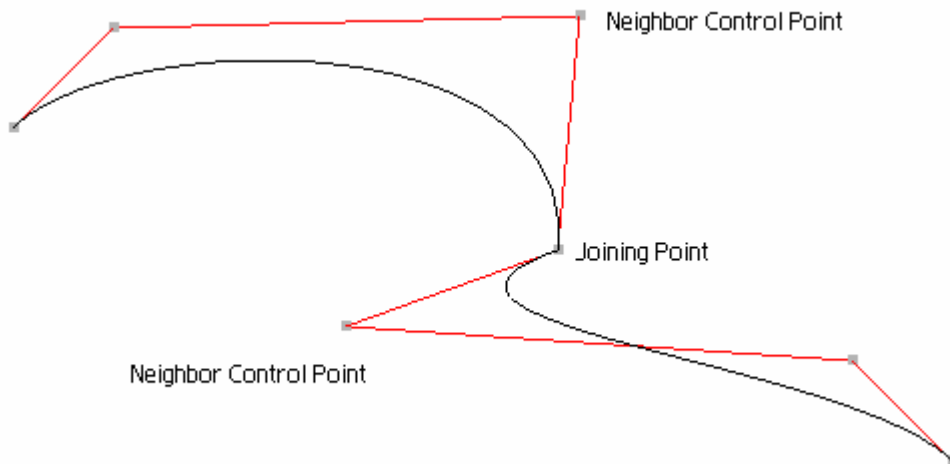


Figure 28 Two Bezier Curves Disjointly Joined

The disjoint join method allows two Bezier curves to be joined without maintaining smoothness. This is accomplished by not forcing the constraint that neighboring control points must be collinear to each other (Figure 28). The option of having two different joining methods available increases the choices a user has in modeling the vocal tract. By increasing the modeling choices available, this can allow users to construct are more realistic vocal tract model.

## 6. Interface Screenshot

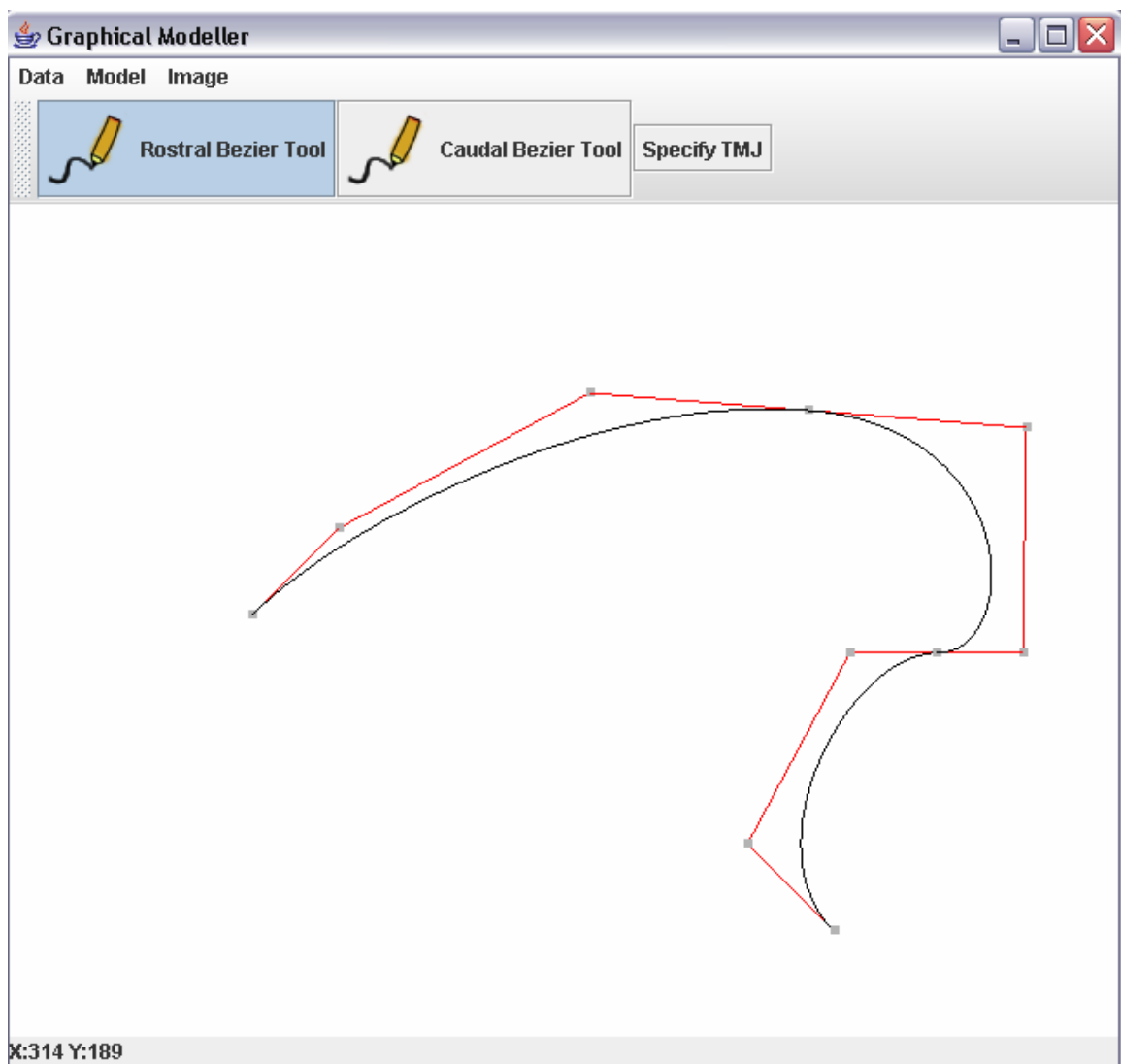


Figure 29 Screenshot of the User Interface

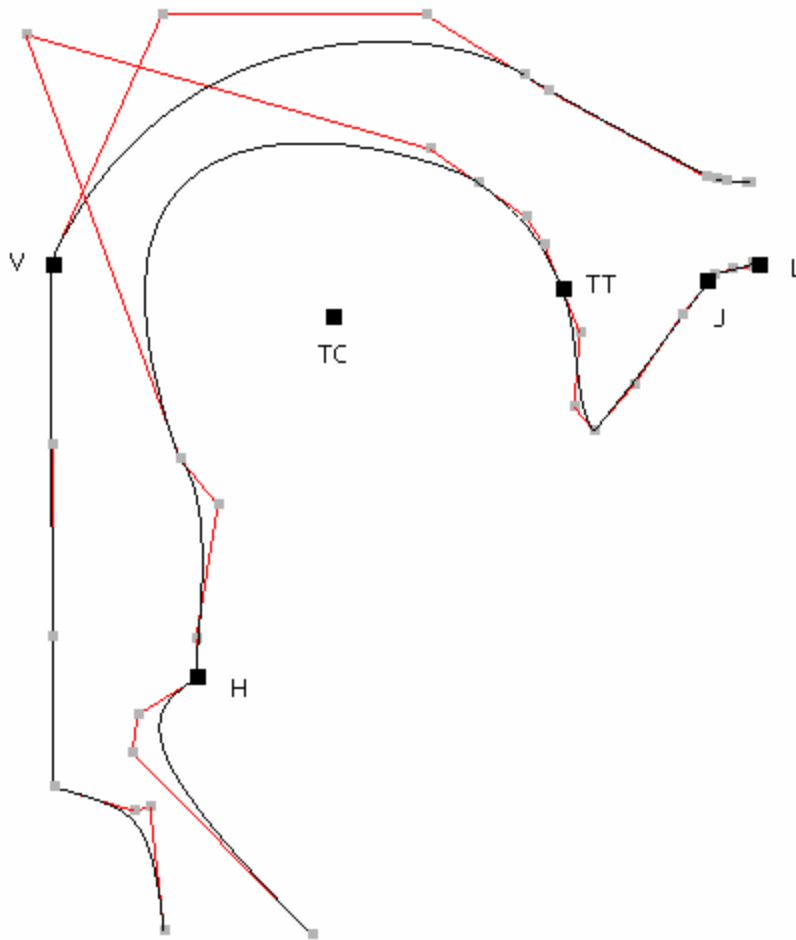
Figure 29 shows a screenshot of the user interface for drawing vocal tract models. The “Rostral Bezier Tool” allows the user to construct the upper wall of the vocal tract, while the “Caudal Bezier Tool” allows the user to build the lower wall of the vocal tract. The “Data” and “Model” menu allows users to save and load vocal tract models from a file. Finally, the “Image” menu permits the user to load an image, such as an MRI image of the vocal tract, to facilitate in the construction and animation of the model.

#### **iv. Vocal Tract Animation Tool**

This section examines the designs used to build the animation tool. It discusses how the model was parameterized, and provides examples of vocal tract animation.

##### ***1. Parameterization of the Vocal Tract Model***

This model’s parameterization is based on Mermelstein’s work in vocal tract modeling [1]. There are six key parameters for this model (Figure 30). ‘V’ stands for velum, which is a muscular flap that controls the opening to the nasopharynx. ‘H’ stands for hyoid, which is also known as the Adam’s apple. ‘TC’ stands for tongue center, and ‘TT’ stands for tongue tip. Finally, ‘J’ stands for jaw, and ‘L’ stands for lips. Animation of the vocal tract model is based on the movements of these six key parameters.



**Figure 30 Parameterization of the Vocal Tract Model**

## ***2. Degrees of Freedom***

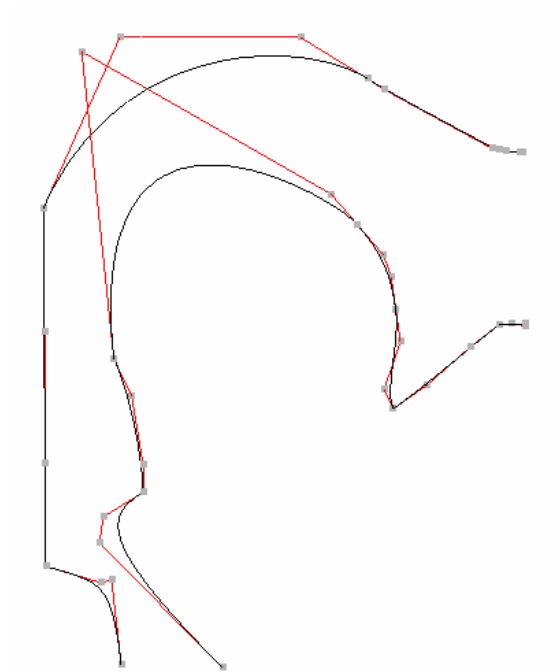
This model has 14 degrees of freedom (DOF) available for animating the model. Of the 14 DOF available, 10 of them were based on Mermelstein's parameterization of the vocal tract [1]. The velum has one DOF, and moves along a straight line. The hyoid has two DOF, which permits it to be translated vertically and horizontally. The tongue center has four degrees of freedom. Two degrees of freedom for the tongue center allows for the a rotation and radial movement from the temporomandibular joint (TMJ). The other two degrees of freedom permits the tongue center to be translated vertically and horizontally. The translational movement of the tongue center was added for convenience in animating

the model, even though two degrees of freedom is sufficient for describing all types of movement on a two dimensional plane. The tongue tip has two degrees of freedom, which allows for rotational and radial movement relative to a fixed point on the tongue body. The jaw has three degrees of freedom, which includes rotational movement around the TMJ, and horizontal/vertical translational movements. Finally, the lips have two degrees of freedom, which are horizontal and vertical translational movement. Additionally, the movement of the upper lip mirrors the movement of the lower lip.

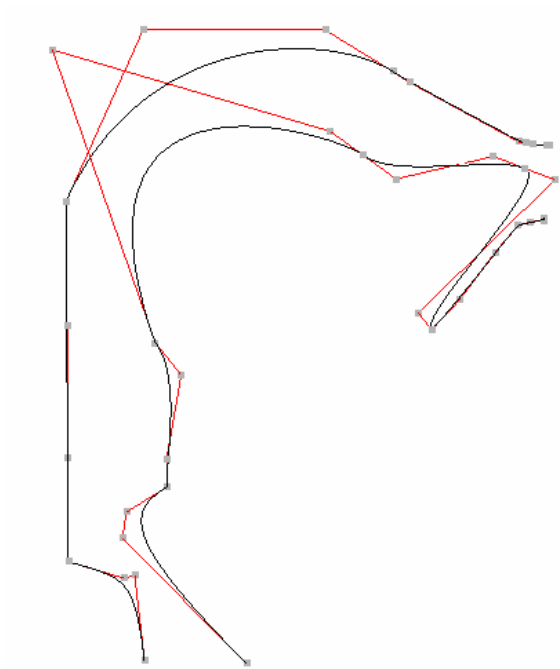
The following figures illustrate the movement of the vocal tract model.



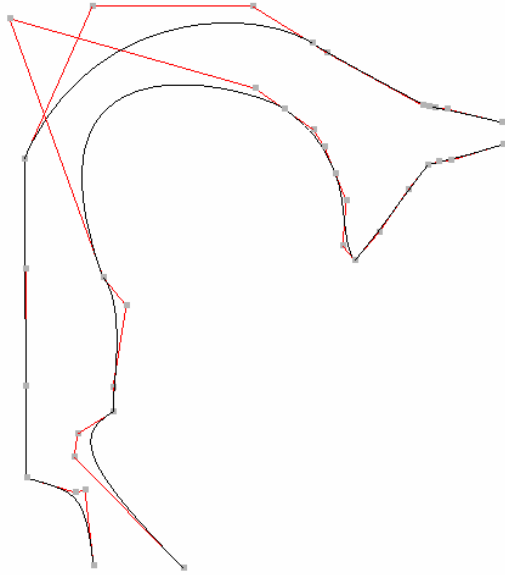
**Figure 31 Opening of the Jaw**



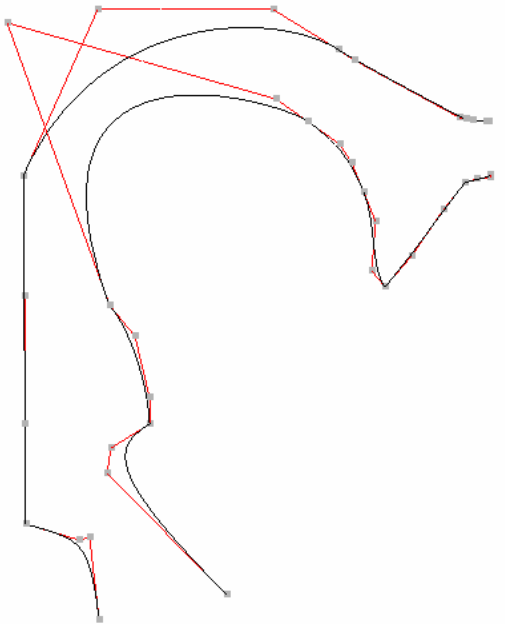
**Figure 32 Opening of the Jaw with Lower Jaw Protruded Forward**



**Figure 33 Movement of the Tongue Tip**



**Figure 34 Movement of the Lips**



**Figure 35 Movement of the Hyoid**

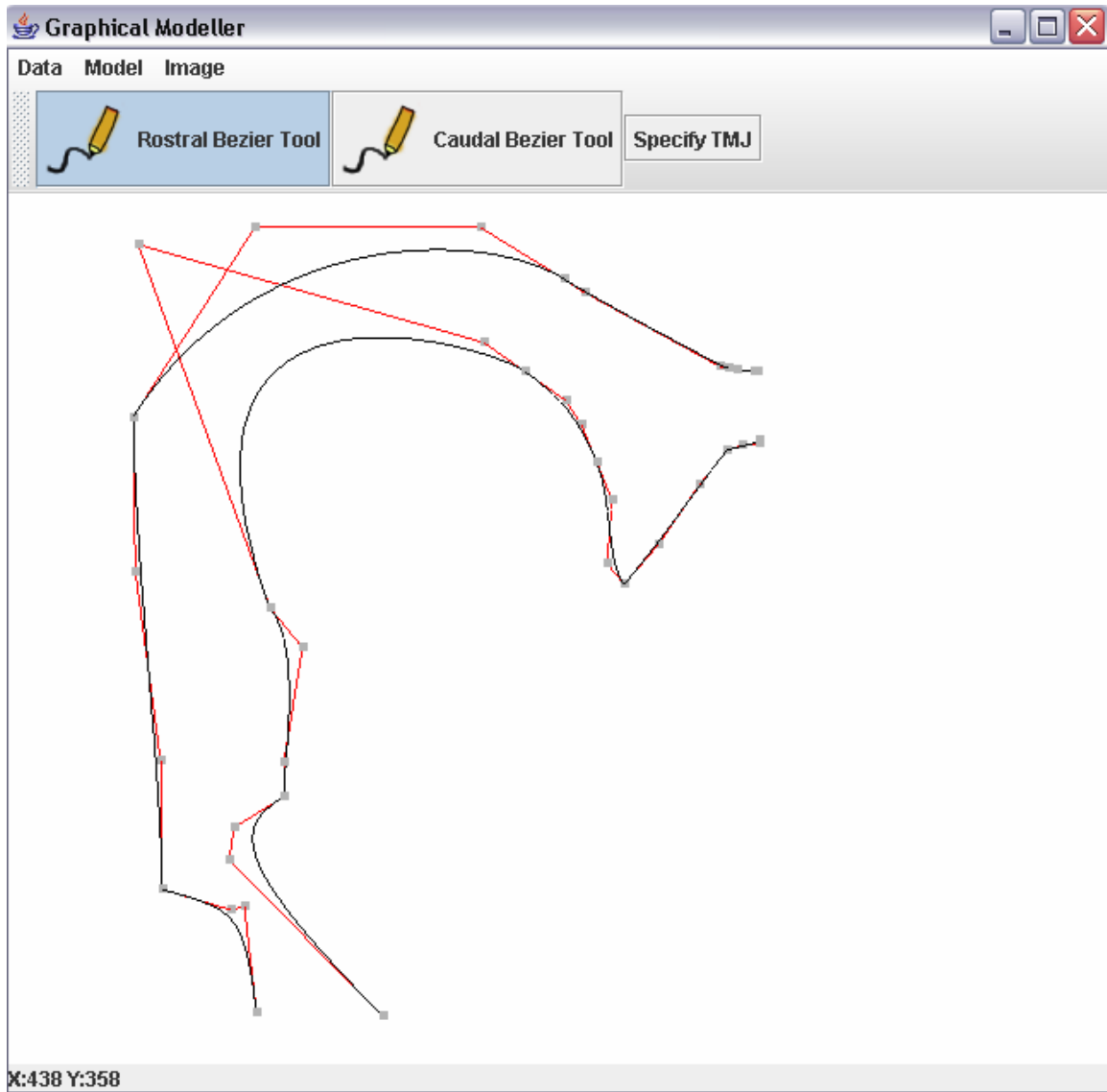


**Figure 36 Movement of the Tongue Center**



**Figure 37 Movement of the Velum**

### 3. Interface Screenshots



**Figure 38 Screenshot of Vocal Model Loaded onto the Graphical Modeler**

The animation tool uses the graphical modeler to display the vocal tract model (Figure 38). The darker curves represent the actual surfaces of the model. Whereas the lighter grey lines is the control polygon allows the user to customize the shape of the model. The slider controls allow the user to animate the model by moving a set of sliders (Figure 39). Changes in the values of the sliders causes movements of the associated parameters of the vocal tract rendered in the graphical modeler.

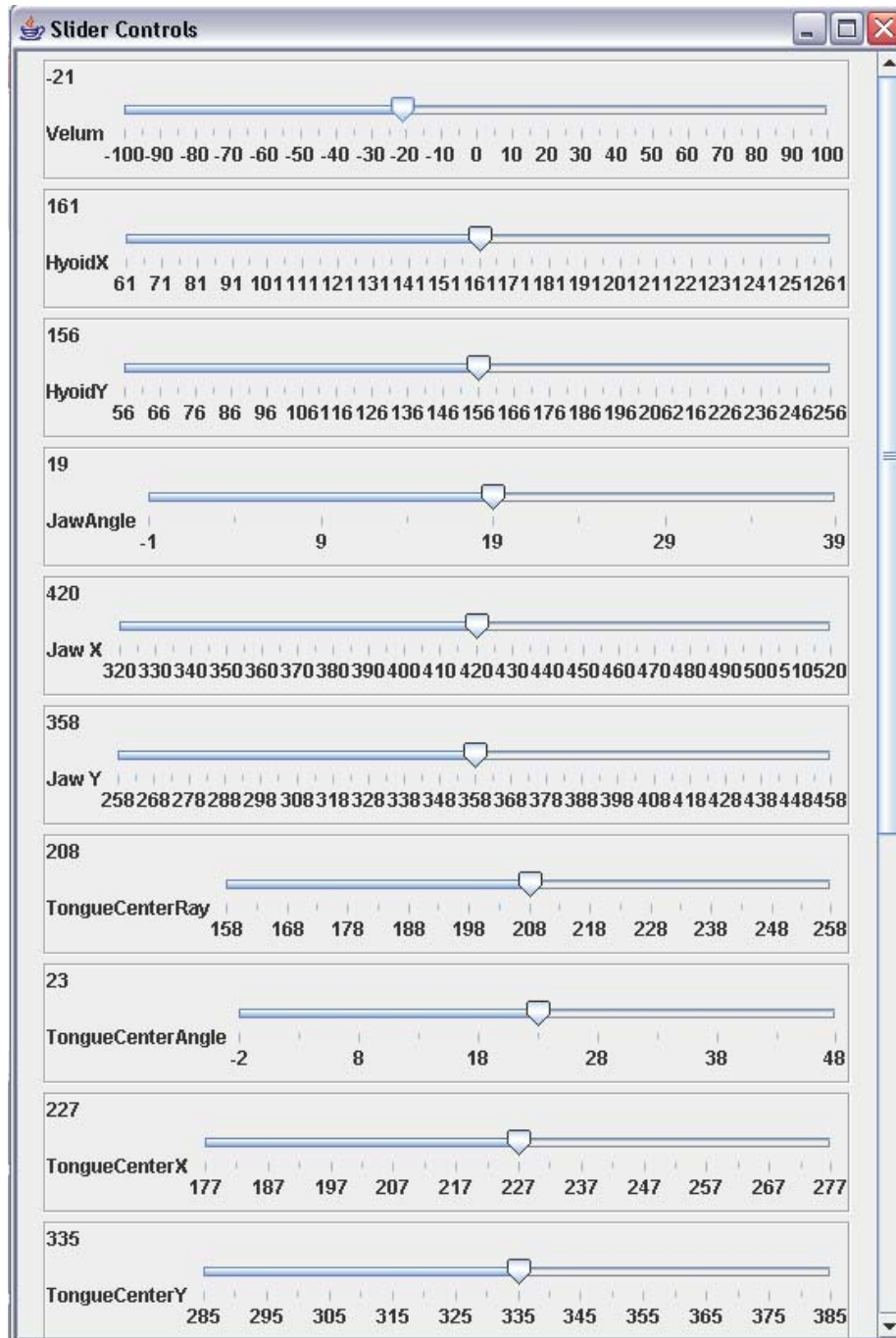


Figure 39 Screenshot of the Slider Controls used to Animate the Model

## 4. DISCUSSION

The following sections discuss the objectives that were achieved, the objectives that were not met, problems that occurred during development, and suggestions for future improvement.

### i. Vocal Tract Modeler Tool

The major goal of the project was to create a two-dimensional vocal tract model for Artisynt. We found that it would be easier to create a vocal tract model if we had a modeling tool. With the tool in place, creating the model would be quick and easy. However, developing the tool would be very time consuming. The amount of time expended to construct the tool, was balanced by the time saved by using the tool.

We met all the objectives for the modeling tool, which had a minimal set of functions to facilitate the drawing of a vocal tract model. It can perform all the basic functions that you would find in a vector graphics modeling tool, such as creating curves, reshaping curves, joining curves, deleting curves, saving models, and deleting models. However, constructing the tool was a large and complex task. We ran into problems with mixing lightweight and heavyweight components in the GUI, and we had problems with file saving using Serialization.

The panel on which OpenGL renders its view is a heavyweight component. However, the GUI elements, such as pop-up menus, and menu bars are lightweight components. Unfortunately, mixing heavyweight and lightweight components is tricky, because lightweight components are not visible if they overlap each other. Therefore, if we wanted a pop-up menu in the drawing area, which is made of a heavyweight component, that pop-up menu would not be visible because it was a lightweight component. We discovered that we could force lightweight components to become heavyweight components. This was easily accomplished with the pop-up menu component, because

there was a built in function to set it to be heavyweight. However, for menus on menu bars, it was a bit more tedious. There was no explicit function to convert a lightweight menu into a heavyweight one, but we discovered that we could cast the menu into a pop-up menu, and then set the component as a heavyweight object. This solved our problem with mixing heavyweight and lightweight components.

We initially used Java Serialization to save the information of the model created in the drawing area, so that the user could load the model from file if needed. However, because we were actively developing the system, Serialization was a bad idea since different code versions of an object will not allow for correct Serialization. We created a text based method for saving model data and constructed a parser. The parser itself is quite simple, and can read a set of data that are delineated by a set of XML-like tags. The parser uses Java's library for regular expressions to aid in parsing. In the future, it would be better to save the model in a valid XML format, and use an XML parser to grab the data from the XML file. The advantages of this is that XML is a common format for exchanging data, and data stored in XML can be used by programs other than the modeling tool.

## **ii. Vocal Tract Animator Tool**

The goal of the animator tool was to create a system that would allow the user to control the movements of any model constructed using the modeling tool. The animator tool would support functions, such as the movement of a set of control parameters using rotation about a point, and horizontal/vertical translation. However, we were not able to meet the objective of having the animator tool support any arbitrary model.

The tool met the objectives of being able to actuate a model interactively. However, the tool currently supports the actuation of only one model that we created. This model's control points can be rotated and translated with the affine transformation package that we created. However, we found that moving a set of Bezier curves is complicated.

Rotations and translations are accomplished by applying an affine transform to the control points of a Bezier curve. We discovered that the affine transforms worked correctly with a single curve. However, transformation of a set of connected Bezier curves did not work correctly. We discovered that some control points on the Bezier curve would end up having the affine transformation performed twice, instead of once as intended. Upon re-examination of how connected Bezier curves were created, we realized that the problem occurred at the joining points of the curves. When a transformation is applied to a Bezier curve, all the control points of the curve are transformed. But if we had two curves, the joining point of the two curves would end up being transformed twice, because that point belonged to both curves. Therefore, we corrected our transformation algorithm by permitting the joining points of Bezier curves to be transformed only once. With the solution implemented, the transformation of multiple Bezier curves performed correctly.

We were unable to meet the objective of allowing the animator tool to work on any vocal tract model. The reasons were that this objective was not necessary for reaching the goal of creating a two dimensional vocal tract model and that implementing the GUI interface to allow the animator tool to work on any vocal tract model would be very time consuming. Nevertheless, it is not a good design to allow a user to draw any vocal tract model they wished, but have the animator tool not working for their model. Future developments of the animator tool can be focused on allowing the tool to work on any vocal tract model. This will create a nice system, where the user can create any vocal tract outline he or she wishes, and be able to design a set of actuators to move the model.

### **iii. Two Dimensional Vocal Tract Model**

The model we constructed, using the modeling tools and animator tools, met the objectives of creating a computational model of the vocal tract (Figure 30). With the tools, the actual time used in creating the model was less than 10 minutes for the developer of this system.

The original model was based on the parameterizations of the vocal tract done by Mermelstein [1]. However, during a presentation of this system to the Artisynth group members, it was highlighted that this model was not optimal. Adding four more degrees of freedom would allow for more possibilities in the actuation of the vocal tract. With the animator tool developed, adding four extra degrees of freedom to the model was a quick and easy task. The extra degree of freedom allows for movements such as the protrusion of the lower jaw. In the end, we have created a vocal tract model that improves on the one parameterized by Mermelstein [1].

#### **iv. Coupling with the Aero-Acoustic Renderer**

One of the objectives of this project was to couple the computation model of the vocal tract with the aero-acoustic renderer in Artisynth. The coupling would allow us to hear sounds created by changing the shape of the vocal tract model. This would have been the most exciting part of the project, because we would be able to hear the system. The key to coupling the two systems is to calculate the cross-sectional dimensions of the vocal tract model, which is used by the aero-acoustic renderer to create sounds. However, calculating cross-sectional dimensions was not a trivial task.

We were unable to meet the objective of calculating the cross-sectional dimensions for our vocal tract model, because of the complexity of the problem, and time restraints. The vocal tract can be viewed as a non-uniform tube. There is no known mathematical solution for finding the cross-sectional dimensions of such a tube. Therefore, if an elegant solution to finding the cross-sectional dimensions of an arbitrary tube is discovered, it would be a breakthrough. We ran out of time in our project to discover an algorithm to calculate the cross-sectional dimensions. However, future development should be focused on this task. An algorithm should first be created to work for the current vocal tract model, so that we can demonstrate that the coupling of the computational model and the aero-acoustic renderer works. Afterwards, development can be focused on creating an

algorithm for finding the cross-sectional area of an arbitrary tube. This algorithm should work for two-dimensional and three-dimensional tubes, so that the algorithm can be used for the three-dimensional vocal tract model as well.

## **v. Integration of the Model into Artisynt**

One of the objectives for this project was to integrate the computational model of the vocal tract into the Artisynt framework. This objective was not met, because of time constraints and because the Artisynt framework was being reworked.

The completion of the animator tool occurred near the end of the time allotted for the development of this project. Therefore, there was insufficient time to integrate the system into the Artisynt framework. In addition, the some key elements of the Artisynt framework was being reworked, therefore it would not have been reasonable to integrate the system at that time.

Integration of the system into Artisynt should be straightforward. Both systems are written in Java, so there is no need to rewrite the system to fit programming language constraints. Additionally, because the system was built using the MVC pattern, we can import the Model and the Controllers from our project into Artisynt, and use Artisynt as the Viewer. Future development should be focused on integrating the system into Artisynt once other objectives, such as the coupling with the aero-acoustics renderer is complete.

## 5. CONCLUSIONS

This project created a computational model of the vocal tract. In addition, tools were created such that users can quickly construct a new model and animate it. These tools will be useful for users, such as linguistics experts, in studying how the shape of the vocal tract can affect speech output. It can also be used to produce articulatory based speech synthesis.

Through the process of building this system, we created the OpenGL monitor object, which allowed safe multi-thread access to the OpenGL state-machine. The OpenGL state sniffer was another innovation that helped us to debug run-time errors in the OpenGL state-machine. Object picking and selection, which are key requirements for any vector graphics drawing program was accomplished using OpenGL's selection mode rendering. Bezier curves, which are the graphical primitives from which our vocal model was constructed, was implemented using a subdivision algorithm. The vocal tract model was animated with 14 degrees of freedom, and parameterized following the work of Mermelstein [1].

Work is still required in coupling the computational model of the vocal tract with the aero-acoustics renderer, to allow the system to produce sounds based on the shape of the vocal tract. In addition, the system must be integrated into the Artisynth framework, which is a tool for speech synthesis and modeling the face and vocal tract.

This project accomplished three large tasks of producing a modeling tool, an animator tool, and a computational model of the vocal tract. We hope that this tool will benefit others in the research of linguistics, and other areas related to vocal tract modeling.

## 6. References

- [1] Haskins Laboratory, "Introduction to Articulatory Phonology and the Gestural Computational Model," [Online Document], [Last Accessed 2005 April 10], Available HTTP:  
<http://www.haskins.yale.edu/haskins/HEADS/ASY/GESTURAL/gestural.html>
- [2] J. Lloyd (private communication), 2005