Learning the Alphabet of Rhythmic Motion

By

Jonathan H. Chu

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
August 20, 2003

Author_____
Department of Electrical Engineering and Computer Science
August 20, 2003

Certified by_____
Jovan Popović
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Learning the Alphabet
of Rhythmic Motion

by
Jonathan H. Chu

Submitted to the
Department of Electrical Engineering and Computer Science

August 20, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

Abstract

Movement recorded from motion capture can be decomposed along two separate dimensions. It can be divided into shorter units of motion with respect to time. Secondly, a movement can also be decomposed into a base motion and a motion texture. We define the term, *basic movement,* to reflect properties from both of these decompositions of movement; it adheres to the atomic length property in time and also describes the baseline graph of a motion. We additionally define the terms, *alphabet,* as the complete set of all distinct basic movements in a motion database, and *grammar rules* as the set of all frequently occurring sequences of the basic movements.

Determining the alphabet and the grammar of a motion capture database assists in addressing the organizational issues plaguing the process of motion capture, as well as provides a notational representation of motion. In this thesis, we introduce a method for determining the alphabet and grammar of rhythmic motion.

Thesis Supervisor: Jovan Popović
Title: Assistant Professor, MIT Laboratory for Computer Science

Acknowledgements

First and foremost, I am grateful to Professor Jovan Popović in believing in my abilities to succeed on this project, despite knowing that I had had no previous experience in the field of computer graphics. He went well beyond his call of duty to bring me up to speed on current issues and projects in the subject and was extremely supportive, suggesting particularly creative ideas and directions whenever I was stuck. Sommer Gentry was indispensable in convincing Lindy Hop dancers to volunteer for the motion capture recording sessions and always maintained a light and fun atmosphere at the often lengthy sessions. I would also like to thank Eugene Hsu who developed much of the framework code, which I built my system on top of and who was always willing to spend time to explain algorithms and implementation details and also assist in the processing of the data files. Furthermore, I am appreciative of the rest of the computer graphics staff and students, who were happy to share much of their knowledge and wisdom with me.

I would also like to thank my friends, who were instrumental in my maintaining my sanity and sense of humor throughout the length of the project, in particular: Marco Hernandez, Judy Liao, Emery Lin, Annie-Jo Cain, Manu Seth, Buddhika Kottahachchi, David Ngo, Audrey Lee, Gary Lee, Debora Lui, Jumaane Jeffries, Carolyn Chen, and Jason Kahn. Finally, I am especially grateful for my immediate family: Mom, Dad and brother Jeff, who encouraged me from start to end of this project and were always there for me when I needed it.

Contents

List of Figures

List of Tables

1 Introduction

1.1. Project Context

Motion capture is rapidly increasing in use and importance, not only in the media entertainment field, such as the film and video games industries, where it is used to create realistic motion, but also in athletics to help professional athletes perfect their form and even in the medical/rehabilitation area in prevention of injury from harmful habits in patients' movements. A few problems, however, hinder the motion capture process. The first of which is caused by the property that, like video data, motion capture takes must be viewed linearly in time. As a result, to find a particular movement, in the worst case scenario, it may be necessary to search through the entire timeline of each take in the database. This problem is further compounded by the fact that at this time, motion capture equipment is expensive to purchase and labs costly to rent. In response, users react by squeezing the most out of their sessions in the motion capture lab, recording duplicate takes and slight variations of the takes to ensure that any and all requirements and possible needs, however remote, are met. Consequently, the resultant motion capture databases that users create tend to be large and dense, further exacerbating this problem of search and organization of the data.

A second issue affecting this area is the need for the capability to reuse and edit motion capture data. In particular, animators in the film and video game industries strongly desire this feature, since in these fields it is often necessary to obtain exact movements. For example, in a film, a particular physical movement may be critical to creating the atmosphere of a scene. In a video game, a character's motion may have to be synchronized to an external event, such as an explosion. As it is often difficult to record the perfect take in one motion capture session and often implausible to repeat a second session because of the cost issue, the need for the ability to edit or reuse existing motion capture data becomes crucial.

We introduce the notion of alphabet and grammar structures of a motion capture database to assist in addressing these issues of motion capture. The alphabet and grammar act as a representation of movement, while inherently providing an organizational structure for motion, which allows users to more easily search for and manipulate motion capture data. To further describe the benefits of the alphabet and grammar, we will first define what we imply by the terms and then describe the valuable properties that result from determining the structures from a motion capture database.

1.2. Alphabet and Grammar

Movement can be decomposed along two separate dimensions. It can be divided into shorter units of motion with respect to time. For example, a walking movement might comprise of a motion unit describing the lifting and placement of the right foot, followed by another describing the lifting and placement of the left foot, and then followed by yet another for the next right foot motion and so on. Secondly, as noted by Li et al. [2002] and Pullen and Bregler [2002], a movement can also be decomposed into a baseline motion and a motion texture. In other words, one can separate out the variations in motion that make one's stride unique from the similarities in motion that make one's stride resemble that of another. We define the term, *basic movement*[1], to reflect properties from both of these decompositions of movement; it adheres to the atomic length property in regards to time and also describes the baseline graph of a motion. For instance, extending the previous example, a walking movement might consist of two basic movements, one describing the units with forward motion of the right foot, and the other describing the units with forward motion of the left foot. We build on top of the notion of a basic movement, by defining the terms, *alphabet,* as the complete set of all distinct basic movements in a motion database, and *grammar rules* as the set of a frequently occurring sequences of the basic movements.

Determining the alphabet and the grammar rules of a motion capture database assists in addressing the issues of organization and reuse of motion capture data through

[1] Note that our definition of basic movement is different than how [Kim et al. 2003] defined the term. Our definition is more akin to their term, prototype movement.

three properties: the notational property, which states that the alphabet can provide a textual representation of movement, the range property, which asserts that the alphabet represents all movements in the database and the hierarchical property, which describes how an organizational structure is implicit in the alphabet and grammar structures. Further explanation of these properties and their benefits follows.

1.2.1. Notational Property

By associating symbolic labels with the basic movements of a database, a user now has access to a notational language for that particular collection of motions and two main beneficial features are produced as a result. First, a motion can now be symbolically recorded. We mentioned earlier the example of the walking motion with the two basic movements. If one labels the first basic movement, "A," and the other, "B," a walk can now be recording as the string sequence "ABAB." This property aids in communication of motion sequences. For instance, suppose that two choreographers, across the country, have access to the same database and that one of the choreographers wants to send a dance routine to the other. Instead of sending the entire dance data structure, which could potentially be very large, he is able to simply send a string sequence through email. Secondly, from a technical perspective, a bridge between motion and textual language has been created. As we will discuss later, we have actually applied well-known text algorithms to the field of motion capture using this property.

The second feature of having a motion notational language is that new motion can now be designed with labels that users are already comfortable with. In the walking example, a user may now define a new circular pivot motion, by specifying the string "AAAA," where the left foot remains planted on the floor and the right foot moves forward. Arikan et al. [2003] have recognized this property and have created an intuitive system for animators to synthesize new motions from their motion capture database using motion annotations.

It should be noted that a language for annotating motion by hand already exists. Laban notation is a system that allows motion to be analyzed and recorded using a number of shapes and shadings. An advantage, however, of determining the alphabet of

a database and utilizing the notational property of the alphabet, over translating motion capture data into Laban notation, is that a user is not subjected to learning a new complex language. Laban notation was designed such that any motion and its style of performance could be described as precisely as possible. As a result, Laban notation may be unnecessarily complicated for users of motion capture to learn and work with. Secondly, the process of matching motions to existing Laban notation patterns provides a much greater technical challenge than determining the alphabet from the data itself. A simple analogy can be found in speech analysis and recognition. There is a great gap in difficulty between trying to uncover common speech patterns through simple signal processing versus attempting to determine what English letter is being pronounced. As a result, the notational property of the alphabet provides a powerful, yet simple method for representing motion in a textual manner.

1.2.2. Range Property

The alphabet is defined as containing all of the distinct basic movements of a motion capture database. Since a basic movement represents a class of movements, the alphabet encapsulates the complete range of motions that occur in the database. This idea, we refer to as the range property of the alphabet. The range property provides several benefits to motion capture users. For animators and motion capture editors, having access to the complete range of motions in a database is analogous to a painter being able to see the full palette of colors that he can use. An animator or editor can select a motion from the "palette" of motions and therefore has an easier interface for the process of editing/reusing motion capture data.

Operations involving entire motion databases gain from the range property in two ways. First, since the alphabet of a motion set contains all the distinct movements in the set, it can act as an icon representation of the actual set of motions. This may be useful in applications comparing two sets of motions. Instead of having to compare all of the takes in the two collections of movements, one can more quickly compare the two alphabets to see where major differences lie. Secondly, when a new motion is added to the database,

it is quicker and more convenient to test whether a similar motion already exists in the database by checking the alphabet instead of all of the motions in the entire database.

1.2.3. Hierarchical Property

The last property is the hierarchical property. Each basic movement can be represented as a two-level hierarchy, where its children are the specific instances of the basic movement. For example, the length of each stride in a subject's walking motion may slightly differ in regards to distance and/or time. We can denote the individuality of each stride by assigning every stride its own number 1 through n, where the odd numbered strides denote the forward left foot motions and the even numbered strides describe the forward right foot motions. As mentioned earlier, there exists some baseline motion that is common to all of the odd numbered strides, such that a basic movement, "A", can be created to describe them. In other words, similar to how the term "fruit" describes specific instances of the fruit, i.e., apple, orange, strawberry, etc., the basic movement "A" describes the specific instances of the forward left foot strides. In this manner, the basic movement, "A" can be represented as a parent of its specific instances, as shown in Figure 1 .

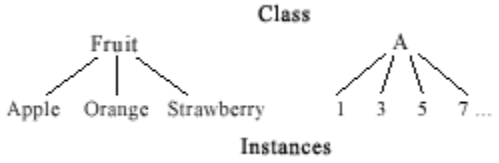

Figure 1 A basic movement as a 2-level hierarchy

By representing the basic movements as 2-level hierarchies, the alphabet becomes a set of hierarchies, which can be leveraged to facilitate easier searching for specific motions.

Furthermore, the addition of a grammar adds more levels to the hierarchies, in that grammar rules contain basic movements and can also contain other grammar rules. As a result, since hierarchies are well known for their organizational capabilities, the fact that the alphabet and grammar of a motion capture database can be seen as a hierarchy helps users to organize and find specific pieces of motion more easily.

1.3 Thesis Outline

In this paper, we present a system for determining the alphabet and grammar of rhythmic motion. We segment motion takes into shorter divisions of motion according to their rhythmic pattern and develop the alphabet of the database through the Support Vector Machine learning algorithm. Lastly, the grammar rules are found by associating symbolic labels with the basic movements and applying the text compression algorithm, hierarchical grammars, upon the sequence of labels.

We organize our findings as follows. In chapter 2, we review related work. In chapter 3, we discuss how motions are segmented into shorter units and other issues that must be addressed before the alphabet can be created. Chapter 4 explores several algorithms that we studied to create the alphabet and chapter 5 details methods for determining the grammar rules. In chapter 6, we provide experimental decisions and results. Finally, in chapter 7, we conclude with an overview of our approach to building the alphabet and grammar and suggest future improvements and follow-up projects.

2 Related Work

Numerous projects in motion capture have focused on the specific application of the reuse and editing of motion. Specifically, much research has been directed towards the development of realistic synthesis of new motions and the creation of smooth transitions between existing motions. Kovar et al. [2002] select transition points in motion and create motion graphs to smoothly connect existing movements. Arikan et al. [2002] cut and paste example motions to create realistic movement that follows new constraints specified by the user.

There also has been research directed towards easier organization and manipulation of motion capture data. Li et al [2002] and Pullen and Bregler [2002] both separate out high-frequency motion textures from lower-frequency baseline motion, which allows users to specify general motions first and then follow up with more detailed constraints. Lee et al. [2002] add connecting transition points into their motion sets and also create a two-level resolution view into the set by clustering motion, allowing for efficient searching into their data set.

To our knowledge, only two other projects have focused on the decomposition of movement into smaller units and the utility of motion notation. Kim et al. [2003] identify smaller segments of rhythmic motion, similarly called basic movements, through the detection of directional changes in movement. A transition graph is created and the basic movements are used for synthesizing realistic movement that satisfies new rhythmic constraints. In the system described by Arikan et al. [2003], users break down movements into shorter segments manually and annotate the motion divisions. This provides the user with the ability to choreograph new movements from their own annotations.

3 Basic movements and Motion Chunks

3.1 Partitioning Takes into Motion Chunks

We begin the process of finding the alphabet of a motion capture database by partitioning each take into shorter units of motion, which we refer to as motion chunks. Because we are working with rhythmic motion, the divisions defining the motion chunks can be placed regularly along the timeline. In particular, as a result of our choice to use Lindy Hop dance motion, determining the exact specifications of motion chunks, such as the length and the starting location of segmentation, becomes a relatively simple task. Each Lindy Hop dance movement can be subdivided into 1-beat segments. We use these inherent divisions in Lindy Hop dance movements to specify our motion chunks.

Establishing motion chunks beforehand is invaluable to the process of determining basic movements. Our approach for finding a basic movement is to build up the hierarchy of the alphabet in a bottom-up manner. In particular, all similar motions are classified into the same set. A basic movement can then be determined through a simple average of all of the members of the set. For example, in Figure 2, the two selected motions are recognized to be similar and are categorized into the same set[2]. A basic movement 'A' is then created by averaging the two movements.

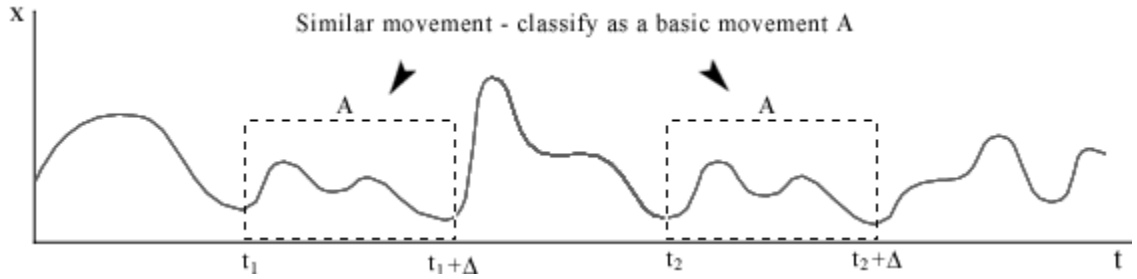

Figure 2 Example of a basic movement

If the take has not been divided into motion chunks, the process of finding these two matching movements becomes a dramatically more difficult task, involving computationally costly searches. (See Appendix D for more details) With the take partitioned into motion chunks, however, the general concept of determining basic

[2] For simplicity sake, we will represent motions as signal graphs of the x-coordinate of a single marker.

movements is simple. It entails comparing all of the motion chunks in the database and grouping the similar ones. In Figure 3, we would compare chunks 1 through 5 and find that chunks 2 and 4 are similar, causing them to be classified as the same basic movement.

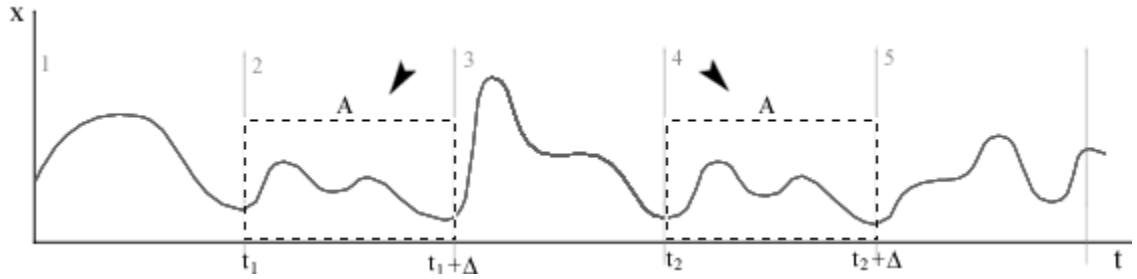

Figure 3 Finding a basic movement using motion chunks.

3.2 Motion Chunk Comparisons

Before basic movements and the alphabet can be found, a few issues arise with motion chunk comparisons. One of the problems is that comparisons may be adversely affected by the orientation of movements. In other words, it is important to ensure that a movement facing the northwest direction should be deemed equivalent to the identical movement facing the southeast direction. There are two methods for solving this problem. One possibility is to represent motion capture data using the subject's joint angles. In this representation, the location and orientation of the subject are relative and therefore, comparisons between motion chunks would not be affected by different orientations. However, as Kovar and Gleicher [2002] found, objective comparisons of motions using joint angles are difficult, since certain joint rotations have greater effect on the overall motion than others (i.e., hip rotation versus ankle rotation). As a result, two motions, seemingly similar to the human eye, might be deemed as nonequivalent because of significant differences in some minor joint angle. Furthermore, weighting each joint to compensate for this joint angle peculiarity is not an effective solution, as the same weighting system may not apply for all motion comparisons.

The other solution to the problem of orientation affecting motion chunk comparison is to continue using the standard representation of motion capture data as 3-dimensional marker locations, and to apply transformations to convert the orientation of one motion chunk to emulate that of the other. This is the method we chose in this

system and used the transformation that Kovar and Gleicher [2002] designed in their project.

Another issue in comparing motion chunks is that it may be necessary to compare two motion chunks of differing lengths. For example, looking in Figure 4, we see two motion capture takes, where the motion chunks of the second take are longer than those of the first. However, there exist motion chunks in both takes that should be classified as the same basic movement.

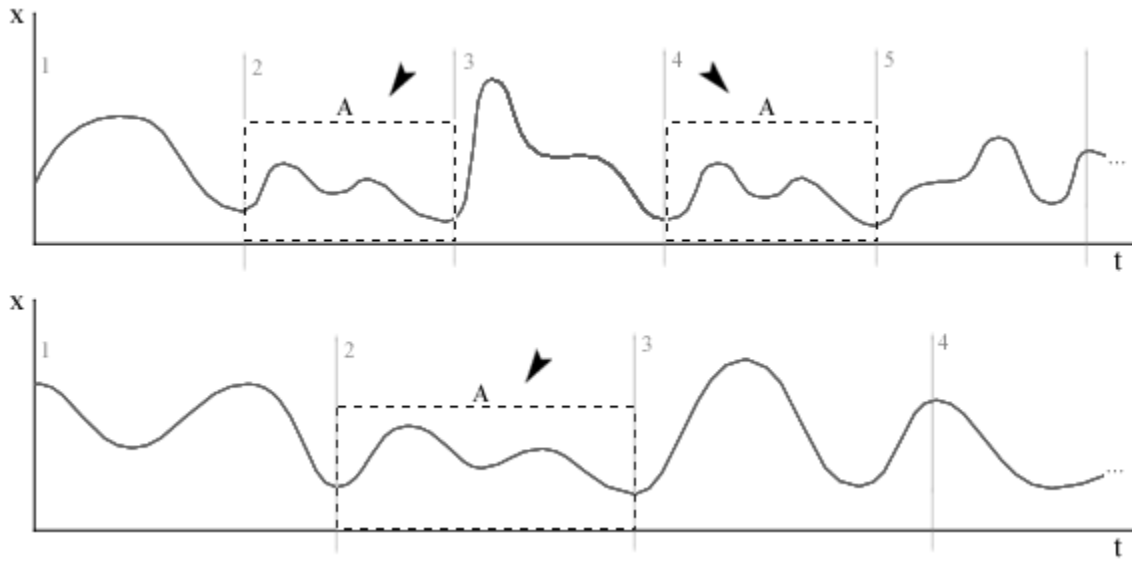

Figure 4 Comparing motion chunks of differing lengths.

To resolve this issue, we determine the maximum length of all of the motion chunks and expand all the other motion chunks out to this maximum length, interpolating any missing data. Since the lengths of motion chunks are small to begin with and do not vary widely in length, this method of normalizing motion chunks of differing sizes is acceptable, and comparisons are not affected by the interpolated data.

4 Alphabet Algorithms and Methods

To determine the alphabet of a motion capture database, a method or algorithm is required to fulfill three requirements. First, it accurately compares motion chunks and performs a comprehensive categorization of all similar motion chunks as the same basic movement. This requirement guarantees that the quality of each basic movement is high and that it describes a specific baseline motion. It also ensures that all motion chunks, which should be classified as a basic movement, are labeled as such. Secondly, the alphabet-building method must produce the complete set of all distinct basic movements. This implies that a method can neither return a few highly accurate basic movements nor generate a set with duplicate basic movements. Lastly, there should be a period of time during or after the method has run, which permits for user review of the alphabet and correction of the alphabet or basic movements if necessary.

4.1 Unsupervised Learning

4.1.1 Hierarchical Clustering

Our initial approach for building the alphabet of a motion capture database was to consider clustering algorithms to group similar motion chunks together. Specifically, we investigated hierarchical clustering, since it is conceptually simple to implement and could produce a hierarchical representation of the process, which would be a useful visualization tool for the user in reviewing clustering results. Hierarchical clustering operates in the following manner. Consider the example in which we desire to cluster n data points into c clusters. The algorithm starts with n clusters, with each data point in its own cluster. The algorithm merges the two most similar clusters together, resulting in one less cluster. It continues to merge clusters, until only c clusters remain. A valuable representation of how hierarchical clustering proceeds can be seen in a tree structure called the dendrogram. It indicates the order in which the clusters are merged, as well as the similarity level at which the merging is done.

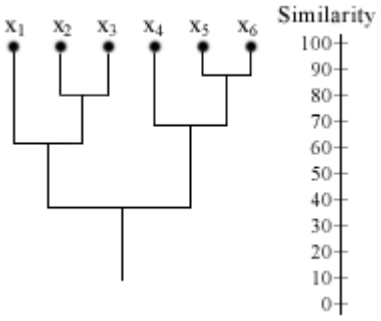

Figure 5 An example dendrogram.

For example, in Figure 5, clustering starts at the top of the dendrogram and proceeds downward. The data points, x_5 and x_6, are first merged at a high similarity level of 87. They are then followed by the joining of x_2 and x_3 at the similarity level of 82. The algorithm continues to merge clusters until the desired number of clusters has been achieved.

One may notice that besides data point comparison, hierarchical clustering also performs cluster comparison to merge clusters. There exist different metrics for measuring cluster distance, the choice of which greatly affects how the cluster merging procedure progresses. Two well-known distance metrics are the single linkage and the complete linkage distance measures. In the single linkage metric, the distance between the two clusters being compared is equal to the distance of the nearest neighbor points. In Figure 6.i, the circled points are the nearest neighbors and the line drawn between them is the distance of the clusters A and B. In the complete linkage metric, the distance between the clusters is equal to the distance between the furthest-neighbor points. In Figure 6.ii, the circled points are the points furthest apart from each and the line drawn between them is the distance between the two clusters.

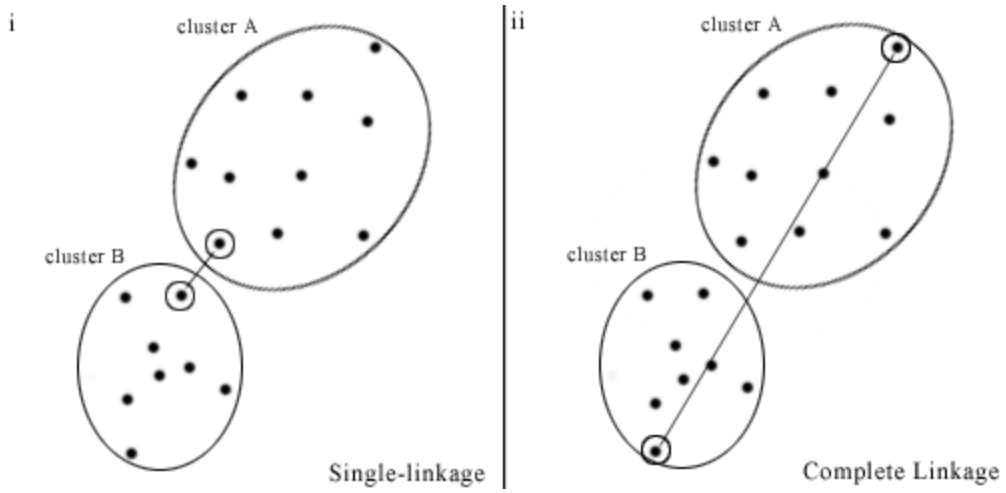

Figure 6 Comparison of the single linkage and complete linkage distance metrics. The distance between clusters A and B are represented by the lines drawn.

Single linkage clustering is sometimes criticized for its high sensitivity to noise in the input and in the worst case scenario, can create single elongated clusters. Complete linkage clustering, on the other hand, while less sensitive to noise, can, in the worst case, impose clustering structure onto the points instead of learning it from the input.

4.1.2 Clustering Difficulties

Conceptually, clustering appears to be a tight match for the requirements of determining the alphabet. It provides a manner for which similar motion chunks can be clustered together and allows for the specification of how many basic movements should be created. However, realistically, clustering fails on two levels. First, user review of the performance can only be performed after clustering has finished. As a result, determining optimal clustering settings is awkward, since users must wait for the completion of clustering before he/she can view the results of any changes in settings that were made. More importantly, even at optimal settings, clustering does not succeed in accurately developing tight collections of similar motion chunks. We implemented and experimented with both single linkage and complete linkage clustering, allowing the user to set the number of desired clusters. For the distance metric of the motion chunks, we used the sum of the squared differences of all of the marker locations.

$$\sum_{\text{all frames}} (\text{chunkA.marker}_{x,y,z} - \text{chunkB.marker}_{x,y,z})^2.$$

In all test cases, there were many misclassifications of motion chunks, requiring the user to examine all results and in many cases, manually reclassify many motion chunks. In total, we discovered that clustering necessitates as much or even more user supervision than as in manual creation of the alphabet. As a consequence, we abandoned the clustering methods and proceeded to examine manual classification of the alphabet with assistance from supervised learning methods.

4.2 Supervised Learning Methods

Since the unsupervised method of clustering demands as much attention from the user as manual classification of the alphabet would have, we consider a new paradigm, in which the user builds the alphabet with heavy assistance from semi-automated processes. One simple method of aiding the user in creating the alphabet is to allow the user to compare motion chunks using a distance metric. We implement this feature in our system by pre-calculating a matrix of the distances between all pairings of the motion chunks, using the same distance measure we used in clustering. When the user selects a particular motion chunk for viewing, the system returns a listing of motion chunks, which meet a similarity threshold value set by the user. While this feature is quite simple and has limited usefulness, it does help the user in getting started in creating the alphabet.

4.2.1 Support Vector Machines

To achieve more accurate and automated classification of basic movements, we look towards supervised learning algorithms. The ideal situation is that learning machines would gather enough information from the initial user actions so that they could guide the user through the creation of the rest of the alphabet. In particular, we examined the Support Vector Machine (SVM) algorithm for two main reasons. First, it works well

when presented with input data of high dimensionality, which is the case here, as we will discuss later on. Secondly, Support Vector Machines are able to classify accurately, even when there are few irrelevant features of the input vector, in other words, when all of the information in the input vectors is equally important and there are no parts, or features, of the vector that can be easily discarded for better classification.

Overview

A Support Vector Machine classifier is a learning machine for classifying input vectors into two groups. The principal of SVM classifiers is the following: input vectors are mapped to a much higher dimensional space, called the feature space, where a decision boundary surface can be easily found and classification decisions quickly performed.

The support vector machines algorithm is capable of accepting input vectors of many dimensions. For introduction's sake, however, we will begin with input vectors of 2-dimensional points. Furthermore, we will discuss how support vector machines are used for classification for a single class, then review how they are generalized for multiple classes and finally discuss how SVMs are used in this system.

Single Class Classification

In single class classification, provided with initial training vectors, a support vector machine returns whether an input vector does or does not belong to a class. Conceptually, the support vector machine classifier accomplishes this by mapping all of the training vectors into the higher dimensional feature space and determining a decision boundary surface that maximizes the distance between the closest positive and negative examples of the class, which are called support vectors. An example linear boundary surface, separating the training vectors, is shown in Figure 7. The support vectors are displayed in gray.

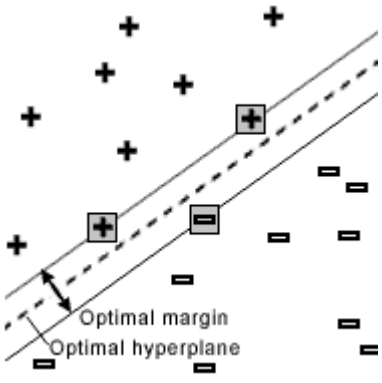

Figure 7 Support Vector Machines for a Single Class

SVM classifiers are universal learners, connoting that they can support different kernels for boundary surfaces. For example, a user can choose linear, quadratic, polynomial or radial basic functions as decision boundaries to fit their particular classification needs.

Support Vector Machine classifiers return a numeric value for each previously unclassified vector, indicating how strong a positive or negative example the vector is, according to its distance from the decision boundary. Typically, implementations of the support vector machine algorithm return positive values for vectors belonging to the class and negative values for those not belonging to the class.

Classification of multiple classes

To categorize vectors into multiple classes, an SVM classifier is created for each desired class. As mentioned in the previous section, when run, each SVM classifier returns a value for how strongly a vector belongs to that particular class. The result of running all of the classifiers on all of the input vectors is a matrix of values indicating how strongly each vector belongs to each of the classes. For instance, if we wish to classify 7 input vectors into classes A through F, the matrix, shown in Figure 8 might be a result.

		Classes					
		A	**B**	**C**	**D**	**E**	**F**
Vectors	**1**	**0.8**	0.15	-0.45	-0.5	-0.35	0.3
	2	-0.2	0.45	0.25	**0.7**	-0.6	-0.7
	3	**0.85**	0.20	-0.3	-0.65	0.05	0.15
	4	0.35	**0.95**	-0.85	-0.3	0.45	0.1
	5	-0.6	-0.2	-0.6	-0.5	**0.7**	0.3
	6	-0.4	-0.5	-0.75	-0.2	-0.3	-0.6
	7	-0.3	0.3	0.15	**0.45**	-0.2	-0.5

Figure 8 Example of a multiple Class SVM Classifier result matrix

Using this matrix, the input vectors can be classified into partitioned classes (i.e., no class can share any members with any other class). For each vector, all of its classifier values are examined and the maximum value is selected. In Figure 8, the maximum values for each vector are shaded in gray. If the maximum value is positive, the vector is labeled as the class, the classifier of which returned that particular value. For example in Figure 8,

the maximum of the classification values for vector 1 is 0.8. As a result, vector 1 would be classified as class A.

Similarly, the vectors can be classified into overlapping classes (class A may share members with class B). A vector is labeled as belonging to every class, the classifier of which returned a positive value. For example, vector 2 would belong to classes B, C, D. Lastly, vectors with no associated positive values remain unclassified. Vector 6 in Figure 8 would be an example of this case.

4.2.2 Support Vector Machines in this system

As mentioned previously, SVM classifiers are one of the methods to assist the user in the process of manually constructing the alphabet. A user initially creates a few basic movements, and categorizes some example motion chunks as these basic movements. An SVM classifier can then be created for each basic movement, trained on its existing members and run on all unclassified motion chunks. The maximum positive value returned by the SVM classifiers for each motion chunk is presented as a classification suggestion to the user.

There are a few benefits to using a supervised learning machine in this type of process. First, the user is constantly responsible for reviewing classification decisions and can therefore ensure a high degree of quality of the alphabet, yet at the same time, the user is not overwhelmed with the task of complete manual classification. Secondly, as the user classifies more and more motion chunks, there are more positive and negative examples for the SVM classifiers to use and as a result, the classifier decision boundaries and the quality of their SVM suggestions improve. This enables users to easily go through and classify all of the motion chunks.

Below we will detail specifics of how support vector machines are applied to classifying motion chunks.

Motion chunk vector representation

To use Support Vector Machines for classification of motion chunks, the information must first be represented in a vector form. Each motion chunk initially is structured as an array of frames, which in turn is an array of marker locations. See Appendix C for an example. To transform the motion chunk structure into vector form, we simply flatten the existing structure, such that only the x, y, and z marker locations are left. See Figure 9.

```
Frame1.left_hip.x
        Frame1.left_hip.y
        Frame1.left_hip.z
        Frame1.right_hip.x
        Frame1.right_hip.y
        Frame1.right_hip.z
        …
        FrameN.left_hip.x
        FrameN.left_hip.y
        FrameN.left_hip.z
        FrameN.right_hip.x
        FrameN.right_hip.y
        FrameN.right_hip.z
```

Figure 9 Vector Representation of Dance Chunk

Training of Support Vector Machines

To train a Support Vector Machine classifier, it is necessary to have positive and negative examples of the basic movement. We employ the user-added members of the basic movement as positive examples and members of other basic movements as negative examples. For instance, if the user has created basic movements A, B and C and the system is building an SVM classifier for A, the members of A are chosen as the positive examples and the members of basic movements, B and C are selected as negative examples.

A minor issue, which arises out of this approach in how classifiers are trained, occurs when a user creates two or more basic movements whose differences are subtle. In this situation, when classifiers are created for those basic movements, there is a set of negative example training vectors that are similar to the set of positive example training vectors. As a consequence, since there is an inherent limitation in how fine a decision

boundary can be, the SVM classifiers for both basic movements will become too restrictive and will not suggest motion chunks for either basic movement. One simple, but not optimal, solution to this problem is to enable the user to change the influence of negative examples have in training classifiers. Specifically, in this system, we allow the user to specify the percentage of members of each class to use as negative examples. By experimenting with this setting, it is possible to resolve this minor issue.

Learning Models

There are two main learning models for Support Vector Machines. The first is the inductive learning model, in which only positive and negative examples of a class are required to train a classifier. This is the model in which Support Vector Machines have been introduced in this paper and the model which is generally used.

There is a second type of learning model, called transductive inference, in which both training and test vectors are used in developing a decision boundary surface. In this model, there are attempts to extract patterns between training and test vectors, resulting in requirements of fewer training examples and in certain cases higher accuracy in classification. [Joachims, 1999]

In this system, we provide both options of learning models. Unfortunately, the performance of the transductive learning model is noticeably more costly; therefore the transductive model is mainly useful only when the user starts the process of classifying motion chunks. Later on, the user can switch to the inductive learning model, which is much faster and provides equally accurate suggestions after a certain number of training examples have been found.

5 Determining the Grammar

Since the grammar system, as we define it, heavily relies on the notational property of the alphabet, in this section, we will first review the notational property, then describe the concept of the grammar system and finally discuss methods we explored for determining the grammar. The notational property of the alphabet works as follows. By determining the alphabet of a motion capture database, every motion chunk can be described by a basic movement in the alphabet. If every basic movement is assigned a distinct symbolic label, every motion chunk can then be represented by the label of its associated basic movement. Furthermore, every motion capture take can be viewed as a sequence of these labels. For example, in Figure 10, every basic movement in the motion capture take has been given a symbolic label and every motion chunk has been denoted by the label of its respective basic movement.

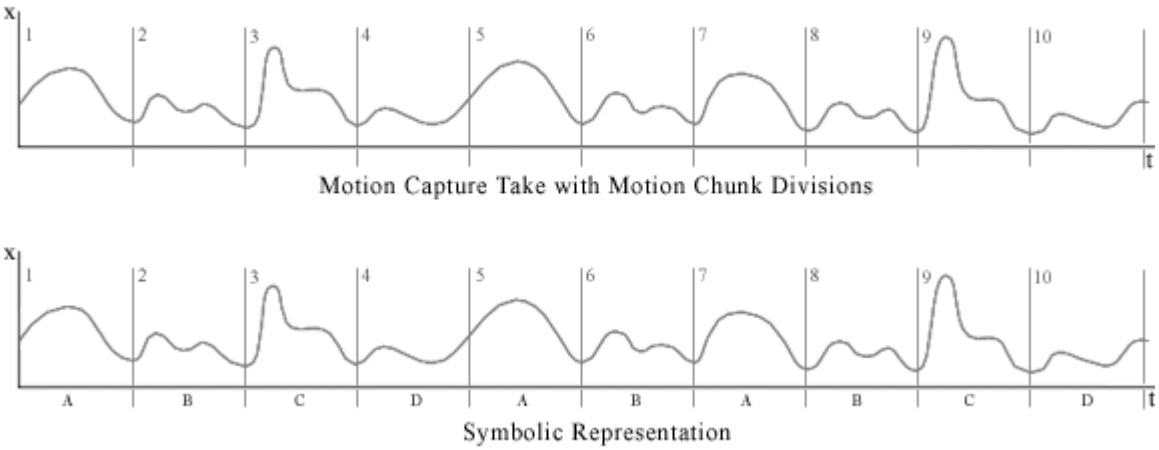

Figure 10 Example of the Notational Property of the alphabet. i) The top graph displays a motion capture take with motion chunk divisions. ii) The bottom graph shows the symbolic representation of the take, using the labels of the basic movements.

Specifically, by observation of their graphs, we notice that motion chunks 1, 5 and 7 should be classified with the same basic movement and indeed they are, with the basic movement labeled as "A."

A grammar rule is defined as a frequently occurring sequence of basic movements, which occurs in more than one context. To clarify, in Figure 11, we display the grammar rules that are determined from the motion capture take shown in Figure 10.

Figure 11 Example grammar rules.

We can see that two grammar rules are uncovered, the sequence "ABCD" and the sequence "AB." From this example, a few nuances of grammar rules can be learned. First, a grammar rule sequence must appear more than once in the database. Secondly, not all frequently occurring sequences are grammar rules. The sequences "BC", "CD", "ABC" and "BCD" are not defined as grammar rules since they only occur within one context, that of Rule 2. For the same reason, "AB" is defined as its own grammar rule since it appears elsewhere besides in the sequence "ABCD." Lastly, grammar rules can be hierarchical and can contain other grammar rules as is exemplified by Rule 2 containing Rule 1.

We approach determining the grammar of a motion capture database by studying whether any text compression algorithms can accept the input of the symbolic representations of motion takes and can determine common text sequences patterns in them.

5.1 Huffman Coding

Huffman coding is one of the most well-known text compression algorithms and for quite some time, had been one of the most popular as well. Therefore, it is a logical first step to examine if Huffman coding could be applied towards the application of determining the grammar of motion sequences. In Huffman coding, each symbol is replaced with an output code, the size of which is inversely proportional to how frequently the symbol occurs in the text. By doing so, the overall size of the text file can be reduced. For example, the symbol 'e' frequently occurs in text and would therefore be mapped to an output code of very small size, hopefully reducing the overall size of the text.

Unfortunately, while the Huffman coding works well for compressing text, it is not useful in determining patterns in sequences of symbols, as it only examines symbols one at a time. However, a newer algorithm, called arithmetic coding, builds upon the idea of Huffman Coding and does inspect the order of symbols to compress text. We study this algorithm next.

5.2 Arithmetic Coding

Similar to Huffman coding, arithmetic coding also records the probabilities at which single symbols occur in the text; however, it is typically extended to retain information on the probability of sequences of symbols, which increases its compression capabilities. In this section, we will discuss how arithmetic coding tracks symbol sequences; however, will refrain from specifying details on how the algorithm compresses text, as it is irrelevant to our goal of determining the grammar.

The simplest mode of arithmetic coding is called order 0 and acts similarly to the behavior of Huffman coding. Arithmetic coding proceeds down the text sequence one letter at a time. With each symbol, it updates a table, recording how often the symbol has appeared and calculates the probability of each symbol from the frequency information. For example,

Input: "ABCDABABCD"

Symbol	Count	(Implied Probability)
A	2	3 / 10
B	3	3 / 10
C	2	2 / 10
D	2	2 / 10
Total	10	

Table 1 Order 0 Table in Arithmetic Coding

In higher order modes, arithmetic coding maintains a record of the frequency of the current symbol as well as the sequence of preceding symbols, referred to as the context. The number of symbols tracked in the context depends on the order number. The higher the order number, the longer the context that is logged. For example, in Table 2, an order

2 context table is shown, displaying the frequencies of occurrence of symbol sequences of length 1, 2 and 3.

Input: “ABCDABABCD”

Order 0

Context: “”	
Symbol	Count
A	3
B	3
C	2
D	2

Order 1

Context: “A”	
Symbol	Count
B	3

Context: “B”	
Symbol	Count
C	2
A	1

Context: “C”	
Symbol	Count
D	2

Context: “D”	
Symbol	Count
A	1

Order 2

Context: “AB”	
Symbol	Count
C	2
A	1

Context: “BC”	
Symbol	Count
D	2

Context: “CD”	
Symbol	Count
A	1

Context: “DA”	
Symbol	Count
B	1

Context: “BA”	
Symbol	Count
B	1

Table 2 Context Table up to Order-2 in Arithmetic Coding

In the order 1 table, the context is 1 symbol long. For example, the “A” context sub table should be interpreted as: the sequence “AB” has occurred three times.

By examining the table and doing simple analysis of the frequency counts, sequences occurring multiple times can be determined. In this example, these sequences would be: “ABC”, “AB”, “BC”, “CD,” and “BCD”. Furthermore, sequences that only occur within other sequences can be found and eliminated, by cross-checking frequencies of the rules. So, for instance, “BC” could be removed, since it occurs twice both in the order 1 table and in the order 2 “BC” sub-table, indicating that it only occurs in the sequence “BCD” and nowhere else.

There are, however, a few issues with using arithmetic coding for finding the grammar. First, the length of grammar rules that can be found is restricted by the order at which arithmetic coding is running. For example, notice that the pattern “ABCD” is not found, because we are in order 2 and are restricted to sequences of length 3 and below. If we try to work around this issue by setting the order to be extremely high, the amount of memory that arithmetic coding requires to run becomes quite large. In addition, it should

be noted that as it is, there is already much unnecessary information being stored in the tables. Arithmetic coding records all sequences that have occurred, no matter what their frequency is.

The next algorithm that we explore, hierarchical grammars, addresses these issues. It has the capacity to find grammar rules of infinite length and does not store unnecessary information.

5.3 Hierarchical Grammars

The Hierarchical grammars algorithm is an elegant compression algorithm developed by Nevill-Manning and Witten [1997]. It works by examining the input sequence, one digram or pair of symbols, at a time and replaces repeated digrams with a rule symbol. For example, the sequence "abcdbc" would be rewritten as "aAdA" where the rule symbol A represents the sequence "bc." This algorithm abides by two properties at all times during runtime:

p_1: No digram appears more than once in the grammar
p_2 : Every rule is used more than once.

Property p_1, referred to as digram uniqueness, states that every digram in the grammar must be unique. Upholding digram uniqueness forces grammar rules to be created and reused when necessary. Property p_2, called digram utility, ensures that a rule remains useful. For example, if the input sequence was "abcabc", we would want the sequence to be rewritten "AA," where A represented the sequence "abc," instead of two rules say "BB," where B was "Ac" and A was "ab."

Running the hierarchical grammars algorithm on the example string sequence "abcdababcd" results in the behavior shown in Figure 12.

Input String: “abcdababcd”

Current Symbol	String so far	Resulting Grammar	Duplicate Digrams	Underused Rules
a	a	S → a		
b	ab	S → ab		
c	abc	S → abc		
d	abcd	S → abcd		
a	abcda	S → abcda		
b	abcdab	S → abcdab - enforce digram uniqueness (create new rule) S → AcdA A → ab	ab	
a	abcdaba	S → AcdAa A → ab		
b	abcdabab	S → AcdAab A → ab - enforce digram uniqueness (reuse rule) S → AcdAA A → bc	ab	
c	Abcdababc	S → AcdAAc A → ab - enforce digram uniqueness (create new rule) S → BdAB A → ab B → Ac	Ac	
d	Abcababcd	S → BdABd A → ab B → Ac - enforce digram uniqueness (create new rule) S → CAC A → ab B → Ac C → Bd - enforce digram utility S → CAC A → ab C → Acd	Bd	B

Figure 12 Hierarchical Grammars behavior

The hierarchical grammars algorithm fits exactly in our context of finding the grammar of the motion capture database, since the set of rules it determines is exactly the one we are looking for, sequences of symbols that appear in more than one context. In addition, unlike the arithmetic coding algorithm, it does not take up any unnecessary storage and can recognize sequences of arbitrary length, making it a better algorithm for our task. Lastly, it can be applied to the application of finding the grammar rules of motion with only the most minor of modifications.

Hierarchical Grammars in our System

The hierarchical grammars algorithm was integrated into our system with only one slight alteration. The symbolic representations of the motion capture takes were concatenated together to form one long text sequence, to be passed as input into the hierarchical grammars algorithm. We altered the algorithm to support a special skip symbol, such that the skip symbol could never be included in any grammar rules. Skip symbols were placed at the end of every motion capture take in the input sequence to prevent rules from being formed to describe sequences spanning two or more motion capture takes.

Take 1:	"abcab"
Take 2:	"caaa"
Skip Symbol:	"|"
Input:	"abcab|caaa"

Figure 13 Example input passed to Hierarchical Grammars algorithm

For example, in Figure 13, had we not included the skip symbol, a new rule describing the sequence "abc" would have falsely been created. (A rule for the sequence "ab", however, should be returned)

6 Experimental Results

6.1 Motion Database

Our Lindy Hop motion database consists of improvised dances and a choreographed dance routine. The motions were recorded in ten motion capture sessions total, with seventeen distinct subjects. Three subjects came to two different sessions and there was one session where just the one subject danced without a partner.

To test the system, we created a Lindy Hop phrase routine, consisting of the eight 8-beat moves shown in Figure 14.

1. Swingout from Open
2. Texas Tommy (Hand Behind the Back)
3. Cross Hand
4. Swingout from Open
5. Lindy Circle to Closed
6. Swingout from Closed
7. Outside Turn
8. Inside Turn

Figure 14 Choreographed Routine

Most subjects were recorded three to five times performing this routine and twenty-one usable takes resulted from all of the sessions.

Improvised dances, without a set routine, were also recorded and we worked with fourteen functional takes of improvised dances, ranging from three to five minutes. For both the phrases and dances, subjects were free to choose the accompanying song from the list in Figure 15.

Figure 15 Song Selection

6.2 Motion Capture Processing

The motions were acquired in a motion capture laboratory, which consists of ten Vicon optical motion capture cameras that record the motions at 120 frames per second. The ten cameras were arranged to create a circular recordable area of twelve feet in diameter. We employed a full-body motion capture marker set, consisting of forty-one markers per subject, a standard Vicon marker set. Since we were capturing Lindy Hop dancers, there were typically two subjects per session, resulting in eighty-two markers being tracked and recorded for each take. A Vicon Hardware Workstation and a dual-processor 1.7 mHz Intel Xeon Workstation with 2.1 gb RAM, running Windows XP were both used to capture the data.

The software/hardware package, Vicon Workstation, records and labels the marker data. These steps require a calibration for which each subject holds a Harley stance, a pose that resembles riding a motorcycle, for 3 seconds. In addition, we also captured each subject doing a T-stance, a pose where the subject stands feet together and arms straight out to the side, resembling the letter T. This stance simplifies skeleton estimation in a separate software package, Kaydara Filmbox, in which marker data post processing was performed. See Appendix B for images of these poses.

The motion capture system infers the 3-dimensional trajectory of each optical marker, but frequently may record random extraneous trajectories created by noise. In addition, because optical motion capture systems require the cameras to have direct line of sight of the markers to calculate their positions, a subject or an object covering a marker may cause a loss of information or a gap in the marker trajectory. Furthermore, as a result of noise or slight shifting of the locations of markers on the subjects' bodies, the Vicon system may mislabel or be unable to label markers, causing marker labels to be missing or inaccurate. Therefore, to obtain a clean database of motion, it is necessary to remove the extra trajectories caused by noise, interpolate the missing information for trajectory gaps and correct the marker labels. In our motion capture pipeline, we removed extraneous marker trajectories and labeled untagged markers in Vicon Workstation. Files were imported into Kaydara Filmbox for more advanced marker corrections, such as fixing incorrect and swapped marker labels and interpolating marker trajectory gaps.

After marker cleanup, the data was exported into the trc file format. The trc files were further processed to include motion chunk division data, which was determined by resychronizing the music with the motion capture data in Alias|Wavefront Maya studio and by recording manually the frame numbers on which the divisions should occur. Motion chunk divisions were made on every other beat of the Lindy Hop dance. Lastly, the marker data and motion chunk division data was saved in a custom designed file format, described in Appendix G.

6.3 Support Vector Machines

To implement Support Vector Machine classifiers, we used the publicly available software library, SVM Lite. We use a radial basic function $<f_1, f_2> = \exp\left(-\frac{|f_1 - f_2|}{\gamma}\right)$ as the decision boundary surface kernel, identical to the usage of SVM classifiers by Arikan et al. [2003]. We allow the user to choose the value for γ, which can provide slight SVM classification improvements for the user's particular dataset. In addition, our system supports both the inductive and transductive learning models.

We tested the quality of the SVM classifiers by using only the choreographed routines. Because the choreographed routines are set sequences of Lindy Hop moves, the motion chunks can be easily manually classified. The results of the SVM classifications can then be compared to the manual classification for a quantitative measure of accuracy. In addition, using only choreographed dances serves as a controlled environment, in which we can observe how the different settings of the SVM classifiers affect classification suggestions.

Segmentation of the twenty-one choreographed routines (including both partners) creates 1,368 total motion chunks. Out of the twenty-one routines, two were selected for test data for all test runs. We varied the number of choreographed routines to use as training data to observe how performance of the SVM classifiers relates to the amount of training information. For γ, the value, 0.17, was a good starting value for our experiments. The results of the SVM classifier testing are shown in Table 3. The running times listed were produced on a 1.2 gHz Athlon with 512 mb of RAM, running Windows 2000. Accuracy was judged using two contexts. We first examined how many SVM suggestions matched the manual classifications out of the total number of unclassified test motion chunks. Secondly, since the SVM classifiers did not suggest classifications for all unclassified chunks, we also assessed how many classifications were correct out of the ones suggested by the classifier.

Num of Training Choreographed Routines	**Number of Training Chunks**	**Learning Mode**	**Running Time (min)**	**Accuracy Rate** (correct / num suggested / num test chunks)
2	106	Inductive	4:20	18 / 19 / 128
3	160	Inductive	4:40	35 / 36 / 128
4	224	Inductive	4:50	36 / 37 / 128
6	416	Inductive	4:55	44 / 47 / 128
8	544	Inductive	5:15	48 / 51 / 128
2	106	Transductive	7:30	88 / 111 / 128
3	160	Transductive	7:55	97 / 116 / 128
4	224	Transductive	8:40	106 / 119 / 128
6	416	Transductive	9:20	107 / 122 / 128
8	544	Transductive	10:10	108 / 119 / 128

Table 3 Performance of Support Vector Machines with different settings.

As can be seen from the results, both the inductive and transductive learning models have high levels of accuracy. In general, the inductive learning model emphasizes maintaining a very low level of misclassification error, which ranged from 2.7% to 6.4%. In achieving this accuracy, however, inductive learning classifies only a subset of the unclassified chunks. The inductive learning model, with a large quantity of training data (greater than 4 choreographed routines) only suggested classifications for 37 to 51 motion chunks out of a possible 128 unclassified motion chunks. On the other hand, the transductive learning model allows for larger misclassification errors (12.3% to 20.7%), but provides more classification suggestions.

For both the inductive and transductive learning models, the number of suggested classifications and the quality of the suggestions improved or remained the same as more training data was introduced. This behavior indicates that the Support Vector Machine algorithm is a robust and scalable learning machine that is capable of handling the large amounts of motion-capture data.

A slow performance was one drawback of Support Vector Machines, especially in regards to the transductive learning model. While the transductive learning model tended to be more useful than the inductive learning model in terms of requiring less work on the user's end in classifying the motion chunks, it also took twice as long to run, reducing its effectiveness, forcing the user to wait longer before reviewing the SVM classifier results. However, for both the inductive learning and transductive learning models, the process of training the SVM classifiers can be separated from the application of the SVM classifiers, which permits the user to create a classifier once and employ it repeatedly for classification of new data. For the running times listed in Table 3, roughly half of the time is used for training the SVM classifiers and should be noted as a one-time performance cost. As a result, the performance of SVM classifiers can be slightly improved by the separation of the training and application processes.

Despite the slower performance, the Support Vector Machine learning algorithm provides an excellent method for creating the alphabet of a motion capture database. Its accuracy and completeness, in the case of the transductive learning model, in classifying motion chunks, allow the user to develop the alphabet quickly and reduce user review and supervision required with clustering methods or manual classification.

6.4 Hierarchical Grammar

Our implementation of the hierarchical grammar algorithm followed the general Sequitur algorithm detailed by Nevill-Manning and Witten [1997]. Checking for existing rules and existing digrams is the main bottleneck of the algorithm. In our implementation, we performed digram and rule lookups using the Standard Template Library map object. While this may not have been the most efficient data structure (Nevill-Manning and Witten suggested a doubly-linked list data structure), performance of our implementation still remained quite fast, processing a sequence of 1,368 symbols in less than three seconds.

We also made the implementation decision to represent a symbol with the string object and a text sequence as a vector of strings. For instance, an example input sequence might be the vector shown in Table 4, where "s1" is the first symbol of the text sequence, "b2" is the second symbol, and so on.

1	"s1"
2	"b2"
3	"d"
4	"ab1"

Table 4 A text sequence as vector of strings

This choice was made to support an unrestricted number of basic movements in the alphabet, each of which requires its own distinct symbol. In addition, we do not know beforehand, how many rules are going to be created by the hierarchical grammar algorithm and likewise, each rule necessitates its own individual symbol. By using a string object, an unlimited number of possible identifier symbols can be created for rules and basic movements.

In addition, the string objects ease differentiation between rules and symbols denoting regular letters. In our implementation, we add the underscore character, "_", as a prefix to any rule in the grammar. Initially each rule is assigned a numeric label (e.g. "_17" or "_129"). The user can also rename rules to have more descriptive labels and therefore, name rules with labels such as "_Swingout" or "_TexasTommy."

We tested the generated grammar rules on the choreographed dance routines, which consist of a sequence of well defined Lindy Hop movements. For example, if the choreographed routine is represented by the sequence of symbols "abcdefgh," we would expect the hierarchical grammar algorithm to return a rule, for example, "_100," which is assigned either this text sequence directly or a sequence of sub-rules, which when flattened would represent this text sequence. Furthermore, we would expect that a certain set of these sub-rules would contain a sequence of symbols or rules that similarly represent the shorter Lindy Hop moves, such as a Swingout or a Texas Tommy.

Using all of the twenty-one choreographed routines for both partners, the hierarchical grammar algorithm identified sixty-eight grammar rules, varying in length from two to twenty-six basic movements. In Table 5, we list the grammar rules that represent recognizable Lindy Hop moves, either directly or when its hierarchy is flattened.

Length (Num of Chunks)	**Rule Name**	**Leader / Follower**	**Lindy Hop Move(s) Represented**
4	_18	Follower	Swing Out from Open
	_56	Follower	Texas Tommy
	_140	Follower	Texas Tommy (variant)
	_17	Follower	Cross Hand
	_139	Follower	Outside Turn
	_29	Leader	Outside Turn
	_32	Leader	Inside Turn
	_109	Leader	Inside Turn (variant)
8	_35	Follower	Swing Out, Texas Tommy
	_78	Leader	Swing Out, Texas Tommy
	_75	Leader	Outside Turn, Inside Turn
16	_160	Follower	2nd half of routine: Lindy Circle, Swingout, Outside Turn, Inside Turn
	_13	Leader	1st half of routine: Swing out, Texas Tommy, Cross Hand, Swing out
	_54	Leader	2nd half of routine: Lindy Circle, Swingout, Outside Turn, Inside Turn
19	_152	Leader	Longer than half of routine
26	_94	Follower	Nearly all of routine

Table 5 Recognizable Lindy Hop moves discovered

From Table 5, a few observations can be made. First, the hierarchical grammar algorithm determines the longest possible frequently occurring sequence. Because there is wide

variation in the motion chunks of the choreographed routines, the entire choreography was not returned as a grammar rule. For example, because of the inaccurate classification, the routine may have been represented by both text sequences, "abcdeXgh" and "abcdeYgh", where "X" and "Y" are misclassified similar movements. The hierarchical grammar algorithm was successful in determining the most lengthy existing sequence. For the leader dancer, the longest sequence, is described by the rule, "_152," which when flattened was 19 basic movements long. For the follower, the rule, "_94", was 26 basic movements long, close to, but not quite the length of the entire choreographed routine (32 basic movements).

Not all frequently occurring recognizable Lindy Hop moves are returned as grammar rules (such as the Swingout and Cross-hand for the leader), because the hierarchical grammar algorithm favors the first-occurring repeating sequence. For instance, if provided with the input sequence "abcbcabc", the hierarchical grammar algorithm will never return "ab" as a grammar rule, since "bc" repeats first in the sequence. The Swingout or Cross-hand moves for the leader were not returned as a grammar rule for this reason. However, the system does recognize that they are frequently occurring patterns as they are implicitly returned in rule "_13," which includes both of these moves. Consequently, uncovering the complete set of all Lindy Hop dance moves can be accomplished by creating new rules out of all subsequences of longer rules. For instance, the hierarchical grammar algorithm, give the input sequence "abcbcabc," would return two rules: A → "bc" and B → "abc." A new rule C → "ab" could be created by examining the longer rule B. Similarly, rule "_13" could be decomposed into a number of rules describing its sub-sequences, three of which would exist that describe the Swing Out, Texas Tommy, and Cross-Hand Lindy Hop moves for the leader subject respectively.

Overall, while the hierarchical grammar algorithm does not directly determine the Lindy Hop moves of the choreography, it remains a very flexible and feasible method for determining the set of grammar rules of a motion capture database. It can quickly uncover the longest frequently occurring sequence in the input, and the complete set of all frequently occurring sequences of arbitrary lengths can be determined from its records. The performance of the hierarchical grammar algorithm is very speedy, there are no

limitations on the length of discovered rules and memory use is efficient, making it a solid algorithm for the task of determining the grammar.

6.5 Simple Alphabet and Grammar Application

We also explored the use of alphabet and grammar for recognition of the Lindy Hop moves in improvised dances. The alphabet and grammar, determined by the analysis of the choreographed routines, was used to train SVM classifiers on the motion chunks of the improvised dances. Next, a simple text matching algorithm applied the grammar rules of the choreographed routines to the dances. For example, if there existed a rule, which described a swing out as the sequence, "abcd," and that particular sequence existed in the improvised dance, the rule would be applied to that sequence and a swing out would be recognized.

We tested this process on three improvised dances, which by visual inspection, we determined had seventeen Lindy Hop moves found in the choreographed routines. The SVM classification and grammar rules automatically distinguished fourteen. With further manual classification of the chunks that were misclassified by the SVM classifiers, all seventeen Lindy Hop moves were recognized.

These experiments show that the combination of the Support Vector Machine and the hierarchical grammar algorithms allow the user to determine the alphabet and grammar rules of a motion capture database quickly and accurately. We have furthermore shown a successful application of the alphabet and grammar in the systematic recognition of longer, more human-identifiable movement units, such as Lindy Hop moves, in improvised dances.

7 Discussion and Future Work

We presented a user supervised system which determined the alphabet and grammar rules of a motion capture database of rhythmic motion. Support Vector Machine classifiers guided the user in building up the alphabet of the database, allowing for quick and highly accurate construction of the basic movements. Dances were then translated into symbolic representations and passed into the text compression algorithm, hierarchical grammars, for grammar rule detection. Afterward, users could view frequently occurring variable length movements.

We believe that this project showcases a unique perspective on how to represent and work with motion and have confidence that this system can be extended in many ways to provide better solutions to the challenges plaguing the process of motion capture. We foresee some of the future directions and follow-up projects for this system:

- Determining the alphabet/grammar of general motion. Our system focused on rhythmic motion alone and as a result, certain issues remained unanswered. It may be useful to find out if an alphabet and grammar could be determined for general motion.

- Motion capture editing/design systems. As mentioned earlier, the alphabet and grammar provide properties that allow for better organization of motion, allowing editors to be able to find specific motions more easily and quickly. In addition, the notational property provides users with an additional and possibly more intuitive method for specifying motion. Algorithms ensuring smooth transitions of existing movements could be added to this system to provide an intuitive method for motion synthesis from existing motion.

- Motion analysis systems. The alphabet and grammar provide intuitive organization and representational of motion that allows for a more systematic analysis of motion.

- Automatic segmentation. Dividing motion into motion chunks was done manually in this system. A small future direction could explore more automated approaches for partitioning motion into smaller units.

References

ARIKAN, O. AND FORSYTH, D. (2002) "Interactive Motion Generation from Examples." In *Proceeding of SIGGRAPH 2002*.

ARIKAN, O., FORSYTH, D. AND O'BRIEN, J. (2003) "Motion Synthesis from Annotations." In *Proceedings for SIGGRAPH 2003* (to appear).

BURGES, C. J. C. (1998) "A Tutorial on Support Vector Machines for Pattern Recognition." In *Data Mining and Knowledge Discovery*. Vol 2. pp. 121-167

CORTES, C. AND VAPNIK, V. (1995) "Support-Vector Networks." In *Machine Learning*. Vol 20. No. 3 pp. 273-297 1995

DUDA, R. O., HART, P. E. AND STORK, D. G. *Pattern Classification*, 2nd ed. pp. 550-563 New York, NY: Wiley, 2001.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns*. Addison-Wesley, 1997.

JOACHIMS, THORSTEN. (1999) "Text Categorization with Support Vector Machines: Learning with Many Relevant Features." In *Proceedings of the European Conference on Machine Learning*. Springer, 1998.

JOACHIMS, THORSTEN. (1999) "Transductive Inference for Text Classification Using Support Vector Machines." In *International Conference on Machine Learning* (ICML) 1999.

KIM, T., PARK, S. I., AND SHIN, S.Y. (2003) "Rhythmic-Motion Synthesis Based of Motion-Beat Analysis." In *Proceedings of SIGGRAPH 2003* (to appear)

KOVAR L., GLEICHER, M., AND PIGHIN, F. (2002) "Motion Graphs." *Proceedings of SIGGRAPH 2002*

LI. Y., WANG, T. AND SHUM, H.-Y. (2002) "Motion Texture: A Two-Level Statistical Model for Character Motion Synthesis." In *Proceedings of SIGGRAPH 2002*.

LEE. J., CHAI, J., REITSMA, P.S., HODGINS, J.K. AND POLLARD, N.S. (2002) "Interactive Control of Avatars Animated with Human Motion Data." In *Proceedings of SIGGRAPH 2002*.

NELSON, M. "Arithmetic Coding + Statistical Modeling = Data Compression." (1991) In *Dr. Dobb's Journal*. February, 1991

NEVILL-MANNING, C. AND WITTEN, I. (1997) "Compression and Explanations Using Hierarchical Grammars." In *The Computer Journal*. Vol. 40 No. 2/3

OSUNA, E., FREUND, R., AND GIROSI,. F (1997) "Training Support Vector Machines: an Application to Face Detection." In *Proceedings of CVPR 1997* June 17-19, Puerto Rico

PULLEN, K.A. AND BREGLER, C. (2002) "Motion capture assisted animation: Texturing and synthesis." In *Proceedings of SIGGRAPH 2002*.

ZELNIK-MANOR. L., AND IRANI, M. (2002) "Event-based Analysis of Video." In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2001.

Appendix A: Motion Capture

Description of Motion Capture

Figure 16 Virtual actors from motion capture data

Motion capture is a process that allows the location, orientation and motion of an object to be recorded and mapped to a virtual representation on the computer. Motion capture is most well known for its application in creating virtual actors for films, animations and video games; however, is also used in biomechanics, athletics and medicine for movement analysis. Currently, there are two methods that are typically used to perform motion capture, optical and magnetic, each with its own set of pros and cons.

Optical Motion Capture

Optical motion capture involves attaching small balls coated in retro-reflective paint, referred to as markers, as close as possible to the subject's joints. A number of specialized high-shutter speed infrared cameras are arranged in a circle and pointed towards a central recording area, such that the 3-dimensional locations of the subject's markers can be calculated and recorded to a computer. This marker data can be used to power a virtual skeleton, on top of which, skin and clothing textures can be layered. The advantages of using optical motion capture are: one, the subjects are relatively unconstrained by any equipment which might alter their

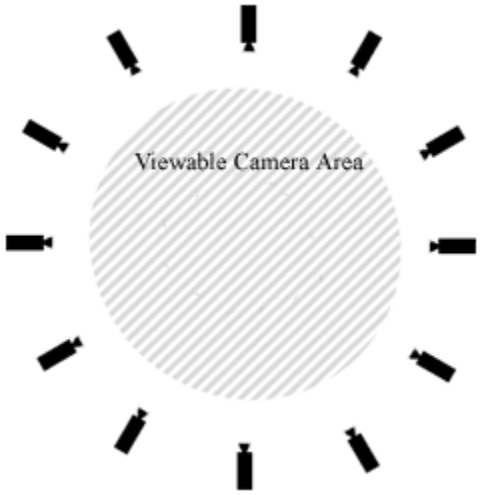

Figure 17 Motion Capture Camera Setup

original motion, two, the viewable camera area is large, allowing for multiple greater freedom of movement or the recording of multiple subjects and thirdly, large amounts of detailed motion can be captured. One of the disadvantages of using optical motion capture is that markers may be occluded from camera sight by body parts or other objects, resulting in loss of marker data. Secondly, light and other reflective material, like that on certain sneakers, can affect the motion capture. Lastly, optical motion capture systems generally are more expensive than other routes of motion capture. Note that in the past, data from optical motion capture systems required more time to process; however, there have been recent motion captures system developments that support real-time optical motion capture processing.

Magnetic Motion Capture

In magnetic motion capture, magnetic sensors are attached to the subject's joints and measure the relative strength of a magnetic field from a source positioned at a set location. By doing so, the sensors provide the absolute positions and orientations of the joints relative to the source location. The strengths and weaknesses of magnetic motion capture tend to be complementary to those of optical motion capture, in that magnetic motion capture tends to be less expensive and sensors can constantly measure data. However, magnetic motion capture can suffer from low accuracy, small recording range and sensitivity to interference caused by metallic objects.

Appendix B: Calibration Poses

Calibration Poses *(Vicon Workstation)*

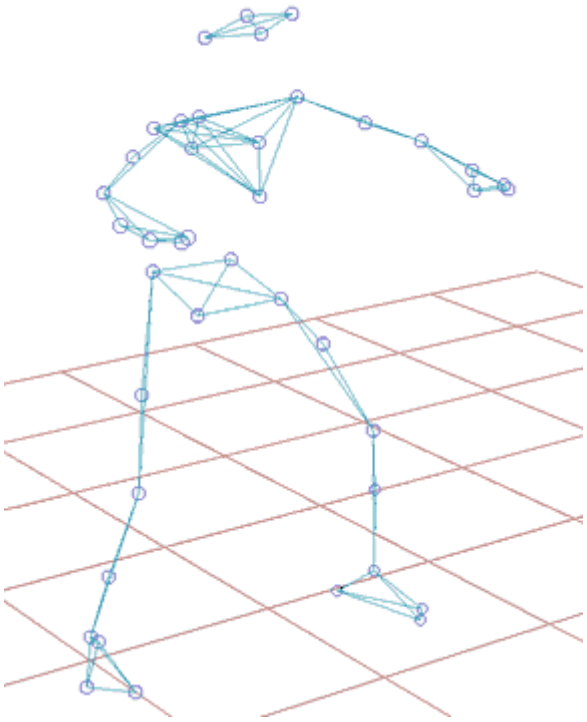

Figure 18 The Harley Stance. One of the poses used for calibration

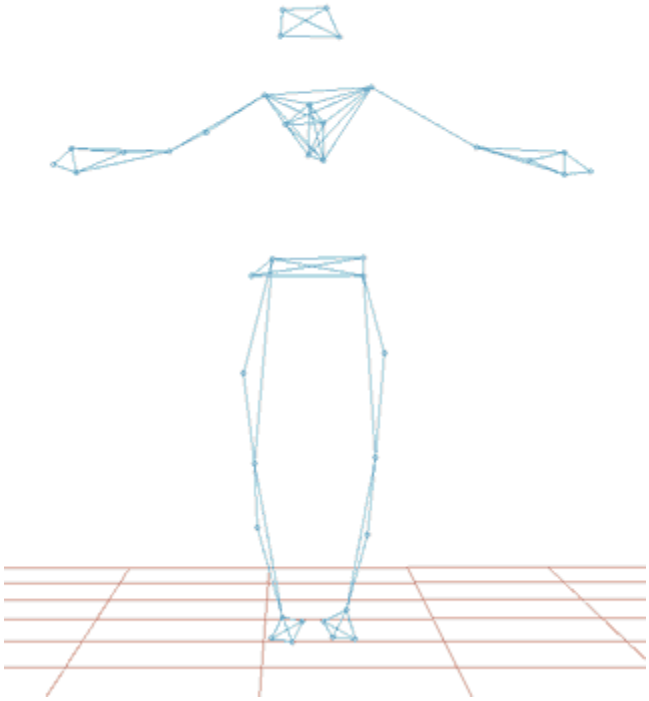

Figure 19 The T-Stance. Another pose used for calibration

Appendix C: Motion Chunk Structure

Each dance chunk is structured as an array of frames, which in turn is an array of marker locations.

- Chunk 1
  - Frame 1
    - Left hip
      - x coordinate
      - y coordinate
      - z coordinate
    - Right hip
      - x coordinate
      - y coordinate
      - z coordinate
    - ….
  - Frame 2
    - Left hip
      - x coordinate
      - y coordinate
      - z coordinate
    - Right hip
      - x coordinate
      - y coordinate
      - z coordinate
    - ….
  - …
  - Frame n
    - Left hip
      - x coordinate
      - y coordinate
      - z coordinate
    - Right hip
      - x coordinate
      - y coordinate
      - z coordinate
      -
    - …

Figure 20 Motion Chunk Data Structure

Appendix D: Determining Basic Movements without the Aid of Motion Chunks

Determining basic movements without first partitioning motion takes into smaller units of motion provides a difficult challenge. We review that the goal of determining a basic movement is to find two windows of frames, for which the movements are similar. For example, in Figure 21, the series of frames t_1 to $t_1+\Delta$ is similar to the frames at t_2 to $t_2+\Delta$ and should be classified as the same basic movement.

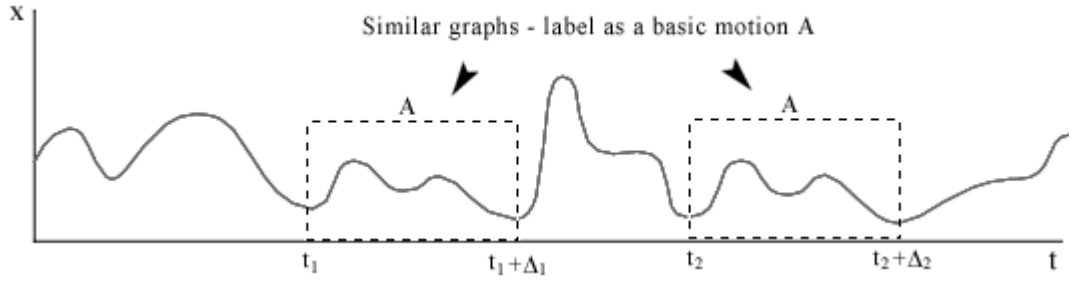

Figure 21 Finding a basic movement without motion chunks

We present some of the issues involved by stepping through a brute-force approach to solving this problem, that in which we arbitrarily select a target frame window (i.e., in Figure 21, the range t_1 to $t_1+\Delta$) and search through the motion capture take by advancing a second query window (the range t_2 to $t_2+\Delta$) forward until a matching motion is found. One of the first issues that we encounter is how to select an appropriately sized target window. In Figure 22, we observe that selecting too lengthy of a query window may result in finding no matches at all.

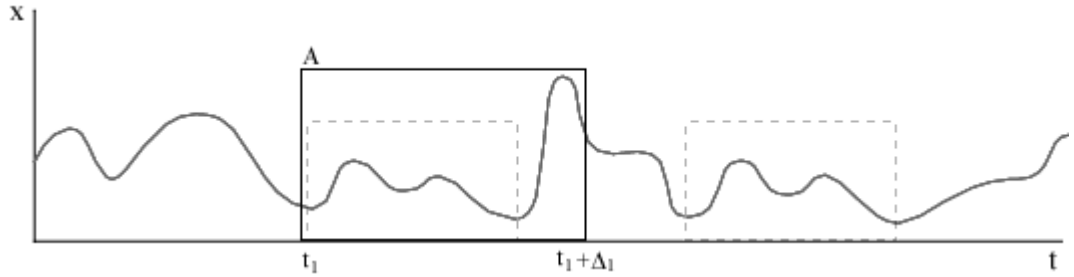

Figure 22 Finding a basic movement using too large of a target window.

However, proceeding in the opposite direction, by choosing a shorter query window, also encounters problems as well. As seen in Figure 23, choosing a small target window may

allow an appropriate matching query window to be determined; however, the window may not be a complete basic movement.

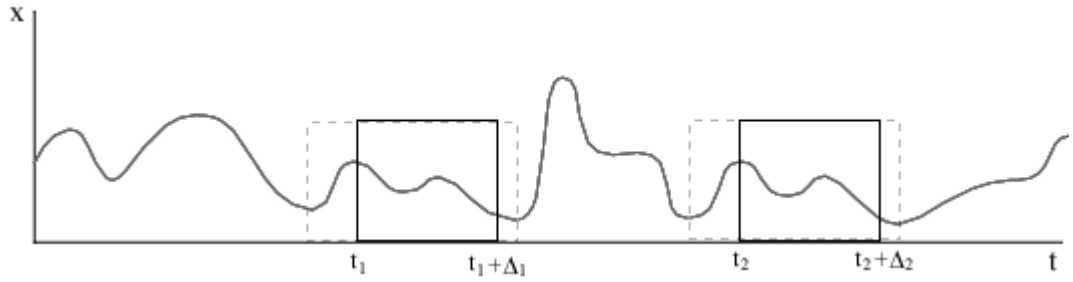

Figure 23 Finding a basic movement using too small of a target window.

Secondly, it is necessary to be able to find similar motions spanning different lengths of time. For example, in Figure 24, the method should be capable of recognizing that the two windows should be classified as the same basic movement.

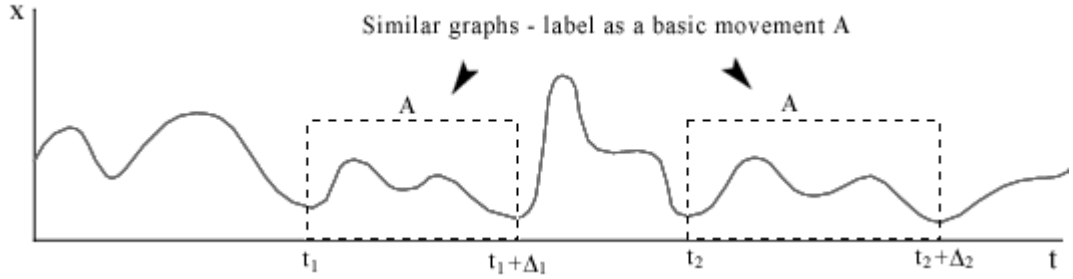

Figure 24 Comparing ranges of frames of different lengths

To solve this, one could vary the query window's length as it is advanced; however, this may prove to be a computationally expensive feature.

Lastly the issue of the granularity of advancement of the query window must be addressed. Moving the query window forward with too great of a step could cause matches not to be found, while pushing the query window forward at too short of a delta step, could prove to be yet another performance costly choice. All in all, determining basic movements without the aid of motion chunk divisions remains a challenging task, which must be addressed if the task of finding the alphabet and grammar of general motion is ever attempted.

Appendix E: Foundation Code Documentation

Main

<main.cpp>
Main driver of program, reads in dance file and sets any calculation into motion

Chunkt

<chunkt.cpp, chunkt.h>
Contains chunk_t definition and chunk methods. If I had more time, I would have converted this to a Chunk class, complete with operators.

Definitions:

Chunk_t

typedef vector< vector<Vec3d> > Chunk_t

A Chunk is simply a vector of frames, where each frame is a vector of markers and each marker is composed of a x, y, z coordinate.

Chunk_t[frame_no].[marker].x/y/z

Example:

```
Chunk_t exampleChunk;
// set x coordinate of marker 13 in the 15th frame of the chunk
exampleChunk[15][13][0]  = 0.5;
```

AXIS

enum AXIS {X=0, Y=1, Z=2}

The AXIS enum is for specifying which axis want data for

MARKER

```
enum  MARKER {
        ROOT,
        LEFT_HIP,
        LEFT_THIGH,
        LEFT_SHIN,
        RIGHT_HIP,
        RIGHT_THIGH,
        RIGHT_SHIN,
        LEFT_SHOULDER,
        LEFT_BICEP,
        LEFT_WRIST,
        RIGHT_SHOULDER,
        RIGHT_BICEP,
        RIGHT_WRIST }
```

The MARKER enum is for specifying which marker want chunk data for

Methods:

convertIntToMarker

MARKER convertIntToMarker(intMarker);

Convert integer marker index to MARKER enum type.

convertIntToAxis

AXIS convertIntToAxis(intAxis);

Convert integer axis index to AXIS enum type.

findTransform

void findTransform(const Chunk_t &a, const Chunk_t &b, Mat4d &T);

Figure out the transformation to bring 'b' to 'a' in a least squares sense (impl from Kovar/Gleicher). Chunks 'a' and 'b' must be equally size (use extendchunk if needed);

transformFrame

void transformFrame(std::vector<Vec3d> &f, const Mat4d &T);

Transforms all the points in frame f, in place.

Parameters:

f – Frame (i.e. vector <Vec3d>)

T – the transform to use

transformChunk

void transformChunk(Chunk_t &a, const Mat4d &T);

Transforms all the frames in chunk a, in place.

a – Chunk to transform

T – the Transform to use

compareChunks

double compareChunks(const Chunk_t &a, const Chunk_t &b);

Compares the chunks a and b and returns a numerical value of the difference between the two chunks. Assumed that chunks 'a' and 'b' are being compared post-transformation!

Parameters:

a, b – two chunks to compare

Returns:

double value of difference

extendChunk

void extendChunk(Chunk_t &a, size_t to);

Extend length of chunk 'a' to length 'to' through interp method.

concatChunk

void concatChunk(Chunk_t &a, const Chunk_t &b);

Concatenates b to the end of a. Attempts to transform b, such that first frame of chunk b matches the last frame of chunk a.

printChunk

void printChunk(std::ostream &out, const Chunk_t &a)

Print all values of marker coordinates for every frame

validChunk

bool validChunk(const Chunk_t &a);

Returns whether every frame contains same number of markers.

accumChunk

void accumChunk(Chunk_t &x, const Chunk_t &y);

Adds chunk y to chunk x.

x = x + y

subChunk

void subChunk(Chunk_t &x, const Chunk_t &y);

Subtracts chunk y from chunk x.

x = x - y

mulChunk

void mulChunk(Chunk_t &x, const Chunk_t &y);

Multiply chunk x times chunk y.

x = x * y;

void mulChunk(Chunk_t &x, double a);

Multiply chunk x times constant a

x = x * a;

averageChunk

```
void averageChunks(const std::vector<int> &chunkIDs,
                   const std::vector<Chunk_t> &chunks,
                   Chunk_t &avgChunk);
```

Average selected chunks.

Parameters:

chunkIDs (input) – vector of chunkIDs to average. Indexes into parameter chunks

chunks (input) – vector of all chunks (from db)

avgChunk (output) – returns this with average of all chunks

stdDevChunk

```
void stdDevChunks(const std::vector<int> &chunkIDs,
                  const std::vector<Chunk_t> &chunks,
                  const Chunk_t &avgChunk,
                  Chunk_t &stdChunk);
```

Take the Standard Deviation of the selected chunks
Parameters:
chunkIDs (input) – vector of chunkIDs to average. Indexes into parameter chunks
chunks (input) – vector of all chunks (from db)
avgChunk (input) – the average of the selected chunks
stdChunk (output) – returns this with standard deviation chunk

createDummyChunk
Chunk_t createDummyChunk(int length);
Create a dummy chunk with length frames.

ChunkAsVector
string ChunkAsVector(Chunk_t theChunk);
Gets vector representation of chunk. Basically this flattens out a chunk: it lists out the marker values of all markers for every 5th frame in a string. This is used for SVM Classification
Returns:
String - chunk as a vector

Class Cluster
<cluster.cpp, cluster.h>
A misnomer. This is just a group of chunks

Fields:

clusterId
int clusterId;
clusterID is mainly used for indexing into vector of clusters in Database

clusterName
string clusterName;
User provided cluster label

memberIds
vector<int> memberIds
Vector of all the chunkIDs of the members belonging to this cluster

average
Chunk_t average
The average of all the chunk member

stddev
Chunk_t stddev
The standard dev of all the chunk members

maxId

int maxId

Yet to be implemented. Together with minId, keeps track of two chunks in cluster with greatest distance.

minId

int minId

See Above.

Constructor:

Cluster

Cluster()

Cluster(int clusterId);

Cluster(int clusterId, std::vector<int> &newMemberIds);

Methods:

getClusterId

int getClusterId() const

getClusterName

string getClusterName() const

setClusterName

void setClusterName(string newClusterName)

getMembers

vector<int> getMembers() const

Returns vector of member chunk Ids

setMembers

void setMembers(std::vector<int> &newMembers)

Set memberIds to vector

addMember

void addMember(int newMember)

Adds new chunkID to memberIds. Does not update average/stddev chunk. If desired, run method calcClusterStats.

removeMember

int removeMember(int existMember);

clear

void clear();

Clears all members

size
int size() const
Returns the number of members in the cluster

getMaxId
int getMaxId() const
Returns the ChunkID of the "Max" Chunk. Not implemented yet.

getMinId
int getMinId() const
Returns the ChunkID of the "Min" Chunk. Not implemented yet.

getAverage
Chunk_t getAverage()
Returns Average chunk

getStdDev
Chunk_t getStdDev();
Returns standard dev chunk

calcClusterStats
void calcClusterStats(const std::vector<Chunk_t> &chunkList);

Operators:
[]
int operator[] (int letterIndex)
References chunk member at index letterIndex

==
bool operator== (const Cluster& rhs);
Returns whether two clusters are equal

!=
bool operator!= (const Cluster& rhs);
Returns whether two clusters are inequal

Class Dance

<dance.cpp, dance.h>
Represents one dance file, a consecutive sequence of chunks.

Fields:
_danceID
int _danceID
The ID of the dance. Used for indexing into vector of dances in Database.

_startChunkID
int _startChunkID
Dances are opened and processed sequentially. For each dance, Chunks (and their resulting chunkIDs) are created sequentially. As a result, each dance represents a sequential range of chunkIDs. startChunkID is the chunkID of the starting chunk of the dance inclusive.

_endChunkID
int _endChunkID
The chunkID of the last chunk in the dance inclusive.

_danceName
string _danceName
A label for the dance. Usually set to the dance filename.

Constructor:
Dance
Dance(int danceID, int startChunkID, int endChunkID, string danceName)
Parameters:
See Above.

Methods:
getDanceID
int getDanceID() const

getStartChunkID
int getStartChunkID() const
getEndChunkID
int getEndChunkID() const

getDanceName
string getDanceName()

size
int size()
Returns the number of chunks in Dance.

containsChunkID
bool containsChunkID(int chunkID)
Returns whether parameter chunkID belongs to Dance

Class Database
<database.cpp, database.h>

Database is the centralized datastructure that contains all chunks and class information. It is a singleton class, meaning that there is only one instantiation of the class Database. (For further information, see *Design Patterns* by Gamma, Helm, Johnson and Vlissides)

Fields:

_instance

Database* _instance

Database keeps track of the lone Database instance.

_chunks

vector<Chunk_t> _chunks

Vector of all chunks as determined by trc files and beat frame number text files

_transformedChunks

vector<Chunk_t> _transformedChunks

Same as _chunks vector, except all of the chunks have been transformed to match up to the first chunk.

_danceList

vector<Dance> _danceList

Vector of all Dances.

_chunkLabels

vector<string> _chunkLabels

Labels of all the chunks. Unclassified chunks have labels of their chunk IDs.

_clusterMap

map <string, Cluster > _clusterMap

Map of class labels to class

_unclassified

Cluster _unclassified

Class containing all unclassified chunks. Mainly used for quick reference to unclassified chunks.

_classCounter

int _classCounter

Class counter. A integer value for counting how many classes have been created. Used for the default class label given to a new class.

For SVM Use

classPredictionMap

map<string, double> classPredictionMap

Maps a class label to its prediction value.

_chunkPredictions

map<int, classPredictionMap>
Maps a chunk ID to a classPredictionMap

_chunkPredictMax
map<int, double>
Maps a chunk ID to the max prediction value. If this value is positive, the chunk will be suggested as belonging to the class that the value is associated with.

_SVMChunkLabels
map<int, string>
Maps a chunk ID to a suggested label, if there exists one.

_SVMClusterMap
map<string, Cluster>
Maps class labels to classes (for the classes for which there are suggested chunks to be added to it)

For Hierarchical Grammars Use
_theSequitur
Sequitur
For running the Sequitur hierarchical grammars

_superChunks
map<string, SuperChunk>
Maps superchunk labels to all superchunks

Constants:
DB_FILE_ID
string
A string tag to insert into the ".db" files to differentiate db files from regular text files.

FILESECTION_CLASS_CTR_ID
string
A string tag to insert into the ".db" files to describe a section for recording the class counter.

FILESECTION_SVM_PREDICT_MEMBER_ID
string
A string tag to insert into the ".db" files to describe a section for recording the SVM suggestions.

FILESECTION_SEQUITUR_S_ID
string

A string tag to insert into the ".db" files to describe a section for recording the S rule for sequitur.

FILESECTION_SEQUITUR_RULES_ID
string
A string tag to insert into the ".db" files to describe a section for recording the grammar rules.

DB_FILE_EXTENSION
string
Specifies what the database file extension is. Currently set to ".db"

DANCE_FILE_EXTENSION
string
Specifies what the dance file extension is. Currently set to ".ch"

Methods:

instance
static Database* instance()
This method is to be used instead of the constructor, which is abstracted away.
Example:

```
Database* theDB = Database::Instance();
theDB->setChunks(tempChunkVect);
```

(get/set)Chunks
vector<Chunk_t> getChunks()
void setChunks(std::vector<Chunk_t> &newChunks)
Gets or sets _Chunks vector

getSelectChunk
Chunk_t getSelectChunk(int chunkID)
Gets specific chunk from _Chunks vector. Recommended over getChunks for retrieving single records.

(get/set)ClusterMap
map<string, Cluster> getClusterMap()
void setClusterMap(map<string, Cluster> &newClusters)
Gets or sets _clusterMap map

getSelectClass
Chunk_t getSelectClass(string className)
Gets specific chunk from _ClusterMap map. Recommended over getSelectClass for retrieving single records.

(get/set)TransformedChunks
vector<Chunk_t> getTransformedChunks()

void setTransformedChunks(std::vector<Chunk_t> &newChunks)
Gets or sets _transformedChunks vector

getSelectTransformedChunk
Chunk_t getSelectTransformedChunk(int chunkID)
Gets specific chunk from _transformedChunks vector. Recommended over getTransformedChunks for retrieving single records

(get/set)DanceList
vector<Dance> getDanceList()
void setDanceList(std::vector<Dance> &newDanceList)
Gets or sets _danceList

getSelectDance
Chunk_t getSelectDance(int chunkID)
Gets specific dance from _danceList vector. Recommended over getDanceList for retrieving single Dances.

(get/set)ChunkLabels
vector<Chunk_t> getChunkLabels()
void setChunkLabels(std::vector<Chunk_t> &newChunks)
Gets or sets _chunkLabels vector

(get/set)ClassCounter
int getClassCounter()
void setClassCounter(int ctrValue)
Gets or sets _classCounter

incClassCounter
void incClassCounter()
Increments _classCounter by 1

getSVMClusterMap
map<string, Cluster> getSVMClusterMap()
Returns _SVMClusterMap

getSuperChunks
map<string, SuperChunk> getSuperChunks()
Returns _superChunks map

getSelectSuperChunk
Chunk_t getSelectSuperChunk(int SChunkID)
Gets specific super chunk from _superChunks vector. Recommended over getSuperChunks for retrieving single records.

numChunks

int numChunks()
Returns the number of chunks in the database.

numClusters
int numClusters()
Returns the number of clusters/classes in the database

addClass
bool addClass(Cluster newCluster)
Adds a new class to the database. Returns true if successfully added, false if not. Database does not allow for classes with duplicate class labels.

renameClass
void renameClass(string oldClusterID, string newClusterID)
Renames a class with a new class label.

deleteClass
void deleteClass(string clusterID)
Removes a class with the provided class label from the database.

existsClass
bool existsClass(string clusterID, map<string, Cluster> clustMap)
Checks the provided cluster map if there exists a particular cluster.

classMembers
vector <Chunk_t> classMembers(string clusterID, bool transformed)
Returns the Chunk_t members of a class. The bool transformed denotes whether the members should transformed to emulate the first member of the class.

calcClassStats
void calcClassStats(string clusterID)
Calculates the average and standard deviation for a class.

labelChunk
void labelChunk (int chunkID, string clustID)
Adds chunk to the class, with the given class label. Updates the label of the chunk accordingly.

unLabelChunk
void unLabelChunk (int chunkID)
Removes the chunk from any class it may have belonged to and adds to the unclassified class. Updates the label of the chunk accordingly.

clearClassifyData
void clearClassifyData()
Clears all classification data, created by user, SVM learning or superchunks.

saveDatabase
bool saveDatabase(string filename)
Saves the database to the provided filename, overwriting the file if file already exists. (File structure is text-based). Returns whether successful in saving.

loadDatabase
bool loadDatabse(string filename)
Loads the database in the provided filename. Returns whether successful in loading.

importCHFile
bool importCHFile(string filename)
Imports a CH dance file into the database. Returns whether successful in saving.

SVM Use

setSVMPredictions
void setupSVMPredictions()
Sets up data structures for new SVM predicting.

tentLabelChunk
void tentLabelChunk(int chunkID, string clustID)
Used for SVM. Create a SVM suggested labels for an unclassified chunk. Chunk is actually still unclassified. If user confirms the label, labelChunk should be called.

tentUnlabelChunk
void tentUnlabelChunk(int chunkID)
Used for SVM. Removes the SVM suggested label of chunk.

Hierarchical Grammar Use

runHierarchicalGrammar
void runHierarchicalGrammar()
Runs the hierarchical grammar algorithm on existing dances.

buildSuperChunks
void buildSuperChunk()
Builds the SuperChunks from the grammar rules

renameRule
void renameRule(string oldRuleID, string newRuleID)
Renames a grammar rule from oldRuleID to newRuleID

findDanceRep

vector<string> findDanceRep(int whichDance, int ruleLimitation)
Finds the representation of dance, of index whichDance, using grammar rules of length limitation, ruleLimitation

getRuleLengths
vector<int> getRuleLengths()
Gets the lengths of all the rules in the grammar

Class UserInterface

<ui.cxx, ui.h>
User interface. Instantiates FLTK widgets and custom Open GL views. Includes some logic for dealing with user interaction with widget

Fields:
_theDB
Database _theDB;
The database is the central place for all data: chunks, clusters, classes, etc, which all modules reference.

Constructor:
UserInterface
UserInterface()

Methods:
Show
void show()
Shows UserInterface

Redraw
void redraw();
Redraws UserInterface

setDB
void setDB(Database &inDB)
Sets _theDB to given parameter.

SuperChunk

<superchunk.cpp, superchunk.h>
Multi-beat chunks.

Constructor:
SuperChunk
SuperChunk()

Fields:
_chunkMembers

vector <Chunk_t>
The chunks that this structure contains stored in sequential order.

_memberLabels
The label of the chunks.

_combinedChunk
The chunk representation of all the chunks concatenated together.

Methods:

addChunk
void addChunk(Chunk_t theChunk, string theChunkLabel)
Adds the chunk (and chunk label) on to the end of the superchunk.

getCombinedChunk
Chunk_t getCombinedChunk()
Returns the _combinedChunk chunk

getMembers
vector <Chunk_t> getMembers()
Returns the vector of chunks

getMemberLabels
vector <string> getMemberLabels()
Returns the vector of chunk labels

numMembers
int numMembers()
Returns the number of chunk members superchunk contains

size
int size()
Returns how many frames long the superchunk is

Operators:

[]
int operator[] (int chunkIndex)
References chunk member at index chunkIndex

==
bool operator== (const Cluster& rhs);
Returns whether two superchunks are equal

!=
bool operator!= (const Cluster& rhs);
Returns whether two superchunks are inequal

UI Helper

<uihelper.cpp, uihelper.h>
General helper functions for the UserInterface.

Methods:

getChunkIDFromBrowser

int getChunkIDFromBrowser(string browserLine)

Retrieves the integer value chunk ID from a string browser line.

createBrowserLineChunkID

string createBrowserLineChunkID(int chunkID)

Returns a browser string line that contains the chunk label and ID.

Utilities

<util.cpp>
General utility functions.

Methods:

Convert

convert <convertToType> (covertFromType);

Converts a variable from one type to another.

Example:

```
double d;
string salary;
string s="12.56";
d=convert <double> (s);      //d equals 12.56
salary=convert <string> (9000.0);    //salary equals "9000"
```

Class GraphView : Fl_Gl_Window

<graphview.cpp, graphview.h>
A openGL viewer for plotting graphs of markers for chunks

Constructor

GraphView

public GraphView(int x, int y, int w, int h, const char *l=0)

Parameters:

x – x coordinate of top left corner
y – y coordinate of top left corner
w – width of GraphView
h – height of GraphView
l – Unused label

Fields

_selectedCluster

protected Cluster **selectedCluster**

Cluster which average and standard deviation will be viewed.

_selectedChunks

protected "Collection"<Chunk_t> **selectedChunks**

Chunks which will be graphed

_selectedMarker

protected int **selectedMarker**

Marker to focus upon.

_selectedFrame

protected size_t **selectedFrame**

Current frame

_viewCluster

protected bool **viewCluster**

Boolean if set to true, will view graph of selectedCluster

_viewChunks

protected bool **viewChunks**

Boolean if set to true, will view graph of selectedChunks

Methods:

draw

void draw( )

Draws graphview. If selectedCluster set to true, draws average and standard deviation graph of Cluster. If selectedChunk set to true, draws chunk graphs. (Overlayed if both are set to true). If there does not exist selectedCluster or selectedChunk, displays nothing. Redraw( ) refreshes graph.

handle
int handle(int event)
Handles any events within GraphView. Currently nothing is set.

addChunk
void addChunk(const Chunk_t &inChunk)
Add new chunk to be viewed

clearChunks
void clearChunks()
Clears all selected chunks

setFrame
void setFrame(size_t frame)
Sets the current frame to input

setSelectedAxis
void setSelectedAxis(AXIS whichAxis)
Set which Axis to be viewed. See Chunk_t for AXIS enumeration.

setSelectedCluster
void setSelectedCluster(const Cluster &inCluster)
Sets GraphView's selectedCluster to new cluster

setSelectedMarker
void setSelectedMarker(MARKER whichMarker)
Set which marker to be viewed. See Chunk_t for MARKER enumeration.

setViewCluster
void setViewCluster(bool viewClust)
Sets whether selectedCluster will be graphed or not

setViewChunks
void setViewChunks(bool viewChunks)
Sets whether selectedChunks will be graphed or not

Class ChunkView : Fl_Gl_Window

<chunkview.cpp, chunkview.h>
A openGL viewer for viewing chunks

Constructor

ChunkView

public ChunkView(int x, int y, int w, int h, Camera*& inCamera, int inCVID, const char *l=0);

Parameters:

x – x coordinate of top left corner
y – y coordinate of top left corner
w – width of ChunkView
h – height of ChunkView
inCamera – Camera to use
l – Unused label

Fields

_chunks1

protected "Collection"<Chunk_t > _chunks1;

List of chunks to be viewed

_chunks2

protected "Collection"<Chunk_t> _chunks2;

List of chunks to be viewed. Shown in different color than chunks1. Possibly to be overlayed on top of chunks1

_chunkViewID

protected int _chunkViewID;

An id in the case ChunkView's need to be differentiated

_distinguishSChunkMembers

protected bool _distinguishSChunkMembers;

Boolean indicating whether to the chunks of the SuperChunk in different colors

_frame

protected size_t _frame;

The current viewing frame.

_ptrCamera

protected Camera* _ptrCamera;

A pointer to the OpenGL camera, mainly used for synching up the camera view between separate ChunkViews

_showOverlayChunk

protected bool _showOverlayChunk;

Boolean indicating whether to show chunks in _chunks2

_superchunks

protected vector<SuperChunk> _superchunks;

List of SuperChunks to be viewed.

Methods

draw

void draw()

Draws the chunks in _chunks1. If _showOverlay is set to true, overlays chunks in _chunks2. If no _chunks1 and _chunks2 contain no chunks, only draws the "floor."

handle

int handle(int event);

Handles events that occur inside ChunkView. Currently handles click and drag of all 3 mouse buttons. Left mouse button: rotate, Middle mouse button: translate, Right mouse button: zoom

addChunks1

void addChunks1(const Chunk_t &newChunk1);

Add chunk to _chunks1

addChunks2

void addChunks2(const Chunk_t &newChunk2);

Add chunk to _chunks2

clearAll

void clearAll();

Clear all data in ChunkView

clearChunks1

void clearChunks1();

Clear all chunks in _chunks1

clearChunks2

void clearChunks2();

Clear all chunks in _chunks2

clearSuperChunks

void clearSuperChunks();

Clear all SuperChunks

distinguishSChunkMembers

bool distinguishSChunkMembers(bool toDistinguish);

Sets _distinguishSChunkMembers

(get/set)Camera

Camera* getCamera() const

Returns a pointer to the ChunkView's camera.

void setCamera(Camera *& inCamera)

Sets what the camera is. Generally ChunkView needs to be redrawn to reflect new Camera

getChunks1

vector<Chunk_t> getChunks1();

Returns _chunks1 collection. This should probably return an iterator to abstract away the implementation of Chunks1.

getChunks2

vector<Chunk_t> getChunks2();

Returns _chunks2 collection. See above for details.

hasChunks1

bool hasChunks1()

Returns boolean whether _chunks1 contains any chunks

hasChunks2

bool hasChunks2()

Returns boolean whether _chunks2 contains any chunks

setBeats

void setBeats(vector<int> &beats);

Sets where the beats are for SuperChunks

setFrame

void setFrame(size_t frame);

Sets what the current frame number is

showOverlayChunks

void showOverlayChunks(bool showChunk)

Sets whether chunks in _chunks2 should be shown or not

Appendix F: Hierarchical Grammars Code Documentation

Class Rule

<rule.cpp, rule.h>
A representation of the hierarchical grammar rule

Constructor

Rule

public Rule();

Fields

_secondletter

typedef map<string,int> _secondletter;

Second letter map. To quickly find 2nd letter in digram lookup

_userCount

int _useCount;

Stores how many times rule is used.

_containsNumRules

int _containsNumRules

Count of how many rule symbols are in string. This is used when rule set is flattened.

_digramMap

map<string,_secondletter> _digramMap;

For easy lookup of digrams in rule

_theString

vector<string> _theString;

Stores the sequence of symbols (string) that rule represents

Methods

add

void add(string letter);

Adds symbol to the end of the rule. Will update digramMap accordingly. This does **not** check if symbol causes duplicate digram.

clear

void clear();

Clears all rule information

erase

void erase(int letterIndex);

Erases symbol at provided index. Like the vector index, indices start at 0.

expandRule

void expandRule(string ruleLetter, const vector<string> replaceString);

When the count of the rule, r1, identified by argument ruleLetter, drops to 1, it should be replaced with the contents of the r1. This method iterates through _theString and replaces every occurrence (there should only be one occurrence) of r1 with the contents of r1.

getContainsNumRules

int getContainsNumRules()

Gets the count of how many rule symbols are contained in the string.

getString

vector<string> getString() const

Returns the contents of the rule

getUseCount

int getUseCount()

Returns the use count of the rule

hasDigram

bool hasDigram(string l1, string l2);

Returns whether the given digram exists in the rule

hasSymbol

bool hasSymbol(string l1);

Returns whether rule contains given symbol.

(inc/dec)UseCount

void incUseCount(int delta);

Increments the count of how many times the rule has been used, by the amount of delta

void decUseCount(int delta);

Decrements the count of how many times the rule has been used, by the amount of delta

(inc/dec)ContainsNumRules

void incContainsNumRules(int delta)

Increments the count of how many rule symbols this rule contains. containsNumRules field must be updated by caller since Rule class cannot tell which symbols are marked as rules symbols.

void incContainsNumRules(int delta)

Increments the count of how many rule symbols this rule contains.

printString

void printString();

Sends the contents of the rule to output

replace

int replace(string oldLetter, string newLetter);

Replaces all occurences of oldLetter with the newLetter. Returns the number of replacements made.

int replace(string l1, string l2, string replaceletter);

Replaces the argument digram with the argument replaceLetter (presumably of the symbol of another rule). Returns the number of replacements made, so that the caller can update the count of the argument rule.

size

int size();

Returns the length of the sequence

Operators

Operator [ ]

string operator[] (int letterIndex);

Returns the symbol at the argument index. Index follows vector indices (starting at 0)

Operator ==

bool operator== (const Rule& rhs);

Returns whether two rules are equal or not

Operator !=

bool operator!= (const Rule& rhs);

Returns whether two rules are inequal or not

Operator <<

friend ostream& operator<<(ostream& os, const Rule& rhs);

Sends _theString to output stream.

Class Sequitur

<sequitur.cpp, sequitur.h>

Our implementation of Nevill-Manning, Witten Sequitur program

Constructor

Sequitur

public Sequitur();

Fields

_verbose
bool _verbose
Indicates whether current state should output to standard output.

_S
Rule _S
The top level rule, containing the representation of the original string with rule symbol replacements.
Ex. Original string: "abcdbcabc"
S → BdAB (See below for definition of rules)

_theRules
map<string, Rule> _theRules
A mapping of rule symbols to rules.
Ex. Original string: "abcdbcabc"
A → bc
B → aA

_theFlatRules
map<string, Rule> _theFlatRules
A mapping of rule symbols to flattened rules. In the set of flattened rules, there exists no rule that contains a rule symbol in its string.
Ex. Original string "abcdbcabc"
A → bc
B → abc

RuleMap
Typedef map<string, Rule>
A map of rule names to Rules

Constants
RULE_ID_PREFIX
String
The prefix for differentiating rules from regular symbols.

Methods

addString
void addString(vector<string> inputtxt, vector<string> outputtxt, bool verbose)

Perform the hierarchical grammar on the existing text and adding the argument inputtxt. Verbose boolean sets the field _verbose. Outputtxt returns the representation of the inputtxt using the grammar rules

getS

Rule getS()

Returns the rule S

getRules

map<string, Rule> getRules()

Returns the map of rule IDs to rules

getFlattenedRules

map<string, Rule> getFlattenedRules()

Returns the map of rule IDs to flattened rules. In the flattened rule set, there is no rule that contains a rule symbol within its string.

getRepresentation

void getRepresentation(vector<string> &inpuutxt, vector<string> &outputtxt, int lengthThresh)

Returns the representation of inputtxt through outputtxt using the existing grammar rules. lengthThresh is a length threshold for the rules to apply. If lengthThresh is -1, there is no threshold imposed.

getRuleName

string getRuleName(string ruleID)

The rule ID contains the string rule prefix. This function returns the name of the rule without the prefix.

getStructuredRepresentation

void getStructuredRepresentation(vector<string> &inpuutxt, vector<string> &outputtxt, int ruleLength)

Returns the representation of inputtxt through outputtxt using the existing grammar rules. ruleLength specifies the length of the rules to apply..

getVerbose

bool getVerbose()

Returns the verbose setting.

isRuleLetter

bool isRuleLetter(string letter)

Returns whether the argument symbol is a rule symbol

markAsRuleLetter

string markAsRuleLetter(string letter)

Marks a symbol as a rule symbol

printGrammar
  void printGrammar()
        Outputs the grammar at current state to standard output.

printFlattenedGrammar
  void printFlattenedGrammar()
        Outputs the flattened grammar at current state to standard output

Sequitur Utilities

<sequiturUtil.cpp, sequiturUtil.h>
Contains helper functions for sequitur program

Convert
   template <class out_type, class in_value>
   out_type convert(const in_value & t)
        Converts from one type to another. See Utilities section in framework documentation

Appendix G: Input File (.Ch) Structure

[#c : Number of Chunks]

[#f : Number of Frames] [#m : Number of Markers] (Chunk: 1)

Frame 1 :	x y z (Marker: 1)	x y z (Marker: 2)	…	x y z (Marker: m)
Frame 2 :	x y z (1)	x y z (2)	…	x y z (m)
Frame 3 :	x y z (1)	x y z (2)	…	x y z (m)
.				
.				
.				
Frame f :	x y z (1)	x y z (2)	…	x y z (m)

[#f : Number of Frames] [#m : Number of Markers] (Chunk: 2)

Frame 1 :	x y z (Marker: 1)	x y z (Marker: 2)	…	x y z (Marker: m)
Frame 2 :	x y z (1)	x y z (2)	…	x y z (m)
Frame 3 :	x y z (1)	x y z (2)	…	x y z (m)
.				
.				
.				
Frame f :	x y z (1)	x y z (2)	…	x y z (m)

.
.
.

[#f : Number of Frames] [#m : Number of Markers] (Chunk: c)

Frame 1 :	x y z (Marker: 1)	x y z (Marker: 2)	…	x y z (Marker: m)
Frame 2 :	x y z (1)	x y z (2)	…	x y z (m)
Frame 3 :	x y z (1)	x y z (2)	…	x y z (m)
.				
.				
.				
Frame f :	x y z (1)	x y z (2)	…	x y z (m)

Appendix H: Database File Structure

File Structure

```
#Mocap_Chunk_DB_Identifier              // Identifies that this is a Database file
Class1 Member1 Member2 .                // Stores members of Class_Name
Class2 Member1 Member2 Member3 .        // Members separated by space char
Class3 Member1 .                        // Mark end of list with '.' char
Class4 Member1 Member2 .
Class5 Member1 Member2 Member3 .
Unclassified Member1 Member2 .          // Store unclassified chunks

#SVM_Predictions_Members                // Indicate SVM Predictions section
Class1 Member1 Member2 .                // Store SVM suggested members
Class3 Member1 .
Class4 Member1 Member 2 .

#Sequitur_S                            //Indicate the S rule of the grammar
Rule1 Letter2 Rule5.                   // Representation of 1st dance
Rule 6 Rule 7.                         // Representatino of 2nd dance

#Sequitur_Rules                        //Indicate grammar rules section
Rule1 Letter5 Letter2 .                // Define rule1
Rule2 Letter1 Letter2 Rule1 .          // Define rule2
Rule5 Rule1 Rule2 .                    // Define rule5

#Class_Counter 22                      // Stores the number a new class
                                       // should be labelled as
```

File Example

```
#Mocap_Chunk_DB_Identifier
GRP10 2 10 14 34 38 42 118 278 6 46 54 274 162 246 174 50 110 182 .
GRP11 75 67 79 87 195 207 151 215 .
GRP12 129 133 141 145 345 153 157 193 205 217 17 21 29 197 209 221 337 349 .
GRP13 225 229 237 241 249 253 257 261 .
GRP14 228 252 256 260 236 240 .
GRP15 289 293 313 317 321 325 301 305 .
GRP16 3 11 15 43 183 175 287 59 .
GRP17 26 66 150 .
GRP19 80 68 76 88 92 208 216 28 64 144 152 192 196 204 220 336 344 20 16 140 332 .
GRP20 22 134 198 70 295 .
GRP21 55 35 47 163 .
GRP6 32 36 44 48 56 60 188 160 172 184 176 280 284 4 272 12 164 .
GRP7 33 37 45 49 109 125 113 1 5 9 13 117 121 173 177 181 185 189 161 165 277 41
273 281 53 .
```

GRP8 65 69 77 81 .
Unclassified 0 7 8 18 19 23 24 25 27 30 31 39 40 51 52 57 58 61 62 63 71 72 73 74 78 82
83 84 85 86 89 90 91 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 111
112 114 115 116 119 120 122 123 124 126 127 128 130 131 132 135 136 137 138 139
142 143 146 147 148 149 154 155 156 158 159 166 167 168 169 170 171 178 179
... *(Abbreviated)*

#SVM_Predictions_Members
GRP10 122 166 243 270 282 .
GRP11 27 91 203 335 351 .
GRP12 89 93 107 142 146 179 299 307 333 .
GRP14 248 .
GRP19 128 132 200 .
GRP6 .
GRP7 57 61 .

#Sequitur_S
_13 .
_35 _F_CrossHand _F_SwingOut2 .
_13 _54 .
_F_SwingOut _F_TexasTommy2 _F_CrossHand _F_SwingOut _55 _116 .
... *(Abbreviated)*

#Sequitur_Rules
_0 MGRP00 MGRP03 .
_100 GRP45 GRP27 .
_3 MGRP08 MGRP13 .
_35 _F_SwingOut2 _F_TexasTommy .
_36 GRP26 GRP29 .
_38 _4 GRP36 .
_F_SwingOut3 _55 GRP41 .
_L_InsideTurn _85 MGRP24 .
_L_InsideTurn2 _85 MGRP13 .
_L_OutsideTurn _122 MGRP23 .
... *(Abbreviated)*

#Class_Counter 22