2006

Visualization of Secondary RNA Structure Prediction Algorithms

Brandon Hunter
San Jose State University

Follow this and additional works at: http://scholarworks.sjsu.edu/etd_projects

Part of the Computer Sciences Commons

Recommended Citation
http://scholarworks.sjsu.edu/etd_projects/20

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.
Visualization of Secondary RNA Structure Prediction Algorithms

A Project Report

Presented to
The Faculty of the Department of Computer Science
San Jose State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Brandon Hunter
2006
Contents

1 Introduction .............................................. 5
  1.1 RNA Secondary Structure Prediction .................................................. 5
  1.2 Importance of RNA Secondary Structure .............................................. 6
  1.3 RNA Secondary Structure Visualization .............................................. 6

2 RNA Introduction ........................................... 8
  2.1 What is RNA ....................................................... 8
    2.1.1 Nitrogenous Bases ..................................................... 9
    2.1.2 Pentose Sugar ....................................................... 10
    2.1.3 Phosphate Group .................................................... 11
  2.2 RNA Synthesis .................................................... 11
  2.3 Types of RNA ...................................................... 12
    2.3.1 Messenger RNA (mRNA) .............................................. 12
    2.3.2 Transfer RNA (tRNA) ............................................... 13
    2.3.3 Ribosomal RNA (rRNA) ............................................. 13
    2.3.4 Small Nuclear RNA (snRNA) ....................................... 13
    2.3.5 Double Stranded RNA (dsRNA) .................................... 14
    2.3.6 Small Interfering RNA (siRNA) .................................... 14
2.3.7 Micro RNA (miRNA) ................................................................. 14
2.3.8 Other RNA Types .................................................................. 15
2.4 RNA World Hypothesis ............................................................. 15
2.5 Implications of World Hypothesis on Structure Prediction .............. 15

3 Secondary RNA Structure .......................................................... 17
3.1 Secondary Structure Formal Description ....................................... 17
3.2 Taxonomy of Algorithms ............................................................ 18
  3.2.1 Deterministic ....................................................................... 18
  3.2.2 Stochastic ........................................................................... 19
3.3 Issues with Algorithms ............................................................... 19
3.4 Secondary Structure Elements .................................................... 19
3.5 Secondary Structure Visualizations .............................................. 21
  3.5.1 String Representation .......................................................... 21
  3.5.2 Bracketed Representation ...................................................... 21
  3.5.3 Linked Graph Representation ............................................... 22
  3.5.4 Planar Graph Representation ............................................... 22
  3.5.5 Tree Representation ............................................................ 23
  3.5.6 Circular Representation ......................................................... 24
  3.5.7 Matrix Representation .......................................................... 25
  3.5.8 Dot Plot Representation ......................................................... 25
  3.5.9 Mountain Representation ...................................................... 26

4 Nussinov Folding Algorithm ......................................................... 28
4.1 Nussinov Algorithm .................................................................. 28
Chapter 1

Introduction

This chapter introduces the secondary structure prediction problem. It describes what the secondary structure prediction problem is and why it is important. Based on the importance of the algorithm it is essential to have a clear means to visually represent the problem. Therefore, this chapter details the high level goals of the visualization. It details how the visualization will visually represent the problem through several simultaneous representations. These visual representations will be tied together in order to increase the understanding of the algorithm.

1.1 RNA Secondary Structure Prediction

Secondary RNA structure prediction is a problem that endeavors to predict the two dimensional structure of an RNA sequence given only the nucleotide bases of the sequence. This problem is very complex because there are an enormous number of possible secondary structures that could be created from any given RNA sequence. As the length of the RNA sequence gets longer, so too does the number of possible secondary structures that could be created from the sequence. The key to the problem of secondary RNA structure prediction is to distinguish between structures which are biologically correct from those that are not correct. In practice, the algorithms that attempt to solve the secondary RNA structure prediction problem are still being perfected but they all suffer computational complexity issues.
1.2 Importance of RNA Secondary Structure

The RNA structure prediction problem is very important to the Bioinformatics field because one very common task performed by researchers is to find all the homologues of a given RNA sequence. The way this is typically done is that a researcher would search a genome database for matches to a sequence and if a similarity threshold were achieved then the sequences would be considered homologous. The database search could be significantly improved if one were able to search for similarities in secondary structure in addition to the sequence itself. This improvement can be achieved because it has been shown that homologous RNA can share secondary structure while not sharing a high degree of sequence similarity.

1.3 RNA Secondary Structure Visualization

The RNA secondary structure problem has been such an important topic that many techniques have been devised to visualize the secondary structure. Instead of creating a visualization where one simply enters the input RNA sequence and the program calculates a visual representation of the output, this paper proposes a visualization which is richer in features and visual representations. Instead of just being an input/output visualization this paper details a visualization where the individual can step through an RNA sequence and interactively view the secondary structure as it’s created. The visualization detailed in this paper is intended to be a teaching tool which would help individuals who are new to the topic quickly and intuitively understand how the secondary structure of a particular RNA is calculated. Since the focus of this visualization is to help increase understanding it follows that the definition of visualization used in this paper is as proposed by Card, Mackinlay and Shneiderman in “Readings in Information Visualization, Using Vision to Think” as: “The use of computer-supported, interactive, visual representations of abstract data to amplify cognition” [4]. The authors of the book propose that the purpose of the visualization should be to amplify cognition or the understanding of a problem. That is exactly the intention of the visualization in this paper. The visualization will amplify cognition by
allowing the individual to visually see the connection between the algorithms matrix used in the secondary structure calculation to the visual representation of the RNA. This visualization will use several techniques to increase cognition all of which were proposed in the book [4]. First, the user of the visualization will have increased memory and processing resources available because they will not have to use much thought to understand the visualization therefore freeing the user to spend resources elsewhere. Second, the user will not have to waste mental resources searching for information because the visualization will provide all relevant information in the display. Third, the visualization will be created in such a way that it enhances the user’s ability to find patterns in the information. Fourth, the visualization will allow the user to perceptually infer details about the problem. Fifth, the visualization will use techniques that draw the attention of the user to the desired location. Lastly, the visualization will increase cognition by encoding the details of the problem in a medium which can be manipulated through a forward and back stepping mechanism.

Since the goal of this paper is to detail a means to display several representations of the secondary RNA structure prediction problem, it is important to understand the make up of RNA. The next chapter will detail the make up of RNA as well as all the known types of RNA. By analyzing all the know types of RNA, it will help give an understanding of which types of RNA are important to study. It is also important to know the typical sequence lengths for each type of RNA in order to know which types of RNA this visualization will be able to model. Since this visualization is intended as a teaching tool it will not in devour to visualize extremely long sequences. The intention is to focus on conveying as much information as possible about how the secondary structure algorithm works. Therefore, the target sequence length is approximately 100 nucleotides or less.
Chapter 2

RNA Introduction

This chapter gives an overview of the components of RNA and starts by describing the actual make up of RNA at the molecular level. It describes the three main structural components of RNA, the nitrogenous base, the pentose sugar and the phosphate group. These three structural components form chains in RNA so it is shown how the chains are bound together. Finally, the chapter describes the types of RNA. Each type of RNA is categorized based on similar function and sequence length.

2.1 What is RNA

RNA is a biological structure contained within all living organisms. The term RNA is actually an acronym which stands for ribonucleic acid. There are two types of nucleic acids found within living organisms. There’s deoxyribonucleic acid and ribonucleic acid. The nucleic acid is composed of chains of nucleotides. The nucleotides are further composed of three components: a nitrogenous base, a pentose sugar and a phosphate group (See Figure 2.1) [25].
Figure 2.1. RNA Nucleotide [25]

Notice that in figure 2.1 above, if the oxygen molecule where the arrow points was removed then this sugar would become a deoxyribose sugar. This is where the deoxyribose name is derived from since it is a ribose sugar without the oxygen molecule at the indicated location.

2.1.1 Nitrogenous Bases

There are five primary nucleotide bases which are common to both DNA and RNA, Adenine, Guanine, Cytosine, Thymine and Uracil [7]. Adenine, Guanine, Cytosine and Thymine are most commonly found in DNA while Thymine is replaced with Uracil in RNA. These nucleotide bases are generally grouped according to the number of atoms that make up the ring in the nucleotides structure. Purine bases have a fused ring composed of 9 atoms, 5 carbon atoms and 4 nitrogen atoms [7]. Adenine and Guanine are Purine bases. Pyrimidine bases have a single ring composed of 6 atoms, 4 carbon and 2 nitrogen [7]. Cytosine, Thymine and Uracil are all Pyrimidine bases (See Figure 2.2).

All nucleotide bases form bonds to other nucleotide bases in specific combinations. These bonds are generally referred to as Watson-Crick base pairs [7]. The Watson-Crick base pairing states that Adenine forms a bond with its complementary base pair Thymine and that Cytosine forms a bond with its complementary base pair Guanine in DNA. In RNA, the Thymine is replaced with Uracil so the complementary pair is Adenine to
Uracil. Adenine bonds to either Thymine or Uracil with two hydrogen bonds while Cytosine bonds to Guanine with three hydrogen bonds. Since the Cytosine and Guanine pair has more bonds it is considered to be a more stable paring.

The five primary bases are only a small fraction of the total number of bases which actually appear in RNA. There are an abundance of modified bases such as Pseudouridine and Thymidine which generally occur in the TΨC loop of tRNA [7]. There are around 100 modified bases which occur in RNA, and there are so many modified bases that most of them are not generally understood [7].

![Figure 2.2. RNA Bases](image)

### 2.1.2 Pentose Sugar

The pentose sugar is a biological structure with five carbon atoms. The root of the name pentose, pent, meaning five suggests as much. There are many types of pentose sugars such as Lyxose, Xylose, Arabinose, and Ribose but RNA contains a ribose sugar [7]. The ribose sugar in RNA has five carbon atoms which are typically labeled 1’ through 5’ with hydroxyl (OH) groups at the 1’, 2’, 3’ and 5’ locations (See Figure 2.1). DNA differs from RNA in that DNA is missing a hydroxyl group at the 2’ location. The ribose sugar of an RNA chain is located in between two phosphates while the nitrogenous base forms bonds on the side. The phosphate bonds to the sugar at the 5’ and 3’ locations while the nitrogenous base bonds at the 1’ location (See Figure 2.3). The phosphate
bonds at the 5' and 3' locations are what allow the RNA to form chains. The chain starts with a phosphate on the 5' side of the sugar followed by the sugar with a nitrogenous base at the 1' side of the sugar and then another phosphate at the 3' side of the sugar. The phosphate on the 3' side of the sugar will bond with the 5' side of the next sugar and the thus the chain is formed [26].

![RNA Chain](image)

**2.1.3 Phosphate Group**

The phosphate consists of a central phosphorus atom surrounded by four oxygen atoms [7]. The phosphate acts as the linking mechanism between the ribose sugars in an RNA chain (See Figure 2.3).

**2.2 RNA Synthesis**

Although there are many types of RNA, each of which perform a specific function, one of the main purposes of RNA is to be the messenger which helps carry the genetic information coded in the DNA and to convert that coded information into proteins. The Central Dogma of Molecular Biology states just that [2]. The Central Dogma states that DNA can replicate itself; it can transition to RNA through a process called translation and it can transition to a protein through a process called transcription.
The synthesis of the RNA which occurs in the transcription phase of the central dogma is performed in the following manner. First, an RNA polymerase binds to the DNA at a promoter site which is typically found upstream of the gene that is going to be transcribed [7]. Next, the DNA is unwound by the polymerase into two individual sequences referred to as the coding strand and the template strand. The coding strand goes in the 5’ to 3’ direction while the template goes in the 3’ to 5’ direction. The RNA is synthesized from the template strand so the polymerase travels down the template strand in the 3’ to 5’ direction. The RNA is created by continually adding nucleotides to the RNA sequence which are complementary to the template strand in a process referred to as elongation [7]. This process will continue until a code in the DNA tells the polymerase to stop.

2.3 Types of RNA

Biologists have discovered that there are several types of RNA which fall into categories based on their biological function and typical sequence length. For the purposes of this visualization it is very important to understand the multitude of RNA types in order to understand which sorts of RNA the visualization will be able to handle. It is also important to understand the function of each type of RNA so that it can be determined if any type of RNA has more significance to the secondary structure prediction problem than another.

2.3.1 Messenger RNA (mRNA)

Messenger RNA (mRNA) is responsible for carrying the genetic information encoded in the DNA to ribosome receptor sites where the genetic information is translated into a protein [7]. The mRNA classes of RNA’s are quite long and are typically in the order of several thousand base pairs. In eukaryotic cells the mRNA does not start the process of protein synthesis until the mRNA has exited the nucleus and is in the cytoplasm. In prokaryotic cells there is no nucleus so protein synthesis can begin as the mRNA is being transcribed from the DNA. Once in the cytoplasm, the mRNA attaches to an organelle called a ribosome. The ribosome helps the transfer RNA (tRNA) attach to the mRNA three base pairs at a time referred to as a codon [2]. The mRNA is read in order while the
tRNA connected to the mRNA one codon at a time. Each tRNA has an attached amino acid that forms a chain as the mRNA is read and the amino acid chain becomes a protein.

2.3.2 Transfer RNA (tRNA)

Transfer RNA (tRNA) is a short sequence of about 74 to 93 nucleotides in length [7]. The tRNA generally has a secondary structure that resembles a clover leaf where the middle clover leaf branch contains three nucleotides referred to as the anti-codon. The 3’ end of the tRNA usually ends with the three nucleotide sequence CCA and then is attached to an amino acid. The tRNA is responsible for binding to the mRNA at the tRNA anti-codon site through hydrogen bonding [7]. As the tRNA bonds to the mRNA at the ribosome bond site, the tRNA connectes its amino acid to the growing polypeptide chain and hence facilitates the synthesis of proteins.

2.3.3 Ribosomal RNA (rRNA)

Ribosomal RNA (rRNA) is a component of the protein synthesis molecule called the ribosome. There are 4 types of rRNA in eukaryotic cells referred to as 18S, 5S, 5.8S and 28S [7]. The 18S rRNA along with around 30 different proteins constitute the small subunit of the ribosome while the 5S, 5.8S and 28S rRNA’s along with around 45 different proteins constitute the large subunit of the ribosome. The “S” after the name of each of the rRNA types refers to Svedberg units which are the units used to measure sediment in the ultracentrifuge. The numbers in the rRNA names are not proportional but they reflect the size of the rRNA molecule [11].

2.3.4 Small Nuclear RNA (snRNA)

Small Nuclear RNA (snRNA) is a short RNA sequence generally in the range of 60 to 300 nucleotides long [12]. The snRNA has several functions but is generally connected to the processing of other types of RNAs. One example of how snRNA is connected to the processing of other RNA’s is that snRNAs are connected with polyadenylation or with the terminating of the mRNA at the Poly-A tail on the 3’ end of the RNA [12].
Another example is snRNA’s are connected to the spliceosome which functions to exclude the interons and connect the exons in the final mRNA [12].

2.3.5 Double Stranded RNA (dsRNA)

Double Stranded RNA (dsRNA) is a form of RNA which has two connected strands. The two strands are complementary, similar to what is found in DNA [7]. Double stranded RNA acts as a mechanism to initiate the process of RNA interference (RNAi). RNA interference occurs when small subsequences in the dsRNA, which are homologous to sequences in a gene, interfere with the expression of the gene. From what is currently known the dsRNA is cut into its individual strands by an enzyme known as a dicer [7]. The individual strands then bond with other RNA sequences which are complementary to the single dsRNA strand which renders the RNA useless. Double stranded RNA is also known to be a component in the formation of small interfering RNA (siRNA) as well as a component of the genetic material in some viruses [7].

2.3.6 Small Interfering RNA (siRNA)

Small Interfering RNA (siRNA) are short RNA sequences in the range of 20 to 25 nucleotides long [7]. Small interfering RNA is a double stranded RNA sequence which is a component of RNA interference (RNAi) (See Double Stranded RNA).

2.3.7 Micro RNA (miRNA)

Micro RNA (miRNA) are short single stranded RNA sequences in the range of 20 to 25 nucleotides long [7]. Micro RNA functions as a mechanism to regulate gene expression. The miRNA is complementary to a portion of an mRNA which it will bond to in order to regulate gene expression. When the miRNA bonds to the mRNA it inhibits that portion of the RNA from being translated into a protein [7].
2.3.8 Other RNA Types

Although the RNA types described so far tend to be the most well known and understood types of RNA, there are many other types of RNA such as Guide RNA (gRNA), Efference RNA (eRNA), Signal Recognition Partical RNA (srpRNA), Phages RNA (pRNA), Transfer Messenger RNA (tmRNA) and many others [12].

2.4 RNA World Hypothesis

The RNA World Hypothesis is a theory that suggests RNA is the precursor that made it possible for DNA to be created. This hypothesis was originally suggested by Carl Woese in his 1967 book titled *The Genetic Code* [9] [10]. The RNA world hypothesis arises from a “chicken-and-egg” type problem that was present in the primordial earth. How would it have been possible for DNA to have been created when it takes proteins to replicate and transcribe DNA? On the other hand, how would it have been possible to have the proteins when DNA is the mechanism that carries the genetic code which is needed to synthesize the proteins? Each is required to synthesize the other. RNA seemed to be the perfect answer to this problem. Since RNA can store genetic information and since its single stranded structure allows it to form many tertiary structures similar to proteins it might have been possible that RNA did the job of both DNA and protein. In the early 1980’s came the discovery that made this hypothesis seem very plausible. It was discovered that there are some RNA’s which are self replicating. This prompted Walter Gilbert to write in 1986, “One can contemplate an RNA World” [9], hence, the theory was coined the “RNA World Hypothesis”.

2.5 Implications of World Hypothesis on Structure Prediction

The implications of the RNA World Hypothesis on secondary RNA structure prediction are that all living organisms could have evolved into their present form from a common type of RNA. This could mean that there are many similarities in the RNA of present day organisms. Even if the sequence of the RNA in current organisms has varied a sufficient amount so as that is would seem they are not related, it might be possible that the RNA actually has a similar secondary or tertiary structure which effectively have the same
function. This is why researchers are very interested in finding computer based algorithms that can accurately predict the secondary structure of RNA. If a database can be created that describes the secondary structure of RNA sequences, then the secondary structure could be used in addition to just using the sequence when performing tasks such as trying to determine if several RNA sequences are homologous.

Since the visualization is intended to be a teaching tool and is not focused on dealing with extremely long RNA sequences, one can determine which RNA classes the program will be useful to work with. The visualization is intended to deal with sequences of 100 base pairs or less. By analyzing the typical lengths of each class of RNA one can determine that the visualization will be able to handle tRNA’s, some of the shorter snRNA’s, siRNA’s and miRNA’s. Of course the program could always visualize short subsequences of longer RNA’s such as mRNA. Now that it has been shown which type of RNA the visualization will be able to handle, the next chapter will give a formal description of the secondary structure prediction problem and will discuss the types of algorithms which have been discovered which attempt to solve the secondary structure prediction problem.
Chapter 3

Secondary RNA Structure

This chapter gives an overview of the secondary RNA structure prediction problem. It starts out by formally describing the secondary structure prediction problem as suggested by Zuker [14]. It describes the different types of algorithms and the classifications that these algorithms fall into. The chapter explains the issues and deficiencies of the algorithms. There is a section which describes the structural elements that the secondary structure is made of. Finally, the chapter describes all ways in which the secondary structure has been visualized.

3.1 Secondary Structure Formal Description

RNA secondary structure refers to the two dimensional shape that RNA would physically fold into under natural conditions. As RNA folds back on itself it forms hydrogen bonds at complimentary base pair locations. These hydrogen bonds formed by the pairing of complementary Watson-Crick bases as well as the weaker wobble pair G-U are described as canonical base pairs [14].

Formally, the secondary structure of RNA can be described as suggested by Zuker [14] as follows: An RNA sequence is represented by R as \( R = r_1, r_2, r_3, \ldots, r_n \), where \( r_i \) is called the \( i^{th} \) nucleotide. Each \( r_i \) belongs to the set \{A, C, G, U\}. A secondary structure, or folding, on \( R \) is a set \( S \) of ordered pairs, written as \( i,j, 1 \leq i \leq j \leq n \) satisfying:
1. \( j - i > 4 \)
2. If \( i,j \) and \( i',j' \) are 2 base pairs, (assuming without loss in generality that \( i <= i' \)), then either:
   a. \( i = i' \) and \( j = j' \) (they are the same base pair),
   b. \( i < j < i' < j' \) (\( i,j \) precedes \( i', j' \)), or
   c. \( i < i' < j' < j \) (\( i,j \) includes \( i',j' \)).

Item 2c above disallows pseudoknots which occur when two base pairs, \( i,j \) and \( i',j' \) satisfy the condition \( i < i' < j < j' [14] \). The formal description does not account for pseudoknots for several reasons. First, the algorithms which try to predict the secondary structure through energy minimization are not able to handle pseudoknots. Energy minimization algorithms can not handle pseudoknots because it is beyond current scientific understanding how to assign energy values to the structures created by pseudoknots. Secondly, pseudoknots are not considered because the dynamic programming based algorithms are not able to handle the loop structures created by pseudoknots.

### 3.2 Taxonomy of Algorithms

There are several methods used in the laboratory in order to determine the secondary and tertiary structure of RNA. These methods include x-ray crystallography and nuclear magnetic resonance spectroscopy [15]. The problem with these methods is that they are very expensive and time consuming to produce the secondary structure results. It would be tremendously preferable if a computer based algorithm could be created which would accurately calculate the secondary structure in mere seconds. This has been the focus of many researches for the past several decades. There are many types of algorithms which have been devised which endeavor to fulfill this goal but the algorithms can be categorized into two main types, deterministic and stochastic [13].

#### 3.2.1 Deterministic

There are a whole host of algorithms which fit into the classification of deterministic. The one fact that is common to all algorithms of this type is that the correct next step in the algorithm only depends on the current state of the algorithm. There is no point in the
algorithm at which there are several next steps that could happen with some unknown way to choose between them. Algorithms that fall into this category are

**Minimum Free Energy** such as Zuker’s algorithm [14], **Kinetic Folding** such as Martinez [16], **5’ – 3’ Folding** [13], **Partition Function** [13], and **Maximum Matching** such as Nussinov [8]. Kinetic Folding and 5’ – 3’ Folding are able to determine pseudoknots.

### 3.2.2 Stochastic

The common theme between all the stochastic algorithms is that they are based on probabilities. One such example is based on a special Monte Carlo procedure known as **Simulated Annealing** [17]. The Simulated Annealing algorithm is able to assign probabilities to both the opening and closing of single base pairs. This allows the algorithm to account for a wide range of secondary structures.

### 3.3 Issues with Algorithms

The main issue with all the computer-based algorithms is that they are no more than first order approximations of the actual secondary structure which would occur in nature. The determination of secondary structure is by no means an exact science. Furthermore, the structure which the algorithm calculates to be the optimal structure might not be the most biologically correct. Many of the algorithms allow for suboptimal structures to be calculated as well taking into account this anomaly.

### 3.4 Secondary Structure Elements

All RNA secondary structures are composed of several basic structural shapes which occur naturally when RNA folds back on itself. These basic structures are usually depicted as two-dimensional pictures which indicate the positions where base pair bonds occur. The regions where base pairs stack on top of each other and form into helical regions are called stems or stacking pairs (See Figure 3.1 Stacking Pairs). Sections of RNA which occur at either the start or end of the sequence that are not part of any structure are called unstructured single strands or free ends (See Figure 3.1 Joint and
Free Ends. All other structures formed by RNA are variations of loop structures which occur when a section of RNA loops around on itself and is bounded by base pairs. Hairpin loops are loops which occur at the end of a stem and consist of three or more base pairs because a three base pair loop is the smallest biologically feasible loop (See Figure 3.1 Hairpin Loop). Hairpin loops are sometime referred to as stem loops but the hairpin name stuck because the loops resemble a hairpin. A bulge which occurs in a single strand of a stem is referred to as a bulge loop (See Figure 3.1 Bulge). When bulges occur on both strands of a stem an interior loop is formed (See Figure 3.1 Interior Loop). When loops occur which have three of more branches (stems) extending out of the loop then a multi-branched loop of formed (See Figure 3.1 Multiple Loop). The last type of loop is referred to as a pseudoknot. A pseudoknot occurs when bases inside a loop are bonded with bases in another section of the RNA which is outside the bounding stem of the loop. Pseudoknots occur relatively infrequently as compared to the other RNA structure elements [1][13].

Figure 3.1. Secondary Structure Elements [13]
3.5 Secondary Structure Visualizations

The human mind is not capable of comprehending problems which have large amounts of data in strictly numerical or character formats. The RNA secondary structure problem is one such problem where any given RNA sequence could be hundreds of bases in length and that fact makes it necessary to devise methods to help the human mind comprehend the data. In order to increase the comprehension of the RNA secondary structure problem, many types of visualizations have been devised which represent the data in alternate formats so that the largest amount of intuitive understanding can be gained from the depictions.

3.5.1 String Representation

The simplest form of representing an RNA sequence is strictly by the bases which make up the sequence (See Figure 3.2). A string is formed where the characters in the string represent the four RNA bases. The characters in the string are positioned as to represent the ordering of the bases in the RNA sequence.

```
AACGGAACCAACAUGGAUCAUGCUUCGGCCCUGGUCGCG
```

Figure 3.2. RNA in String Representation

3.5.2 Bracketed Representation

The bracketed representation is one of the simplest ways to visualize the secondary structure of RNA. This representation is sometimes referred to as bracket dot notation because of the brackets and dots used in the representation [31]. The bracketed representation consists of using the string representation of RNA on one line and then on a line directly below the string representation a sequence of open or close brackets and dots are used to represent nucleotides which are bonded as pairs (See Figure 3.3). If a bond exists between nucleotides at position i and position j then an open bracket ‘(‘ is used at position i and a close bracket ‘)’ is used at position j to represent the bond. If no bond exists then a dot is placed at the nucleotide position to represent that no bond exists.
For every open bracket there must be a corresponding closing bracket to represent the pairs.

AACGGAACCAACAUGGAUUAUGCUUCGGCCUCGGUCGCG
.(.(.(..))((..))((.))((() )))(( () ))((() ))(( () ))((()) )

Figure 3.3. Secondary Structure in Bracketed Representation

3.5.3 Linked Graph Representation

For the linked graph representation the nucleotide bases are drawn on a line at equidistant intervals. Arcs are then drawn which connect base pairs which have bonds [18]. This representation makes it very easy to determine if pseudoknots exist by examining the graph for arcs that cross one another. If any arcs cross then a pseudoknot exists.

Figure 3.4. Linked Graph Representation of tRNA\textsuperscript{phe} [18]

3.5.4 Planar Graph Representation

The planar graph representation is the most intuitive representation. This is the closest approximation to what the actual two dimensional structures would look under natural conditions. The planar graph is merely a topology therefore structures drawn in the graph which visually seem close together may actually be distant in reality.
3.5.5 Tree Representation

The tree representation of RNA secondary structure not only produce a visual display of the secondary structure but also allows mathematical properties of tree theory to be used in the process of examining the tree. The tree graph is actually a forest where paired bases correspond to internal nodes. The labels on the internal nodes are the bases which are paired. The leaf nodes correspond to the unpaired bases whose label is a single base [20]. One useful operation that can be performed on two forest graphs is creating a mathematical value which represents how similar two forests are.

Figure 3.5. Planar Graph Representation of tRNA<sub>phe</sub> [19]
3.5.6 Circular Representation

The circular representation of secondary RNA structure can be thought of as an extension to the Linked Graph representation where the ends of the string have been wrapped around into a circle [8]. The circular representation uses a circle and then places the nucleotide bases at equidistant intervals around the circle. Chords are then drawn on the interior of the circle between base pairs that form a bond. This representation also allows for easy visual detection of pseudoknots by examining the graph for any chords that intersect. If any chords intersect then a pseudoknot is present. This representation was first devised by Ruth Nussinov [8].
3.5.7 Matrix Representation

The matrix representation is a visual representation of the dynamic programming matrix which is created in algorithms like Nussinov’s [1]. The nucleotide bases are listed horizontally and vertically along the edges of the matrix and then the scores from the algorithm fill the interior of the matrix. Some representations show the trace back path through the matrix as a color scale path.

![Figure 3.8. Matrix Representation (Dynamic Programming Matrix)](image)

3.5.8 Dot Plot Representation

A dot plot is a graph which is setup as a triangular array. The RNA sequence it places along the to axis of the triangular array and a dot is placed in the graph corresponding to pair i,j at the ith row and jth column. Dot plots are generally used for comparative analysis because many dot plots can be superimposed on the same graph where the plot can be easily compared [14].
3.5.9 Mountain Plot Representation

The roots of the mountain plot were originally devised in a paper written by Paulien Hogeweg in 1984 [22]. Later, another related paper written by Danielle Konings in 1989 furthered the mountain plot “by identifying ‘(’, ‘)’, and ‘.’, with “up”, “down”, and “horizontal”, respectively” [18]. The mountain plot is represented by three elements: peaks, plateaus, and valleys. The peaks correspond to hairpin loops, the plateaus correspond to unpaired bases and the valleys indicate either unpaired sequences between the branches of a multi-branch loop or unpaired sequences which join components of the secondary structure [18].
Now that the RNA secondary structure prediction problem has been formally described, the next chapter will focus specifically on the Nussinov base pair maximization algorithm. There are actually two versions of the Nussinov algorithm, the standard version and the stochastic context free grammar version. The next chapter will detail both versions.
Chapter 4

Nussinov Folding Algorithm

This chapter gives a detailed description of the two versions of the Nussinov algorithm which will be implemented in the visualization. First, the standard Nussinov algorithm is detailed, a formal algorithm definition is described and then the fill and traceback stages are explained. A list of enhancements to the original algorithm are suggested which will be incorporated into the visualization. Second, the stochastic context free grammar version of the Nussinov algorithm is detailed. A formal algorithm definition is described and the fill and traceback stages are explained just as in the standard version.

4.1 Nussinov Algorithm

The Nussinov algorithm is a base pair maximization algorithm. In other words, the algorithm tries to calculate the secondary structure which has the maximum possible number of base pairs. The algorithm was first conceived by Ruth Nussinov in a paper that she wrote in 1978 [8]. The algorithm makes use of a dynamic programming algorithm to recursively find the optimal structure for small subsequences and then recursively works its way out to larger and larger subsequences. At any point during the recursive calculation there are only four possible ways to obtain the optimal structure from the optimal structures of the smaller subsequences (See Figure 4.1) [1]. The four ways to obtain the optimal structure as described in the Durbin book are [1]:
1. add unpaired position $i$ onto best structure for subsequence $i + 1, j$;
2. add unpaired position $j$ onto best structure for subsequence $i, j - 1$;
3. add $i, j$ pair onto best structure found for subsequence $i + 1, j - 1$;
4. combine two optimal substructures $i, k$ and $k + 1, j$.

When two substructures are combined as in item 4 it is referred to as a bifurcation.

![Nussinov Structures](image)

**Figure 4.1. Nussinov Structures [1]**

### 4.1.1 Formal Algorithm Definition

Given a sequence $x$ of length $L$ with symbols $x_1, \ldots, x_L$. Let $\delta(i, j) = 1$ if $x_i$ and $x_j$ are a complementary base pair; else $\delta(i, j) = 0$. Then recursively calculate scores $\delta(i, j)$ which are the maximal number of base pairs that can be formed for subsequences $x_i, \ldots, x_j$ [1].

### 4.1.2 Fill Stage

The fill stage of the Nussinov algorithm works by first creating a rectangular array for the dynamic programming matrix. The matrix is initialized so that both the diagonal and the lower diagonal are set to zero. The algorithm then proceeds along the diagonals calculating the score as described below [1].

**Initialization:**

$$
\begin{align*}
\gamma(i, i - 1) &= 0 & \text{for } i = 2 \text{ to } L; \\
\gamma(i, i) &= 0 & \text{for } i = 1 \text{ to } L.
\end{align*}
$$

**Recursion:** starting with all subsequences of length 2, to length $L$:

$$
\gamma(i, j) = \max \left\{ \gamma(i + 1, j), \gamma(i, j - 1), \gamma(i + 1, j - 1) + \delta(i, j), \max_{i < k < j} [\gamma(i, k) + \gamma(k + 1, j)] \right\}
$$

The fill stage is $O(L^2)$ in memory and $O(L^3)$ in time [1].
4.1.3 Traceback Stage

The traceback stage makes use of a stack construct in order to handle the case in which there is a bifurcation. When a bifurcation happens two optimal substructures are joined together and the stack is used to travel down one structure and then pop back so it can travel down the second structure. The traceback stage works by first pushing the element in array position (1, L) onto the stack. This element represents the maximum number of base pairs in the optimally aligned structure. It is possible that there are other positions in the array with the same number of base pairs as the optimally aligned structure. The algorithm then traces back down the array as described below [1].

**Initialization:** Push (1, L) onto stack.

**Recursion:** Repeat until stack is empty:

- pop (i, j).
- if i >= j continue;
  - else if γ(i + 1, j) = γ(i, j) push (i + 1, j);
  - else if γ(i, j - 1) = γ(i, j) push (i, j - 1);
  - else if γ(i + 1, j - 1) + δij = γ(i, j):
    - record i, j base pair.
    - push (i + 1, j - 1).
  - else for k = i + 1 to j - 1: if γ(i, k) + γ(k + 1, j) = γ(i, j):
    - push (k + 1, j).
    - push (i, k).
    - break.

The traceback stage is linear in memory and time [1].

4.1.4 Enhancements to Algorithm

There are several enhancements which can be made to the Nussinov algorithm. First, the formal definition of the algorithm only allows for Watson-Crick pairs. Second, both A-U and C-G pairs are given the same value in the scoring matrix. It might be desirable to allow C-G pairs to have a higher value since the C-G bonds are slightly stronger because of three hydrogen bonds instead of two for A-U. There is also a case in nature where G-U bonds occur. It might be desirable to allow these so called wobble pairs to occur in the algorithm. With just a minor modification to the scoring function both of the cases can be accounted for.
Another enhancement to the algorithm would be to put a limit on how small a hairpin loop can be. There is a biological limit on the minimum length of hairpin loop to 3 bases. The algorithm allows for hairpin loops of lengths less then three, so another minor modification could be made to limit the hairpin loops to three bases.

### 4.2 Nussinov SCFG Version

The stochastic context free grammar version of the Nussinov algorithm is a probabilistic algorithm. A grammar is first setup which has all the production rules for the grammar which includes a single non-terminal (See Figure 4.1). Probabilities are then assigned to each production rule based on any one of several methods. Probabilities could be assigned by counting state transitions in known RNA and then converting the counts to probabilities [1]. Another method is to use an Expectation Minimization algorithm to calculate the probabilities [32]. Finally, the probabilities could be gained from subjective estimation [1].

![Figure 4.1. Production Rules [1]](image)

#### 4.2.1 Formal Algorithm Definition

The SCFG version of the Nussinov algorithm actually uses a slightly modified CYK (Cocke – Younger – Kasami) parsing algorithm. Typically CYK parsing algorithms are used on grammars that in Chomsky normal form but by slightly modifying the CYK algorithm a more efficient approach can be used [1].

The algorithm starts out by letting the probability parameters for the grammar’s productions rules be denoted by \( p(aS) \), \( p(cS) \), \( p(gS) \), \( p(uS) \), \( p(Sa) \), \( p(Sc) \), \( p(Sg) \), \( p(Su) \), \( p(aSu) \), \( p(cSg) \), \( p(gSc) \), \( p(uSa) \), and \( p(SS) \). Next the fill stage begins as follows [1]:

\[
egin{align*}
S &\rightarrow aS | cS | gS | uS & \text{(i unpaired)} \\
S &\rightarrow Sa | Sc | Sg | Su & \text{(j unpaired)} \\
S &\rightarrow aSu | cSg | gSc | uSa & \text{(i, j pair)} \\
S &\rightarrow SS & \text{(bifurcation)}
\end{align*}
\]
4.2.2 Fill Stage

The SCFG version of the Nussinov algorithm starts out similar to the standard Nussinov algorithm by creating a rectangular array for the dynamic programming matrix. The diagonal of the array is initialized to be negative infinity. The lower diagonal is initialized to the maximum of the probability values for the character at the given position through either the i-unpaired or j-unpaired rule. Then the recursion phase begins and the algorithm travels down the diagonals calculating the probabilities based on the algorithm.

Initialization:
\[
\gamma(i, i - 1) = -\infty \text{ for } i = 2 \text{ to } L;
\]
\[
\gamma(i, i) = \max \left\{ \frac{\log p(x_iS)}{\log p(Sx_i)} \right\} \text{ For } i = 1 \text{ to } L
\]

Recursion: for \(i = 1 \text{ to } L - 1, j = i + 1 \text{ to } L\):
\[
\gamma(i, j) = \max \left\{ \begin{array}{l}
\gamma(i + 1, j) + \log p(x_iS); \\
\gamma(i, j - 1) + \log p(Sx_j); \\
\gamma(i + 1, j - 1) + \log p(x_iSx_j); \\
\max_{i<k<j} \gamma(i, k) + \gamma(k + 1, j) + \log p(SS)
\end{array} \right\}
\]

4.2.3 Traceback Stage

The traceback stage of the SCFG version of the Nussinov algorithm works almost exactly like the standard version of the algorithm. The traceback starts at the array position \((1, L)\) pushes that onto the stack and then recursively traceback through the array until the stack is empty. The main difference between this traceback and the standard traceback is that instead of using the scoring matrix during traceback the algorithm instead uses the probability values.

Now that the Nussinov algorithm has been detailed as well as all known methods to visualize the RNA secondary structure, all these components can be joined together to describe the visualization interface. The next chapter will show how the Nussinov algorithm is used to calculate the secondary structure and display the secondary structure in several ways simultaneously. The next chapter will also detail the way in which the
visualization will tie information from two different representations together through the use of three dimensional lines linking significant information.
Chapter 5

The Visualization Interface

This chapter gives a high level description of the interface used by the visualization. It starts out by describing which data representations will be used by the visualization. It then shows how the visualization uses a three dimensional stage model to draw the data representations onto facets of the stage. Some of the visualizations unique aspects are described in this chapter. The chapter describes how the visualization can step through an RNA sequence and display intermediate values in the calculation. It is also shown how the visualization makes use of three dimensional lines in order to draw connections between pertinent information in the Nussinov calculation.

5.1 Visualization Interface

The interface to the visualization is composed of a three dimensional space which is used to display the results of the Nussinov algorithm. The main component in the three dimensional space is an object referred to as the stage. The stage is actually a three dimensional model which resembles a theatrical stage. The stage is composed of several angular faces which are referred to as facets (See Figure 5.1). Each facet contains a unique representation of the Nussinov algorithm calculation. For example the back facet, referred to as the planar graph facet, contains the two dimensional representation of the secondary structure of the RNA sequence. The facet on the back right, referred to as the circular facet, contains the circular graph representation of the secondary structure.
back left facet, referred to as the bracketed facet, contains the bracketed representation. Finally, the bottom facet, referred to as the matrix facet, contains the matrix representation of the dynamic programming matrix. The stage has been constructed in such a way as to convey the maximum amount of information while at the same time maintaining a clean interface which is not cluttered with too much information.

There are two unique aspects to this interface. First, the interface makes use of three dimensional space in order to display multiple two dimensional representations of data. Most other secondary structure prediction programs only allow the data to be represented one way at a time. By using three dimensional space and by displaying more information it is intended that the user will be able to have a more intuitive understanding and will be able to draw more connections and infer information from the visualization. The second unique aspect of the visualization is that it allows the user to step through the RNA sequence as the secondary structure is being calculated and displayed.

Figure 5.1. The Stage Object
5.2 Stepping Through the Algorithm

The visualization program allows the user to enter any given RNA sequence and then step through the sequence. In order to facilitate the stepping features, the visualization contains four buttons located on the main interface (See Figure 5.1). These buttons are labeled Step First, Step – 1, Step + 1 and Step Last. The stepping feature works as follows: if an RNA sequence consisting of 25 bases has been entered into the visualization, then after entering the sequence the visualization will automatically step to the last position to show the complete secondary structure of the sequence. The user can then click on any of the four stepping buttons. By clicking the Step First button the visualization will move to the first base in the sequence and all the facets will be recalculated based on the first nucleotide base in the sequence. If the Step + 1 button were now clicked then the next base in the sequence would be included and all the facets would again be recalculated. The user could continue to step through the whole sequence in this manner.

5.3 Three Dimensional Lines

Another unique feature of the visualization is that it takes advantage of the three dimensional space in order to draw connections between the stage facets. The visualization will draw three dimensional lines between the planar graph facet and the matrix facet in order to show where each nucleotide in the planar graph originated from in the dynamic programming matrix. In addition, the color of the three dimensional lines corresponds to the Nussinov structure type that produced the nucleotide in the algorithm. There are four color coded boxes in front of the matrix graph facet (See Figure 5.1). The purple box represents the i-unpaired structure, the red box represents the j-unpaired structure, the blue box represents the i,j-paired structure and the green box represents a bifurcation. The lines are color coded the same color as the boxes to show which structure was used to calculate the current position.
This chapter gave an overview of the visualization interface so now the next chapter will detail how the stage object is constructed. The next chapter will detail how the Microsoft DirectX API is used to create a three dimensional visualization space.
Chapter 6

Three Dimensional Stage

This chapter describes how the Microsoft DirectX API has been used to create the three dimensional stage object of the visualization. The chapter starts by giving a brief overview of the DirectX API, and then describes how some of the DirectX constructs are used to create the stage object. The chapter details the triangle strips that were created to complete the stage object. It also details the internal dimensions of the stage object. Next, it is shown how an index buffer is used in order to increase the performance of the drawing functions. Finally, the chapter details how the DirectX texturing mechanism is used to draw images onto the stage object.

6.1 Three Dimensional Stage

The three dimensional stage is the structure that the visualization uses as a canvas to paint its multiple data representations upon. The stage is actually a three dimensional model composed of polygons that with the help of Microsoft’s DirectX 9.0 software development kit, are rendered to the screen. The DirectX SDK is a collection of low level application programming interfaces that are used for creating high performance 2D and 3D graphics. The DirectX SDK consists of three main API interfaces, Direct3D, DirectInput, and DirectSound [27]. Since this visualization is both highly graphical and the visualization is utilizing a three dimensional view space, both of which are cpu
intensive, performance was a critical consideration. Based on this consideration the Direct3D interface offered by DirectX was a perfect fit for the visualization.

6.2 Direct3D And The Three Dimensional Stage

Direct3D is the application programming interface used for displaying three dimensional objects. The main advantage of Direct3D is that it utilizes whichever graphics accelerator device is installed on the machine without requiring the programmer to write any code specific to the particular brand of graphics accelerator. The Direct3D API provides a set of functions which all graphics accelerator manufacturers support.

In order to create a three dimensional stage object using Direct3D, the API provides several types of primitive objects which can be utilized. The Direct3D primitives include a Point List, Line List, Line Strip, Triangle List, Triangle Strip, and a Triangle Fan. In addition each primitive consists of one or more vertices which define the endpoints of the primitive type. The Direct3D API provides a flexible mechanism for defining the vertices depending on the requirements of the application. For example, if the application only requires the three dimensional position of a vertex without color, lighting or texturing then you could define a vertex as follows:

```c
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ)
struct CUSTOMVERTEX
{
    float x,y,z; // Position of vertex in 3D space
};
```

**Listing 6.1. Custom Vertex (Position Only)**

Alternately, if a vertex requires the three dimensional position as well as color then the vertex could be defined as follows:

```c
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_DIFFUSE)
struct CUSTOMVERTEX
{
    float x,y,z; // Position of vertex in 3D space
    DWORD color; // Diffuse color of vertex
};
```

**Listing 6.2. Custom Vertex (Position and Color)**
The types of vertices required by the visualization program are first vertices used for the stage object and second vertices used for the lines which are drawn to connect different facets of the stage. The stage vertices need to have a three dimensional position as well as normal vectors for lighting calculations as well as texture coordinates so that the visualization output can be drawn onto the stage. A stage vertex is defined as follows:

```c
#define STAGE_D3DFVF_CUSTOMVERTEX(D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_TEX1)
struct STAGE_CUSTOMVERTEX
{
    float x, y, z;    //Position of vertex in 3D space
    float nx, ny, nz; //Normal vector for lighting calculations
    float tu, tv;    //Texture coordinates
};
```

### Listing 6.3. Custom Vertex used by stage object

The vertices for the three dimensional lines need to have a three dimensional position as well as normal vectors for lighting as well as a color component. A line vertex is defined as follows:

```c
#define LINE_D3DFVF_CUSTOMVERTEX(D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_DIFFUSE)
struct LINE_CUSTOMVERTEX
{
    float x, y, z;    // Position of vertex in 3D space
    float nx, ny, nz; // Normal vector for lighting calculations
    DWORD color;    // Diffuse color of vertex
};
```

### Listing 6.4. Custom Vertex used by visualization lines

Once the vertices have been defined then the vertices can be used to create the primitive types used by Direct3D. The simplest of the primitive types is the point list. In order to draw a set of points using Direct3D one would first specify an array of vertices with the required vertex attributes as described above. Once the array of vertices was created then the Direct3D API function used from drawing primitive types would be called. The code for drawing this point list would be as follows [27]:

```c
struct CUSTOMVERTEX
{
    float x, y, z;
};

CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},  // (0, 5, 0)  (10, 5, 0)  (20, 5, 0)
    {-5.0, -5.0, 0.0},  // (0, 5, 0)  (10, 5, 0)  (20, 5, 0)
    {5.0, -5.0, 0.0},   // (0, 5, 0)  (10, 5, 0)  (20, 5, 0)
};
```
{ 0.0, 5.0, 0.0},  
{ 5.0, -5.0, 0.0},  
{10.0, 5.0, 0.0},  
{15.0, -5.0, 0.0},  
{20.0, 5.0, 0.0} 
);

d3dDevice->DrawPrimitive(D3DPT_POINTLIST, 0, 6);  

Listing 6.5. Point List

The code for drawing a line list would be as follows [27]:

```
struct CUSTOMVERTEX  
{  
  float x,y,z;  
};

CUSTOMVERTEX Vertices[] =  
{
  { -5.0, -5.0, 0.0},
  {  0.0,  5.0, 0.0},
  {  5.0, -5.0, 0.0},
  { 10.0,  5.0, 0.0},
  { 15.0, -5.0, 0.0},
  { 20.0,  5.0, 0.0}
};

d3dDevice->DrawPrimitive(D3DPT_LINELIST, 0, 3);  

Listing 6.6. Line List

The code for drawing a line strip would be as follows [27]:

```
struct CUSTOMVERTEX  
{  
  float x,y,z;  
};

CUSTOMVERTEX Vertices[] =  
{
  { -5.0, -5.0, 0.0},
  {  0.0,  5.0, 0.0},
  {  5.0, -5.0, 0.0},
  { 10.0,  5.0, 0.0},
  { 15.0, -5.0, 0.0},
  { 20.0,  5.0, 0.0}
};

d3dDevice->DrawPrimitive(D3DPT_LINESTRIP, 0, 5);  

Listing 6.7. Line Strip

The code for drawing a triangle list would be as follows [27]:
```c
struct CUSTOMVERTEX
{
    float x, y, z;
};

CUSTOMVERTEX Vertices[] =
{
    { -5.0, -5.0, 0.0 },
    {  0.0,  5.0, 0.0 },
    {  5.0, -5.0, 0.0 },
    {10.0,  5.0, 0.0 },
    {15.0, -5.0, 0.0 },
    {20.0,  5.0, 0.0 }
};
d3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2);

Listing 6.8. Triangle List

The code for drawing a triangle strip would be as follows [27]:

```c
struct CUSTOMVERTEX
{
    float x, y, z;
};

CUSTOMVERTEX Vertices[] =
{
    { -5.0, -5.0, 0.0 },
    {  0.0,  5.0, 0.0 },
    {  5.0, -5.0, 0.0 },
    {10.0,  5.0, 0.0 },
    {15.0, -5.0, 0.0 },
    {20.0,  5.0, 0.0 }
};
d3dDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 4);

Listing 6.9. Triangle Strip

The code for drawing a triangle fan would be as follows [27]:

```c
struct CUSTOMVERTEX
{
    float x, y, z;
};

CUSTOMVERTEX Vertices[] =
{
    {  0.0,  0.0,  0.0 },
    { -5.0,  5.0,  0.0 },
    { -3.0,  7.0,  0.0 },
    {  0.0, 10.0,  0.0 },
    {  3.0,  7.0,  0.0 },
    {  5.0,  5.0,  0.0 },
};
```

42
6.2.1 Stage Triangle Strips

The three dimension stage object is composed of triangle strips as shown above. In order to create the stage object, eight triangle strips were used. The first triangle strip defines the front panel where the progress bar will be placed. The second triangle strip defines the region of the stage where the four Nussinov structure types will be placed. The third and fourth triangle strips are some unused area but are necessary to connect the structure together. The fifth and seventh triangle strips are also unused sections. The sixth triangle strip is the panel where the matrix graph will be placed. The eighth triangle strip has sections for the bracketed graph, planar graph and circular graph panels (See Figure 6.1).

![Figure 6.1. Stage model showing triangle strips](image)

6.2.2 Stage Dimensions

Now that the triangle strips have been defined the next step is to create dimensions for the stage object and to define the vertices positions. The stage object will have its own coordinate system so the dimensions are made relative to the size of the facets needed in the stage object. The facets which will contain the matrix graph, bracketed graph, planar
graph and circular graph will all be the same size so define those facets as 10 units by 10 units. The four facets which will contain the four Nussinov structure types will be the same with as one of the 10 by 10 facets so each of those facets has to be 10 / 4 = 2.5 units. The progress bar facet is 22 units wide by 2 units high. When completely put together the dimensions turn out as follows: (See Figure 6.2)

![Figure 6.2. Stage model show dimensions and vertices]

6.2.3 Direct3D Index Buffer

Direct3D offers a structure called an index buffer which helps to increase the performance of drawing the model by combining vertices which overlap. When the stage object is put together and the triangle strips are arranged next to each other there are some vertices with the same coordinates. For example vertex 1 in triangle strip 1 is at the same coordinate as vertex 14 in triangle strip 3. This in fact happens many times in the model so the index buffer is used as an integer offset into the vertex buffer. This buffered offset increases the performance of the drawing functions [27]. The index buffer for the stage object would be setup as follows:

```c
//Index Buffer Values
WORD pIndices[] = { 0, 1, 2, 3, 2, 1, //Polygon 0
                   4, 5, 6, 7, 6, 5, //Polygon 1
                   8, 9,10,11,10, 9, //Polygon 2
                  12,13,14,15,14,13, //Polygon 3
                16,17,18,19,18,17, //Polygon 4
               20,21,22,23,22,21, //Polygon 5
```
Now that it has been shown how to create the three dimensional stage, the next operation would be to draw something onto the stage. The process of drawing a picture onto a polygon is called texturing the polygon. A texture is simply an image file such as a bitmap or jpeg. The picture is arranged in such a way that portions of the picture can be extracted from the image file and connected to vertices in the model. The texture image is a two dimensional picture where the x, y coordinates are represented by the letters u and v. The dimensions of the texture image are from 0 to 1 so the upper left corner of a texture image would be at coordinate (0, 0) and the lower right coordinate would be (1, 1) (See Figure 6.3).

This visualization program uses the Direct3D texturing mechanism to draw the data representations on the stage. The texture file used for the texture is dynamically calculated by the program and then mapped onto the stage object. Recall that each vertex in the stage object has several components. The vertex contains the x, y, z position of the vertex as well as the nx, ny, nz normal vector for lighting. The last component that the vertex contains in the tu, tv texture coordinates (See Listing 6.3). The mapping works by connecting the tu, tv texture coordinate of the vertex to the correct location within the texture image (See Figure 6.4).
This chapter detailed how the stage object was constructed and displayed through the help of Microsoft DirectX. The chapter ended showing how the texturing functionality of DirectX could be used to draw an image onto the stage object. The next chapter will explain how to create the image that is used as the texture. There are actually several calculations which make up the components of the image so each one will be detailed individually.
Chapter 7

Texture Construction

The secondary RNA structure visualization has five main facets. The five facets are the planar graph facet, the circular graph facet, the matrix graph facet, the bracketed graph facet and the progress bar facet. This chapter goes into detail on how each of these facets are constructed. Each of the facets share a common coordinate system so this chapter first details how the coordinate system is setup and then each facet has a brief overview before diving into the detail of the construction. Each facet is divided into a subsection which first details the input to the facet calculation, then the actual calculation is detailed and finally the final result is shown.

7.1 Coordinate System Setup

The coordinate system used by Microsoft Windows is a Cartesian coordinate system where the origin, position (0, 0), is positioned at the upper left hand corner of the screen. The X axis moves in the positive direction as you move to the right across the screen and the Y axis moves in the positive direction as you move down the screen (See Figure 7.1). All calls to Windows GDI drawing functions would use this coordinate system, with the origin at the upper left corner of the screen, as the basis for their drawing. This coordinate setup may not be ideal for all applications. For example you might want the origin to be at the center of the screen and you might want the Y axis to move in the positive direction as it moves up instead of moving down like it does. Luckily, this is just
the default setup for the coordinate system and it can easily be changed. Each of the main facets in this visualization are setup as 512 x 512 pixel bitmaps. The origin of the coordinate system is moved so that it is positioned at the exact center of the bitmap drawing area. The Y axis is also reconfigured so that the positive Y axis points up as would be expected in a standard Cartesian coordinate setup.

![Figure 7.1. Windows Coordinate System [23]](image)

The Win32 API provides a rich set of functions that can be used to manipulate the coordinate system. To change the position of the origin so that it is at the center of the 512 x 512 drawing area of the facet, a call to the `SetViewportOrgEx` function is used (See Listing 7.1). The SetViewportOrgEx function is used to reposition the origin of the coordinate system. The function takes four parameters, a handle to the current drawing device context, the x-coordinate origin position, the y-coordinate origin position and a pointer to the original origin position. After this function call the coordinate system is setup where the origin is position as desired but the Y axis is still upside down (See figure 7.2).

```c
SetViewportOrgEx(hdc, (int)(fTextureSize / 2.0f), (int)(fTextureSize / 2.0f), NULL);
```

**Listing 7.1. SetViewportOrgEx() Function (Windows API) [24]**
To change the direction of the Y axis and setup the coordinate system as desired several more Win32 API functions are need. First a call to the \texttt{SetMapMode} function (See \textbf{Listing 7.2}) is needed in order to specify the coordinate systems unit of measure. The \texttt{SetMapMode} function has two parameters. The first parameter is a handle to the current drawing surface device context and the second parameter is the desired mapping mode. The images drawn for each of the facets will use the \texttt{MM\_ISOTROPIC} mapping mode which means that one unit in the horizontal axis is equivalent to one unit in the vertical axis. The last step is to set the scaling factor between the window and the viewport. To set the horizontal and vertical extents of the window the \texttt{SetWindowExtEx} function (See \textbf{Listing 7.2}) is used. This function takes four parameters, a handle to the current drawing surface device context, the horizontal window extent, the vertical window extend and a pointer to the original window extents. To set the horizontal and vertical extents of the viewport the \texttt{SetViewportExtEx} function (See \textbf{Listing 7.2}) is used. This function works in exactly the same way as the \texttt{SetWindowExtEx} function except that it modifies the viewport extents. The interesting thing to notice is that the vertical viewport extent is set to -1. This has the affect of reversing the direction of the Y axis so that it points in the direction as desired (See \textbf{Figure 7.3}).

\begin{verbatim}
SetMapMode(hdc, MM\_ISOTROPIC);
SetWindowExtEx(hdc, 1, 1, NULL);
SetViewportExtEx(hdc, 1, -1, NULL);

int SetMapMode(
    HDC \textit{hdc},   // handle to device context
    int \textit{fnMapMode}   // new mapping mode
)

BOOL SetWindowExtEx(
    HDC \textit{hdc},   // handle to device context
    int \textit{xExt},   // horizontal window extent
    int \textit{yExt},   // vertical window extent
    \textit{pOriginalExt}   // pointer to original window extents
)
\end{verbatim}

\textbf{Figure 7.2. Coordinate With Modified Origin [23]}

\begin{center}
\includegraphics[width=0.5\textwidth]{figure7_2}
\end{center}
**Listing 7.2 Windows API Functions [24]**

```c
HDC hdc,      // handle to device context
int nXExtent, // new horizontal window extent
int nYExtent, // new vertical window extent
LPSIZE lpSize // original window extent
);    

BOOL SetViewportExtEx(
    HDC hdc,      // handle to device context
    int nXExtent, // new horizontal viewport extent
    int nYExtent, // new vertical viewport extent
    LPSIZE lpSize // original viewport extent
);    
```

**Figure 7.3. Final Coordinate System Configuration [23]**

### 7.2 Circular Graph Calculation

The circular facet as described in section 3.5.6, has several components to it. First, there are small blue tick marks on the outer edge of the circle that help identify the position of the RNA nucleotides in the circular representation. Every 10\textsuperscript{th} tick mark is slightly longer so that when long sequences are visualized there is an easy division that helps keep track of the current position in the overall sequence. Second, there are small red tick marks on the inner edge of the circle that represent nucleotides which do not have bonds. Lastly, there are red arcs on the inner surface of the circle that connected nucleotides that have a bond in the secondary structure. All of these elements will be drawn onto a 512 x 512 bitmap using windows GDI (graphical device interface) commands.
### 7.2.1 Circular Graph Calculation Input

The input to the circular facet calculation is the output from the Nussinov algorithm which is an array that represents the secondary structure in bracketed form. For example, given the RNA sequence GGGAAAUCC the output from the Nussinov algorithm would have been an array of nine elements in this form:

```
Position     0  1  2  3  4  5  6  7  8
Sequence     G  G  G  A  A  A  U  C  C  
Nussinov = [ -, (, (, -, -, (, ), ), ) ]
```

The circular facet actually uses a slight variation on the array from above. Instead of the array being a character array where each element in the array holds a dash or a bracket the actual input to the circular calculation uses an integer array where dashes are replaced with the positions array index and each bracket position holds the index to the matching bracket. For example, the bracket at array position 1 matches with the bracket at array position 8 so the new array will store 8 in array index 1. When all brackets are replaced with their matching index values then the following array is obtained:

```
Position     0  1  2  3  4  5  6  7  8
Nussinov = [ 0, 8, 7, 3, 4, 6, 5, 2, 1 ]
```

### 7.2.2 Bounding Box Dimensions

The circular representation is going to be drawn onto a bitmap surface that is 512 x 512 pixels. There will be a border around the bitmap which will take 5 pixels from each side of the bitmap. Additionally, there will be 20 pixels of padding on the interior of the border. Let `fTextureSize = 512.0f`, `fBorder = 5.0f` and `fPadding = 20.0f`. Therefore, the radius of the circle will be:

```
fRadius = (fTextureSize / 2.0f) – fBorder – fPadding;
fRadius = 231.0f
```
7.2.3 Calculating Tick Marks

In order to draw the tick marks around the circular representation the first thing that needs to be done is to determine where on the circle the tick marks should be drawn. Given that the input array contains a sequence of length \( x \), \( x = 9 \) in this example, the circle must be segmented into \( x \) equal pie pieces. Since the circle is 360 degrees then dividing 360 by \( x \) tells us the angle between each item on the circle. In the example sequence \( x = 9 \) so \( 360 / 9 = 40 \) degrees between each tick mark. There are two types of tick marks each of which have a slightly different length. The smaller tick marks will be drawn on the exterior of the circle and they will be aligned with nucleotide positions. The longer tick marks will be drawn every 10\(^{th}\) tick mark to give a better indication of the position in the sequence. Since both the smaller and larger tick marks are drawn on the exterior of the circle then two imaginary circles will be used to draw the tick marks. The first imaginary circle will have radius 5 pixels larger than the main circle and the second imaginary circle will have a radius that is 10 pixels larger. The smaller tick marks will be created by drawing lines from positions on the circle with radius + 5 to the center of the circle and the larger tick marks will be created by drawing lines from the circle with radius + 10 to the center of the circle. The lines are drawn using the windows GDI \texttt{MoveToEx} and \texttt{LineTo} functions (See Listing 7.3).
Listing 7.3 Windows API Functions [24]

Since we know the angle between each position on the circle and we know the coordinate of the center of the circle then we need to calculate the coordinate of the starting point of each line. The program loops through the length of the sequence, at each position the angle to the position is calculated and then the coordinate of the starting point of the line is calculated using the following formula: (See Figure 7.5)

\[
\text{Center of circle} = (px, py) \\
x = r \cos(\theta) + px \\
y = r \sin(\theta) + py
\]

The program checks to see if it is on an increment of 10 by checking if the modulus of the loop increment equals 0. If it is then the longer tick marks are drawn.

```c
// Draw the tick marks around the circle
hPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
hOldPen = (HPEN)SelectObject(hdc, hPen);
for(int x = 0; x < iStep; x++)
{
    if((x % 10) == 0) // Draw longer tick every 10th position
    {
        xPos = (int)(((fRadius + 10.0f) * cos(x * fAngle)) + fCenterX)
        yPos = (int)(((fRadius + 10.0f) * sin(x * fAngle)) + fCenterY)
        MoveToEx(hdc, xPos, yPos, NULL);
    }
    Else // Draw shorter tick
    {
        xPos = (int)(((fRadius + 5.0f) * cos(x * fAngle)) + fCenterX)
        yPos = (int)(((fRadius + 5.0f) * sin(x * fAngle)) + fCenterY)
        MoveToEx(hdc, xPos, yPos, NULL);
    }
    LineTo(hdc, (int)fCenterX, (int)fCenterY);
}
SelectObject(hdc, hOldPen);
DeleteObject(hPen);
```
Once all the lines have been drawn then the main circle will be drawn with the interior filled with the background color. This will both draw the circle and mask the interior of the lines.

![Circle with arcs](image)

**Figure 7.5. Circular Facet (Degrees between positions)**

### 7.2.4 Calculating Arcs

The first piece of information needed in the calculation used to draw the arcs onto the circle is to know which RNA nucleotides will be paired so that arcs can be drawn between these pairs. This information can be determined from the input array. The input array was setup so that each element in the array either holds the index of the matching pair or it holds its linear position if no matching pair exists.

<table>
<thead>
<tr>
<th>Position</th>
<th>0 1 2 3 4 5 6 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nussinov</td>
<td>[ 0, 8, 7, 3, 4, 6, 5, 2, 1 ]</td>
</tr>
</tbody>
</table>

To determine which positions need arcs to be drawn, a loop is setup on the array index. While looping through the array if the element at the current index equals the index then no arc is necessary. If the element at the current index contains a value greater than the current index then an arc is necessary. Notice in the array above that the position of the arc can be determined at two places. For example, at array index 1 the value is 8. That
means that an arc is necessary from position 1 to position 8. But this information can also be determined from array index 8 which has a value of 1. By convention the arcs will always be drawn when the trailing end of the arc has been encountered. This can be determined by checking to see if the value of the element at the current index is less than the index. If true then the trailing side of the arc has been encountered and the arc should be drawn. The arcs will be drawn with the windows GDI Arc function (See Listing 7.4).

```c
BOOL Arc(
    HDC hdc,       // handle to device context
    int nLeftRect,  // x-coord of rectangle's upper-left corner
    int nTopRect,   // y-coord of rectangle's upper-left corner
    int nRightRect, // x-coord of rectangle's lower-right corner
    int nBottomRect,// y-coord of rectangle's lower-right corner
    int nXStartArc, // x-coord of first radial ending point
    int nYStartArc, // y-coord of first radial ending point
    int nXEndArc,   // x-coord of second radial ending point
    int nYEndArc    // y-coord of second radial ending point
);
```

Listing 7.4. Windows API Arc() Function [24]

If it has been determined that an arc should be drawn from array index \( x \) to index \( y \) then the first thing to do is to calculate where the corresponding points should be on the circle. Let the angle between the positions on the circle be represented by \( \theta \). Let \( \theta_1 \) be the angle to the first point which can be calculated by \( \theta_1 = x \times \theta \). Let \( \theta_2 \) be the angle to the second point which can be calculated by \( \theta_2 = y \times \theta \). Now that the angles to the positions on the circle can be determined, next the coordinate positions of the start and end points of the arc need to be determined. Let \( P_1 \) be the start point and \( P_2 \) be the end point. Let \( P_0 \) be the center of the circle. Points \( P_1 \) and \( P_2 \) and be calculated as follows: (See Figure 7.6)

\[
P_1 = (r \cos(\theta_1) + P_0x, r \sin(\theta_1) + P_0y)
\]

\[
P_2 = (r \cos(\theta_2) + P_0x, r \sin(\theta_2) + P_0y)
\]
Figure 7.6. Circular Facet (Intermediate Calculation)

Now that the start and end points of the arc have been calculated to be \( P_1 \) and \( P_2 \), the next calculation is to determine the apex of the arc. The first step in this calculation is to find point \( P_3 \) and \( P_4 \) which are the end points of a chord passing through the center of the circle and parallel to the chord \( P_1, P_2 \) (See Figure 7.6). The angle to point \( P_3 \) can be calculated by finding the angle half way between \( \theta_1 \) and \( \theta_2 \) and subtracting 90 degrees from that angle. Let \( \theta_x \) be the angle half way between \( \theta_1 \) and \( \theta_2 \).

\[
\theta_x = \frac{\theta_1 + (\theta_2 - \theta_1)}{2}
\]

\[
\theta_3 = \theta_x - 90 \text{ degrees}
\]

\[
P_3 = (r \cos(\theta_3) + P_{0x}, r \sin(\theta_3) + P_{0y})
\]

\[
\theta_4 = \theta_x + 90 \text{ degrees}
\]

\[
P_4 = (r \cos(\theta_4) + P_{0x}, r \sin(\theta_4) + P_{0y})
\]

Now that point \( P_1, P_2, P_3 \) and \( P_4 \) are known it’s time to find the position of the apex of the arc. The position is calculated by drawing lines from \( P_1 \) to \( P_4 \) and from \( P_2 \) to \( P_3 \). The intersection of these lines will be the apex of the arc (See Figure 7.7).
Figure 7.7. Circular Facet (Arc Apex Determination)

Let the position of the intersection of the lines be represented by the point \( P_x \). To calculate the position of \( P_x \) it is necessary to find the equation for line \( P_1P_4 \) and \( P_2P_3 \) and then to solve two equations and two unknowns to find \( P_x \). First determine the equation of lines \( P_1P_4 \) and \( P_2P_3 \) in slope intercept form:

\[
\begin{align*}
  y &= m_1x + b_1 \\
  y &= m_2x + b_2
\end{align*}
\]

To calculate the slope and the y intercept use the equation of slope and the intercept equation:

\[
\begin{align*}
  m &= \frac{y_2 - y_1}{x_2 - x_1} \\
  b &= y - mx
\end{align*}
\]

Then solve the two equations and two unknowns to find the x, y coordinate of \( P_x \).

Given:

\[
\begin{align*}
  y &= m_1x + b_1 \\
  y &= m_2x + b_2
\end{align*}
\]

Substitute \( m_1x + b_1 \) for y:

\[
\begin{align*}
  m_1x + b_1 &= m_2x + b_2 \\
  m_1x - m_2x &= b_2 - b_1 \\
  x(m_1 - m_2) &= b_2 - b_1 \\
  x &= \frac{b_2 - b_1}{m_1 - m_2}
\end{align*}
\]
Rewrite equation of line as $x = (y - b_1) / m_1$

Substitute $(y - b_1) / m_1$ for $x$:

$y = m_2((y - b_1) / m_1) + b_2$

$y = ((m_2y - m_2b_1) / m_1) + b_2$

$m_1y = m_2y - m_2b_1 + m_1b_2$

$m_1y - m_2y = m_1b_2 - m_2b_1$

$y = (m_1b_2 - m_2b_1) / (m_1 - m_2)$

$P_x = ((b_2 - b_1) / (m_1 - m_2), (m_1b_2 - m_2b_1) / (m_1 - m_2))$

Now that the start and end points of the arc as well as the arcs apex have been calculated, the last remaining piece is to calculate the bounding box that is needed for the Windows GDI Arc function. The way it works is to imagine that the arc was actually tracing out a complete circle. Find the bounding box that precisely fits the imaginary circle then align that bounding box with the x and y coordinates (See Figure 7.8).

**Figure 7.8. Circular Facet (Arc Bounding Rectangle)**

Before the bounding box can be calculated a few more pieces of information are required. The center of the imaginary circle needs to be calculated as well as the circles radius. To calculate the center of the circle it is necessary to find the midpoint between lines $P_1P_x$ and $P_2P_x$ and label those points $P_5$ and $P_6$ respectively. $P_5$ and $P_6$ can be calculated using the midpoint formula.
MidPoint = \( ((X1 + X2) / 2, (Y1 + Y2) / 2) \)

Next calculate a line that is perpendicular to \( P1Px \) and passing through \( P5 \) and calculate another line perpendicular to \( P2Px \) and passing through point \( P6 \). The intersection of these lines will be the center of the circle labeled \( Px2 \) (See Figure 7.8). The slope of line \( P1Px \) is the same as line \( P1P4 \) which was calculated previously and the slope of line \( P2Px \) is the same as line \( P2P3 \) which was also calculated previously. Since we know the slope of these lines we can determine the slope of the perpendicular line by using the fact that the slope of the perpendicular line is the inverse reciprocal of the slope.

\[ m1 = -(1/m2) \]

Now the y intercept can be calculated by using the slope that was just calculated and by plugging in the points \( P5 \) and \( P6 \) into the y intercept formula.

\[ b = y - mx \]

Again we will have the equation of two lines with two unknowns so it can be solved as shown previously to acquire the intersection of the lines which will be the origin of the imaginary circle.

\[ y = m1x + b1 \]
\[ y = m2x + b2 \]

\[ Px2 = \left( \frac{(b2 - b1)}{(m1 - m2)}, \frac{(m1b2 - m2b1)}{(m1 - m2)} \right) \]

The last piece of information required to find the bounding box of the imaginary circle is the radius of the circle. The radius can be calculated by using the distance formula between point \( Px2 \) and \( Px \).

\[ d = \text{Sqr}((x2 - x1)^2 + (y2 - y1)^2) \]

Now it is a simple process to calculate the upper left and lower right coordinates of the bounding box. To find the x coordinate of the upper left corner subtract the radius from the x coordinate part of the center of the circle. To find the y coordinate of the upper left corner add the radius to the y coordinate part of the center of the circle. To calculate the lower right coordinate of the bounding box follow the same procedure of using the center of the circle and adding or subtracting the radius.

Finally, the arc can be drawn with the Windows GDI Arc function as follows:
7.2.5 Drawing The Circular Graph

The circular graph is actually drawn onto the texture with a call to the \textit{Draw} (See Listing 7.5) method of the \texttt{CircularGraph} class. When the \texttt{Draw} method is called the program calculates all of the arcs up to the current step position. That is to say if the sequence is 25 nucleotides long and the step position is currently at position 20 of 25 then the \texttt{Draw} method will draw the circular graph with all the arc necessary up to position 20. The first parameter of the \texttt{Draw} method is a handle to a device context on which to draw so the final picture will be drawn onto the device context and would appear as in Figure 7.9.

\begin{verbatim}
void CircularGraph::Draw(HDC hdc, char* Sequence, int iStep, int* Pairing, float xsize, float ysize)
\end{verbatim}

\textbf{Listing 7.5. CircularGraph.Draw Method}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{circular_graph.png}
\caption{Circular Graph}
\end{figure}
7.3 Planar Graph Calculation

The planar graph calculation as described in section 3.5.4 is the closest approximation to how the secondary structure would actually appear under natural conditions. The planar graph usually consists of the letters A, C, G, and U which represent the nucleotide bases, positioned on a two dimensional graph in such a way as to represent their distance from each other as well as their pairing. Sometimes the bases are omitted and lines are drawn to represent the secondary structure where a joint in the line represents the place where a nucleotide base would be positioned. The pairing between the bases is represented by a blue circle for an A-U bond and a red circle for a G-C bond. Sometimes the circles are replaced with red or blue lines. The planar graph representation has numbers which represent the loops that are formed in the sequence. For larger sequences a positional number is placed outside the loop at a given interval so that the linear position in the sequence can be easily determined.

7.3.1 Planar Graph Base Program

The base of the planar graph calculation used in the visualization was originally written by Robert E. Bruccoleri. Bruccoleri wrote a program called NAView which stands for nucleic acid view [29]. The NAView program takes a Zuker .CT file as input, runs through its calculation and outputs a .PLT2 plotter file. If another output type is needed then a companion program written by Darrin Stewart could be used to convert the plotter file to a postscript file. Darrin Stewart wrote a program called PLT22PS which takes the .PLT2 output from NAView as input and Darrin’s program outputs a .PS postscript file. From this point the output could be printed or another program such as Adobe Acrobat Distiller could be used to convert the postscript file into a .PDF (portable document format) file which can be viewed with the Adobe Acrobat Reader (See Figure 7.10).
Figure 7.10. Process Required To Create PDF

The visualization program needed to be able to calculate the secondary structure of any sequence and output the secondary structure as a bitmap. Additionally, the visualization program needed to be extremely efficient so the use of an external program was not feasible. Based on these two requirements it was necessary to take elements from the NAView program and the PLT22PS program and incorporate them into the visualization. The NAView program was turned into a C++ class called RNAGraph which is the heart of the calculation. The main modification to the program was that instead of taking a Zuker .CT file as input it was modified so that it could also take input passed into it through an interface method. It was also modified so that instead of instantly sending its output to a plotter .PLT2 file it stored its output in memory. Next the RNAGraph class what extended through inheritance into a class called RNAGraphBMP. The extended class took some drawing elements from the PLT22PS program so that it was able to take the output from RNAGraph and calculate a bitmap in memory (See Figure 7.11). The bitmap could then be used as part of the texture that gets drawn on the stage object.

Figure 7.11. Sequence to Bitmap
7.3.2 Planar Graph Calculation Input

There are four elements which are needed as input to the planar graph calculation. First, an integer is needed which holds the length of the sequence. Second, a character array is needed which holds the title of the given sequence. Third, a character array is needed which holds the actual sequence. And fourth, an integer array is needed which holds the pairing in the sequence. This pairing array is the same as the input to the circular graph in section 7.2.1.

```
int Length   = 9;
char* Title  = ['Y', 'e', 'a', 's', 't']
int* Pairing = [ 0,   8,   7,   3,   4,   6,   5,   2,   1]
```

7.3.3 Planar Graph Walkthrough

The planar graph calculation starts with a call to the `LoadFromRNASequence` method (See Listing 7.6). This method is used as the gateway where the input to the calculation gets passed into the RNAGraph class. The four variables being passed into the method are as described in section 7.3.2.

```
void RNAGraph::LoadFromRNASequence(char* Title, char* Sequence, int Length, int* Pairing)
```

Listing 7.6. LoadFromRNASequence Method Prototype

The `LoadFromRNASequence` method starts out by dynamically creating an array of structures of type Base (See Listing 7.7). The array labeled as `bases` is dynamically create by calling the c++ malloc function. The `bases` array is one element longer than the length of the input sequence and that is so that the element at array index zero can be used to store a start symbol. Type start symbol will be represented on the planar graph by a small black dot and it is nearly a placeholder that allows you to visually determine where the sequence starts.
typedef struct base_tag {
    char name;
    int mate, hist_num;
    float x, y;
    bool extracted;
    struct region_tag *region;
} Base;

Listing 7.7. Structure of type Base

The `LoadFromRNASequence` method then enters a loop where the member variables of each structure in the array are initialized. The `name` variable is used to store the nucleotide base symbol so each array element is updated with the sequence symbols. The `mate` variable is used to store the index of the nucleotide base which bonds with the current base. If the nucleotide base does not have a bonding pair then `mate` is populated with zero. The `hist_num` variable is simply the linear position of the structure in the array. The variables `x` and `y` are initialized with the fixed number 9999.0 but they are updated later in the calculation. These `x` and `y` variables are the main output of the calculation and will store the two dimensional position of the nucleotide base in the resulting graph. The two remaining variables, `extracted` and `region` are initialized to false and null respectively and are used later in the calculation. The resulting array created in this first method is as follows (See Figure 7.12):

```c
nbase = 9
bases = (struct Base *) malloc(sizeof(struct Base) * (nbase + 1));
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>char name</td>
<td>'o'</td>
<td>'G'</td>
<td>'G'</td>
<td>'G'</td>
<td>'A'</td>
<td>'A'</td>
<td>'A'</td>
<td>'U'</td>
<td>'C'</td>
<td>'C'</td>
</tr>
<tr>
<td>int mate</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>int hist_num</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>float x</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
</tr>
<tr>
<td>float y</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
</tr>
<tr>
<td>bool extracted</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>region* region</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>
```

Figure 7.12. Array of Base structures

Now that the `bases` array has been created and initialized the calculation goes into a method where it tries to find regions within the `bases` array. A region is section of the sequence that is connected in some way such as a stem or the start and ending pairs of a loop. The region finding method starts out by dynamically creating two arrays. The first array label as the `mark` array is an array of boolean variables and is only used for the region finding phase of the calculation and then it is removed. The `mark` array is used to
determine if the element at any given index in the array has been found to be part of a region. If the mark variable at the given index is true then the element is already part of the region and the calculation moves on to the next element. The second array label as the *regions* array in an array of Region structures (See Listing 7.8) and is used to store the start and end positions of the regions within the sequence.

```c
typedef struct region_tag {
    int start1, end1, start2, end2;
}Region;

Listing 7.8. Structure of type Region
```

The calculation then enters another loop where it goes through the *bases* array and updates the *regions* array with all regions that are found to exist. During this process the *region* pointer variable in the *bases* array also gets updated so that it points to the matching index in the *regions* array. As can be seen in Figure 7.4 there is a stem region that starts at *bases* array index 2 and ends at index 3 and the corresponding nucleotides start at *bases* array index 8 and ends at index 9. This is reflected in the *regions* array where a region has been determined to exist so the variables start1, end1, start2 and end2 are updated with those array indeces. A second region for the given sequence was found to exist at bases array index 6 and ending at index 7. Therefore, the arrays that exist after the region finding phase are as follows (See Figure 7.13):

| nbase = 9 |
|---|---|---|---|---|---|---|---|---|
| bases |
| char name | 'o' | 'G' | 'G' | 'G' | 'A' | 'A' | 'U' | 'C' | 'C' |
| int mate | 0 | 0 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| int hist_num | 9999 | 9999 | 9999 | 9999 | 9999 | 9999 | 9999 | 9999 | 9999 |
| float x | 9999 | 9999 | 9999 | 9999 | 9999 | 9999 | 9999 | 9999 | 9999 |
| bool extracted | False | False | False | False | False | False | False | False | False |

| mark = (bool *) malloc(sizeof(int) * (nbase + 1)); |
|---|---|---|---|---|---|---|---|---|
| nbase = 9 |
| bool mark | False | False | True | True | False | False | True | True | True |
| nregion = 2 |
| regions = (struct Region *) malloc(sizeof(struct Region) *(nbase + 1)); |
|---|---|---|---|---|---|---|---|---|
| int start1 | 2 | 6 | 3 | 6 | 8 | 7 | 9 | 7 | 7 |
| int end1 | 6 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |

Figure 7.13. Arrays after region finding phase
At this point both the *bases* array and the *regions* array have been initialized and all regions have been identified. The next phase of the calculation is to determine the loops that exist in the given sequence. The calculation now enters another method where all the previous input is used to identify the loops. The process starts by dynamically creating an array of Loop (See Listing 7.9) structures labeled as *loops* which is again one more than the length of the sequence. A recursive method is then executed which goes through the *bases* and *regions* arrays to determine where loops exist. As the loops are determined the calculation dynamically creates another array of Connection (See Listing 7.10) structures which holds the connections between the loops.

```c
typedef struct loop_tag {
    int nconnection;
    struct connection_tag **connections;
    int number;
    int depth;
    bool mark;
    float x, y, radius;
} Loop;

Listing 7.9. Structure of type Loop
```

```c
typedef struct connection_tag {
    struct loop_tag *loop;
    struct region_tag *region;
    int start, end; // Start and end form the 1st base pair of the region.
    float xrad, yrad, angle;
    bool extruded; // True if segment between this connection and
                   // the next must be extruded out of the circle
    bool broken;  // True if the extruded segment must be drawn long.
} Connection;

Listing 7.10. Structure of type Connection
```

As the recursive method progresses, loops will be identified and stored into the *loops* array. The loops array has member variables that will store the *x* and *y* coordinate of the loop as well as the *radius* of the loop. The *x*, *y* coordinates and the radius are used later in the calculation to determine the position of the nucleotide bases around the loop of a given radius. The *loops* array also stores a pointer to a *connection* array which connects the loops to each other as well as the regions. The arrays that are setup after this phase of the calculation are as follows (See Figure 7.14):
nbase = 9
bases

<table>
<thead>
<tr>
<th>char name</th>
<th>'o'</th>
<th>'G'</th>
<th>'G'</th>
<th>'G'</th>
<th>'A'</th>
<th>'A'</th>
<th>'U'</th>
<th>'C'</th>
<th>'C'</th>
</tr>
</thead>
<tbody>
<tr>
<td>int mate</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>int hist_num</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>float x</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
</tr>
<tr>
<td>float y</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
<td>9999</td>
</tr>
<tr>
<td>bool extracted</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

nregion = 2
regions

| int start1 | 2 | 6 |
| int end1   | 3 | 6 |
| int start2 | 8 | 7 |
| int end2   | 9 | 7 |

loop_count = 0
loops = (struct Loop *) malloc(sizeof(struct Loop) * (nbase + 1));

| int nconnection | 1 | 2 | 1 |
| connection**    | **A | **B | **C |
| int number      | 1 | 2 | 3 |
| int depth       | 0 | 0 | 0 |
| bool mark       | True | True | True |
| float x         | 0.0 | 0.0 | 0.0 |
| float y         | 0.0 | 0.0 | 0.0 |
| float radius    | 0.0 | 0.0 | 0.0 |

connection = (struct Connection **)realloc(retloop->connections, (++retloop->nconnection + 1) * sizeof(struct Connection *));

A

| Loop* loop | Loop[2] | Null |
| Region* region | Region[0] | |
| int start | 2 |
| int end   | 9 |
| float xrad | 0.0 |
| float yrad | 0.0 |
| float angle | 0.0 |
| bool extruded | False |
| bool broken | False |

connection = (struct Connection **)realloc(retloop->connections, (++retloop->nconnection + 1) * sizeof(struct Connection *));

B

| Region* region | Region[1] | Region[1] | |
| int start | 6 | 8 |
| int end   | 7 | 3 |
| float xrad | 0.0 | 0.0 |
| float yrad | 0.0 | 0.0 |
| float angle | 0.0 | 0.0 |
bool extruded | False | False | False
bool broken | False | False | False

connection = (struct Connection **)realloc(retloop->connections, (+retloop->nconnection + 1) * sizeof(struct Connection *));

C

<table>
<thead>
<tr>
<th>Loop* loop</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region* region</td>
<td>Loop[2]</td>
<td>Null</td>
</tr>
<tr>
<td>int start</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>int end</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>float xrad</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>float yrad</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>float angle</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>bool extruded</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>bool broken</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.14. Arrays after loop finding phase

At this point the calculation now enters another recursive method called traverse_loop (See Listing 7.11). This method is really the heart of the calculation. Everything previous to this method was identifying regions, loops and connections and preparing the necessary data required for this calculation. The traverse_loop method uses all the arrays and recursively travels through the arrays. As the method travels through the arrays it updates all the coordinate and radius information about where the objects should be positioned on the resulting planar graph. The method does not return anything and therefore its return type is void. What the methods output is, is that it updates all of the member variables in all the arrays. When this method completes all information necessary to draw the planar graph is present.

void RNAGraph::traverse_loop(struct Loop *lp, struct Connection *anchor_connection)

Listing 7.11. traverse_loop method prototype

7.3.4 Drawing The Planar Graph

Since all information necessary to draw the planar graph has now been calculated the program now enters its last phase which is to actually draw the planar graph. The drawing functionality is actually located in a class called RNAGraphBMP which was inherited from the RNAGraph class. The RNAGraphBMP class has access to all the
arrays and variables which have been created in the previous calculations and uses that information to draw the planar graph.

To actually draw the planar graph on to the texture a call to the Draw (See Listing 7.12) method of the RNAGraphBMP class is used. The Draw method takes 13 parameters on its method call. Most of the parameters are used in order to turn on or off certain features in the planar graph such as marking the loops or drawing a circle or line between bases.

```c
void RNAGraphBMP::Draw(HDC hdc, float BMPScale, float xsize, float ysize, bool mark_loops, bool draw_bases, float csz, int label_rate, bool dot_pairs, int mosaicx, int mosaicy, float glob_rot, RNA *pRNA)
```

Listing 7.12. RNAGraphBMP.Draw Method

In order to draw the image in a desired height and width the Draw method inputs the desired image size through its xsize and ysize parameters. The RNAGraph class calculated all of the coordinates necessary to draw the planar graph but the calculation used its own internal units. That means that the Draw method has to setup vectors for scaling and translation in order to properly draw the image. The Draw method follows this general order of events. It first sets up the scaling and translation vectors. Next it draws the Sequence Title centered across the bottom of the image and scaled accordingly. Next it draw the small circle which represents the start of the sequence. Next it draws the sequence lines, the lines that connect the nucleotide bases. Next it draws the circles or lines between the pairs which have a bond. Next it draws the nucleotide base symbol. Next it draws the linear position number based on the label rate. And finally it draws the loop numbers. All of this is drawn onto the device context which was passed into the method and the result is as follows (See Figure 7.15):
7.4 Matrix Graph Calculation

The matrix graph is the graphical representation of the dynamic programming matrix created in the Nussinov secondary structure algorithm. The matrix graph consists of a two-dimensional rectangular grid where each element in the grid contains a score from the Nussinov algorithm. The RNA sequence is aligned with the matrix across the top and along the left side of the matrix. The Nussinov algorithm uses an upper triangular matrix so that portion of the matrix will be the portion that has significant data. There are two versions of the matrix graph that are used in the visualization based on the version of the Nussinov algorithm selected. If the standard Nussinov algorithm is selected then each cell in the matrix will contain an integer value which represents the intermediate score of the Nussinov algorithm. If the SCFG Nussinov algorithm is selected then each cell of the matrix will contain a floating point number which is based on the probability calculation used in the algorithm. Finally, the matrix will have cells with colored backgrounds which correspond to the traceback path of the algorithm. These colored cells use a color scale to indicate the distance traveled along the traceback path.
7.4.1 Drawing The Matrix Graph

The matrix graph is drawn by calling the `Draw` (See Listing 7.13) method of the `MatrixGraph` class. The draw method takes four parameters which are used to supply all information necessary to draw the graph. The `pRNA` parameter supplies all information about the RNA sequence such as the nucleotides in the sequence the sequence length and the current step position. The `pNussinov` parameter supplies all information about which version of the Nussinov algorithm has been selected (Standard or SCFG) as well as the dynamic programming matrix. The `xsize` and `ysize` parameters are used to supply the height and width of the image that should be drawn. Finally, `hdc` is a handle to the device context on which the image will be drawn.

```c
void MatrixGraph::Draw(HDC hdc, RNA *pRNA, Nussinov *pNussinov, float xsize, float ysize)
```

Listing 7.13. MatrixGraph.Draw Method

The `Draw` method starts out by calculating the cell dimensions based on the image size variables `xsize` and `ysize`, and dividing that by the length of the RNA sequence. That calculation will return the cell height and with needed so that the whole matrix will fit within the image size. Next, the `Draw` method goes into a loop which draws the RNA sequence across the top and left borders. The loop starts with the nucleotide base, draws that base scaled to a size that fits within the cell, then moves one cell distance down and starts again. Once the RNA sequence has been drawn then the program enters a loop nested in another loop. The outer loop travels down the columns of the matrix while the inner loop travels down the rows. As the program comes across a cell with a value from the dynamic programming matrix it takes that value, converts it to a string, scales it so that it fits within the cell and draws the value. If the current cell is determined to be on the traceback path then the background of the cell is colored according the current color scale position. The final output from the calculation is as follow (See Figure 7.16):
7.5 Bracketed Graph Calculation

The main purpose of the bracketed graph representation is to visually convey the bonds between the nucleotide bases in the RNA sequence. The bonds are displayed in the graph by aligning a bracket with the RNA sequence. An open bracket indicates the first nucleotide base of the pair and the close bracket indicates the second base in the pair. A dash in the sequence indicates that no bond exists at the current location.

Since the bracketed graph representation doesn’t take a large area to represent there was room on this facet to display additional information. The facet also contains the traceback path which is the result of the Nussinov algorithm. In addition, the facet contains some statistics such as which Nussinov algorithm was used, Standard or SCFG, the length of the sequence, the current step position within the sequence and the optimal score of the Nussinov algorithm.
7.5.1 Drawing The Bracketed Graph

The bracketed graph is drawn through a call to the `Draw (See Listing 7.14)` method in the `BracketedGraph` class. The method takes five parameters and returns a void. The results of the `Draw` method are that the image is drawn onto the `hdc` parameter. The `hdc` parameter is a handle to a device context which is a bitmap that eventually becomes the texture. The other parameters `*pRNA`, `*pNussinove`, `xsize` and `ysize` are used to gather the information used for the statistics as well as the traceback and bracketed graph representation.

```cpp
void BracketedGraph::Draw(HDC hdc, RNA *pRNA, Nussinov *pNussinov, float xsize, float ysize)
```


The `Draw` method works by determining the size of the image through the `xsize` and `ysize` parameters. Based on the size of the image the method starts by drawing the statistics at the top of the image. The algorithm type is obtained through the `*pNussinov` class pointer and the sequence length, step position and optimal score are all obtained through the `*pRNA` class pointer. The `Draw` method then enters a loop which is used to draw the traceback path. The traceback path is stored as a linked list which is referenced through the `*pNussinov` class. The program loops through the list and draws the row and column position of the cell within the dynamic programming matrix. Finally, the `Draw` method enters another loop which is used to draw the bracketed representation. The bracketed graph is stored as an array of integers in the `*pRNA` class and is drawn by drawing the sequence on a row and then aligning either a open bracket, close bracket or dash on the next row. The final output from the bracketed graph calculation is as follows (See Figure 7.17):
7.6 Progress Bar Calculation

The progress bar representation is used to show the linear position of the current step within the RNA sequence. The progress bar works like a standard Windows progress bar except that the progress is tied directly to a particular nucleotide base in the sequence. As the program steps through the sequence the current position is covered by the blue progress bar to indicate that the current nucleotide is part of the calculation. All nucleotides which are not covered with the blue bar are beyond the current step position and are not included in the Nussinov calculation.

7.6.1 Drawing The Progress Bar

The progress bar is drawn through the Draw (See Listing 7.15) method in the ProgressGraph class. The Draw method takes five parameters in order to supply the information necessary for the calculation. The result of the method is that the progress bar image is drawn onto the device context of a bitmap which is passed in as the hdc parameter. The Sequence, RNALength and StepPosition are all parameters supplying information about the current state of the visualization.
void ProgressGraph::Draw(HDC hdc, char* Sequence, int RNALength, int StepPosition, float xsize, float ysize)

Listing 7.15. BracketedGraph.Draw Method

The Draw method starts out by determining the cell size that each character in the sequence must fit into and then calculates how many cells per line will fit onto the progress bar based on the current xsize and ysize parameters. Once it is known how many cells fit per line then the Draw method enters a loop and starts to draw the RNA sequence. As each nucleotide base in the sequence is encountered the program determines if the nucleotide base is either within or beyond the current step position. If the nucleotide base is within the current step position then the base is drawn with a blue background so that it’s part of the progress bar. If the base is beyond the step position then the base is drawn with a white background and is not part of the blue progress bar. When the loop hits the end of a line the program moves to the next line and starts again until the whole sequence has been covered. The output of the progress bar is as follows (See Figure 7.18):

![Progress Bar](image)

Figure 7.18. Progress Bar
Chapter 8

Program Usage

This chapter gives an overview of program usage and requirements. A minimum recommended hardware and software requirement is detailed. The chapter finishes by detailing the usage of the visualization program.

8.1 Program Requirements

The visualization program was written in C++ using the Microsoft Visual Studio 2005 development environment which is dependent on the Microsoft .Net 2.0 framework. This dependency means that the visualization requires that the .Net 2.0 framework has been installed. The visualization also uses Microsoft DirectX 9.0c in order to handle the three dimensional drawing functions so DirectX 9.0c must be installed. Both the .Net 2.0 framework and DirectX 9.0c are free downloads which can be obtained through the Microsoft website.

Since the visualization utilizes a three dimensional interface, the program has to run on fairly strong hardware. Although the program will run on a computer without a 3D accelerator card through software emulation, it is recommended that the computer have a 3D accelerator card. Therefore, recommended hardware would be a computer with a 2.0GHz CPU, 512MB RAM and a 3D accelerated graphics card.
With the release of Microsoft’s next version of Windows code named Vista, the DirectX API will be built into the operating system [30]. This means that with very minimal program modification to reference the new DirectX 10 API, the operating system will automatically support the three dimensional components of the visualization.

### 8.2 Program Usage

In order to use the visualization, the program should first be launched by executing the RNAVis.exe program. Upon launching the visualization, the program will automatically load an example RNA sequence into the display interface and all the visualization options will be available (See Figure 8.1). In the upper left corner of the interface there is a button that allows the help commands to be toggled on or off. The upper right corner of the interface contains the Nussinov type buttons as well as the algorithm stepping buttons. The main middle section of the interface contains the visualization of the Nussinov algorithm. The lower left corner of the interface displays statistics about the visualization. Finally, the lower right corner of the interface has three buttons for visualization options, a button to toggle between full screen, and a button for Direct3D options.

![Figure 8.1. The main visualization interface](image-url)
From this point one could step through the example sequence by clicking any one of the four stepping buttons or a new sequence could be loaded into the visualization. To load a new sequence one of two buttons could be clicked depending upon whether the standard or SCFG versions of the Nussinov algorithm are desired. After clicking the desired Nussinov type button, a new window will pop up which has several options related to the type of Nussinov algorithm selected.

If the standard Nussinov algorithm was selected then a window is displayed that allows one to enter the RNA sequence as well as several parameters related to the standard algorithm (See Figure 8.2). To give the sequence a name one would simply type in the sequence name into the edit box. To enter the RNA sequence one could either type the sequence directly into the edit box or one could paste a FASTA sequence into the box. To enter the scoring matrix one could enter any integer parameters into the scoring matrix fields but the visualization automatically defaults a scoring matrix upon program startup. In addition, there are some scoring matrix buttons which will populate the fields with commonly used parameters. Finally, there’s an edit box that is used to enter the minimum length that a hairpin loop is allowed to be. The hairpin loop value should be an integer between 0 and 10.

![Figure 8.2. Standard Nussinov Options](image-url)
If the SCFG Nussinov algorithm was selected then a window is displayed that allows one to enter the RNA sequence as well as several parameters related to the SCFG algorithm (See Figure 8.3). Just as in the standard version, the SCFG version has edit boxes to enter the sequence name as well as the sequence itself. The SCFG pop up has additional edit boxes to enter all of the probabilities for the grammar options. All together there are thirteen edit box related to the probabilities any of which can be modified from the default which is populated at program startup. Finally, there is an edit box for the minimum hairpin loop length just like the one available in the standard version.

![SCFG Nussinov Options](image)

**Figure 8.3. SCFG Nussinov Options**

At the bottom left of the main visualization interface there is a button for visualization options. When this button is clicked a pop up windows is displayed that contains several options which can be set (See Figure 8.4). There is a check box which is used to toggle the three dimensional lines on or off between the matrix graph and the planar graph. The other four options in the window are all related to the planar graph. The **Mark Loops** check box toggles the drawing of an integer which indicates the loop number, on or off. The **Draw Bases** check box toggles the drawing of the nucleotide base symbol, on or off. The **Dot Pairs** check box toggles between drawing a circular dot or a line between the
nucleotide bases with a bond. Finally, the **Label Rate** edit box is used to enter the rate at which a linear position number will be drawn on the planar graph.

---

**Figure 8.4. Visualization Options Window**

The main interface also contains a button which is used for Direct3D Settings *(See Figure 8.5)*. This window was taken directly from an example project provided by Microsoft which was part of the DirectX SDK. This window has options for selecting the display adapter if more than one is present. The render device can be selected which can be either the hardware video card or software drivers. The Direct3D Settings window provides additional options all of which are related to Direct3D.

---

**Figure 8.5. Direct3D Settings**
Chapter 9

Conclusion

The objective of this project was to create a visualization that would be used as a teaching tool in order to increase the intuitive understanding of the Nussinov secondary structure prediction algorithm. In order to facilitate this objective the visualization took advantage of several sophisticated techniques with the purpose of conveying information about the algorithm. One of the primary techniques used to communicate the information about the algorithm was to illustrate the data with several representations simultaneously. The reasoning behind showing several representations simultaneously is to allow the user to draw connections between the data by visually analyzing the representations. The need to show several representations of data simultaneously led to the first innovation of the visualization. The program uses a three dimensional interface with the intention that the data representations can all be viewed together without cluttering the interface. A significant amount of attention was paid to creating an aesthetically pleasing interface which draws the attention of the user to the significant information while other ancillary information is conveyed on the perimeter. Additionally, the data representations where spatially aligned in such a way as to allow additional connections to be drawn by the visualization. For instance, the visualization draws three dimensional lines between the planar graph and the dynamic programming matrix in order to show how the calculation obtained the results. Another innovative feature of the visualization is its ability to allow the user to step through the RNA sequence and view intermediate results of the calculation. By starting with a small sequence and gradually stepping through the
sequence, information can be gleamed about the inner workings of the algorithm. Through the use of the techniques outlined here, the objective of creating a visualization teaching tool has been achieved.
Bibliography


[19] University of Southern California, XRNA, http://rna.ucsc.edu/rnacenter/xrna/xrna.html (Free Software)


[27] Microsoft DirectX 9.0C Documentation, Microsoft Corporation, 2006


[30] Shah, Sarju and Yu, James, Windows Vista and DirectX 10, 

[31] Vienna RNA Package, Representations Secondary Structures, 

Lecture 17, May 2003.
c:\Thesis\RNAVis\RNAVis.cpp

1 //-----------------------------------------------------------------------------------
2 // Copyright (c): 2006, All Rights Reserved
3 // Project:         SJSU Masters Project
4 // File:            RNAVis.cpp
5 // Purpose:         This is the main interface for the RNA Visualization program.
6 //                   This program creates a 3D interface where several representations
7 //                   of the RNA secondary structure can be viewed simultaneously.
8 //
9 // Start Date:      10/11/2006
10 // Programmer:      Brandon Hunter
11 //
12 // Based on the Basic starting point for new Direct3D samples
13 //
14 // Copyright (c) Microsoft Corporation. All rights reserved.
15 //-----------------------------------------------------------------------------------
16
17 #include "dxstdafx.h"
18 #include "resource.h"
19 #include "Stage.h"       // Header file for stage class
20 #include "RNA.h"        // Header file for RNA class
21 #include "RNAGraphBMP.h" // Header file for RNAGraphBMP class
22 #include "Nussinov.h"    // Header file for Nussinov class
23 #include "CircularGraph.h" // Header file for CircularGraph class
24 #include "MatrixGraph.h" // Header file for MatrixGraph class
25 #include "ProgressGraph.h" // Header file for ProgressGraph class
26 #include "BracketedGraph.h" // Header file for BracketedGraph class
27 #include "directio.h"    // for the sprintf function used to convert float to
28 // char*
29 #include "LineList.h"    // Header for linked list of lines
30 #include <direct.h>      // Used for getcwd() function
31
32 //#define DEBUG_VS     // Uncomment this line to debug vertex shaders
33 //#define DEBUG_PS    // Uncomment this line to debug pixel shaders
34
35 //-----------------------------------------------------------------------------------
36 // Global variables
37 //-----------------------------------------------------------------------------------
38
39 ID3DXFont* g_pFont = NULL;     // Font for drawing text
40 ID3DXSprite* g_pTextSprite = NULL; // Sprite for batching draw text
41 //ID3DXEffect* g_pEffect = NULL; // D3DX effect interface
42 /CMODELVIEWERCAMERA g_Camera; // A model viewing camera
43 bool g_bShowHelp = false;      // If true, it renders the UI
44
call control text
45 CDXUTDialogResourceManager g_DialogResourceManager; // manager for shared resources
46 of dialogs
47 CD3DSettingsDlg g_SettingsDlg; // Device settings dialog
48 CDXUTDialog g_HUD;             // dialog for standard controls
49 CDXUTDialog g_SampleUI;        // dialog for sample specific
50
call controls
51 RNA* g_pRNA;                   // Global pointer to RNA class
52 RNAGraphBMP* g_pRNAGraphBMP;   // Global pointer to RNAGraphBMP
53 Nussinov* g_pNussinov;         // Global pointer to Nussinov class
54 CircularGraph* g_pCircular;    // Global pointer to CircularGraph
55 MatrixGraph* g_pMatrix;        // Global pointer to MatrixGraph
56

class ProgressGraph* g_pProgress; // Global pointer to ProgressGraph
67 BracketedGraph* g_pBracketed; // Global pointer to BracketedGraph

DWORD m_dwFrames;
DWORD m_dwStartTime;
DWORD m_dwEndTime;
DWORD m_dwTotalPolygons;

// Globals For Visualization Options
bool g_Drawlines = true; // Draw 3D lines from Matrix Graph to Planar Graph
bool g_MarkLoops = true; // Draw loop numbers
bool g_DrawBases = true; // Draw the base character
bool g_DotBases = true; // Draw bonded bases with a dot
int g_LabelRate = 50; // Rate at which to draw base number

// UI control IDs
#define IDC_TOGGLEFULLSCREEN 1
#define IDC_CHANGETDEVICE 2
#define IDC_NUSSINOVSTANDARD 3
#define IDC_NUSSINOVSCFG 4
#define IDC_STEPFIRST 5
#define IDC_STEPREV 6
#define IDC_STEPNEXT 7
#define IDC_STEPLAST 8
#define IDC_VISUALIZATIONOPTIONS 9

// Forward declarations
bool CALLBACK IsDeviceAcceptable(D3DCAPS9* pCaps, D3DFORMAT AdapterFormat,
D3DFORMAT BackBufferFormat, bool bWindowed, void* pUserContext);
bool CALLBACK ModifyDeviceSettings(DXUTDeviceSettings* pDeviceSettings, const
D3DCAPS9* pCaps, void* pUserContext);
HRESULT CALLBACK OnCreateDevice(IDirect3DDevice9* pd3dDevice, const D3DSURFACE_DESC*
pBackBufferSurfaceDesc, void* pUserContext);
HRESULT CALLBACK OnResetDevice(IDirect3DDevice9* pd3dDevice, const D3DSURFACE_DESC*
pBackBufferSurfaceDesc, void* pUserContext);
void CALLBACK OnFrameMove(IDirect3DDevice9* pd3dDevice, double fTime, float
fElaspedTime, void* pUserContext);
void CALLBACK OnFrameRender(IDirect3DDevice9* pd3dDevice, double fTime, float
fElaspedTime, void* pUserContext);
LRESULT CALLBACK MsgProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam, bool* pbNoFurtherProcessing, void* pUserContext);
void CALLBACK KeyboardProc(UINT nChar, bool bKeyDown, bool bAltDown, void* pUserContext);
void CALLBACK OnGUIEvent(UINT nEvent, int nControlID, CDXUTControl* pControl, void* pUserContext);
void CALLBACK OnLostDevice(void* pUserContext);
void CALLBACK OnDestroyDevice(void* pUserContext);
INT_PTR CALLBACK VisualizationOptions(HWND hWnd, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK NussinovStandard(HWND hWnd, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK NussinovSCFG(HWND hWnd, UINT, WPARAM, LPARAM);
int checkRNA(char* str);
int checkFASTA(char* str);
void InitApp();
void SetupCamera(IDirect3DDevice9* pd3dDevice);
void Render3D(IDirect3DDevice9* pd3dDevice);
void RenderText();
void CreateDynamicTexture();
void UpdateStageLines();
void GetCurrentPath(char* buffer);

//-----------------------------------------------------------------------------------
// Entry point to the program. Initializes everything and goes into a message processing
// loop. Idle time is used to render the scene.
//-----------------------------------------------------------------------------------

INT WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    // Enable run-time memory check for debug builds.
    #if defined(DEBUG) | defined(_DEBUG)
        _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
    #endif

    // Set the callback functions. These functions allow DXUT to notify
    // the application about device changes, user input, and windows messages. The
    // callbacks are optional so you need only set callbacks for events you're interested
    // in. However, if you don't handle the device reset/lost callbacks then the
    // framework won't be able to reset your device since the application must first
    // release all device resources before resetting. Likewise, if you don't handle
    // the
    // device created/destroyed callbacks then DXUT won't be able to
    // recreate your device resources.
    DXUTSetCallbackDeviceCreated(OnCreateDevice);
    DXUTSetCallbackDeviceReset(OnResetDevice);
    DXUTSetCallbackDeviceLost(OnLostDevice);
    DXUTSetCallbackDeviceDestroyed(OnDestroyDevice);
    DXUTSetCallbackMsgProc(MsgProc);
    DXUTSetCallbackKeyboard(KeyboardProc);
    DXUTSetCallbackFrameRender(OnFrameRender);
    DXUTSetCallbackFrameMove(OnFrameMove);

    // Show the cursor and clip it when in full screen
    DXUTSetCursorSettings(true, true);

    InitApp();

    // Initialize DXUT and create the desired Win32 window and Direct3D
    // device for the application. Calling each of these functions is optional, but they
    // allow you to set several options which control the behavior of the framework.
    DXUTInit(true, true, true); // Parse the command line, handle the default hotkeys,
    // and show msgboxes
    DXUTCreateWindow(L"RNA Secondary Structure Visualization");
    DXUTCreateDevice(D3DADAPTER_DEFAULT, true, 1024, 768, IsDeviceAcceptable, ModifyDeviceSettings);

    // Pass control to DXUT for handling the message pump and
    // dispatching render calls. DXUT will call your FrameMove
    // and FrameRender callback when there is idle time between handling window
    // messages.
    DXUTMainLoop();

    // Perform any application-level cleanup here. Direct3D device resources are
    // released within the
    // appropriate callback functions and therefore don't require any cleanup code
    // here.

    if (g_pRNA != NULL)
        delete g_pRNA;
if (g_pNussinov != NULL)
  delete g_pNussinov;
if (g_pRNAGraphBMP != NULL)
  delete g_pRNAGraphBMP;
if (g_pCircular != NULL)
  delete g_pCircular;
if (g_pMatrix != NULL)
  delete g_pMatrix;
if (g_pProgress != NULL)
  delete g_pProgress;
if (g_pBracketed != NULL)
  delete g_pBracketed;
return DXUTGetExitCode();
}

// Initialize the app
void InitApp()
{
  // Initialize dialogs
  g_SettingsDlg.Init(&g_DialogResourceManager);
  g_HUD.Init(&g_DialogResourceManager);
  g_SampleUI.Init(&g_DialogResourceManager);
  g_HUD.SetCallback(OnGUIEvent);
  g_HUD.AddButton(IDC_VISUALIZATIONOPTIONS, L"Visualization Options", 0, 0, 125, 22);
  g_HUD.AddButton(IDC_TOGGLEFULLSCREEN, L"Toggle Full Screen", 135, 0, 125, 22);
  g_HUD.AddButton(IDC_CHANGEDEVICE, L"Direct3D Settings (F2)", 270, 0, 125, 22, VK_F2);
  g_SampleUI.SetCallback(OnGUIEvent);
  int iX = 0, iY = 10;
  g_SampleUI.AddButton(IDC_NUSSINOVSTANDARD, L"Nussinov Standard", iX, iY, 125, 22);
  g_SampleUI.AddButton(IDC_NUSSINOVSCFG, L"Nussinov SCFG", iX, iY += 24, 125, 22);
  g_SampleUI.AddButton(IDC_STEPFIRST, L"Step First", iX += 135, iY = 10, 125, 22);
  g_SampleUI.AddButton(IDC_STEPPREV, L"Step - 1", iX, iY += 24, 125, 22);
  g_SampleUI.AddButton(IDC_STEPLAST, L"Step Last", iX += 135, iY = 10, 125, 22);
  g_SampleUI.AddButton(IDC_STEPNEXT, L"Step + 1", iX, iY += 24, 125, 22);
  m_dwFrames = 0;
  m_dwStartTime = 0;
  m_dwEndTime = 0;
  m_dwTotalPolygons = 0;
  // Visualization started, so record time
  m_dwStartTime = timeGetTime();

  char CurrentPath[150];
  GetCurrentPath(CurrentPath); // use the function to get the path

  // Load the default image ("C:\\Thesis\\RNAVis\\StageTexture.bmp")
  g_hbm = (HBITMAP)LoadImageA(0, strcat(CurrentPath, "\\StageTexture.bmp"), IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE);

  char* Title = new char[31]; // char* Title = (char *) malloc(sizeof(char) * 21);
  strcpy(Title, "Default Visualization Sequence");
  int Length = 25;
  char* Sequence = new char[Length + 1]; // char* Sequence = (char *) malloc(sizeof
(char) * Length);
strcpy(Sequence, "GGGGUUUUUGUCAGUUGGGAGAGC");
g_pRNA = new RNA(Title, Sequence, Length); // Instantiate the RNA class object
g_pNussinov = new Nussinov(Nussinov::NussinovType::NussinovStandard, g_pRNA->getSequence(), g_pRNA->getLength()); // Instantiate an Nussinov class object
g_pNussinov->FillStage();
g_pNussinov->TraceBack(g_pRNA); // Set the pairing array
g_pRNAGraphBMP = new RNAGraphBMP(0.7f); // Instantiate the RNAGraphBMP class object
g_pRNAGraphBMP->LoadFromRNASequence(g_pRNA->getTitle(), g_pRNA->getSequence(),
    g_pRNA->getStepPosition(), g_pRNA->getPairing());
g_pCircular = new CircularGraph(); // Instantiate the CircularGraph class object
g_pMatrix = new MatrixGraph(); // Instantiate the MatrixGraph class object
g_pProgress = new ProgressGraph(); // Instantiate the ProgressGraph class object
g_pBracketed = new BracketedGraph(); // Instantiate the BracketedGraph class object
CreateDynamicTexture();

bool CALLBACK IsDeviceAcceptable(D3DCAPS9* pCaps, D3DFORMAT AdapterFormat,
    D3DFORMAT BackBufferFormat, bool bWindowed, void* v
        pUserContext)
{
    // Skip backbuffer formats that don't support alpha blending
    IDirect3D9* pD3D = DXUTGetD3DObject();
    if(FAILED(pD3D->CheckDeviceFormat( pCaps->AdapterOrdinal, pCaps->DeviceType,
        AdapterFormat, D3DUSAGE_QUERY_POSTPIXELSHADER_BLENDING,
        D3DRTYPE_TEXTURE, BackBufferFormat)))
        return false;
    return true;
}

bool CALLBACK ModifyDeviceSettings(DXUTDeviceSettings* pDeviceSettings, const
    D3DCAPS9* pCaps, void* pUserContext)
{
    // If device doesn't support HW T&L or doesn't support 1.1 vertex shaders in HW
    // then switch to SWVP.
    if((pCaps->DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT) == 0 ||
        pCaps->VertexShaderVersion < D3DVS_VERSION(1,1))
        {
            pDeviceSettings->BehaviorFlags = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
        }
// Debugging vertex shaders requires either REF or software vertex processing
// and debugging pixel shaders requires REF.
#define DEBUG_VS
if( pDeviceSettings->DeviceType != D3DDEVTYPE_REF )
{
   pDeviceSettings->BehaviorFlags &= ~D3DCREATE_HARDWARE_VERTEXPROCESSING;
   pDeviceSettings->BehaviorFlags &= ~D3DCREATEPUREDEVICE;
   pDeviceSettings->BehaviorFlags |= D3DCREATE_SOFTWARE_VERTEXPROCESSING;
}
#endif DEBUG_VS
#define DEBUG_PS
pDeviceSettings->DeviceType = D3DDEVTYPE_REF;
#endif DEBUG_PS

// For the first device created if its a REF device, optionally display a warning dialog box.
static bool s_bFirstTime = true;
if(s_bFirstTime)
{
   s_bFirstTime = false;
   if(pDeviceSettings->DeviceType == D3DDEVTYPE_REF)
   DXUTDisplaySwitchingToREFWarning();
}
return true;

//-----------------------------------------------------------------------------------------------

// This callback function will be called immediately after the Direct3D device has been created, which will happen during application initialization and windowed/full screen toggles. This is the best location to create D3DPOOL_MANAGED resources since these resources need to be reloaded whenever the device is destroyed. Resources created here should be released in the OnDestroyDevice callback.

HRESULT CALLBACK OnCreateDevice(IDirect3DDevice9* pd3DDevice, const D3DSURFACE_DESC* pBackBufferSurfaceDesc, void* pUserContext)
{
   HRESULT hr;
   V_RETURN(g_DialogResourceManager.OnCreateDevice(pd3DDevice));
   V_RETURN(g_SettingsDlg.OnCreateDevice(pd3DDevice));
   // Initialize the font
   V_RETURN(D3DXCreateFont(pd3Device, 15, 0, FW_BOLD, 1, FALSE, DEFAULT_CHARSET,
                           OUT_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH | FF_DONTCARE,
                           L"Arial", &g_pFont));
   g_pStage = new Stage(pd3Device, 10.0f, 0.0f, -5.0f, 0.0f);
   g_pStage->SetTextureFromBitmap(g_hbm);
   UpdateStageLines();
   // Define DEBUG_VS and/or DEBUG_PS to debug vertex and/or pixel shaders with the shader debugger. Debugging vertex shaders requires either REF or software vertex processing, and debugging pixel shaders requires REF. The D3D3XSHADER_FORCE_SOFTWARE_NOOPT flag improves the debug experience in the shader debugger. It enables source level debugging, prevents instruction reordering, prevents dead code elimination, and forces the compiler to compile
HRESULT CALLBACK OnResetDevice(D3DDEVICE9* pd3dDevice, void* pUserContext)
{
    HRESULT hr;
    V_RETURN(g_DialogResourceManager.OnResetDevice());
    V_RETURN(g_SettingsDlg.OnResetDevice());
    if(g_pFont)
        V_RETURN(g_pFont->OnResetDevice());
    //if(g_pEffect)
    //    V_RETURN(g_pEffect->OnResetDevice());
    // Create a sprite to help batch calls when drawing many lines of text
    V_RETURN(D3DXCreateSprite(pd3dDevice, &g_pTextSprite));
    // Setup the camera's projection parameters
    float fAspectRatio = pBackBufferSurfaceDesc->Width / (FLOAT) pBackBufferSurfaceDesc->Height;
    //
    g_Camera.SetProjParams(D3DX_PI/4, fAspectRatio, 0.1f, 1000.0f);
    // Setup the camera's view parameters
    D3DXVECTOR3 vecEye(0.0f, 0.0f, -5.0f);
    D3DXVECTOR3 vecAt (0.0f, 0.0f, -0.0f);
    g_Camera.SetViewParams(&vecEye, &vecAt);
    return S_OK;
}

// This callback function will be called immediately after the Direct3D device has been reset, which will happen after a lost device scenario. This is the best location to create D3DPool_Default resources since these resources need to be reloaded whenever the device is lost. Resources created here should be released in the OnLostDevice callback.

HRESULT CALLBACK OnResetDevice(IDirect3DDevice9* pd3dDevice, void* pUserContext)
{
    HRESULT hr;
    V_RETURN(g_DialogResourceManager.OnResetDevice());
    V_RETURN(g_SettingsDlg.OnResetDevice());
    if(g_pFont)
        V_RETURN(g_pFont->OnResetDevice());
    //if(g_pEffect)
    //    V_RETURN(g_pEffect->OnResetDevice());
    // Create a sprite to help batch calls when drawing many lines of text
    V_RETURN(D3DXCreateSprite(pd3dDevice, &g_pTextSprite));
    // Setup the camera's projection parameters
    float fAspectRatio = pBackBufferSurfaceDesc->Width / (FLOAT) pBackBufferSurfaceDesc->Height;
    //
    g_Camera.SetProjParams(D3DX_PI/4, fAspectRatio, 0.1f, 1000.0f);
8

```cpp
// g_Camera.SetWindow(pBackBufferSurfaceDesc->Width, pBackBufferSurfaceDesc->Height);

// g_HUD.SetLocation(pBackBufferSurfaceDesc->Width - 405, pBackBufferSurfaceDesc->Height - 30);
g_HUD.SetSize(400, 170);
g_SampleUI.SetLocation(pBackBufferSurfaceDesc->Width - 405, 0);
g_SampleUI.SetSize(375, 300);

// Release the stage object
if (g_pStage != NULL) {
    delete g_pStage;
    g_pStage = NULL;
}
g_pStage = new Stage(pd3DDevice, 10.0f, 0.0f, -5.0f, 0.0f);
g_pStage->SetTextureFromBitmap(g_hbm);
UpdateStageLines();

return S_OK;

//-----------------------------------------------------------------------------------
// This callback function will be called once at the beginning of every frame. This is the
// best location for your application to handle updates to the scene, but is not intended to contain actual rendering calls, which should instead be placed in the
// OnFrameRender callback.
//-----------------------------------------------------------------------------------
void CALLBACK OnFrameMove(IDirect3DDevice9* pd3DDevice, double fTime, float fElapsedTime, void* pUserContext) {
    // Update the camera's position based on user input
    g_Camera.FrameMove(fElapsedTime);
}

//-----------------------------------------------------------------------------------
// This callback function will be called at the end of every frame to perform all the
// rendering calls for the scene, and it will also be called if the window needs to be repainted. After this function has returned, DXUT will call
// IDirect3DDevice9::Present to display the contents of the next buffer in the swap chain.
//-----------------------------------------------------------------------------------
void CALLBACK OnFrameRender(IDirect3DDevice9* pd3DDevice, double fTime, float fElapsedTime, void* pUserContext) {
    HRESULT hr;
    //D3DXMATRIXA16 mWorld;
    //D3DXMATRIXA16 mView;
    //D3DXMATRIXA16 mProj;
    //D3DXMATRIXA16 mWorldViewProjection;
    // If the settings dialog is being shown, then
    // render it instead of rendering the app's scene
    if (g_SettingsDlg.IsActive()) {
        g_SettingsDlg.OnRender(fElapsedTime);
        return;
    }
```
void SetupCamera(IDirect3DDevice9* pd3DDevice)
{
    // Here we will setup the camera.
    // The camera has three settings: "Camera Position", "Look at Position" and "Up Direction"
    D3DXMATRIX matView;

    // Clear the render target and the zbuffer
    V(pd3DDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, D3DCOLOR_ARGB(0, 45, 50, 170), 1.0f, 0));
    V(pd3DDevice->Clear(0L, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER, D3DCOLOR_ARGB(0, 66, 75, 121), 1.0f, 0L));

    // Setup the light
    D3DLIGHT9 light;
    D3DXVECTOR3 vecLightDirUnnormalized(0.0f, -1.0f, 1.0f);
    ZeroMemory(&light, sizeof(D3DLIGHT9));
    light.Type = D3DLIGHT_DIRECTIONAL;
    light.Diffuse.r = 1.0f;
    light.Diffuse.g = 1.0f;
    light.Diffuse.b = 1.0f;
    D3DXVECTOR3Normalize((D3DXVECTOR3*)&light.Direction, &vecLightDirUnnormalized);

    light.Position.x = 0.0f;
    light.Position.y = -1.0f;
    light.Position.z = 1.0f;
    light.Range = 1000.0f;
    V(pd3DDevice->SetLight(0, &light));
    V(pd3DDevice->LightEnable(0, TRUE));
    V(pd3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE));
    V(pd3DDevice->SetRenderState(D3DRS_AMBIENT, D3DCOLOR_XRGB(60, 60, 60)));

    // Render the scene
    if(SUCCEEDED(pd3DDevice->BeginScene()))
    {
        // Get the projection & view matrix from the camera class
        //mWorld = *g_Camera.GetWorldMatrix();
        //mProj = *g_Camera.GetProjMatrix();
        //mView = *g_Camera.GetViewMatrix();

        //mWorldViewProjection = mWorld * mView * mProj;

        // Update the effect's variables. Instead of using strings, it would
        // be more efficient to cache a handle to the parameter by calling
        // ID3DEffect::GetParameterByName
        //V(g_pEffect->SetMatrix("m_wWorldViewProjection", &mWorldViewProjection));
        //V(g_pEffect->SetMatrix("m_wWorld", &mWorld));
        //V(g_pEffect->SetFloat("g_fTime", (float)fTime));

        // Setup camera and perspective
        SetupCamera(pd3DDevice);

        // Now that the 3D camera is setup, render the 3D objects
        Render3D(pd3DDevice);

        DXUT_BeginPerfEvent(DXUT_PERFEVENTCOLOR, L"HUD / Stats"); // These events are to help PIX identify what the code is doing
        RenderText();
        V(g_HUD.OnRender(fElapsedTime));
        V(g_SampleUI.OnRender(fElapsedTime));
        DXUT_EndPerfEvent();

        V(pd3DDevice->EndScene());
        m_dwFrames++;
    }
}

// This function sets up the Camera
void SetupCamera(IDirect3DDevice9* pd3DDevice)
{
    // Here we will setup the camera.
    // The camera has three settings: "Camera Position", "Look at Position" and "Up Direction"
    D3DXMATRIX matView;
D3DXMatrixLookAtLH(&matView, &D3DXVECTOR3(0.0f, 5.0f, -15.0f), //Camera Position
 &D3DXVECTOR3(0.0f, 0.0f, 0.0f), //Look At Position
 &D3DXVECTOR3(0.0f, 1.0f, 0.0f)); //Up Direction
pd3DDevice->SetTransform(D3DTS_VIEW, &matView);

//Here we specify the field of view, aspect ratio and near and far clipping planes.
D3DXMATRIX matProj;
D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI/4, 1.0f, 1.0f, 2000.0f);
pd3DDevice->SetTransform(D3DTS_PROJECTION, &matProj);

//Make sure that the z-buffer and lighting are enabled
pd3DDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
pd3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
}

void Render3D(IDirect3DDevice9* pd3DDevice)
{

D3DXMATRIX MatWorld; // The final world matrix
D3DXMATRIX MatScale; // Matrix for scaling
D3DXMATRIX MatRot; // Final rotation matrix, applied to pMatWorld.

// Left-To-Right order of matrix concatenation is important

// Apply Translation
D3DXMatrixTranslation(&MatWorld, g_pStage->m_xPos, g_pStage->m_yPos, g_pStage->m_zPos);

// Apply Rotations
if(g_pStage->m_fPitch || g_pStage->m_fYaw || g_pStage->m_fRoll) {
    //D3DXMatrixIdentity(&MatRot);
    D3DXMatrixRotationYawPitchRoll(&MatRot, D3DXToRadian(g_pStage->m_fYaw),
    D3DXToRadian(g_pStage->m_fPitch), D3DXToRadian(g_pStage->m_fRoll));
    D3DXMatrixMultiply(&MatWorld, &MatRot, &MatWorld); // Apply matrix to world matrix.
}

// Apply Scaling
D3DXMatrixScaling(&MatScale, g_pStage->m_fScale, g_pStage->m_fScale, g_pStage->m_fScale);
D3DXMatrixMultiply(&MatWorld, &MatScale, &MatWorld);

//Render our objects
pd3DDevice->SetTransform(D3DTS_WORLD, &MatWorld);
_m_dwTotalPolygons += g_pStage->Render();

//Render the help and statistics text. This function uses the ID3DXFont interface for efficient text rendering.

void RenderText()
{
    // The helper object simply helps keep track of text position, and color
    // and then it calls pFont->DrawText( m_pSprite, strMsg, -1, &rc, DT_NOCLIP, m_clr);
    // If NULL is passed in as the sprite object, then it will work however the
    // pFont->DrawText() will not be batched together. Batching calls will improves performance.
    CDXUTTextHelper txtHelper(g_pFont, g_pTextSprite, 15);

    // Output statistics
txtHelper.Begin();
txtHelper.SetInsertionPos(5, 5);
txtHelper.SetForegroundColor(D3DCOLOR(1.0f, 1.0f, 1.0f, 1.0f));
DWORD dwDuration = (timeGetTime() - m_dwStartTime) / 1000;
if(dwDuration <= 0)
{
    txtHelper.DrawTextLine(L"Calculating..." 上);}
else
{
    // Draw help
    const D3DSURFACE_DESC* pd3dsdBackBuffer = DXUTGetBackBufferSurfaceDesc();
    if(g_bShowHelp)
    {
        txtHelper.SetInsertionPos(5, 5);
        //txtHelper.SetForegroundColor(D3DCOLOR(1.0f, 0.75f, 0.0f, 1.0f));
        txtHelper.DrawTextLine(L"Controls (F1 to hide):" 上);

        txtHelper.SetInsertionPos(40, 25);
        txtHelper.DrawTextLine( L"X Axis: Z,z\n"
L"Y Axis: X,x\n"
L"Z Axis: C,c" 上);

        txtHelper.SetInsertionPos(140, 25);
        txtHelper.DrawTextLine( L"Roll: A,a\n"
L"Pitch: S,s\n"
L"Yaw: D,d" 上);

        txtHelper.SetInsertionPos(240, 25);
        txtHelper.DrawTextLine( L"Stage: F,f\n"
L"Scale: +,-\n"
L"Wireframe: F8" 上);

        txtHelper.SetInsertionPos(340, 25);
        txtHelper.DrawTextLine( L"Panel 1: 1\n"
L"Panel 2: 2\n"
L"Panel 3: 3" 上);

        txtHelper.SetInsertionPos(440, 25);
        txtHelper.DrawTextLine( L"Panel 4: 4\n"
L"Reset: Home\n"
L"Quit: ESC" 上);

        //txtHelper.SetForegroundColor(D3DCOLOR(1.0f, 1.0f, 1.0f, 1.0f));
        dwDuration, m_dwFrames, (m_dwFrames / dwDuration), g_pStage->m_fPitch, g_pStage->m_fYaw, g_pStage->m_fRoll, g_pStage->m_fScale, m_nMouseLeft, m_nMouseRight, m_nMouseX, m_nMouseY);
        wchar_t buffer[255];
        swprintf(buffer, L"Duration: %d s Frames: %d FPS: %d Pitch: %f. Yaw: %f. Roll: %f. Scale: %f. Left: %d. Right: %d. X = %d. Y = %d." 上);
        g_pStage->m_fPitch, g_pStage->m_fYaw, g_pStage->m_fRoll, g_pStage->m_fScale)
        txtHelper.DrawTextLine(buffer);

        swprintf(buffer, L"X: %f. Y: %f. Z: %f." 上);
g_pStage->m_xPos, g_pStage->m_zPos)
        txtHelper.DrawTextLine(buffer);

        //txtHelper.SetForegroundColor(D3DCOLOR(1.0f, 1.0f, 0.0f, 1.0f));
        //txtHelper.DrawTextLine(DXUTGetFrameStats()) 上);
        //txtHelper.DrawTextLine(DXUTGetDeviceStats()) 上);
    }
}
else
{
    txtHelper.SetInsertionPos(5, 5);
    txtHelper.SetForegroundColor(D3DCOLOR(1.0f, 1.0f, 1.0f, 1.0f));
    txtHelper.DrawTextLine(L"Press F1 for help");
}

// Before handling window messages, DXUT passes incoming windows
// messages to the application through this callback function. If the application
// sets
// *pbNoFurtherProcessing to TRUE, then DXUT will not process this message.
LRESULT CALLBACK MsgBoxProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam, bool* pbNoFurtherProcessing, void* pUserContext)
{
    // Always allow dialog resource manager calls to handle global messages
    // so GUI state is updated correctly
    *pbNoFurtherProcessing = g_DialogResourceManager.MsgProc(hWnd, uMsg, wParam, lParam);
    if(*pbNoFurtherProcessing)
    return 0;

    if(g_SettingsDlg.IsActive())
    {
        g_SettingsDlg.MsgProc(hWnd, uMsg, wParam, lParam);
        return 0;
    }

    // Give the dialogs a chance to handle the message first
    *pbNoFurtherProcessing = g_HUD.MsgProc(hWnd, uMsg, wParam, lParam);
    if(*pbNoFurtherProcessing)
    return 0;

    *pbNoFurtherProcessing = g_SampleUI.MsgProc(hWnd, uMsg, wParam, lParam);
    if(*pbNoFurtherProcessing)
    return 0;

    // Pass all remaining windows messages to camera so it can respond to user input
    g_Camera.HandleMessages(hWnd, uMsg, wParam, lParam);

    switch( uMsg )
    {
        // Use WM_CHAR to handle parameter adjustment so
        // that we can control the granularity based on
        // the letter cases.
        case WM_CHAR:
        {
            switch( wParam )
            {
                case 'A':
                case 'a':
                    if( 'a' == wParam )
                        g_pStage->m_fRoll -= 0.5f;
                    else
                        g_pStage->m_fRoll += 0.5f;
                    break;
                case 'S':
                case 's':
                    if( 's' == wParam )
                        break;
            }
        }
        break;
    }
g_pStage->m_fPitch = 0.5f;
else
    g_pStage->m_fPitch += 0.5f;
break;
case 'D':
case 'd':
    if ( 'd' == wParam)
        g_pStage->m_fYaw -= 0.5f;
    else
        g_pStage->m_fYaw += 0.5f;
break;
case 'F':
case 'f':
    if ( 'f' == wParam )
        g_pStage->SetStageAngle(g_pStage->m_fTheta = 0.0f);
    else
        g_pStage->SetStageAngle(g_pStage->m_fTheta += 0.0f);
break;
case 'Z':
case 'z':
    if ( 'z' == wParam )
        g_pStage->m_xPos -= 0.5f;
    else
        g_pStage->m_xPos += 0.5f;
break;
case 'X':
case 'x':
    if ( 'x' == wParam )
        g_pStage->m_yPos -= 0.5f;
    else
        g_pStage->m_yPos += 0.5f;
break;
case 'C':
case 'c':
    if ( 'c' == wParam )
        g_pStage->m_zPos -= 0.5f;
    else
        g_pStage->m_zPos += 0.5f;
break;
case 'I':
case 'i':
    g_pStage->m_fScale = 0.5f;
    if(g_pStage->m_fScale < 1.0f) // Set lower bound on scaling
        g_pStage->m_fScale = 1.0f;
break;
case 'L':
case 'l':
    g_pStage->m_fScale += 0.5f;
    if(g_pStage->m_fScale > 20.0f) // Set upper bound on scaling
        g_pStage->m_fScale = 20.0f;
break;
case '1': // Show Matrix Panel
    g_pStage->m_xPos = 0.0f;
    g_pStage->m_yPos = -7.0f;
    g_pStage->m_zPos = 0.0f;
    g_pStage->m_fRoll = 0.0f;
    g_pStage->m_fPitch = -72.0f;
    g_pStage->m_fYaw = 0.0f;
    g_pStage->m_fScale = 1.0f;
    g_pStage->SetStageAngle(D3DXToRadian(10.0f));
break;
case '2': // Show Planar Graph Panel
    g_pStage->m_xPos = 0.0f;
    g_pStage->m_yPos = -0.5f;
    g_pStage->m_zPos = -15.0f;
void KeyboardProc(UINT nChar, bool bKeyDown, bool bAltDown, void* pUserContext)
{
    switch(nChar)
    {
    case VK_F1:
        g_bShowHelp = !g_bShowHelp;
        break;
    case VK_HOME:
        g_pStage->m_xPos = 0.0f;
        g_pStage->m_yPos = -5.0f;
        g_pStage->m_zPos = 0.0f;
        g_pStage->m_fRoll = 0.0f;
        g_pStage->m_fPitch = 0.0f;
        g_pStage->m_fYaw = 0.0f;
        g_pStage->m_fScale = 1.0f;
        break;
    case VK_ESCAPE:
        case VK_DOWN:
            case VK_LEFT:
                case VK_RIGHT:
                    case VK_HOME:
                        case VK_END:
                            case VK_PAGE_DOWN:
                                case VK_PAGE_UP:
                                    break;
    return 0;
    }
}
return 0;
}
void CALLBACK OnGUIEvent(UINT nEvent, int nControlID, CDXUTControl* pControl, void* pUserContext)
{
    switch(nControlID)
    {
    case IDC_VISUALIZATIONOPTIONS:
        {
            bool bWindowed = DXUTIsWindowed();
            if( !bWindowed )
                DXUTToggleFullScreen();
            DialogBox(NULL, MAKEINTRESOURCE(IDD_VISUALIZATIONOPTIONS), DXUTGetHWND(), VisualizationOptions);
            if( !bWindowed )
                DXUTToggleFullScreen();
            break;
        }
    case IDC_TOGGLEFULLSCREEN:
        DXUTToggleFullScreen();
        break;
    case IDC_CHANGEDEVICE:
        g_SettingsDlg.SetActive(!g_SettingsDlg.IsActive());
        break;
    case IDC_NUSSINOVSTANDARD:
        {
            bool bWindowed = DXUTIsWindowed();
            if( !bWindowed )
                DXUTToggleFullScreen();
            DialogBox(NULL, MAKEINTRESOURCE(IDD_NUSSINOVSTANDARD), DXUTGetHWND(), NussinovStandard);
            if( !bWindowed )
                DXUTToggleFullScreen();
            break;
        }
    case IDC_NUSSINOVSCFG:
        {
            bool bWindowed = DXUTIsWindowed();
            if( !bWindowed )
                DXUTToggleFullScreen();
            DialogBox(NULL, MAKEINTRESOURCE(IDD_NUSSINOVSCFG), DXUTGetHWND(), NussinovSCFG);
            if( !bWindowed )
                DXUTToggleFullScreen();
            break;
        }
    case IDC_STEPFIRST:
        g_pRNA->setStepPosition(1);
        g_pNussinov->TraceBack(g_pRNA); // Set the pairing array
        g_pRNA->getStepPosition(), g_pRNA->getPairing());
        CreateDynamicTexture();

        break;
    default:
        break;
    }
}
```cpp
// This callback function will be called immediately after the Direct3D device has entered a lost state and before IDirect3DDevice9::Reset is called. Resources created in the OnResetDevice callback should be released here, which generally includes all D3DPOOL_DEFAULT resources. See the "Lost Devices" section of the documentation for information about lost devices.

void CALLBACK OnLostDevice(void* pUserContext)
{
    g_DialogResourceManager.OnLostDevice();
    g_SettingsDlg.OnLostDevice();
    if (g_pFont)
        g_pFont->OnLostDevice();
    if (g_pEffect)
        g_pEffect->OnLostDevice();
    SAFE_RELEASE(g_pTextSprite);
    // Release the stage object
    if (g_pStage != NULL)
    {
        delete g_pStage;
    }
```
```
// This callback function will be called immediately after the Direct3D device has
// been destroyed, which generally happens as a result of application termination or
// windowed/full screen toggles. Resources created in the OnCreateDevice callback
// should be released here, which generally includes all D3DPOOL_MANAGED resources.

void CALLBACK OnDestroyDevice(void* pUserContext)
{
    g_DialogResourceManager.OnDestroyDevice();
    g_SettingsDlg.OnDestroyDevice();
    SAFE_RELEASE(g_pEffect);
    SAFE_RELEASE(g_pFont);
}

// FUNCTION: CreateDynamicTexture
// Used to dynamically create the texture that will be drawn
//@param undefined void
//@return void
void CreateDynamicTexture()
{
    HPEN hPen, hOldPen;
    float fTextureSizeX, fTextureSizeY;
    float fOriginX, fOriginY;
    HDC hdc;
    HBITMAP hbmOld = (HBITMAP)SelectObject(hdc, g_hbm);

    //BITMAP bm;
    //HDC hdc = CreateCompatibleDC(NULL);
    //HBITMAP hbmOld = (HBITMAP)SelectObject(hdc, g_hbm);
    //GetObject(g_hbm, sizeof(bm), &bm);
    //BitBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, hdc, 0, 0, SRCCOPY);

    /// Planar Graph
    fTextureSizeX = 512.0f;
    fTextureSizeY = 512.0f;
    fOriginX = 0.0f;
    fOriginY = 0.0f;
    hPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
    hOldPen = (HPEN)SelectObject(hdc, hPen);
    Rectangle(hdc, (int)(fOriginX), (int)(fOriginY), (int)fTextureSizeX, (int)fTextureSizeY);
    SelectObject(hdc, hOldPen);
    DeleteObject(hPen);
    g_pRNAGraphBMP->Draw(hdc, 1.0, fTextureSizeX, fTextureSizeY, g_MarkLoops,
     g_DrawBases, 0.5, g_LabelRate, g_DotPairs, 1, 1, 0.0, g_pRNA);

    /// Circular Graph
    fTextureSizeX = 512.0f;
    fTextureSizeY = 512.0f;
    fOriginX = 512.0f;
    fOriginY = 0.0f;
    hPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
    hOldPen = (HPEN)SelectObject(hdc, hPen);
    Rectangle(hdc, (int)(fOriginX), (int)(fOriginY), (int)(fOriginX + fTextureSizeX),
     (int)(fOriginY + fTextureSizeY));
```
SelectObject(hdc, hOldPen);
deleteObject(hPen);
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 1, -1, NULL);
SetViewportExtEx(hdc, 1, 1, NULL);
SetViewportOrgEx(hdc, (int)(fOriginX + (fTextureSizeX / 2.0f) + 0.5f), (int)(fOriginY + (fTextureSizeY / 2.0f) + 0.5f), NULL);
g_pCircular->Draw(hdc, g_pRNA->getSequence(), g_pRNA->getStepPosition(), g_pRNA->getPairing(), fTextureSizeX, fTextureSizeY);

// Matrix Graph
fTextureSizeX = 512.0f;
fTextureSizeY = 512.0f;
fOriginX = 0.0f;
fOriginY = 512.0f;
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 1, 1, NULL);
SetViewportExtEx(hdc, 1, 1, NULL);
SetViewportOrgEx(hdc, (int)(fOriginX), (int)(fOriginY), NULL);
hPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
hOldPen = (HPPEN)SelectObject(hdc, hPen);
Rectangle(hdc, 0, 0, (int)(fTextureSizeX), (int)(fTextureSizeY));
SelectObject(hdc, hOldPen);
deleteObject(hPen);
g_pMatrix->Draw(hdc, g_pRNA, g_pNussinov, fTextureSizeX, fTextureSizeY);

// Bracketed Graph
fTextureSizeX = 512.0f;
fTextureSizeY = 512.0f;
fOriginX = 512.0f;
fOriginY = 512.0f;
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 1, 1, NULL);
SetViewportExtEx(hdc, 1, 1, NULL);
SetViewportOrgEx(hdc, (int)(fOriginX), (int)(fOriginY), NULL);
hPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
hOldPen = (HPPEN)SelectObject(hdc, hPen);
Rectangle(hdc, 0, 0, (int)(fTextureSizeX), (int)(fTextureSizeY));
SelectObject(hdc, hOldPen);
deleteObject(hPen);
g_pBracketed->Draw(hdc, g_pRNA, g_pNussinov, fTextureSizeX, fTextureSizeY);

// Progress Graph
fTextureSizeX = 1024.0f;
fTextureSizeY = 128.0f;
fOriginX = 0.0f;
fOriginY = 1152.0f;
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 1, 1, NULL);
SetViewportExtEx(hdc, 1, 1, NULL);
SetViewportOrgEx(hdc, (int)(fOriginX), (int)(fOriginY), NULL);
g_pProgress->Draw(hdc, g_pRNA->getSequence(), g_pRNA->getLength(), g_pRNA->getStepPosition(), fTextureSizeX, fTextureSizeY);

// Message handler for Visualization Options
INT_PTR CALLBACK VisualizationOptions(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
1063 }  
1064 UNREFERENCED_PARAMETER(1_Param);
1065 switch (message)
1066 {
1067 case WM_INITDIALOG:
1068 // Initialize the Draw Lines Check Box
1069 if (g_DrawLines)
1070 SendDlgItemMessage(hDlg, IDC_DRAW3DLINES, BM_SETCHECK, BST_CHECKED, (LPARAM)0);
1071 else
1072 SendDlgItemMessage(hDlg, IDC_DRAW3DLINES, BM_SETCHECK, BST_UNCHECKED, (LPARAM)0);
1073 // Initialize the Mark Loops Check Box
1074 if (g_MarkLoops)
1075 SendDlgItemMessage(hDlg, IDC_MARKLOOPS, BM_SETCHECK, BST_CHECKED, (LPARAM)0);
1076 else
1077 SendDlgItemMessage(hDlg, IDC_MARKLOOPS, BM_SETCHECK, BST_UNCHECKED, (LPARAM)0);
1078 // Initialize the Draw Bases Check Box
1079 if (g_DrawBases)
1080 SendDlgItemMessage(hDlg, IDC_DRAWBASES, BM_SETCHECK, BST_CHECKED, (LPARAM)0);
1081 else
1082 SendDlgItemMessage(hDlg, IDC_DRAWBASES, BM_SETCHECK, BST_UNCHECKED, (LPARAM)0);
1083 // Initialize the Dot Pairs Check Box
1084 if (g_DotPairs)
1085 SendDlgItemMessage(hDlg, IDC_DOTPAIRS, BM_SETCHECK, BST_CHECKED, (LPARAM)0);
1086 else
1087 SendDlgItemMessage(hDlg, IDC_DOTPAIRS, BM_SETCHECK, BST_UNCHECKED, (LPARAM)0);
1088 SetDlgItemInt(hDlg, IDC_LABELRATE, g_LabelRate, true); // Initialize Label
1089 return (INT_PTR)TRUE;
1090 case WM_COMMAND:
1091 switch (LOWORD(wParam))
1092 {
1093 case IDOK:
1094 {
1095 if (SendDlgItemMessageA(hDlg, IDC_DRAW3DLINES, BM_GETCHECK, (LPARAM)0, BST_CHECKED) == (LPARAM)0) {
1096 g_DrawLines = true;
1097 else
1098 g_DrawLines = false;
1099 if (SendDlgItemMessageA(hDlg, IDC_MARKLOOPS, BM_GETCHECK, (LPARAM)0, BST_CHECKED) == (LPARAM)0) {
1100 g_MarkLoops = true;
1101 else
1102 g_MarkLoops = false;
1103 if (SendDlgItemMessageA(hDlg, IDC_DRAWBASES, BM_GETCHECK, (LPARAM)0, BST_CHECKED) == (LPARAM)0) {
1104 g_DrawBases = true;
1105 else
1106 g_DrawBases = false;
1107 if (SendDlgItemMessageA(hDlg, IDC_DOTPAIRS, BM_GETCHECK, (LPARAM)0, BST_CHECKED) == (LPARAM)0) {
1108 g_DotPairs = true;
1109 else
1110 g_DotPairs = false;
g_DotPairs = false;

// Get Label Rate from the text box
BOOL success;
g_LabelRate = GetDlgItemInt(hDlg, IDC_LABELRATE, &success, true);
if (!success)
{
    MessageBoxA(hDlg, "The Label Rate Is Invalid", "Invalid Value", MB_OK);
    break;
}
CreateDynamicTexture();
g_pStage->SetTextureFromBitmap(g_hbm);
UpdateStageLines();
EndDialog(hDlg, LOWORD(wParam));
return (INT_PTR)TRUE;
}
break;
case IDCANCEL:
    EndDialog(hDlg, LOWORD(wParam));
    return (INT_PTR)TRUE;
    break;
break;
}
return (INT_PTR)FALSE;

// Message handler for RNAInputBox.
INT_PTR CALLBACK NussinovStandard(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    UNREFERENCED_PARAMETER(lParam);
    switch (message)
    {
    case WM_INITDIALOG:
        // Initialize Title
        SetDlgItemTextA(hDlg, IDC_TITLE, g_pRNA->getTitle());
        break;
        // Initialize Sequence
        SetDlgItemTextA(hDlg, IDC_SEQUENCE, g_pRNA->getSequence());
        break;
        // Initialize the Scoring Matrix
        SetDlgItemInt(hDlg, IDC_AA, g_pNussinov->getScoringMatrix(0, 0), true);
        SetDlgItemInt(hDlg, IDC_AC, g_pNussinov->getScoringMatrix(0, 1), true);
        SetDlgItemInt(hDlg, IDC_AG, g_pNussinov->getScoringMatrix(0, 2), true);
        SetDlgItemInt(hDlg, IDC_AU, g_pNussinov->getScoringMatrix(0, 3), true);
        SetDlgItemInt(hDlg, IDC_CA, g_pNussinov->getScoringMatrix(1, 0), true);
        SetDlgItemInt(hDlg, IDC_CC, g_pNussinov->getScoringMatrix(1, 1), true);
        SetDlgItemInt(hDlg, IDC_CG, g_pNussinov->getScoringMatrix(1, 2), true);
        SetDlgItemInt(hDlg, IDC_CU, g_pNussinov->getScoringMatrix(1, 3), true);
        break;
        SetDlgItemInt(hDlg, IDC_GA, g_pNussinov->getScoringMatrix(2, 0), true);
        SetDlgItemInt(hDlg, IDC_GC, g_pNussinov->getScoringMatrix(2, 1), true);
        SetDlgItemInt(hDlg, IDC_GG, g_pNussinov->getScoringMatrix(2, 2), true);
        SetDlgItemInt(hDlg, IDC_GU, g_pNussinov->getScoringMatrix(2, 3), true);
        break;
        SetDlgItemInt(hDlg, IDC_UA, g_pNussinov->getScoringMatrix(3, 0), true);
        SetDlgItemInt(hDlg, IDC_UU, g_pNussinov->getScoringMatrix(3, 3), true);
        break;
        // Initialize Hairpin Loop Length
        SetDlgItemInt(hDlg, IDC_HAIRPINLENGTH, g_pNussinov->getMinHairpinLength(), true); // Set default minimum hairpin length
    return (INT_PTR)TRUE;
case WM_COMMAND:
    switch (LOWORD(wParam))
    {
        case IDOK:
            
            // Get the Title from the text box
            int TitleLen = (int)SendDlgItemMessageA(hDlg, IDC_TITLE, EM_LINELENGTH, (WPARAM)0, (LPARAM)0); // Get number of characters.
            char* szTitleBuffer = new char[TitleLen + 1];
            GetDlgItemTextA(hDlg, IDC_TITLE, szTitleBuffer, TitleLen + 1); // Get RNA Sequence from edit box

            // Get the RNA Sequence from the text box
            int lineCount = SendDlgItemMessageA(hDlg, IDC_SEQUENCE, EM_GETLINECOUNT, (WPARAM)0, (LPARAM)0); // Get number of lines in edit box
            int lineOffset;
            int RNALen = 0;
            for(int i = 0; i < lineCount; i++)
            {
                lineOffset = SendDlgItemMessageA(hDlg, IDC_SEQUENCE, EM_LINEINDEX, (WPARAM)i, (LPARAM)0);
                RNALen += (int)SendDlgItemMessageA(hDlg, IDC_SEQUENCE, EM_LINELENGTH, (WPARAM)lineOffset, (LPARAM)0) + 1;
            }
            // Get RNA Sequence from edit box

            // Get Scoring Matrix from the text boxes
            BOOL success;
            int AA, AC, AG, AU, CA, CC, CG, CU, GA, GC, GG, GU, UA, UC, UG, UU;
            AA = GetDlgItemInt(hDlg, IDC_AA, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: A, Column: A", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            AC = GetDlgItemInt(hDlg, IDC_AC, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: A, Column: C", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            AG = GetDlgItemInt(hDlg, IDC_AG, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: A, Column: G", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            AU = GetDlgItemInt(hDlg, IDC_AU, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: A, Column: U", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            CA = GetDlgItemInt(hDlg, IDC_CA, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, Column: A", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            GU = GetDlgItemInt(hDlg, IDC_GU, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, Column: U", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            UA = GetDlgItemInt(hDlg, IDC_UA, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: U, Column: A", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            UA = GetDlgItemInt(hDlg, IDC_UA, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: U, Column: U", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            UC = GetDlgItemInt(hDlg, IDC_UC, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: U, Column: C", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            UG = GetDlgItemInt(hDlg, IDC_UG, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: U, Column: G", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            GC = GetDlgItemInt(hDlg, IDC_GC, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: G, Column: C", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            GG = GetDlgItemInt(hDlg, IDC_GG, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: G, Column: G", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            CU = GetDlgItemInt(hDlg, IDC_CU, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, Column: U", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            CG = GetDlgItemInt(hDlg, IDC_CG, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, Column: G", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            CA = GetDlgItemInt(hDlg, IDC_CA, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, Column: A", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            CG = GetDlgItemInt(hDlg, IDC_CG, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, Column: G", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            AU = GetDlgItemInt(hDlg, IDC_AU, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: U, Column: A", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            GU = GetDlgItemInt(hDlg, IDC_GU, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: U, Column: G", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            UC = GetDlgItemInt(hDlg, IDC_UC, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: U, Column: C", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            GG = GetDlgItemInt(hDlg, IDC_GG, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: G, Column: G", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            CA = GetDlgItemInt(hDlg, IDC_CA, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, Column: A", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            GU = GetDlgItemInt(hDlg, IDC_GU, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, Column: U", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            UC = GetDlgItemInt(hDlg, IDC_UC, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, Column: C", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            AU = GetDlgItemInt(hDlg, IDC_AU, &success, true);
            if(!success)
            {
                MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: A, Column: U", "Invalid Scoring Matrix Entry", MB_OK);
                break;
            }
            }
break;
} else {
    CC = GetDlgItemInt(hDlg, IDC_CC, &success, true);
    if(!success)
        MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, 
Column: C", "Invalid Scoring Matrix Entry", MB_OK);
    break;
} else {
    CG = GetDlgItemInt(hDlg, IDC_CG, &success, true);
    if(!success)
        MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, 
Column: G", "Invalid Scoring Matrix Entry", MB_OK);
    break;
} else {
    CU = GetDlgItemInt(hDlg, IDC_CU, &success, true);
    if(!success)
        MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: C, 
Column: U", "Invalid Scoring Matrix Entry", MB_OK);
    break;
} else {
    GA = GetDlgItemInt(hDlg, IDC_GA, &success, true);
    if(!success)
        MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: G, 
Column: A", "Invalid Scoring Matrix Entry", MB_OK);
    break;
} else {
    GC = GetDlgItemInt(hDlg, IDC_GC, &success, true);
    if(!success)
        MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: G, 
Column: C", "Invalid Scoring Matrix Entry", MB_OK);
    break;
} else {
    GG = GetDlgItemInt(hDlg, IDC_GG, &success, true);
    if(!success)
        MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: G, 
Column: G", "Invalid Scoring Matrix Entry", MB_OK);
    break;
} else {
    GU = GetDlgItemInt(hDlg, IDC_GU, &success, true);
    if(!success)
        MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: G, 
Column: U", "Invalid Scoring Matrix Entry", MB_OK);
    break;
} else {
    UA = GetDlgItemInt(hDlg, IDC_UA, &success, true);
    if(!success)
        MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: U, 
Column: A", "Invalid Scoring Matrix Entry", MB_OK);
    break;
} else {
    UC = GetDlgItemInt(hDlg, IDC_UC, &success, true);
    if(!success)
        MessageBoxA(hDlg, "Scoring Matrix Has Invalid Data In Row: U, 
Column: C", "Invalid Scoring Matrix Entry", MB_OK);
    break;
} else {
    UG = GetDlgItemInt(hDlg, IDC_UG, &success, true);
    if(!success)
Instantimate the RNA class 

delete g_pRNA; 
g_pRNA = new RNA(szTitleBuffer, szRNABuffer, lengthRNA); //

Instantimate the RNA class object

delete g_pNussinov; 
g_pNussinov = new Nussinov(Nussinov::NussinovType::NussinovStandard, 

g_pRNA->getSequence(), g_pRNA->getLength()); // Instantiate an Nussinov class object

g_pNussinov->setScoringMatrix(AA, AC, AG, AU, CA, CC, CG, CU, GA, GC); 

hairpin loop length

g_pNussinov->FillStage(); 
g_pNussinov->TraceBack(g_pRNA); // Set the pairing array

delete g_pRNAGraphBMP; 
g_pRNAGraphBMP = new RNAGraphBMP(0.7f); // Instantiate the

class object

delete g_pCircular; 
g_pCircular = new CircularGraph; // Instantiate the CircularGraph

class object

delete g_pMatrix; 
g_pMatrix = new MatrixGraph; // Instantiate the MatrixGraph class

class object

delete g_pProgress; 
g_pProgress = new ProgressGraph; // Instantiate the ProgressGraph

class object

delete g_pBracketed; 
g_pBracketed = new BracketedGraph; // Instantiate the BracketedGraph

class object

CreateDynamicTexture();
g_pStage->SetTextureFromBitmap(g_hbm);
UpdateStageLines();

EndDialog(hDlg, LOWORD(wParam));
return (INT_PTR)TRUE;
    }
    break;
    case IDCANCEL:
        EndDialog(hDlg, LOWORD(wParam));
        return (INT_PTR)TRUE;
        break;
    case IDC_STANDARD:
        // Initialize standard Scoring Matrix
        SetDlgItemInt(hDlg, IDC_AA, 0, true);
        SetDlgItemInt(hDlg, IDC_AC, 0, true);
        SetDlgItemInt(hDlg, IDC_AG, 0, true);
        SetDlgItemInt(hDlg, IDC_AU, 1, true);
        SetDlgItemInt(hDlg, IDC_CA, 0, true);
        SetDlgItemInt(hDlg, IDC_CC, 0, true);
        SetDlgItemInt(hDlg, IDC_CG, 1, true);
        SetDlgItemInt(hDlg, IDC_CU, 0, true);
        SetDlgItemInt(hDlg, IDC_GA, 0, true);
        SetDlgItemInt(hDlg, IDC_GC, 1, true);
        SetDlgItemInt(hDlg, IDC_GG, 0, true);
        SetDlgItemInt(hDlg, IDC_GU, 0, true);
        SetDlgItemInt(hDlg, IDC_UA, 1, true);
        SetDlgItemInt(hDlg, IDC_UC, 0, true);
        break;
    case IDC_STANDARDGU:
        // Initialize standard plus GU Scoring Matrix
        SetDlgItemInt(hDlg, IDC_AA, 0, true);
        SetDlgItemInt(hDlg, IDC_AC, 0, true);
        SetDlgItemInt(hDlg, IDC_AG, 0, true);
        SetDlgItemInt(hDlg, IDC_AU, 1, true);
        SetDlgItemInt(hDlg, IDC_CA, 0, true);
        SetDlgItemInt(hDlg, IDC_CC, 0, true);
        SetDlgItemInt(hDlg, IDC_CG, 1, true);
        SetDlgItemInt(hDlg, IDC_CU, 0, true);
        SetDlgItemInt(hDlg, IDC_GA, 0, true);
        SetDlgItemInt(hDlg, IDC_GC, 1, true);
        SetDlgItemInt(hDlg, IDC_GG, 0, true);
        SetDlgItemInt(hDlg, IDC_GU, 1, true);
        SetDlgItemInt(hDlg, IDC_UA, 1, true);
        SetDlgItemInt(hDlg, IDC_UC, 0, true);
        break;
    case IDC_COMPLEX:
        // Initialize standard Scoring Matrix
        SetDlgItemInt(hDlg, IDC_AA, 0, true);
        SetDlgItemInt(hDlg, IDC_AC, 0, true);
        SetDlgItemInt(hDlg, IDC_AG, 0, true);
        SetDlgItemInt(hDlg, IDC_AU, 2, true);
        SetDlgItemInt(hDlg, IDC_CA, 0, true);
        SetDlgItemInt(hDlg, IDC_CC, 0, true);
        SetDlgItemInt(hDlg, IDC_CG, 3, true);
        SetDlgItemInt(hDlg, IDC_CU, 0, true);
        SetDlgItemInt(hDlg, IDC_GA, 0, true);
        SetDlgItemInt(hDlg, IDC_GC, 3, true);
        SetDlgItemInt(hDlg, IDC_GG, 0, true);
        SetDlgItemInt(hDlg, IDC_GU, 1, true);
        SetDlgItemInt(hDlg, IDC_UA, 2, true);
        SetDlgItemInt(hDlg, IDC_UC, 0, true);
        SetDlgItemInt(hDlg, IDC_UG, 1, true);
        SetDlgItemInt(hDlg, IDC_UU, 0, true);
        break;
    }
    break;
    break;
}
return (INT_PTR)FALSE;

// Message handler for about box.
INT_PTR CALLBACK NussinovSCFG(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_INITDIALOG:
        // Initialize Title
        SetDlgItemTextA(hDlg, IDC_TITLE, g_pRNA->getTitle());
        // Initialize Sequence
        SetDlgItemTextA(hDlg, IDC_SEQUENCE, g_pRNA->getSequence());
        // Initialize the Scoring Matrix
        char strNumber[64];
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(0, 0)));
        SetDlgItemTextA(hDlg, IDC_AS, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(1, 0)));
        SetDlgItemTextA(hDlg, IDC_CS, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(2, 0)));
        SetDlgItemTextA(hDlg, IDC_GS, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(3, 0)));
        SetDlgItemTextA(hDlg, IDC_US, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(0, 1)));
        SetDlgItemTextA(hDlg, IDC_SA, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(1, 1)));
        SetDlgItemTextA(hDlg, IDC_SC, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(2, 1)));
        SetDlgItemTextA(hDlg, IDC_SU, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(3, 1)));
        SetDlgItemTextA(hDlg, IDC_SS, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(0, 2)));
        SetDlgItemTextA(hDlg, IDC_ASU, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(1, 2)));
        SetDlgItemTextA(hDlg, IDC_CSG, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(2, 2)));
        SetDlgItemTextA(hDlg, IDC_GSC, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbMatrix(3, 2)));
        SetDlgItemTextA(hDlg, IDC_USA, strNumber);
        sprintf(strNumber, "%f", exp(g_pNussinov->getProbSS()));
        SetDlgItemTextA(hDlg, IDC_SSU, strNumber);
        // Initialize Hairpin Loop Length
        SetDlgItemInt(hDlg, IDC_HAIRPINLENGTH, g_pNussinov->getMinHairpinLength(), true); // Set default minimum hairpin length
        return (INT_PTR)TRUE;
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
        case IDOK:
            // Get the Title from the text box
            int TitleLen = (int)SendDlgItemMessageA(hDlg, IDC_TITLE, EM_LINELENGTH, (WPARAM) 0, (LPARAM) 0); // Get number of characters.
            char* szTitleBuffer = new char[TitleLen + 1];
            GetDlgItemTextA(hDlg, IDC_TITLE, szTitleBuffer, TitleLen + 1); // Get RNA Sequence from edit box
            // Get the RNA Sequence from the text box
            int lineCount = SendDlgItemMessageA(hDlg, IDC_SEQUENCE, EM_GETLINECOUNT, (WPARAM)0, (LPARAM)0); // Get number of lines in edit box
            int lineOffset;
int RNALen = 0;
for(int i = 0; i < lineCount; i++)
{
    lineOffset = SendMessageA(hDlg, IDC_SEQUENCE, EM_LINEINDEX, (LPARAM)i, (LPARAM)0);
    RNALen += (int)SendMessageA(hDlg, IDC_SEQUENCE, EM_LINELENGTH, (LPARAM)lineOffset, (LPARAM)0) + 1;
}
//int RNALen = (int) SendMessageA(hDlg, IDC_SEQUENCE, EM_LINELENGTH, (LPARAM)0); // Get number of characters.
char* szRNABuffer = new char[RNALen + 1];
GetDlgItemTextA(hDlg, IDC_SEQUENCE, szRNABuffer, RNALen + 1); // Get RNA Sequence from edit box

//atof(str);
// Get Probability Matrix from the text boxes
char strNumber[64];
float aS, cS, gS, uS, Sa, Sc, Sg, Su, aSu, cSc, gSc, uSa, SS;

GetDlgItemTextA(hDlg, IDC_AS, strNumber, 63);
aS = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_CS, strNumber, 63);
cS = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_GS, strNumber, 63);
gS = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_US, strNumber, 63);
uS = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_SA, strNumber, 63);
Sa = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_SC, strNumber, 63);
Sc = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_SG, strNumber, 63);
Sg = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_SU, strNumber, 63);
Su = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_AU, strNumber, 63);
aU = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_ASG, strNumber, 63);
cSc = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_GSC, strNumber, 63);
gSc = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_USA, strNumber, 63);
uSa = (float)atof(strNumber);
GetDlgItemTextA(hDlg, IDC_SSA, strNumber, 63);
SS = (float)atof(strNumber);

// Get Scoring Matrix from the text boxes
BOOL success;
int minHairpin;
minHairpin = GetDlgItemInt(hDlg, IDC_HAIRPINLENGTH, &success, true);
if(!success)
{
    MessageBoxA(hDlg, "The Minimum Hairpin Length Is Invalid", MB_OK);
    break;
}

checkFASTA(szRNABuffer); // If FASTA sequence then remove extra text
and leave sequence
int lengthRNA = checkRNA(szRNABuffer); // Verify correct RNA symbols have been entered

delete g_pRNA;
g_pRNA = new RNA(szTitleBuffer, szRNABuffer, lengthRNA); // Instantiate the RNA class object

delete g_pNussinov;
c:\Thesis\RNAVis\RNAVis.cpp

1533  g_pNussinov = new Nussinov(Nussinov::NussinovType::NussinovSCFG, ✓
1534    g_pRNA->getSequence(), g_pRNA->getLength()); // Instantiate a Nussinov class ✓
1535    object
1536    g_pNussinov->setProbMatrix(aS, cS, gS, uS, Sa, Sc, Sg, Su, aSu, cSG, ✓
1537    gSc, uSa, SS); // Set scoring matrix
1538    g_pNussinov->setMinHairpinLength(minHairpin); // Set the minimum ✓
1539  hairpin loop length
1540    g_pNussinov->FillStage();
1541    g_pNussinov->TraceBack(g_pRNA);
1542    delete g_pRNAGraphBMP;
1543    g_pRNAGraphBMP = new RNAGraphBMP(0.7f); // Instantiate the ✓
1544    RNAGraphBMP class object
1545    g_pRNAGraphBMP->LoadFromRNASequence(g_pRNA->getTitle(), g_pRNA-> ✓
1546    getSequence(), g_pRNA->getStepPosition(), g_pRNA->getPairing());
1547    delete g_pCircular;
1548    g_pCircular = new CircularGraph; // Instantiate the CircularGraph ✓
1549    class object
1550    delete g_pMatrix;
1551    g_pMatrix = new MatrixGraph; // Instantiate the MatrixGraph class ✓
1552    object
1553    delete g_pProgress;
1554    g_pProgress = new ProgressGraph; // Instantiate the ProgressGraph ✓
1555    class object
1556    delete g_pBracketed;
1557    g_pBracketed = new BracketedGraph; // Instantiate the BracketedGraph ✓
1558    class object
1559    CreateDynamicTexture();
1560    g_pStage->SetTextureFromBitmap(g_hbm);
1561    UpdateStageLines();
1562    EndDialog(hDlg, LOWORD(wParam));
1563    return (INT_PTR)TRUE;
1564 }
1565  break;
1566 case IDCANCEL:
1567    EndDialog(hDlg, LOWORD(wParam));
1568    return (INT_PTR)TRUE;
1569    break;
1570 }
1571  return (INT_PTR)FALSE;
1572
1573 // checkFASTA: Checks to see if the sequence is a FASTA sequence and if so it removes
1574 // the FASTA text so that only the RNA sequence remains
1575 int checkFASTA(char* str) {
1576    int i = 0; // index to copy to
1577    int j = 0; // index to copy from
1578 if(*str == '>') // This is a FASTA sequence
1579    {  
1580      while(*str != '\n') { // Loop to end of first line
1581        j++;
1582      }
1583      while(*str != '\0') { // Loop to end of sequence
1584        *(str + i++) = *(str + j++);
1585      }
1586      *(str + i) = '\0'; // Terminate the character array
1587      j = 0;
1588    }
1589 return i; // Return length of updated character array
1590 }
c:\Thesis\RNAVis\RNAVis.cpp

1590  // checkRNA: Checks to make sure that only valid characters (ACGU) are in
1591  // the RNA sequence and removes everything else. Converts lower case to upper case
1592  int checkRNA(char* str) {
1593      int i = 0; // index to copy to
1594      int j = 0; // index to copy from
1595      while(*str) { // Loop to end of sequence
1596          if(*str == 'A' || *str == 'C' || *str == 'G' || *str == 'U')
1597              *(str + i++) = *(str + j++);
1598          else
1599              if(*str == 'a' || *str == 'c' || *str == 'g' || *str == 'u')
1600                  *(str + i++) = *(str + j++) - 32; // Convert to upper case
1601          j++;
1602      }
1603      *str = '\0'; // Terminate the character array
1604      return i; // Return length of updated character array
1605  }
1606  
1607  // FUNCTION: UpdateStageLines
1608  // Used to update the lines that will be drawn between the planar graph and the
1609  // matrix graph
1610  
1611  // @param undefined void
1612  // @return void
1613  void UpdateStageLines() {
1614      LineList listLines; // Instantiate the list which holds lines
1615      if(g_DrawLines)
1616          { // Draw a line from current traceback cell to base at position i
1617              List<
1618                      Node_type* node_ptr = g_pNussinov->listTraceback.head;
1619                      while(node_ptr != NULL)
1620                          { // i Unpaired
1621                              if(node_ptr->next != NULL)
1622                                  { // j Unpaired
1623                                      if(node_ptr->next->i == node_ptr->i + 1 && node_ptr->next->j == node_ptr->j)
1624                                          { // Draw a line from current traceback cell to base at position j
1625                                              listLines.Add(node_ptr->xPos, node_ptr->yPos, g_pRNA->GetPlanarPos()[(node_ptr->i).PlanarX, g_pRNA->GetPlanarPos()[(node_ptr->i).PlanarY, D3DCOLOR_RGBA(102, 9, 123, 255)];
1626                                          }
1627                                      }
1628                                      else
1629                                          { // j Unpaired
1630                                              if(node_ptr->next->i == node_ptr->i && node_ptr->next->j == node_ptr->j - 1)
1631                                                  { // Draw a line from current traceback cell to both base
1632                                                      listLines.Add(node_ptr->xPos, node_ptr->yPos, g_pRNA->GetPlanarPos()[(node_ptr->i).PlanarX, g_pRNA->GetPlanarPos()[(node_ptr->i).PlanarY, D3DCOLOR_RGBA(223, 0, 41, 255)];
1633                                                  }
1634                                          }
1635                                      }
1636                                  }
1637                                  else
1638                                      { if(node_ptr->next->i == node_ptr->i + 1 && node_ptr->next->j == node_ptr->j - 1)
1639                                          { // i,j Paired
1640                                              if(node_ptr->next->i == node_ptr->i && node_ptr->next->j == node_ptr->j)
1641                                                  { // Draw a line from current traceback cell to both base
1642                                                      listLines.Add(node_ptr->xPos, node_ptr->yPos, g_pRNA->GetPlanarPos()[(node_ptr->i).PlanarX, g_pRNA->GetPlanarPos()[(node_ptr->i).PlanarY, D3DCOLOR_RGBA(223, 0, 41, 255)];
1643                                                  }
1644                                          }
1645                                      }
1646                      }
1647      }
1648  }
1649
D3DCOLOR_RGBA(0, 159, 98, 255));

    listLines.Add(node_ptr->xPos, node_ptr->yPos, g_pRNA->GetPlanarPos()[node_ptr->j].PlanarX, g_pRNA->GetPlanarPos()[node_ptr->j].PlanarY, D3DCOLOR_RGBA(0, 159, 98, 255));

    else
    
    } // Bifurcation
    // Draw a line from current traceback cell to both base

    listLines.Add(node_ptr->xPos, node_ptr->yPos, g_pRNA->GetPlanarPos()[node_ptr->i].PlanarX, g_pRNA->GetPlanarPos()[node_ptr->i].PlanarY, D3DCOLOR_RGBA(0, 159, 98, 255));

    listLines.Add(node_ptr->xPos, node_ptr->yPos, g_pRNA->GetPlanarPos()[node_ptr->j].PlanarX, g_pRNA->GetPlanarPos()[node_ptr->j].PlanarY, D3DCOLOR_RGBA(0, 159, 98, 255));

    else
    { // Bifurcation
        listLines.Add(node_ptr->xPos, node_ptr->yPos, g_pRNA->GetPlanarPos()[node_ptr->i].PlanarX, g_pRNA->GetPlanarPos()[node_ptr->i].PlanarY, D3DCOLOR_RGBA(0, 0, 0, 255));

        node_ptr = node_ptr->next;
    }

    g_pStage->CreateLines(&listLines);

    listLines.Clear();

    // FUNCTION: GetCurrentPath
    // Used to obtain the current working directory

    void GetCurrentPath(char* buffer) {
        getcwd(buffer, 150);
    }
c:\Thesis\RNAVis\Stage.h

1 //----------------------------------------------------------------------
2 // Copyright (c): 2006, All Rights Reserved
3 // Project: SJSU Masters Project
4 // File: Stage.h
5 // Purpose: Header file for Stage class. This class is the main object which
6 // the facets will be drawn on.
7 //
8 // Start Date: 2/1/2006
9 // Programmer: Brandon Hunter
10 //----------------------------------------------------------------------
11 
12 #pragma once
13 
14 #include <d3dx9.h>
15 #include "LineList.h" // Header file for linked list of lines
16 
17 //Define a FVF for the Stage
18 #define STAGE_D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1)
19 //Define a FVF for the lines
20 #define LINE_D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_DIFFUSE)
21 #define LINE_D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE)
22 
23 class Stage
24 {
25   private:
26     //Define a custom vertex for our stage
27     struct STAGE_CUSTOMVERTEX
28     {
29       float x, y, z; //Position of vertex in 3D space
30       float nx, ny, nz; //Lighting Normal
31       float tu, tv; //Texture coordinates
32     };
33 
34     //struct LINE_CUSTOMVERTEX
35     {//
36       float x, y, z; // Position of vertex in 3D space
37       DWORD color; // Color of vertex
38     };
39     struct LINE_CUSTOMVERTEX
40     {
41       float x, y, z; // Position of vertex in 3D space
42       float nx, ny, nz; // Normal vector for lighting calculations
43       DWORD color; // Diffuse color of vertex
44     };
45 
46   public:
47     float m_xPos;
48     float m_yPos;
49     float m_zPos;
50     float m_fTheta; // This is the angle of the wings on the stage
51     float m_fScale;
52     float m_fPitch;
53     float m_fYaw;
54     float m_fRoll;
55     bool SetMaterial(D3DCOLORVALUE rgbaDiffuse, D3DCOLORVALUE rgbaAmbient, //
56       D3DCOLORVALUE rgbaSpecular, D3DCOLORVALUE rgbaEmissive, float rPower);
57     bool SetTexture(const char* szTextureFilePath);
58     bool SetTextureFromBitmap(HBITMAP hBitmap);
59     bool SetPosition(float x, float y, float z);
60     bool SetStageAngle(float Theta);
61     DWORD Render();
62     Stage(LPDIRECT3DDevice9 pd3DDevice, float Theta = D3DXToRadian(25.0f), float x = 0.0f, float y = 0.0f, float z = 0.0f); //
63     ~Stage();
```c
void CreateLines(LineList* plistLines);

private:
  bool CreateIndexBuffer();
  D3DXVECTOR3 GetTriangleNormal(D3DXVECTOR3* vVertex1, D3DXVECTOR3* vVertex2, D3DXVECTOR3* vVertex3);
  bool UpdateVertices();
  HRESULT CreateVertexBuffer();
  LPDIRECT3DDEVICE9 m_pD3DDevice;
  LPDIRECT3DVERTEXBUFFER9 m_pVertexBuffer;
  LPDIRECT3DTEXTURE9 m_pTexture;
  D3DMATERIAL9 m_metMaterial;
  LPDIRECT3DINDEXBUFFER9 m_pIndexBuffer;
  LPDIRECT3DVERTEXBUFFER9 m_pLineVertexBuffer;
  DWORD m_dwNumOfVertices;
  DWORD m_dwNumOfIndices;
  DWORD m_dwNumOfPolygons;
  int m_lineCount;
};
```
// Copyright (c): 2006, All Rights Reserved
// Project: SJSU Masters Project
// File: Stage.h
// Purpose: This is the class implementation of the Stage object. This class is
// the main object which the facets will be drawn on.
// Start Date: 2/1/2006
// Programmer: Brandon Hunter

#include "Stage.h"

// FUNCTION: Stage
// Default Constructor

Stage::Stage(LPDIRECT3DDevice9 pD3DDevice, float Theta, float x, float y, float z)
{
    m_pD3DDevice = pD3DDevice;
    m_pVertexBuffer = NULL;
    m_pIndexBuffer = NULL;
    m_pTexture = NULL;
    m_pLineVertexBuffer = NULL;
    m_dwNumOfVertices = 50;
    m_dwNumOfIndices = 72;
    m_dwNumOfPolygons = 24;
    m_lineCount = 0;
    m_fTheta = D3DXToRadian(Theta);
    m_xPos = x;
    m_yPos = y;
    m_zPos = z;
    m_fScale = 1.0f;
    m_fPitch = 0.0f;
    m_fYaw = 0.0f;
    m_fRoll = 0.0f;
    //Set material default values (R, G, B, A)
    D3DCOLORVALUE rgbaDiffuse = {1.0, 1.0, 1.0, 0.0};
    D3DCOLORVALUE rgbaAmbient = {1.0, 1.0, 1.0, 0.0};
    D3DCOLORVALUE rgbaSpecular = {0.0, 0.0, 0.0, 0.0};
    D3DCOLORVALUE rgbaEmissive = {0.0, 0.0, 0.0, 0.0};
    SetMaterial(rgbaDiffuse, rgbaAmbient, rgbaSpecular, rgbaEmissive, 0);

    //Initialize Vertex Buffer
    if(SUCCEEDED(CreateVertexBuffer()))
    {
        if(CreateIndexBuffer())
        {
            UpdateVertices();
        }
    }
c:\Thesis\RNAVis\Stage.cpp

// FUNCTION: ~Stage
// Default Destructor
//
// @param undefined void
// @return void
Stage::~Stage()
{
    if(m_pTexture != NULL)
    {
        m_pTexture->Release();
        m_pTexture = NULL;
    }
    if(m_pIndexBuffer != NULL)
    {
        m_pIndexBuffer->Release();
        m_pIndexBuffer = NULL;
    }
    if(m_pVertexBuffer != NULL)
    {
        m_pVertexBuffer->Release();
        m_pVertexBuffer = NULL;
    }
    if(m_pLineVertexBuffer != NULL)
    {
        m_pLineVertexBuffer->Release();
        m_pLineVertexBuffer = NULL;
    }
}

// FUNCTION: Render
// Used to render the stage to the screen
//
// @param undefined void
// @return DWORD
DWORD Stage:::Render()
{
    m_pD3DDevice->SetStreamSource(0, m_pVertexBuffer, 0, sizeof(CUSTOMVERTEX));
    //m_pD3DDevice->SetVertexShader(STAGE_D3DFVF_CUSTOMVERTEX);
    m_pD3DDevice->SetFVF(STAGE_D3DFVF_CUSTOMVERTEX);
    if(m_pTexture != NULL)
    {
        //A texture has been set. We want our texture to be shaded based
        //on the current light levels, so use D3DTOP_MODULATE.
        m_pD3DDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
        m_pD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
        m_pD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_CURRENT);
    }
    else
    {
        //No texture has been set
        m_pD3DDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG2);
        m_pD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
        m_pD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_CURRENT);
    }
    m_pD3DDevice->SetMaterial(&m_matMaterial); //Select the material to use
    m_pD3DDevice->SetIndices(m_pIndexBuffer); //Select index buffer
    m_pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0, m_dwNumOfVertices, 0, m_dwNumOfPolygons); //Render polygons from index buffer
    // Render lines
    if(m_lineCount > 0)
    {
        m_pD3DDevice->SetStreamSource(0, m_pLineVertexBuffer, 0, sizeof(CUSTOMVERTEX));
LINE_CUSTOMVERTEX));
    m_p3DDevice->SetFVF(LINE_D3DFVF_CUSTOMVERTEX);
    m_p3DDevice->DrawPrimitive(D3DPT_LINELIST, 0, m_lineCount);
}

//Return the number of polygons rendered
return m_dwNumOfPolygons;

// FUNCTION: CreateLines
// Used to create the lines drawn between the planar graph and matrix graph
// @param plistLines LineList* pointer to list of lines
// @return void
void Stage::CreateLines(LineList* plistLines)
{
    VOID* pVertices;
    m_lineCount = 0;
    //Initialize Line Vertex Buffer
    if(m_pLineVertexBuffer != NULL)
    {
        m_pLineVertexBuffer->Release();
        m_pLineVertexBuffer = NULL;
    }

    //Create the Line vertex buffer from our device.
    if(SUCCEEDED(m_p3DDevice->CreateVertexBuffer((plistLines->count * 2) * sizeof STRUCT LINE_CUSTOMVERTEX),
                      0, LINE_D3DFVF_CUSTOMVERTEX,
                      D3DPOOL_DEFAULT, &m_pLineVertexBuffer, NULL))
    {
        LINE_CUSTOMVERTEX* p_cvLineVertices = new LINE_CUSTOMVERTEX[plistLines->count * 2];

        float scaleX = 10.0f / 512.0f;
        float scaleY = 10.0f / 512.0f;
        int x = 0;
        LineList::Node_type* node_ptr = plistLines->head;
        while(node_ptr != NULL)
        {
            p_cvLineVertices[x].x = (node_ptr->startX * scaleX) - 5.0f;
            p_cvLineVertices[x].y = 0.0f;
            p_cvLineVertices[x].z = 12.5f - (node_ptr->startY * scaleY);
            p_cvLineVertices[x].nx = 0.0f;
            p_cvLineVertices[x].ny = 1.0f;
            p_cvLineVertices[x].nz = 0.0f;
            p_cvLineVertices[x].color = node_ptr->color;
            x++;
            p_cvLineVertices[x].x = (node_ptr->endX * scaleX) - 5.0f;
            p_cvLineVertices[x].y = 10.0f - (node_ptr->endY * scaleY);
            p_cvLineVertices[x].z = 12.5;
            p_cvLineVertices[x].nx = 0.0f;
            p_cvLineVertices[x].ny = 1.0f;
            p_cvLineVertices[x].nz = 0.0f;
            p_cvLineVertices[x].color = node_ptr->color;
            x++;
            node_ptr = node_ptr->next;
        }

        //Get a pointer to the Line vertex buffer vertices and lock the vertex buffer
        if(SUCCEEDED(m_pLineVertexBuffer->Lock(0, sizeof(LINE_CUSTOMVERTEX) * (plistLines->count * 2), (void**)&pVertices, 0)))
        {
            //Copy our stored vertices values into the vertex buffer
        }
    }
Create the vertex buffer from our device.

```c
HRESULT Stage::CreateVertexBuffer()
{
    // FUNCTION: CreateVertexBuffer
    // Used to create the vertex buffer
    // @param undefined void
    // @return HRESULT
    HRESULT pBufferIndices;
    // Create the vertex buffer from our device
    if (FAILED(m_pD3DDevice->CreateVertexBuffer(m_dwNumOfVertices * sizeof(struct STAGE_CUSTOMVERTEX),
                                                      0, STAGE_D3DFVF_CUSTOMVERTEX,
                                                      D3DPOOL_DEFAULT, &m_pVertexBuffer, NULL)))
    {
        return E_FAIL;
    }
    return S_OK;
}
```
// Get a pointer to the index buffer indices and lock the index buffer
m_pIndexBuffer->Lock(0, m_dwNumOfIndices * sizeof(WORD), (void**)&pBufferIndices, 0);

// Copy our stored indices values into the index buffer
memcpy(pBufferIndices, pIndices, m_dwNumOfIndices * sizeof(WORD));

// Unlock the index buffer
m_pIndexBuffer->Unlock();
return true;

// FUNCTION: GetTriangleNormal
// Calculate the triangle normals. Used to lighting
D3DXVECTOR3 Stage::GetTriangleNormal(D3DXVECTOR3* vVertex1, D3DXVECTOR3* vVertex2, D3DXVECTOR3* vVertex3)
{
    D3DXVECTOR3 vNormal;
    D3DXVECTOR3 v1;
    D3DXVECTOR3 v2;
    D3DXVECTOR3 v3;
    D3DXVECTOR3 v1v2 = D3DXVECTOR3Subtract(&v1, vVertex2, vVertex1);
    D3DXVECTOR3 v1v3 = D3DXVECTOR3Subtract(&v1, vVertex3, vVertex1);
    D3DXVECTOR3 v2v3 = D3DXVECTOR3Subtract(&v2, vVertex3, vVertex1);
    D3DXVECTOR3 vCross = D3DXVECTOR3Cross(&vNormal, &v1v2, &v2v3);
    D3DXVECTOR3 vNormalize = D3DXVECTOR3Normalize(&vNormal, &vNormal);
    return vNormal;
}

// FUNCTION: UpdateVertices
// Used to update the vertices
bool Stage::UpdateVertices()
{
    DWORD i;
    VOID* pVertices;
    WORD* pBufferIndices;
    D3DXVECTOR3 vNormal;
    DWORD dwVertex1;
    DWORD dwVertex2;
    DWORD dwVertex3;

    WORD* pNumOfSharedPolygons = new WORD[m_dwNumOfVertices]; //Array holds how many times this vertex is shared
    D3DXVECTOR3* pSumVertexNormal = new D3DXVECTOR3[m_dwNumOfVertices]; //Array holds sum of all face normals for shared vertex

    //Clear memory
    for(i = 0; i < m_dwNumOfVertices; i++)
    {
        pNumOfSharedPolygons[i] = 0;
        pSumVertexNormal[i] = D3DXVECTOR3(0, 0, 0);  
    }
STAGE_CUSTOMVERTEX cvVertices[] =
{
    // Face 0
    (-5 - (10 * sin(m_fTheta)), -2, 0, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f,), // Vertex 0
    (-5 - (10 * sin(m_fTheta)), 0, 0, 0.0f, 0.0f, 0.0f, 0.0f, 0.9f,),  // Vertex 1
    (5 + (10 * sin(m_fTheta)), -2, 0, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f,), // Vertex 2
    (5 + (10 * sin(m_fTheta)), 0, 0, 0.0f, 0.0f, 0.0f, 1.0f, 0.9f,),  // Vertex 3
    // Face 1
    (-5, 0, 0, 0.0f, 0.0f, 0.0f, 0.0f, 0.9f,), // Vertex 4
    (-5, 0, 2.5, 0.0f, 0.0f, 0.0f, 0.0f, 0.8f,), // Vertex 5
    (-2.5, 0, 0, 0.0f, 0.0f, 0.0f, 0.125f, 0.9f,), // Vertex 6
    (-2.5, 0, 2.5, 0.0f, 0.0f, 0.0f, 0.125f, 0.8f,),  // Vertex 7
    // Face 2
    (-2.5, 0, 0, 0.0f, 0.0f, 0.0f, 0.125f, 0.9f,), // Vertex 8
    (-2.5, 0, 2.5, 0.0f, 0.0f, 0.0f, 0.125f, 0.8f,), // Vertex 9
    (0, 0, 0, 0.0f, 0.0f, 0.0f, 0.25f, 0.9f,), // Vertex 10
    (0, 0, 2.5, 0.0f, 0.0f, 0.0f, 0.250f, 0.8f,), // Vertex 11
    // Face 3
    (0, 0, 0, 0.0f, 0.0f, 0.0f, 0.25f, 0.9f,), // Vertex 12
    (0, 0, 2.5, 0.0f, 0.0f, 0.0f, 0.25f, 0.8f,), // Vertex 13
    (2.5, 0, 0, 0.0f, 0.0f, 0.0f, 0.375f, 0.9f,), // Vertex 14
    (2.5, 0, 2.5, 0.0f, 0.0f, 0.0f, 0.375f, 0.8f,); // Vertex 15
    // Face 4
    (2.5, 0, 0, 0.0f, 0.0f, 0.0f, 0.375f, 0.9f,); // Vertex 16
    (2.5, 0, 2.5, 0.0f, 0.0f, 0.0f, 0.375f, 0.8f,); // Vertex 17
    (5, 0, 0, 0.0f, 0.0f, 0.0f, 0.5f, 0.9f,); // Vertex 18
    (5, 0, 2.5, 0.0f, 0.0f, 0.0f, 0.5f, 0.8f,); // Vertex 19
    // Face 5
    (-5 - (10 * sin(m_fTheta)), 0, 0, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 20
    (-5 - (10 * sin(m_fTheta)), 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 21
    (5 + (10 * sin(m_fTheta)), 0, 0, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 22
    (5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 23
    // Face 6
    (5, 0, 0, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 24
    (5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 25
    (5 + (10 * sin(m_fTheta)), 0, 0, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 26
    // Face 7
    (5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 27
    (5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 28
    (5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 29
    // Face 8
    (5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 30
    // Face 9
    (5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 31
    (5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 32
    (5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 33
    (5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f,); // Vertex 34
// Face 9
7 5, 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 35
    5, 0, 12.5, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 36
7 (5 + (10 * sin(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 37
7 // Face 10
7 (-5 - (10 * sin(m_fTheta)), 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 38
7 (-5 - (10 * sin(m_fTheta)), 10, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, // vertex 39
7 (-5, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 40
7 (-5, 10.125, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 41
7 // Face 11
7 (5, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 42
7 (5, 10.125, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 43
7 (5, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 44
7 (5, 10.125, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 45
7 // Face 12
7 (5, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 46
7 (5, 10.125, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 47
7 (5 + (10 * sin(m_fTheta)), 0, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 48
7 (5 + (10 * sin(m_fTheta)), 10, 12.5 - (10 * cos(m_fTheta)), 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, // vertex 49
7 
7 // Get a pointer to the index buffer indices and lock the index buffer
7 m_pIndexBuffer->Lock(0, m_dwNumOfIndices * sizeof(WORD), (void**)&pBufferIndices, D3DLOCK_READONLY);
7 
7 // For each triangle, count the number of times each vertex is used and
7 // add together the normals of faces that share a vertex
7 for(i = 0; i < m_dwNumOfIndices; i += 3)
7 {
7    dwVertex1 = pBufferIndices[i];
7    dwVertex2 = pBufferIndices[i + 1];
7    dwVertex3 = pBufferIndices[i + 2];
7    
7 // Get the triangle normal from the vertices
7    vNormal = GetTriangleNormal(&D3DXVECTOR3(cvVertices[dwVertex1].x, cvVertices[dwVertex1].y, cvVertices[dwVertex1].z),
7                              &D3DXVECTOR3(cvVertices[dwVertex2].x, cvVertices[dwVertex2].y, cvVertices[dwVertex2].z),
7                              &D3DXVECTOR3(cvVertices[dwVertex3].x, cvVertices[dwVertex3].y, cvVertices[dwVertex3].z));
7    
7    pNumOfSharedPolygons[dwVertex1]++;
7    pNumOfSharedPolygons[dwVertex2]++;
7    pNumOfSharedPolygons[dwVertex3]++;
7    
7    pSumVertexNormal[dwVertex1].x += vNormal.x;
7    pSumVertexNormal[dwVertex1].y += vNormal.y;
7    pSumVertexNormal[dwVertex1].z += vNormal.z;
7    pSumVertexNormal[dwVertex2].x += vNormal.x;
7    pSumVertexNormal[dwVertex2].y += vNormal.y;
7    pSumVertexNormal[dwVertex2].z += vNormal.z;
7    pSumVertexNormal[dwVertex3].x += vNormal.x;
7    pSumVertexNormal[dwVertex3].y += vNormal.y;
7    pSumVertexNormal[dwVertex3].z += vNormal.z;
7 
7 }
void SetStageAngle(float Theta) {
    m_fTheta = Theta;
    UpdateVertices();
    return true;
}

bool SetPosition(float x, float y, float z) {
    m_xPos = x;
    m_yPos = y;
    // Clean up
    delete pNumOfSharedPolygons;
    delete pSumVertexNormal;
    pNumOfSharedPolygons = NULL;
    pSumVertexNormal = NULL;
    return true;
}

FUNCTION: SetStageAngle
Used to set the stage angle
//@param Theta float the angle to set the stage
//@return bool

FUNCTION: SetPosition
Used to set the stage position
//@param x float the x coordinate position
//@param y float the y coordinate position
//@param z float the z coordinate position
//@return bool

c:\Thesis\RNAVis\Stage.cpp

    m_zPos = z;
    UpdateVertices();
    return true;
}

// FUNCTION: SetTexture
// Used to set the texture from a disk file
//@param szTextureFilePath const char* the path to the texture file
//@return bool
bool Stage::SetTexture(const char *szTextureFilePath)
{
    if(FAILED(D3DXCreateTextureFromFileA(m_pD3DDevice, szTextureFilePath, &
        m_pTexture)))
    {
        return false;
    }
    return true;
}

// FUNCTION: SetTextureFromBitmap
// Used to create the texture from a bitmap in memory
//@param hBitmap HBITMAP handle to a bitmap
//@return bool
bool Stage::SetTextureFromBitmap(HBITMAP hBitmap)
{
    if(m_pTexture == NULL)
    {
        D3DXCreateTexture(m_pD3DDevice, 1024, 1280, 1, 0, D3DFMT_A8R8G8B8, DPOOL_MANAGED, &m_pTexture);
        //D3DXCreateTexture(m_pD3DDevice, 1024, 1280, 1, D3DUSAGE_AUTOGENMIPMAP, D3DFMT_A8R8G8B8, DPOOL_MANAGED, &m_pTexture);
    }
    if(hBitmap == NULL)
    {
        return m_pTexture;
    }
    return hBitmap;
    D3DSURFACE_DESC d3dsd;
    m_pTexture->GetLevelDesc(0, &d3dsd);
    SIZE size = {d3dsd.Width, d3dsd.Height};
    D3DLOCKED_RECT rcLockedRect = { 0 };
    RECT rc = { 0, 0, size.cx, size.cy };
    m_pTexture->LockRect(0, &rcLockedRect, &rc, D3DLOCK_DISCARD);
    BITMAP bmpX;
    ZeroMemory(&bmpX, sizeof(bmpX));
    GetObject(hBitmap, sizeof(bmpX), &bmpX);
    BYTE* pTextureBits = (BYTE*)rcLockedRect.pBits;
    GetBitmapBits(hBitmap, bmpX.bmpWidthBytes * bmpX.bmpHeight, pTextureBits);
    m_pTexture->UnlockRect(0);
    //BYTE* pTextureBits = (BYTE*)rcLockedRect.pBits;
    //DWORD* pBitmapBits = (DWORD*)bmp_info.bmBits;
    //DWORD* pBitmapBits = (DWORD*)GlobalAlloc(GPTR, bmpX.bmpWidthBytes * bmpX.
    //    bmHeight); //allocate memory for image byte buffer
    //DWORD* pBitmapBits = new DWORD[bmpX.bmpWidthBytes * bmpX.bmpHeight]; // Create
Memory for Bitmap Bits
// DWORD* pBitmapBits = (DWORD*)malloc(sizeof(DWORD) * (bmpX.bmpWidthBytes * bmpX.bmpHeight));
//GetBitmapBits(hBitmap, bmpX.bmpWidthBytes * bmpX.bmpHeight, pBitmapBits);
BYTE* pPixels = (BYTE*)pLineTextureBits;
for (int j = 0; j < size.cy; j++)
{
    for (int i = 0; i < size.cx; i++)
    {
        pPixels = *pBitmapBits;
        pPixels++;
        pBitmapBits++;
    }
    pLineTextureBits += rcLockedRect.Pitch;
}
delete pBitmapBits; // Free Dynamic Memory
//free(pBitmapBits);
//GlobalFree(pBitmapBits);
//m_pTexture->UnlockRect(0);

m_pD3DDevice->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);
//m_pD3DDevice->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
//m_pD3DDevice->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR);
//m_pD3DDevice->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_ANISOTROPIC);
//m_pD3DDevice->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_ANISOTROPIC);
//m_pD3DDevice->SetSamplerState(0, D3DSAMP_MIPFILTER, D3DTEXF_ANISOTROPIC);

FUNCTION: SetMaterial
Used to set the material properties used in the visualization

@param rgbaDiffuse D3DCOLORVALUE
@param rgbaAmbient D3DCOLORVALUE
@param rgbaSpecular D3DCOLORVALUE
@param rgbaEmissive D3DCOLORVALUE
@param rPower float
@return bool

bool Stage::SetMaterial(D3DCOLORVALUE rgbaDiffuse, D3DCOLORVALUE rgbaAmbient, D3DCOLORVALUE rgbaSpecular, D3DCOLORVALUE rgbaEmissive, float rPower)
{
    //Set the RGBA for diffuse light reflected from this material.
    m_matMaterial.Diffuse = rgbaDiffuse;
    //Set the RGBA for ambient light reflected from this material.
    m_matMaterial.Ambient = rgbaAmbient;
    //Set the color and sharpness of specular highlights for the material.
    m_matMaterial.Specular = rgbaSpecular;
    m_matMaterial.Power = rPower;
    //Set the RGBA for light emitted from this material.
    m_matMaterial.Emissive = rgbaEmissive;

    return true;
class CStack
{
public:
   typedef struct item_type {
      int i;
      int j;
   } Item_type;
   bool IsEmpty();
   void Push(Item_type item);
   void Pop(Item_type *item);
   CStack();
   virtual ~CStack();
private:
   typedef struct node_tag {
      Item_type info;
      struct node_tag *next;
   } Node_type;
   Node_type *top;
   Node_type* MakeNode(Item_type item);
   void PushNode(Node_type *node_ptr);
   void PopNode(Node_type **node_ptr);
};
#include "StdAfx.h"
#include "Stack.h"

// FUNCTION: CStack
// Default Constructor
// @param undefined void
// @return void
CStack::CStack()
{
    top = NULL; // Initialize top of stack
}

// FUNCTION: ~CStack
// Default Destructor
// @param undefined void
// @return void
CStack::~CStack()
{
    while (top != NULL) {
        Node_type *node_ptr;
        node_ptr = top;
        top = node_ptr->next;
        delete node_ptr;
    }
}

// FUNCTION: Push
// Used to make a new node with item and push it onto stack
// @param item Item_type the item to add to the stack
// @return void
void CStack::Push(Item_type item) {
    PushNode(MakeNode(item));
}

// FUNCTION: MakeNode
// Used to make a new node and insert item
// @param item Item_type the item to add
// @return Node_type*
CStack::Node_type* CStack::MakeNode(Item_type item) {
    Node_type *p;
    if ((p = new Node_type) == NULL)
        throw "CStack: Out of Memory";
    else {
        p->info = item;
        p->next = NULL;
    }
    return p;
}

// FUNCTION: PushNode
// Used to push node onto the linked stack
void CStack::PushNode(Node_type *node_ptr) {
    if (node_ptr == NULL)
        throw "CStack: Attempted to push a nonexistent node";
    else {
        node_ptr->next = top;
        top = node_ptr;
    }
}

void CStack::Pop(Item_type *item) {
    Node_type *node_ptr;
    PopNode(&node_ptr);
    *item = node_ptr->info;
    delete node_ptr;
}

void CStack::PopNode(Node_type **node_ptr) {
    if (top == NULL)
        throw "CStack: Empty Stack";
    else {
        *node_ptr = top;
        top = (*node_ptr)->next;
    }
}

bool CStack::IsEmpty() {
    return top == NULL;
}
Copyright (c) 2006, All Rights Reserved
Project: SJSU Masters Project
File: RNAGraphBMP.cpp
Purpose: To draw the secondary structure of RNA into a bitmap surface using windows GDI
Start Date: 8/1/2006
Programmer: Brandon Hunter

This program is partially based on a program called plt22ps.c by Darrin Stewart Feb 26, 1998

#pragma once
#include "rnagraph.h"
#include "RNA.h"

class RNAGraphBMP :
public RNAGraph
{
private:
  float m_line_length;
  float m_init_view_x; // Initial View Matrix, x dimension
  float m_init_view_y; // Initial View Matrix, y dimension
  COLORREF m_current_color;
  float m_font_height;
  float m_line_width;
  float m_origin_x, m_origin_y;
  float m_user_tran_x, m_user_tran_y;
  float m_view_x, m_view_y;
  float m_dpiY; // Dots Per Inch, y dimension
  float m_xoff, m_yoff;

  float CalcFontHeight(float char_size, float scale);
  void remove_quotes(char *string);
  void DrawLine(HDC hdc, float x1, float y1, float x2, float y2);
  void DrawCircle(HDC hdc, float radius, float x, float y);
  void DrawLabel(HDC hdc, float angle, float x, float y, int number);
  void DrawCenteredText(HDC hdc, float x, float y, char *string);
  void DrawBase(HDC hdc, float x, float y, char base);
  void DrawLoopNumber(HDC hdc, float x, float y, int number);
  void DrawTitle(HDC hdc, char *title, int BMPSizeX, int BMPSizeY);
  float user_to_screen_x(float x);
  float user_to_screen_y(float y);

public:
  RNAGraphBMP(float MinBaseSeparation);
  ~RNAGraphBMP(void);

  void Draw(HDC hdc, float BMPScale, float xsize, float ysize, bool mark_loops, bool /*draw bases, float csz, int label_rate, bool dot_pairs, int mosaicx, int mosaicy, float glob_rot, RNA *pRNA);*/
#include "StdAfx.h"
#include "RNAGraphBMP.h"
#include <math.h>

// FUNCTION: RNAGraphBMP
// Default Constructor
// @param MinBaseSeparation float the minimum base separation
// @return void
RNAGraphBMP::RNAGraphBMP(float MinBaseSeparation) : RNAGraph(MinBaseSeparation) {
    // Initialize Variables
    m_line_length = 1.1f; // Initial View Matrix, x dimension
    m_init_view_x = 1.0f; // Initial View Matrix, y dimension
    m_origin_x = 0.0; // Set the origin of the screen coordinates
    m_origin_y = 0.0;
    m_user_tran_x = 0.0;
    m_user_tran_y = 0.0;
    m_line_width = 0.0;
}

// FUNCTION: ~RNAGraphBMP
// Default Destructor
// @param undefined void
// @return void
RNAGraphBMP::~RNAGraphBMP(void) {
}

// FUNCTION: Draw
// Used to draw the planar graph on the device context
// screen coordinates=view matrix*(user coordinates-user_trans_vector)+screen_trans_vec
// @param hdc HDC handle to the bitmap device context
// @param BMPScale float the scale size of the bitmap
// @param xsize float the x coordinate of the bitmap size
// @param ysize float the y coordinate of the bitmap size
// @param mark_loops bool flag to determine if loop numbers should be drawn
// @param draw_bases bool float to determine if base symbols should be drawn
// @param csz float character size
// @param label_rate int the rate at which to label the sequence
// @param dot_pairs bool flag to determine if a dot or line is drawn between bases
// @param mosaicx int the number of frames in x dimension
// @param mosaicy int the number of frames in y dimension
// @param glob_rot float the global rotation factor
// @param pRNA RNA* pointer to an RNA class
// @return void
void RNAgraphBMP::Draw(HDC hdc, float BMPscale, float xsize, float ysize, bool
mark_loops, bool draw_bases, float csz, int label_rate, bool dot_pairs, int
mosaicx, int mosaicy, float glob_rot, RNA *pRNA) {

int i, mate, imx, imy, istart;
float xmin, xmax, ymin, ymax, scalex, scaley, scale, scalecsz;
float xmx, ymx;
float xn, yn, r, x1, y1, x2, y2, xs, ys, ct, st, xr, yr;
Loop* lp;
#define okx(x) (m_user_tran_x - scale <= (x) && (x) <= xmx + scale)
#define oky(y) (m_user_tran_y - scale <= (y) && (y) <= ymx + scale)

if(label_rate < 0) // Make sure label_rate is valid
    label_rate = 0;
if(mosaicx < 1) // Make sure mosaic is valid
    mosaicx = 1;
if(mosaicy < 1)
    mosaicy = 1;
if(BMPscale < 0.05) {
    BMPscale = 0.05f;
    //printf("\n The scale was too small, was reset to 0.05\n");
}
if(BMPscale > 1.00) {
    BMPscale = 1.0;
    //printf("\n The scale was too large, was reset to 1.0\n");
}

//m_dpiX = GetDeviceCaps(hdc, LOGPIXELSX);
//m_dpiY = GetDeviceCaps(hdc, LOGPIXELSY); // 96.0f;
ct = cos(glob_rot * GetPi() / 180.0f);
st = sin(glob_rot * GetPi() / 180.0f);
for (i = 0; i <= GetNBase(); i++) {
    xr = GetBases()[i].x * ct + GetBases()[i].y * st;
    yr = GetBases()[i].y * ct - GetBases()[i].x * st;
    GetBases()[i].x = xr;
    GetBases()[i].y = yr;
}
for (i = 0; i < GetLoopCount(); i++) {
    lp = GetLoops() + i;
    xr = lp->x * ct + lp->y * st;
    yr = lp->y * ct - lp->x * st;
    lp->x = xr;
    lp->y = yr;
}
xmax = ymagmax = -GetBigNum();
xmin = ymin = GetBigNum();
for (i = 0; i <= GetNBase(); i++) {
    if (GetBases()[i].x > GetAnum() - 100.0) {
        //printf("\nError -- base %d position is undefined.\n",i);
    }
    else {
        xmin = minf2(xmin, GetBases()[i].x);
        ymagmax = maxf2(ymagmax, GetBases()[i].y);
        xmin = minf2(xmin, GetBases()[i].x);
        ymagmax = maxf2(ymagmax, GetBases()[i].y);
    }
}
scalex = (xmax - xmin) / xsize / mosaicx;
scaley = (ymagmax - ymin) / ysize / mosaicy;
scale = maxf2(scalex, scaley) * 1.2f; // The 1.02 makes padding around the image
m_view_x = m_init_view_x / scale; // Set the view matrix, depends on naview input
m_view_y = m_init_view_y / scale;
// Scale the view matrix
if(BMPScale != 0.0) {
    m_view_x = m_view_x / BMPScale;
    m_view_y = m_view_y / BMPScale;
}
else

    // Check to find offsets for centering
xs = xsize * scale;
ys = ysize * scale;
m_xoff = (xs * mosaicx - xmax + xmin) / 2.0f;
m_yoff = (ys * mosaicy - ymax + ymin) / 2.0f;

    // Set the Font to scaled size
m_font_height = CalcFontHeight(csz, scale);

for (imx = 1; imx <= mosaicx; imx++) {
    for (imy = 1; imy <= mosaicy; imy++) {
        if (GetTitle()[0] != '\0') {
            if (imx == (mosaicx + 1) / 2 && imy == 1) {
                DrawTitle(hdc, GetTitle(), xs, ys);
            }
        }
    }
    m_user_tran_x = xmin + (imx - 1) * xs - m_xoff; // Set user translation vector
    m_user_tran_y = ymin + (imy - 1) * ys - m_yoff;
    xmx = xmin + imx * xs - m_xoff;
    ymx = ymin + imy * ys - m_yoff;
    m_current_color = RGB(0, 0, 0); // Set Color to Black

    if (GetBases()[0].x != GetANum()) {
        DrawCircle(hdc, 0.0f, GetBases()[0].x, GetBases()[0].y); // Draw the starting black circle
    }

    // SEQUENCE LINES
    if (draw_bases)
        scalecsz = scale * csz * 1.8f;
    else
        scalecsz = 0.0;

    for (i = 0; i <= GetNBase() - 1; i++) {
        if (GetBases()[i].x != GetANum() & GetBases()[i + 1].x != GetANum()) {
            if (okx(GetBases()[i].x) && oky(GetBases()[i].y) || okx(GetBases()[i + 1].x) && oky(GetBases()[i + 1].y)) {
                pRNA->GetPlanarPos()[i].PlanarX = user_to_screen_x(GetBases()[i].x); // record postions for line from matrix to planar graph
                pRNA->GetPlanarPos()[i].PlanarY = user_to_screen_y(GetBases()[i].y); // record postions for line from matrix to planar graph
                xn = GetBases()[i + 1].x - GetBases()[i].x;
                yn = GetBases()[i + 1].y - GetBases()[i].y;
                r = sqrt(xn * xn + yn * yn);
                if (r > scalecsz) {
                    x1 = GetBases()[i].x + xn * scalecsz / 2.0f;
                    y1 = GetBases()[i].y + yn * scalecsz / 2.0f;
                    x2 = GetBases()[i + 1].x - xn * scalecsz / 2.0f;
                    y2 = GetBases()[i + 1].y - yn * scalecsz / 2.0f;

                    DrawLine(hdc, x1, y1, x2, y2); // Draw line
                }
            }
        }
    }
// BASE PAIRING LINES WITH BASE PAIRS
#define draw_lbp \
if (dot_pairs) { \
x1 = (GetBases()[i].x + GetBases()[mate].x) / 2.0f; \
y1 = (GetBases()[i].y + GetBases()[mate].y) / 2.0f; \
if (okx(x1) && oky(y1)) { \
    DrawCircle(hdc, 0.0, x1, y1); \
} \
} \
else { \
xn = GetBases()[mate].x - GetBases()[i].x; \
yn = GetBases()[mate].y - GetBases()[i].y; 
if (r > scalecsz) { 
    x1 = GetBases()[i].x + xn * scalecsz / 2.0f; 
    y1 = GetBases()[i].y + yn * scalecsz / 2.0f; 
    x2 = GetBases()[mate].x - xn * scalecsz / 2.0f; 
    y2 = GetBases()[mate].y - yn * scalecsz / 2.0f; 
    if (okx(x1) && oky(y1)) || okx(x2) && oky(y2)) { 
        DrawLine(hdc, x1, y1, x2, y2); 
    } 
} \
}

m_current_color = RGB(255, 0, 0); // Set Color to Red 
for (i = 0; i <= GetNBase(); i++) { 
    if ((mate = GetBases()[i].mate) && i < mate) { 
        if (GetBases()[i].x != GetANum() && GetBases()[mate].x != GetANum()) { 
            if (GetBases()[i].name == 'G' && GetBases()[mate].name == 'C') || GetBases()[i].name == 'C' && GetBases()[mate].name == 'G') { 
                draw_lbp; 
            } 
        } 
    } 
} \

m_current_color = RGB(0, 0, 255); // Set Color to Blue 
for (i = 0; i <= GetNBase(); i++) { 
    if ((mate = GetBases()[i].mate) && i < mate) { 
        if (GetBases()[i].x != GetANum() && GetBases()[mate].x != GetANum()) { 
            if (!GetBases()[i].name == 'G' && GetBases()[mate].name == 'C' || GetBases()[i].name == 'C' && GetBases()[mate].name == 'G')) { 
                draw_lbp; 
        } 
    } 
} \

m_current_color = RGB(0, 0, 0); // Set Color to Black 
if (label_rate > 0) { 
    // LABELS 
    istart = label_rate * (1 + GetBases()[1].hist_num / label_rate) - 1; 
    GetBases()[1].hist_num += 1; 
    for (i = istart; i <= GetNBase(); i += label_rate) { 
        if (GetBases()[i].x != GetANum()) { 
            if (okx(GetBases()[i].x) && oky(GetBases()[i].y)) { 
                float dx, dy, angle; 
                int ia; 
                if (i == GetNBase()) { 
                    dx = GetBases()[i].x - GetBases()[i-1].x; 
                    dy = GetBases()[i].y - GetBases()[i-1].y; 
                } 
            } 
        } 
    } 
}
else {
    dx = GetBases()[i+1].x - GetBases()[i].x;
    dy = GetBases()[i+1].y - GetBases()[i].y;
    angle = atan2(dy, dx) / GetDTor() - 90.0f;
    ia = angle;
    GetBases()[i].hist_num);
}

if (mark_loops) {
    // LOOP LABELS
    for (i = 0; i < GetLoopCount(); i++) {
        lp = GetLoops() + i;
        if (okx(lp->x) && oky(lp->y))
            DrawLoopNumber(hdc, lp->x, lp->y, lp->number);
    }
}

if (draw_bases) {
    // BASES
    for (i = 1; i <= GetNBase(); i++) {
        if (GetBases()[i].x != GetANum()) {
            if (okx(GetBases()[i].x) && oky(GetBases()[i].y)) {
                [i].name);
                }
        }
    }
}

// FUNCTION: CalcFontHeight
// Used to scale the font size
// @param char_size float the character size
// @param scale float the scale factor
// @return float
float RNAGraphBMP::CalcFontHeight(float char_size, float scale) {
    float nHeight; // in points
    nHeight = MulDiv(PointSize, GetDeviceCaps(hDC, LOGPIXELSY), 72);
    nHeight = ((char_size * m_dpiY) / 72) / scale;
    if(nHeight > 29.0) // Put an upper bound on the font size
        nHeight = 29.0;
    return nHeight;
}

/*
* Remove quotes and remove line feeds
*/

/*
* FUNCTION: remove_quotes
* @param string char* the string to remove the characters from
* @return void
void RNAGraphBMP::remove_quotes(char *string) {
    int i, len;
    len = (int)strlen(string);
    for(i = 0; i < len; i++) {
        if(string[i] == '"')
```cpp
void RNAGraphBMP::DrawTitle(HDC hdc, char *title, int BMPSizeX, int BMPSizeY) {
    int nHeight = 15.0f;
    remove_quotes(title); // remove unwanted characters
    HFONT hFont = CreateFont(nHeight, 0, 0, 0, FW_SEMIBOLD, FALSE, FALSE, FALSE, ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, FF_DONTCARE, TEXT("Arial"));
    HFONT hOldFont = (HFONT)SelectObject(hdc, hFont);
    int len = strlen(title);
    SIZE sizeRect;
    GetTextExtentPointA(hdc, title, len, &sizeRect);
    TextOutA(hdc, (BMPSizeX / 2) - (sizeRect.cx / 2), BMPSizeY - sizeRect.cy - 10, title, len);
    SelectObject(hdc, hOldFont);
    DeleteObject(hFont);
}

void RNAGraphBMP::DrawLine(HDC hdc, float x1, float y1, float x2, float y2) {
    float bmp_x1, bmp_x2, bmp_y1, bmp_y2;
    float line_width;
    bmp_x1 = user_to_screen_x(x1);
    bmp_x2 = user_to_screen_x(x2);
    bmp_y1 = user_to_screen_y(y1);
    bmp_y2 = user_to_screen_y(y2);
    if(m_current_color == RGB(0, 0, 0))
        line_width = 1.0f; // 0.69f * m_view_x + 0.1f; // set width for black lines
    else
        line_width = 2.0f; // 1.2f * m_view_x + 0.15f; // set width for non black lines
    if(m_line_width != line_width) {
        m_line_width = line_width;
    }
    HPEN hPen = CreatePen(PS_SOLID, m_line_width, m_current_color);
    HPEN hOldPen = (HPEN)SelectObject(hdc, hPen);
    MoveToEx(hdc, (int)bmp_x1, (int)bmp_y1, NULL);
    ```
LineTo(hdc, (int)bmp_x2, (int)bmp_y2);
SelectObject(hdc, hOldPen);
DeleteObject(hPen);
}

// FUNCTION: DrawCircle
// Draws a circle at current user coordinates, radius of zero is converted to look nice
// @param hdc HDC handle to bitmap device context
// @param radius float radius of circle
// @param x float x coordinate of circle center
// @param y float y coordinate of circle center
// @return void
void RNAGraphBMP::DrawCircle(HDC hdc, float radius, float x, float y) {
    float bmp_x, bmp_y, radius_in_p;
    bmp_x = user_to_screen_x(x);
    bmp_y = user_to_screen_y(y);
    // basepair circles result from zero radius
    if(radius != 0.0) {
        radius_in_p = (m_dpiY * radius * m_view_x);
        MoveToEx(hdc, (int)bmp_x, (int)bmp_y, NULL);
    }
    else // set radius of circle
    radius_in_p = (m_dpiY * 0.002f * m_view_x);
    // set a maximum on radius size
    if(radius_in_p > 20) {
        radius_in_p = 20;
    }
    // fprintf(psfp,"%.2f %.2f %.2f 0 360 arc closepath fill \n", ps_x,ps_y, radius_in_p);
    //HPEN hPen = CreatePen(PS_SOLID, m_line_width, m_current_color);
    //HPEN hOldPen = (HPEN)SelectObject(hdc, hPen);
    HBRUSH hBrush = CreateSolidBrush(m_current_color);
    HBRUSH hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);
    Ellipse(hdc, (int)(bmp_x - radius_in_p), (int)(bmp_y - radius_in_p), (int)(bmp_x +
        radius_in_p), (int)(bmp_y + radius_in_p));
    SelectObject(hdc, hOldBrush);
    DeleteObject(hBrush);
    //SelectObject(hdc, hOldPen);
    //DeleteObject(hPen);
}

// FUNCTION: DrawLabel
// Draw string with left character at x,y in user coordinates given user coordinates
// @param hdc HDC handle to bitmap device context
// @param angle float the current angle
// @param x float the x coordinate
// @param y float the y coordinate
// @param number int the number to darw
// @return void
void RNAGraphBMP::DrawLabel(HDC hdc, float angle, float x, float y, int number) {
    float xl, y1; // location to place label
    float bmp_sx, bmp_xl, bmp_x2;
    float bmp_sy, bmp_y1, bmp_y2;
    float line_width;
    int len;
    char strNumber[15];
    itoa(number, strNumber, 10);
    len = strlen(strNumber);
        // place label m_line_length away based on angle
        x1 = x + m_line_length * (float)cos((double)(angle * -3.14159 / 180.0));  // convert to radians
        y1 = y + m_line_length * (float)sin((double)(angle * -3.14159 / 180.0));
        bmpsx = user_to_screen_x(x1);
        bmpsy = user_to_screen_y(y1);
        x1 = x + 0.65f * m_line_length * (float)cos((double)(angle * -3.14159 / 180.0));  // convert to radians
        y1 = y + 0.65f * m_line_length * (float)sin((double)(angle * -3.14159 / 180.0));
        bmpx1 = user_to_screen_x(x1);
        bmpy1 = user_to_screen_y(y1);
        x1 = x + 0.25f * m_line_length * (float)cos((double)(angle * -3.14159 / 180.0));  // convert to radians
        y1 = y + 0.25f * m_line_length * (float)sin((double)(angle * -3.14159 / 180.0));
        bmpx2 = user_to_screen_x(x1);
        bmpy2 = user_to_screen_y(y1);
        linewidth = 1.0f;  // 0.4f * m_view_x * + 0.10f; // sets line width for labels
        if(m_line_width != line_width) {
            m_line_width = line_width;

            HFONT hFont = CreateFont(m_font_height, 0, 0, 0, FW_THIN, FALSE, FALSE, FALSE, ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, FF_DONTCARE, TEXT("Arial"));
            HFONT hOldFont = (HFONT)SelectObject(hdc, hFont);
            SIZE objSize;
            GetTextExtentPointA(hdc, strNumber, len, &objSize);
            TextOutA(hdc, bmpsx - (objSize.cx / 2), bmpsy - (objSize.cy / 2), strNumber, len);
            SelectObject(hdc, hOldFont);
            DeleteObject(hFont);
        }

        // FUNCTION: DrawCenteredText
        // Draw text centered at user coordinates x,y. This code is used for most
        // text. Converts to screen coordinates
        //
        // @param hdc HDC the bitmap device context
        // @param x float the x coordinate
        // @param y float the y coordinate
        // @param string char* the string to draw
        // @return void
        void RNAGraphBMP::DrawCenteredText(HDC hdc, float x, float y, char *string) {
            float screen_x, screen_y;
            screen_x = user_to_screen_x(x);
            screen_y = user_to_screen_y(y);
            HFONT hFont = CreateFont(m_font_height, 0, 0, 0, FW_THIN, FALSE, FALSE, FALSE, ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, FF_DONTCARE, TEXT("Arial"));
            HFONT hOldFont = (HFONT)SelectObject(hdc, hFont);
            int len = strlen(string);
            SIZE objSize;
            GetTextExtentPointA(hdc, string, len, &objSize);
            TextOutA(hdc, screen_x - (objSize.cx / 2), screen_y - (objSize.cy / 2), string, len);
            SelectObject(hdc, hOldFont);
            DeleteObject(hFont);
        }
    }
// FUNCTION: DrawBase
// Used to Draw the base symbol
//@param hdc HDC handle to bitmap device context
//@param x float x coordinate
//@param y float y coordinate
//@param base char base symbol
//@return void
void RNAGraphBMP::DrawBase(HDC hdc, float x, float y, char base) {
    char strBase[2];
    strBase[0] = base;
    strBase[1] = '\0';
    DrawCenteredText(hdc, x, y, strBase);
}

// FUNCTION: DrawLoopNumber
// Used to draw the loop number
//@param hdc HDC handle to bitmap device context
//@param x float the x coordinate
//@param y float the y coordinate
//@param number int the number to darw
//@return void
void RNAGraphBMP::DrawLoopNumber(HDC hdc, float x, float y, int number) {
    char strNumber[15];
    itoa(number, strNumber, 10);
    DrawCenteredText(hdc, x, y, strNumber);
}

// FUNCTION: user_to_screen_x
// Used to convert the x coordinate from user to screen coordinates
//@param x float the number to convert
//@return float
float RNAGraphBMP::user_to_screen_x(float x) {
    float x_screen;
    x_screen = m_view_x * (x - m_user_tran_x) + m_origin_x;
    return x_screen;
}

// FUNCTION: user_to_screen_y
// Used to convert the y coordinate from user to screen coordinates
//@param y float the number to convert
//@return float
float RNAGraphBMP::user_to_screen_y(float y) {
    float y_screen;
    y_screen = m_view_y * (y - m_user_tran_y) + m_origin_y;
    return y_screen;
}
class RNAGraph
{
    #define maxiter 500
    #define type_alloc(type) (type *) malloc(sizeof(type))
    #define struct_alloc(struct_name) type_alloc(struct struct_name)

    // Forward Declarations
    typedef struct region_tag;
    typedef struct connection_tag;
    typedef struct loop_tag;
    typedef struct base_tag;
    typedef struct radloop_tag;

typedef struct region_tag {
    int start1, end1, start2, end2;
} Region;

typedef struct connection_tag {
    struct loop_tag *loop;
    struct region_tag *region;
    int start, end; // Start and end form the 1st base pair of the region.
    float xrad, yrad, angle;
    bool extruded; // True if segment between this connection and
    // the next must be extruded out of the circle
    bool broken; // True if the extruded segment must be drawn long.
} Connection;

typedef struct loop_tag {
    int nconnection;
    struct connection_tag **connections;
    int number;
    int depth;
    bool mark;
    float x, y, radius;
} Loop;
typedef struct base_tag {
    char name;
    int mate, hist_num;
    float x, y;
    bool extracted;
    struct region_tag *region;
} Base;

typedef struct radloop_tag {
    float radius;
    int loopnumber;
    struct radloop_tag *next, *prev;
} Radloop;

private:
float m_pi;
float m_dtor;
float m_bignum;
float m_anum;
float m_minbaseseparation;
struct Base *bases;
int nbases, nregion, loop_count;
struct Loop *anchor, *root, *loops;
struct Region *regions;
struct Radloop *rlphead, *rliptail;

#define LINEMX 500
char line[LINEMX], title[LINEMX];

//struct loop_tag *construct_loop();
void CalcGraph();
void find_regions();
void dump_loops();
void find_central_loop();
bool connected_connection(struct Connection *cp, struct Connection *cpnext);
int find_ic_middle(int icstart, int icend, struct Connection *anchor_connection, struct Connection *acp, struct Loop *lp);
void determine_depths();
int depth(struct Loop *lp);
void traverse_loop(struct Loop *lp, struct Connection *anchor_connection);
void determine_radius(struct Loop *lp, float lenCut);  
void generate_region(struct Connection *cp);
void construct_extruded_segment(struct Connection *cp, struct Connection *cpnext);
void construct_circle_segment(int start, int end);
void find_center_for_arc(int n, float b, float *hp, float *thetap);
void trim(char *st, int *stlenp);
void chcnbl(char *st, int stlen);
struct loop_tag *construct_loop(int ibase);
void FreeMemory(void);
void CalcSequenceWithoutLoops(void);

defined:
RNAGraph(float MinBaseSeparation);
~RNAGraph(void);

void LoadFromCTFile(char *inputfile);
void LoadFromRNASequence(char* Title, char* Sequence, int Length, int* Pairing);
void Write_PCL2_Output(char *dev, char *outputfile, float xsize, float ysize, bool mark_loops, bool draw_bases, float csz, int label_rate, bool dot_pairs, int mosaicx, int mosaicy, float glob_rot);  
float minf2(float x1, float x2);
float maxf2(float x1, float x2);

void SetMinBaseSeparation(float MinBaseSeparation);
float GetMinBaseSeparation(void);
void ChangeLoopRadii();
float GetPi(void);
struct base_tag *GetBases(void);
int GetNBase(void);
struct loop_tag *GetLoops();
float GetBigNum(void);
float GetANum(void);
char *GetTitle(void);
float GetDTor(void);
c:\Thesis\RNAVis\RNAGraph.cpp

1 /*****************************************************************************************/
2 * Copyright (c): 2006, All Rights Reserved
3 * Project: SJSU Masters Project
4 * File: RNAGraph.cpp
5 * Purpose: To calculate a graph of RNA secondary structure
6 *
7 * Start Date: 8/1/2006
8 * Programmer: Brandon Hunter
9 *
10 ***************************************************************************************
11 *
12 * This program is based on a program called NAVIEW written
13 * by Robert E. Bruccoleri
14 *
15 * NAVIEW -- A program to make a modified radial drawing of an RNA
16 * secondary structure.
17 *
18 * Copyright (c) 1988 Robert E. Bruccoleri
19 * Copying of this software, in whole or in part, is permitted
20 * provided that the copies are not made for commercial purposes,
21 * appropriate credit for the use of the software is given, this
22 * copyright notice appears, and notice is given that the copying
23 * is by permission of Robert E. Bruccoleri. Any other copying
24 * requires specific permission.
25 *
26 * See R. Bruccoleri and G. Heinrich, Computer Applications in the
28 ***************************************************************************************
29 *
30 * Modified August 2006 by Brad Hunter.
31 * Turned the code into a class
32 * Modified code so that small sequences with no loops are handled
33 */
34 */
35 #include "StdAfx.h"
36 #include "RNAGraph.h"
37 #include <math.h>
38 #include <stdio.h>
39
40 // FUNCTION: RNAGraph
41 // Default Constructor
42 //
43 // @param MinBaseSeparation float the minimum base separation
44 // @return void
45 RNAGraph::RNAGraph(float MinBaseSeparation) {
46     SetMinBaseSeparation(MinBaseSeparation);
47     m_pi = 3.141592653589793f;
48     m_dtor = (3.141592653589793f / 180.0f);
49     m_bignum = 1.7e38f;
50     m_anum = 9999.0f;
51     bases = NULL;
52     regions = NULL;
53     loops = NULL;
54     rlphead = rlptail = NULL; // Initialize the loop radius linked list
55 }
56
57 // FUNCTION: ~RNAGraph
58 // Default Destructor
59 //
60 // @param undefined void
61 // @return void
62 RNAGraph::~RNAGraph(void) {
63     FreeMemory(); // Free the dynamically allocated memory
64 }
// FUNCTION: FreeMemory
// Used to free memory that was dynamically allocated

void RNAGraph::FreeMemory(void) {
    if(bases != NULL) {
        free(bases);
        bases = NULL;
    }
    if(regions != NULL) {
        free(regions);
        regions = NULL;
    }
    if(loops != NULL) {
        for(int x = 0; x < loop_count; x++) {
            for(int y = 0; y < loops[x].nconnection; y++) {
                free(*((loops[x].connections + y));
            }
            free(loops[x].connections);
        }
        free(loops);
        loops = NULL;
    }
    // Free the radloop list
    struct Radloop *rlp = rlphead;
    while(rlp) {
        rlp = rlp->next;
        free(rlphead);
        rlphead = rlp;
    }
}

// FUNCTION: LoadFromRNASequence
// Used to load the sequence into the class

void RNAGraph::LoadFromRNASequence(char* Title, char* Sequence, int Length, int* Pairing) {
    FreeMemory(); // Free Memory From Previous Calculation
    nbase = Length;
    strcpy(title, Title);
    bases = (struct Base *) malloc(sizeof(struct Base) * (Length + 1));
    // Set up an origin
    bases[0].name = 'o';
    bases[0].mate = 0;
    bases[0].hist_num = 0;
    bases[0].extracted = false;
    bases[0].x = m_anum;
    bases[0].y = m_anum;
    for(int i = 1; i <= Length; i++) {
        bases[i].name = Sequence[i - 1];
        if(Pairing[i - 1] == i - 1)
            bases[i].mate = 0;
        else
            bases[i].mate = Pairing[i - 1] + 1;
bases[i].hist_num = i + 1;
bases[i].extracted = false;
bases[i].x = m_anum;
bases[i].y = m_anum;
}
CalcGraph();

// FUNCTION: LoadFromCTFile
// Used to load the sequence from a CT-File
//@param    inputfile char*    the file path to where the ct file exists
//@return   void
void RNAGraph::LoadFromCTFile(char *inputfile) {
    FILE *inf;
    char *cp;
    int i, linel;
    FreeMemory(); // Free Memory From Previous Calculation
    inf = fopen(inputfile, "r");
    if (inf == NULL) {
        throw "RNAGraph: Unable to open input file";
    }
    if ((cp = fgets(line, LINEMX, inf)) == NULL) {
        throw "RNAGraph: Unable to read header record";
    }
    if (
        (cp = fgets(line, LINEMX, inf)) == NULL) {
        throw "RNAGraph: Unable to read header record";
    }
    line[76] = '\0';
    linel = strlen(line);
    chcnbl(line, linel);
    trim(line, &linel);
    line[linel] = '\0';
    sscanf(line, "%5d", &nbase);
    //printf("%d bases specified in file. Title is \n%\n", nbase, &line[5]);
    strcpy(title, &line[5]);
    bases = (struct Base *) malloc(sizeof(struct Base) * (nbase + 1));
    // Set up an origin
    bases[0].name = 'o';
    bases[0].mate = 0;
    bases[0].hist_num = 0;
    bases[0].extracted = false;
    bases[0].x = m_anum;
    bases[0].y = m_anum;
    for (i = 0; i <= nbase; i++)
        if ((cp = fgets(line, LINEMX, inf)) == NULL)
            break;
    sscanf(line, "%*d %c %*d %*d %d", &bases[i].name, &bases[i].mate, &bases[i].x, hist_num);
    // The code in this program depends on mate being false (namely zero) where there is no mate.
    bases[i].extracted = false;
    bases[i].x = m_anum;
    bases[i].y = m_anum;
}
if (--i != nbase) {
    throw "RNAGraph: Number of bases read from file, doesn't match CT File header record";
    nbase = i;
}
CalcGraph();
// FUNCTION:  CalcGraph
// Used to calculate the secondary structure
// @param undefined  void
// @return  void
void RNAGraph::CalcGraph()
{
    // Constructing tree of loops
    nregion = 0;
    loop_count = 0;
    regions = (struct Region *) malloc(sizeof(struct Region) * (nbase + 1));
    find_regions();
    if(nregion == 0) {
        CalcSequenceWithoutLoops();
    }
    else {
        loops = (struct Loop *) malloc(sizeof(struct Loop) * (nbase + 1));
        anchor = construct_loop(0);
        find_central_loop();
        //if(debug) dump_loops();
        traverse_loop(root, NULL); // Constructing the drawing
    }
}

// FUNCTION:  find_regions
// Identifies the regions in the structure
// @param undefined  void
// @return  void
void RNAGraph::find_regions() {
    int i, mate, nbase;
    bool *mark;
    nbase = nbase + 1;
    mark = (bool *) malloc(sizeof(int) * nbase);
    for (i = 0; i < nbase; i++) {
        mark[i] = false;
        nregion = 0;
        for(i = 0; i <= nbase; i++) {
            if((mate = bases[i].mate) && !mark[i]) {
                regions[nregion].start1 = i;
                regions[nregion].end2 = mate;
                mark[i] = true;
                mark[mate] = true;
                bases[i].region = bases[mate].region = &regions[nregion];
                for(i++, mate--; i < mate && bases[i].mate == mate; i++, mate--) {
                    mark[i] = mark[mate] = true;
                    bases[i].region = bases[mate].region = &regions[nregion];
                }
                regions[nregion].end1 = --i;
                regions[nregion].start2 = mate + 1;
                //if(debug) {
                //    if(nregion == 0) printf("\nRegions are:\n"");
                //    printf("Region %d is %d-%d and %d-%d with gap of %d.\n",
                //        nregion+1,regions[nregion].start1,regions[nregion].end1,
                //        regions[nregion].start2,regions[nregion].end2,
                //        regions[nregion].start2-regions[nregion].end1+1);
                //}
                nregion++;
            }
        }
    }
    free(mark);
}

// FUNCTION:  construct_loop
// Starting at residue ibase, recursively construct the loop containing
// said base and all deeper bases
// @param ibase int the base location
// @return struct loop_tag*
struct RNAGraph::loop_tag *RNAGraph::construct_loop(int ibase) {
    int i, mate;
    struct Loop *retloop, *lp;
    struct Connection *cp;
    struct Region *rp;
    struct Radloop *rlp;

    retloop = &loops[loop_count++];
    retloop->nconnection = 0;
    retloop->connections = (struct Connection **) malloc(sizeof(struct Connection *));
    retloop->depth = 0;
    retloop->number = loop_count;
    retloop->radius = 0.0;

    for (rlp = rlphead; rlp; rlp = rlp->next)
        if (rlp->loopnumber == loop_count)
            retloop->radius = rlp->radius;

    i = ibase;

    do {
        if ((mate = bases[i].mate) != 0) {
            rp = bases[i].region;
            if (!bases[rp->start1].extracted) {
                if (i == rp->start1) {
                    bases[rp->start1].extracted = true;
                    bases[rp->end1].extracted = true;
                    bases[rp->end2].extracted = true;
                    lp = construct_loop(rp->end1 < nbase ? rp->end1 + 1 : 0);
                }
                else if (i == rp->start2) {
                    bases[rp->start2].extracted = true;
                    bases[rp->end2].extracted = true;
                    bases[rp->end1].extracted = true;
                    lp = construct_loop(rp->end2 < nbase ? rp->end2 + 1 : 0);
                }
            }
            else {
                throw "RNAGraph: Error detected in construct_loop. i = %d not found in region table";
            }

            retloop->connections = (struct Connection **) realloc(retloop->
                connections, (++retloop->nconnection + 1) * sizeof(struct Connection *));
            retloop->connections[retloop->nconnection] = NULL;
            cp->loop = lp;
            cp->region = rp;
            if (i == rp->start1) {
                cp->start = rp->start1;
                cp->end = rp->end2;
            }
            else {
                cp->start = rp->start2;
                cp->end = rp->end1;
            }
            cp->extruded = false;
            cp->broken = false;
            lp->connections = (struct Connection **) realloc(lp->connections, (++
                lp->nconnection + 1) * sizeof(struct Connection *));
            lp->connections[lp->nconnection-1] = cp = struct_alloc(Connection);
            lp->connections[lp->nconnection] = NULL;
            cp->loop = retloop;
        }
    } while (i != mate);
    return retloop;
}
```c
    cp->region = rp;
    if (i == rp->start1) {
        cp->start = rp->start2;
        cp->end = rp->end1;
    }
    else {
        cp->start = rp->start1;
        cp->end = rp->end2;
    }
    cp->extruded = false;
    cp->broken = false;
    i = mate;
    if (++i > nbase) i = 0;
    }
    while (i != ibase);
    return retloop;
}

// FUNCTION: dump_loops
// Displays all the loops
//@param undefined void
//@return void
void RNAGraph::dump_loops() {
    int il, ilp, irp;
    struct Loop *lp;
    struct Connection *cp,**cpp;
    //printf("\nRoot loop is %d\n", (root-loops)+1);
    for (il=0; il < loop_count; il++) {
        lp = &loops[il];
        //printf("Loop %d has %d connections:\n", il+1, lp->nconnection);
        for (cpp = lp->connections; cp = *cpp; cpp++) {
            ilp = (lp->loop - loops) + 1;
            irp = (lp->region - regions) + 1;
            //printf(" Loop %d Region %d (%d-%d)\n", ilp, irp, cp->start, cp->end);
        }
    }

    // FUNCTION: find_central_loop
    // Find node of greatest branching that is deepest
    //@param undefined void
    //@return void
    void RNAGraph::find_central_loop() {
        struct Loop *lp;
        int maxconn, maxdepth, i;
        determine_depths();
        maxconn = 0;
        maxdepth = -1;
        for(i = 0; i < loop_count; i++) {
            lp = &loops[i];
            if(lp->nconnection > maxconn) {
                maxdepth = lp->depth;
                maxconn = lp->nconnection;
                root = lp;
            }
            else if(lp->depth > maxdepth && lp->nconnection == maxconn) {
                maxdepth = lp->depth;
                root = lp;
            }
        }
    }
```
void RNAGraph::determine_depths() {
    struct Loop *lp;
    int i, j;
    for(i = 0; i < loop_count; i++) {
        lp = &loops[i];
        for(j = 0; j < loop_count; j++)
            loops[j].mark = false;
        lp->depth = depth(lp);
    }
}

void RNAGraph::CalcSeqeuceWithoutLoops(void) {
    float dt, angleinc, angleadjust, sumn, sumd, radius;
    dt = 2.0f * m_pi;
    angleinc = dt / (nbase + 1);
    sumn = dt * (1.0f / nbase + 1.0f);
    sumd = dt * dt / nbase;
    radius = sumn / sumd;
    angleadjust = 0.5f * m_pi;
    for(int i = 0; i <= nbase; i++) {
        bases[i].x = radius * cos(angleadjust + (angleinc * i));
        bases[i].y = radius * sin(angleadjust + (angleinc * i));
    }
}
// FUNCTION: traverse_loop
// This is the workhorse of the display program. The algorithm is
// recursive based on processing individual loops. Each base pairing
// region is displayed using the direction given by the circle diagram,
// and the connections between the regions is drawn by equally spaced
// points. The radius of the loop is set to minimize the square error
// for lengths between sequential bases in the loops. The "correct"
// length for base links is 1. If the least squares fitting of the
// radius results in loops being less than 1/2 unit apart, then that
// segment is extruded.
// The variable, anchor_connection, gives the connection to the loop
// processed in an previous level of recursion.

void RNAGraph::traverse_loop(struct Loop *lp, struct Connection *anchor_connection) {
    float xs, ys, xe, ye, xn, yn, angleinc, r;
    float radius, xc, yc, xo, yo, astart, aend, a;
    struct Connection *cp, *cpnext, *cpp, *cpprev;
    int i, j, n, ic;
    float da, maxang;
    int count, icstart, icend, icmiddle, icroot;
    bool done, done_all_connections, rooted;
    int sign;
    float midx, midy, nrx, nry, mx, my, vx, vy, dotmv, nmidx, nmidt;
    int icstartl, icup, icdown, icnext, icdirection;
    float dan, dx, dy, rr;
    float cpx, cpy, cpnextx, cpnexty, cnx, cny, rcn, rc, lnx, lny, rl, ac, acn, sx, sy, dcp;
    int imaxloop;

    angleinc = 2 * m_pi / (nbase + 1);
    acp = NULL;
    icroot = -1;
    for(cpp = lp->connections, ic = 0; cp = *cpp; cpp++, ic++) {
        // xs = cos(angleinc*cp->start);
        // ys = sin(angleinc*cp->start);
        // xe = cos(angleinc*cp->end);
        // ye = sin(angleinc*cp->end);
        xs = -sin(angleinc * cp->start);
        ys = cos(angleinc * cp->start);
        xe = -sin(angleinc * cp->end);
        ye = cos(angleinc * cp->end);
        xn = ye - ys;
        yn = xs - xe;
        r = sqrt(xn * xn + yn * yn);
        cp->xrad = xn / r;
        cp->yrad = yn / r;
        cp->angle = atan2(yn, xn);
        if(cp->angle < 0.0) cp->angle += 2 * m_pi;
        if(anchor_connection != NULL && anchor_connection->region == cp->region) {
            acp = cp;
            icroot = ic;
        }
    }
    set_radius:
    determine_radius(lp, m_minbaseseparation);
    radius = lp->radius;
    if (anchor_connection == NULL)
        xc = yc = 0.0;
    else {
        
    }
x = (bases[acp->start].x + bases[acp->end].x) / 2.0f;
y = (bases[acp->start].y + bases[acp->end].y) / 2.0f;
x = xo - radius * acp->xrad;
y = yo - radius * acp->yrad;
}

// The construction of the connectors will proceed in blocks of
// two connected connectors, where a connected connector pairs means
// two connectors that are forced out of the drawn circle because they
// are too close together in angle.

// First, find the start of a block of connected connectors
if(icroot == -1)
icstart = 0;
else icstart = icroot;
cp = lp->connections[icstart];
count = 0;
//if (debug) printf("Now processing loop %d\n",lp->number);
done = false;
do {
    j = icstart - 1;
    if(j < 0) j = lp->nconnection - 1;
    cpprev = lp->connections[j];
    if(!connected_connection(cpprev,cp)) {
        done = true;
    }
    else {
        icstart = j;
        cp = cpprev;
    }
    if(++count > lp->nconnection) {
        // Here everything is connected. Break on maximum angular separation
        // between connections.
        maxang = -1.0;
        for(ic = 0; ic < lp->nconnection; ic++) {
            j = ic + 1;
            if(j >= lp->nconnection) j = 0;
            cp = lp->connections[ic];
            cpnext = lp->connections[j];
            ac = cpnext->angle - cp->angle;
            if(ac < 0.0) ac += 2 * pi;
            if(ac > maxang) {
                maxang = ac;
                imaxloop = ic;
            }
        }
        icend = imaxloop;
        icstart = imaxloop + 1;
        if(icstart >= lp->nconnection) icstart = 0;
        cp = lp->connections[icend];
        cp->broken = true;
        done = true;
    }
} while(!done);
done_all_connections = false;
icstart1 = icstart;
//if (debug) printf("Icstart1 = %d\n",icstart1);
while(!done_all_connections) {
    count = 0;
    done = false;
    icend = icstart;
    rooted = false;
    while(!done) {
        cp = lp->connections[icend];
        }
if(icend == icroot) rooted = true;
j = icend + 1;
if(j >= lp->nconnection) {
    j = 0;
}
cpnext = lp->connections[j];
if(connected_connection(cp,cpnext)) {
    if(++count >= lp->nconnection)
        break;
    icend = j;
}
else {
    done = true;
}
}

icmiddle = find_ic_middle(icstart,icend,anchor_connection,acp,lp);
ic = icup = icdown = icmiddle;
// if (debug)
// printf("IC start = %d middle = %d end = %d\n", icstart,icmiddle,icend);
done = false;
direction = 0;
while (!done) {
    if(direction < 0) {
        ic = icup;
    }
    else if(direction == 0) {
        ic = icmiddle;
    }
    else {
        ic = icdown;
    }
    if(ic >= 0) {
        cp = lp->connections[ic];
        if(anchor_connection == NULL || acp != cp) {
            if(direction == 0) {
                astart = cp->angle - asin(1.0f / 2.0f / radius);
                aend = cp->angle + asin(1.0f / 2.0f / radius);
                bases[cp->start].x = xc + radius * cos(astart);
                bases[cp->start].y = yc + radius * sin(astart);
                bases[cp->end].x = xc + radius * cos(aend);
                bases[cp->end].y = yc + radius * sin(aend);
            }
            else if(direction < 0) {
                j = ic + 1;
                if(j >= lp->nconnection)
                    j = 0;
                cp = lp->connections[j];
                cpnext = lp->connections[j];
                cpx = cp->xrad;
                cpy = cp->yrad;
                ac = (cp->angle + cpnext->angle) / 2.0f;
                if(cp->angle > cpnext->angle)
                    ac -= m_pi;
                cnx = cos(ac);
                cny = sin(ac);
                lnx = cny;
                lny = -cnx;
                da = cpnext->angle - cp->angle;
                if(da < 0.0)
                    da += 2 * m_pi;
                if(cp->extruded) {
                    if(da <= m_pi / 2)
                        r1 = 2.0;
                    else
                        r1 = 1.5;
                }
                else {
rl = 1.0;
}
bases[cp->end].x = bases[cpnext->start].x + rl * lnx;
bases[cp->end].y = bases[cpnext->start].y + rl * lny;

bases[cp->start].x = bases[cp->end].x + cpx;
bases[cp->start].y = bases[cp->end].y - cpx;
}
else {
    j = ic - 1;
    if(j < 0)
        j = lp->nconnection - 1;
    cp = lp->connections[j];
cpnext = lp->connections[ic];
cpnextx = cpnext->xrad;
cpnexty = cpnext->yrad;
ac = (cp->angle + cpnext->angle) / 2.0f;
    if(cp->angle > cpnext->angle)
        ac -= m_pi;
    cnx = cos(ac);
cny = sin(ac);
    lnx = -cny;
lny = cnx;
da = cpnext->angle - cp->angle;
    if(da < 0.0)
        da += 2 * m_pi;
    if(cp->extruded) {
        if(da <= m_pi / 2)
            rl = 2.0;
        else
            rl = 1.5;
    }
else {
    rl = 1.0;
}
}
_bases[cpnext->start].x = bases[cp->end].x + rl * lnx;
bases[cpnext->start].y = bases[cp->end].y + rl * lny;
_bases[cpnext->end].x = bases[cpnext->start].x - cpnexty;
bases[cpnext->end].y = bases[cpnext->start].y + cpnextx;
}
}

if(direction < 0) {
    if(icdown == icend) {
        icdown = -1;
    }
else {
    if(icdown >= 0) {
        if(++icdown >= lp->nconnection) {
            icdown = 0;
        }
}
    direction = 1;
}
else {
    if(icup == icstart)
        icup = -1;
else {
    if(icup >= 0) {
        if(--icup < 0) {
            icup = lp->nconnection - 1;
        }
    }
    direction = -1;
}
    done = icup == -1 && icdown == -1;
icnext = icend + 1;
if(icnext >= lp->nconnection)
    icnext = 0;
if(icend != icstart && (!icstart == icstart1 && icnext == icstart1)) {
    /*
     * Move the bases just constructed (or the radius) so
     * that the bisector of the end points is radius distance
     * away from the loop center.
     */
    if(!icnext)
        icnext = 0;
    cpnext = icend;
    if(icnext)
        cpnext = icstart;
    cp = lp->connections[icstart];
    cpnext = lp->connections[icend];
    dx = bases[cpnext->end].x - bases[cp->start].x;
    dy = bases[cpnext->end].y - bases[cp->start].y;
    midx = bases[cp->start].x + dx / 2.0f;
    midy = bases[cp->start].y + dy / 2.0f;
    rr = sqrt(dx * dx + dy * dy);
    mx = dx / rr;
    my = dy / rr;
    vx = xc - midx;
    vy = yc - midy;
    rr = sqrt(dx * dx + dy * dy);
    vx /= rr;
    vy /= rr;
    dotmv = vx * mx + vy * my;
    nrx = dotmv * mx - vx;
    nry = dotmv * my - vy;
    rr = sqrt(nrx * nrx + nry * nry);
    nrx /= rr;
    nry /= rr;
    // Determine which side of the bisector the center should be.
    dx = bases[cp->start].x - xc;
    dy = bases[cp->start].y - yc;
    ac = atan2(dy, dx);
    if(ac < 0.0)
        ac += 2 * m_pi;
    dx = bases[cpnext->end].x - xc;
    dy = bases[cpnext->end].y - yc;
    acn = atan2(dy, dx);
    if(acn < 0.0)
        acn += 2 * m_pi;
    if(acn - ac > m_pi)
        sign = -1;
    else
        sign = 1;
    nmidx = xc + sign * radius * nrx;
    nmidy = yc + sign * radius * nry;
    if(rooted) {
        xc -= nmidx - midx;
        yc -= nmidy - midy;
    }
    else {
        for(ic=icstart; ++ic >= lp->nconnection ? (ic = 0) : 0) {
            cp = lp->connections[ic];
            i = cp->start;
            bases[i].x += nmidx - midx;
            bases[i].y += nmidy - midy;
            i = cp->end;
            bases[i].x += nmidx - midx;
            bases[i].y += nmidy - midy;
            if(ic == icend)
                break;
        }
    }
    icstart = icnext;
```c

done_all_connections = icstart == icstart1;

for (ic = 0; ic < lp->nconnection; ic++) {
    cp = lp->connections[ic];
    j = ic + 1;
    if (j >= lp->nconnection)
        j = 0;
    cpnext = lp->connections[j];
    dx = bases[cp->end].x - xc;
    dy = bases[cp->end].y - yc;
    rc = sqrt((dx * dx + dy * dy));
    ac = atan2(dy, dx);
    if (ac < 0.0)
        ac += 2 * m_pi;
    dx = bases[cpnext->start].x - xc;
    dy = bases[cpnext->start].y - yc;
    acn = atan2(dy, dx);
    if (acn < 0.0)
        acn += 2 * m_pi;
    if (acn < ac)
        acn += 2 * m_pi;
    dan = acn - ac;
    dcp = cpnext->angle - cp->angle;
    if (dcp <= 0.0)
        dcp += 2 * m_pi;
    if (fabs(dan - dcp) > m_pi) {
        if (cp->extruded) {
            //printf(\"Warning from traverse_loop. Loop %d has crossed regions\n\", cp->number);
            goto set_radius; // Forever shamed
        }

        if (cp->extruded) {
            construct_extruded_segment(cp, cpnext);
        }
    }
    else {
        n = cpnext->start - cp->end;
        if (n < 0)
            n += nbase + 1;
        angleinc = dan / n;
        for (j = 1; j < n; j++) {
            i = cp->end + j;
            if (i > nbase)
                i -= nbase + 1;
            a = ac + j * angleinc;
            rr = rc + (rcn - rc) * (a - ac) / dan;
            bases[i].x = xc + rr * cos(a);
            bases[i].y = yc + rr * sin(a);
        }
    }
    for (ic=0; ic < lp->nconnection; ic++) {
        if (icroot != ic) {
            cp = lp->connections[ic];
            generate_region(cp);
            traverse_loop(cp->loop, cp);
        }
    }
    n = 0;
    sx = 0.0;
    sy = 0.0;
    for (ic = 0; ic < lp->nconnection; ic++) {
        j = ic + 1;
```
The radius is selected by a least squares procedure where the sum of the squares of the deviations of length from the ideal value of \( l \) is used as the error function.

```c++
void determine_radius(float minbaseseparation, struct Loop *lp) {
  float rt2_2 = 0.7071068f;
  count = 0;
  do {
    mindit = 1.0e10;
    for (sumn=0.0, summ=0.0, i=0; i < lp->nconnection; i++) {
      cp = lp->connections[i];
      j = i + 1;
      if (j >= lp->nconnection)
        j = 0;
      cpnext = lp->connections[j];
      end = cp->end;
      start = cpnext->start;
      if (start < end)
        start += nbase + 1;
      dt = cpnext->angle - cp->angle;
      if (dt <= 0.0)
        dt += 2 * m_pi;
      if (!lp->extruded)
        ci = start - end;
      else {
        if (dt <= m_pi / 2)
          ci = 2.0;
        else
          ci = 1.5;
      }
      sumn += dt * (1.0f / ci + 1.0f);
    }
    count ++;
    if (count > 1)
      mindit = sumn / count;
  } while (mindit > rt2_2);
  // set radius
  for (j = cp->start; j != cpnext->start; j++)
    if (j > nbase)
      j -= nbase + 1;
    n++;
15

```cpp
sumd += dt * dt / ci;

dit = dt / ci;
if (dit < mindit && !cp->extruded && ci > 1.0) {
    mindit = dit;
    imindit = i;
}

radius = sumn / sumd;
if (radius < rt2_2)
    radius = rt2_2;
if (mindit * radius < minbaseseparation) {
    lp->connections[imindit]->extruded = true;
}
}
while (mindit * radius < minbaseseparation);
if (lp->radius > 0.0)
    radius = lp->radius;
else
    lp->radius = radius;

// FUNCTION: connected_connection
// Determines if the connections cp and cpnext are connected
//@param cp struct Connection* the pointer to the connection
//@param cpnext struct Connection* the pointer to the next connection
//@return bool
bool RNAGraph::connected_connection(struct Connection *cp, struct Connection *cpnext)
{
    if (cp->extruded) {
        return true;
    }
    else if (cp->end+1 == cpnext->start) {
        return true;
    }
    else {
        return false;
    }
}

// FUNCTION: find_ic_middle
// Finds the middle of a set of connected connectors. This is normally
// the middle connection in the sequence except if one of the connections
// is the anchor, in which case that connection will be used.
//@param icstart int the icstart position
//@param icent int the icent position
//@param anchor_connection struct Connection* the anchor connection pointer
//@param acp struct Connection* pointer to connection
//@param lp struct Loop* pointer to loop
//@return int
int RNAGraph::find_ic_middle(int icstart, int icend, struct Connection *anchor_connection, struct Connection *acp, struct Loop *lp) {
    int count, ret, ic, i;
    bool done;
    count = 0;
    ret = -1;
    ic = icstart;
    done = false;
    while (!done) {
        if (count++ > lp->nconnection * 2) {
            throw "RNAGraph: Infinite loop detected in find_ic_middle";
        }
        if (anchor_connection != NULL && lp->connections[ic] == acp) {
            ret = ic;
        }
    }
```
c:\Thesis\RNAVis\RNAGraph.cpp

993 }
994     done = ic == icend;
995     if (++ic >= lp->nconnection) {
996         ic = 0;
997     }
998 }
999 if (ret == -1) {
1000     for (i=1, ic=icstart; i<(count+1)/2; i++) {
1001         if (++ic >= lp->nconnection) ic = 0;
1002     }
1003     ret = ic;
1004 }
1005 return ret;
1006 }
1007 // FUNCTION: generate_region
1008 // Generates the coordinates for the base pairing region of a connection
1009 // given the position of the starting base pair.
1010 //
1011 // @param  cp  struct Connection*  pointer to Connection structure
1012 // @return  void
1013 void RNAGraph::generate_region(struct Connection *cp) {
1014     int l, start, end, i, mate;
1015     struct Region *rp;
1016
1017     rp = cp->region;
1018     l = 0;
1019     if (cp->start == rp->start1) {
1020         start = rp->start1;
1021         end = rp->end1;
1022     } else {
1023         start = rp->start2;
1024         end = rp->end2;
1025     }
1026     if (bases[cp->start].x > m_anum - 100.0 || bases[cp->end].x > m_anum - 100.0) {
1027         throw "RNAGraph: Bad region passed to generate_region. Coordinates not defined";
1028     }
1029     for (i=start+1; i<=end; i++) {
1030         l++;
1031         bases[i].x = bases[cp->start].x + l * cp->xrad;
1032         bases[i].y = bases[cp->start].y + l * cp->yrad;
1033         mate = bases[i].mate;
1034         bases[mate].x = bases[cp->end].x + l * cp->xrad;
1035         bases[mate].y = bases[cp->end].y + l * cp->yrad;
1036     }
1037 }
1038 // FUNCTION: construct_circle_segment
1039 // Draws the segment of residue between the bases numbered start
1040 // through end, where start and end are presumed to be part of a base
1041 // pairing region. They are drawn as a circle which has a chord given
1042 // by the ends of two base pairing regions defined by the connections.
1043 //
1044 // @param  start  int  the start of the circle segment
1045 // @param  end    int  the end of the circle segment
1046 // @return  void
1047 void RNAGraph::construct_circle_segment(int start, int end) {
1048     float dx, dy, rr, h, angleinc, midx, midy, xn, yn, nrx, nry, mx, my, a;
1049     int l, j, i;
1050
1051     dx = bases[end].x - bases[start].x;
1052     dy = bases[end].y - bases[start].y;
1053     rr = sqrt(dx*dx + dy*dy);
1054     l = end - start;
1055     if (l < 0)
if (rr >= 1) {
  dx /= rr;
  dy /= rr;
  for (j = 1; j < 1; j++) {
    i = start + j;
    if (i > nbase)
      i -= nbase + 1;
    bases[i].x = bases[start].x + dx * (float)j / (float)1;
    bases[i].y = bases[start].y + dy * (float)j / (float)1;
  }
}
else {
  find_center_for_arc(l-l, rr, &h, &angleinc);
  dx /= rr;
  dy /= rr;
  midx = bases[start].x + dx * rr / 2.0f;
  midy = bases[start].y + dy * rr / 2.0f;
  xn = dy;
  yn = -dx;
  nrx = midx + h * xn;
  nry = midy + h * yn;
  mx = bases[start].x - nrx;
  my = bases[start].y - nry;
  rr = sqrt(mx * mx + my * my);
  a = atan2(my, mx);
  for (j = 1; j < 1; j++) {
    i = start + j;
    if (i > nbase)
      i -= nbase + 1;
    bases[i].x = nrx + rr * cos(a + j * angleinc);
    bases[i].y = nry + rr * sin(a + j * angleinc);
  }
}

// FUNCTION: construct_extruded_segment
// Constructs the segment between cp and cpnext as a circle if possible.
// However, if the segment is too large, the lines are drawn between
// the two connecting regions, and bases are placed there until the
// connecting circle will fit.
//@param cp struct Connection* pointer to a Connection structure
//@param cpnext struct Connection* pointer to a Connection structure
//@return void
void RNAGraph::construct_extruded_segment(struct Connection *cp, struct Connection * cpnext) {
  float astart, aend1, aend2, aave, dx, dy, a1, a2, ac, rr, da, dac;
  int start, end, n, nstart, nend;
  bool collision;

  astart = cp->angle;
  aend2 = aend1 = cpnext->angle;
  if (aend2 < astart)
    aend2 += 2 * m_pi;
  aave = (astart + aend2) / 2.0f;
  start = cp->end;
  end = cpnext->start;
  n = end - start;
  if (n < 0)
    n += nbase + 1;
  da = cpnext->angle - cp->angle;
  if (da < 0.0) {
    da += 2 * m_pi;
  }
  if (n == 2)
    construct_circle_segment(start, end);
function

1125    else {
1126        dx = bases[end].x - bases[start].x;
1127        dy = bases[end].y - bases[start].y;
1128        rr = sqrt(dx * dx + dy * dy);
1129        dx /= rr;
1130        dy /= rr;
1131        if (rr >= 1.5 && da <= m_pi / 2) {
1132            nstart = start + 1;
1133            if (nstart > nbase)
1134                nstart -= nbase + 1;
1135            nend = end - 1;
1136            if (nend < 0)
1137                nend += nbase + 1;
1138            bases[nstart].x = bases[start].x + 0.5f * dx;
1139            bases[nstart].y = bases[start].y + 0.5f * dy;
1140            bases[nend].x = bases[end].x - 0.5f * dx;
1141            bases[nend].y = bases[end].y - 0.5f * dy;
1142            start = nstart;
1143            end = nend;
1144        }
1145        do {
1146            collision = false;
1147            construct_circle_segment(start, end);
1148            nstart = start + 1;
1149            if (nstart > nbase)
1150                nstart -= nbase + 1;
1151            dx = bases[nstart].x - bases[start].x;
1152            dy = bases[nstart].y - bases[start].y;
1153            a1 = atan2(dy, dx);
1154            if (a1 < 0.0)
1155                a1 += 2 * m_pi;
1156            dac = a1 - astart;
1157            if (dac < 0.0)
1158                dac += 2 * m_pi;
1159            if (dac > m_pi)
1160                collision = true;
1161            nend = end - 1;
1162            if (nend < 0)
1163                nend += nbase + 1;
1164            dx = bases[nend].x - bases[end].x;
1165            dy = bases[nend].y - bases[end].y;
1166            a2 = atan2(dy, dx);
1167            if (a2 < 0.0)
1168                a2 += 2 * m_pi;
1169            dac = aend1 - a2;
1170            if (dac < 0.0)
1171                dac += 2 * m_pi;
1172            if (dac > m_pi)
1173                collision = true;
1174            if (collision) {
1175                ac = minf2(aave, astart + 0.5f);
1176                bases[nstart].x = bases[start].x + cos(ac);
1177                bases[nstart].y = bases[start].y + sin(ac);
1178                start = nstart;
1179                ac = maxf2(aave, aend2 - 0.5f);
1180                bases[nend].x = bases[end].x + cos(ac);
1181                bases[nend].y = bases[end].y + sin(ac);
1182                end = nend;
1183                n -= 2;
1184            }
1185        } while (collision && n > 1);
1186    }
1187 }
1188
1189 // FUNCTION:  find_center_for_arc
1190 // Given n points to be placed equidistantly and equiangularly on a
1191 // polygon which has a chord of length, b, find the distance, h, from the
// midpoint of the chord for the center of polygon. Positive values
// mean the center is within the polygon and the chord, whereas
// negative values mean the center is outside the chord. Also, the
// radial angle for each polygon side is returned in theta.
// The procedure uses a bisection algorithm to find the correct
// value for the center. Two equations are solved, the angles
// around the center must add to 2*pi, and the sides of the polygon
// excluding the chord must have a length of 1.

// @param n int
// @param b float
// @param hp float*
// @param thetap float*
// @return void
void RNAGraph::find_center_for_arc(int n, float b, float *hp, float *thetap) {
    float h, hhi, hlow, r, disc, theta, e, phi;
    int iter;
    hhi = (n + 1) / m_pi;
    hlow = - hhi - b / (n + 1 - b);
    iter = 0;
    do {
        h = (hhi + hlow) / 2.0f;
        r = sqrt(h * h + b * b / 4.0f);
        disc = 1.0f - 1.0f / 2.0f / (r * r);
        if (fabs(disc) > 1.0) {  // disc;
            throw "RNAGraph: Unexpected large magnitude discriminant = %g"; // disc;
        }
        theta = acos(disc);
        phi = acos(h / r);
        e = theta * (n + 1) + 2 * phi - 2 * m_pi;
        if (e > 0.0) {
            hlow = h;
        }
    } else {
        hhi = h;
    }
    while (fabs(e) > 0.0001 && ++iter < maxiter);
    if (iter >= maxiter) {
        //printf("Iteration failed in find_center_for_arc\n");
        h = 0.0;
        theta = 0.0;
    }
    *hp = h;
    *thetap = theta;
}

// FUNCTION: Write PLT2 Output
// Writes the coordinates as a PLT2 command stream

// @param dev char* PLT2 device string
// @param outputfile char* The file name of the PLT2 file to create
// @param xsize float The x coordinate size
// @param ysize float The y coordinate size
// @param mark_loops bool flag to determine if loop numbers should be drawn
// @param draw_bases bool flag to determine if the base symbol should be drawn
// @param csz float the character size
// @param label_rate int the rate a which to label the bases
// @param mosaicy int number of frames in mosaic y
// @param mosaicx int number of frames in mosaic x
// @param dot_pairs int number of pairs drawn
// @param glob_rot float global rotation angle
// @return void
void RNAGraph::Write_PLT2_Output(char *dev, char *outputfile, float xsize, float ysize, bool mark_loops, bool draw_bases, float csz, int label_rate, bool dot_pairs, int mosaicx, int mosaicy, float glob_rot) {
int i, mate, imx, imy, pagecnt, istart;
float xmin, xmax, ymin, ymax, scalex, scaley, scale, scalepsz;
float xn, yn, x, y1, y2, xs, ys, xoff, yoff, ct, st, xr, yr;
struct { float *lp; }
#define okx(x) ((xmin - scale) <= (x) && (x) <= (xmax + scale))
#define oky(y) ((ymin - scale) <= (y) && (y) <= (ymx + scale))

FILE *outf;
outf = fopen(outputfile, "w");
if (outf == NULL) {
    throw "RNAGraph: Unable to open output file";
}

if(label_rate < 0) // Make sure label_rate is valid
    label_rate = 0;

if(mosaicx < 1) // Make sure mosaic is valid
    mosaicx = 1;

if(mosaicy < 1)
    mosaicy = 1;

fprintf(outf, "DEV %sn", dev);
fprintf(outf, "CSZ %10.4f\n", csz);
c = cos(glob_rot * m_pi / 180.0f);
st = sin(glob_rot * m_pi / 180.0f);
for (i = 0; i < nbase; i++) {
xr = bases[i].x * ct + bases[i].y * st;
yr = bases[i].y * ct - bases[i].x * st;
bases[i].x = xr;
bases[i].y = yr;
}
for (i = 0; i < loop_count; i++) {
lp = loops + i;
xr = lp->x * ct + lp->y * st;
yr = lp->y * ct - lp->x * st;
lp->x = xr;
lp->y = yr;
}
xmax = ymmax = -m_bignum;
xmin = ymin = m_bignum;
for (i = 0; i < nbase; i++) {
    if (bases[i].x > m_anum - 100.0f) {
        fprintf("\nError in write_plt2_output -- base %d position is undefined.\n", i);
    }
else {
xmax = maxf2(xmax, bases[i].x); 
ymax = maxf2(ymax, bases[i].y);
xmin = minf2(xmin, bases[i].x); 
ymin = minf2(ymin, bases[i].y);
}
}

scalex = (xmax - xmin) / xsize / (float) mosaicx;
scaley = (ymax - ymin) / ysize / (float) mosaicy;
scale = maxf2(scalex, scaley) * 1.02f;
// The line below was added by Darrin Stewart $$$$$$$$*/
csz = csz / scale;
// It works best when csz is .3, but numbers from .4 to .1 are reasonable*/
// The character size is now depended on the internal scaling of the image*/
// This seems to be quite accurate. */
xs = xsize * scale;
ys = ysize * scale;
xoff = (xs * mosaicx - xmax + xmin) / 2.0f;
yoff = (ys * mosaicy - ymax + ymin) / 2.0f;
fprintf(outf, "SA %g\n", scale);
forall (imx = 1; imx <= mosaicx; imx++) {
    for (imy = 1; imy <= mosaicy; imy++) {
        if (++pagecnt > 1) {
            fprintf(outf, "DUMP\n");
        }
        if (title[0] != '\0') {
            if (imx == (mosaicx + 1) / 2 && imy == 1) {
                fprintf(outf, "CSZ 0.4\n");
                fprintf(outf, "CTA %10.3f 1.0 \"%a\"\n", xsize / 2.0, title);
            }
            fprintf(outf, "CSZ %10.4f\n", csz);
        }
        imx = imx + (imx = 1) * xs - xoff;
        ymn = ymn + (imy = 1) * ys - yoff;
        ymx = ymn + imx * xs - xoff;
        fprintf(outf, "OD %10.3f %10.3f\n", xmn, ymn);
        fprintf(outf, "COLOR WHITE\n");
        if (bases[0].x != m_anum)
            fprintf(outf,"BRI 0\nMOV %10.3f %10.3f\nCIA 0.05\nCIA 0.1\nCIA 0.15\n", bases[0].x, -bases[0].y);
        fprintf(outf, "BRI 3\n");
    }
    if (draw_bases) {
        fprintf(outf, "CM BASES %d\n", nbase); /* Zuker adds nbase */
        for (i = 1; i <= nbase; i++) {
            if (bases[i].x != m_anum)
                if (okx(bases[i].x) && oky(bases[i].y))
                    fprintf(outf, "CTX %10.3f %10.3f \"c\"\n", bases[i].x, -oky(bases[i].y));
            fprintf(outf, "CM SEQUENCE LINES\n");
        }
        if (draw_bases)
            scalecsz = scale * csz / 1.8f;
        else
            scalecsz = 0.0;
        for (i = 0; i <= nbase-1; i++) {
            if (bases[i].x != m_anum && bases[i+1].x != m_anum) {
                if (okx(bases[i].x) && oky(bases[i].y) || okx(bases[i+1].x) && oky(bases[i+1].y)) {
                    xn = bases[i+1].x - bases[i].x;
                    yn = bases[i+1].y - bases[i].y;
                    r = sqrt((xn * xn) + (yn * yn));
                    if (r > scalecsz) {
                        xn /= r;
                        yn /= r;
                        x1 = bases[i].x + xn * scalecsz / 2.0f;
                        y1 = bases[i].y + yn * scalecsz / 2.0f;
                        x2 = bases[i+1].x - xn * scalecsz / 2.0f;
                        y2 = bases[i+1].y - yn * scalecsz / 2.0f;
                        fprintf(outf,"LI %10.3f %10.3f 0.0 %10.3f %10.3f 0.0\n", x1, -y1, x2, -y2);
                    }
                }
            }
        }
    }
    fprintf(outf, "CM BASE PAIRING LINES WITH BASE PAIRS\n");
    fprintf(outf, "COLOR RED\nBRI \n");
    #define draw_lbp \
    if (dot_pairs) {
        x1 = (bases[i].x + bases[mate].x) / 2.0f;
        y1 = (bases[i].y + bases[mate].y) / 2.0f;
        if (okx(x1) && oky(y1)) \
            fprintf(outf,"MOV %10.3f %10.3f 5d 5d\nCIA 0.0\n",x1,-y1,i, okx(x1));
    }
mate);}  

  else {
    x = bases[mate].x - bases[i].x;
    y = bases[mate].y - bases[i].y;
    r = sqrt(x*x + y*y);
    if (r > scalecsz) {
      x /= r;
      y /= r;
      x1 = bases[i].x + x * scalecsz / 2.0f;
      y1 = bases[i].y + y * scalecsz / 2.0f;
      x2 = bases[mate].x - x * scalecsz / 2.0f;
      y2 = bases[mate].y - y * scalecsz / 2.0f;
      if (okx(x1) && oky(y1) || okx(x2) && oky(y2)) {
        fprintf(outf,"LI %10.3f %10.3f 0.0 %10.3f 0.0 %5d %5d\n", x1, -y1, x2, -y2, i, mate);
      }
    }
  }

  for (i = 0; i <= nbase; i++) {
    if (((mate = bases[i].mate) && i < mate) {  
      if (bases[i].x != m_anum && bases[mate].x != m_anum) {
        if (!bases[i].name == 'G' && bases[mate].name == 'C' || bases[i].name == 'C' && bases[mate].name == 'G') {  
          draw_lbp;
        }
      }
    }
  }

  fprintf(outf,"COLOR MAGENTA\nBRI 1\n\n");
  for (i = 0; i <= nbase; i++) {
    if (((mate = bases[i].mate) && i < mate) {  
      if (bases[i].x != m_anum && bases[mate].x != m_anum) {
        if (!bases[i].name == 'G' && bases[mate].name == 'C' || bases[i].name == 'C' && bases[mate].name == 'G') {  
          draw_lbp;
        }
      }
    }
  }

  fprintf(outf,"COLOR WHITE\nBRI 3\n\n");
  if (label_rate > 0) {  
    fprintf(outf,"CM LABELS %d\nCSZ %10.4f\n", nbase, csz);
    istart = label_rate*(1 + bases[0].hist_num / label_rate) - bases[0].hist_num + 1;
    for (i = istart; i <= nbase; i += label_rate) {
      if (bases[i].x != m_anum) {
        if (okx(bases[i].x) && oky(bases[i].y)) {  
          float dx, dy, angle;
          int ia;
          if (i == nbase) {  
            dx = bases[i].x - bases[i-1].x;
            dy = bases[i].y - bases[i-1].y;
          } else {  
            dx = bases[i+1].x - bases[i].x;
            dy = bases[i+1].y - bases[i].y;
            angle = atan2(dy, dx) / m_dtor - 90.0f;
            ia = angle;
            fprintf(outf,"TEX %d %10.3f %10.3f \" %d\n", i, angle, bases[i].x, bases[i].hist_num);
          }
        }
      }
    }
if (mark_loops) {
    fprintf(outf,"CM LOOP LABELS\n");
    for (i = 0; i < loop_count; i++) {
        lp = loops + i;
        if (okx(lp->x) && oky(lp->y))
            fprintf(outf, "CTX %10.3f %10.3f \"d\"\n", lp->x, -lp->y, lp->number);
    }
}
}
}
}
}
}
// FUNCTION: minf
// Computes the minimum of two floating point numbers
//@param x1 float the first float
//@param x2 float the second float
//@return float
float RNAGraph::minf(float x1, float x2) {
    return x1 < x2 ? x1 : x2;
}

// FUNCTION: maxf
// Computes the maximum of two floating point numbers
//@param x1 float the first float
//@param x2 float the second float
//@return float
float RNAGraph::maxf(float x1, float x2) {
    return x1 > x2 ? x1 : x2;
}

// FUNCTION: trim
// Trims blanks off the end of st, *stlenp gives the current length of st
//@param st char* the character array to trim
//@param stlen int* the length of the character array
//@return void
void RNAGraph::trim(char *st, int *stlenp) {
    int stlen = *stlenp;
    while (stlen > 0) {
        if (st[stlen-1] != ' ') break;
        stlen --;
    }
    *stlenp = stlen;
}

// FUNCTION: chcnbl
// Converts all non-printable control characters into blanks
//@param st char* the character array to clear characters from
//@param stlen int* the length of the character array
//@return void
void RNAGraph::chcnbl(char *st, int stlen) {
    int i;
    for (i = 1; i <= stlen; i++) {
        if (*st < 32 || *st > 126)
            *st = ' ';  
st += 1;
    }
}
// FUNCTION:  setMinBaseSeparation
// Used to specify the minimum permissible separation between bases (Ex. 1.0)
//@param MinBaseSeparation float the minimum base separation
//@return void
void RNAGraph::SetMinBaseSeparation(float MinBaseSeparation) {
  //printf("A size > 1.0 will be reduced to 1.0.\n");
  MinBaseSeparation = maxf2(0.0, minf2(1.0, MinBaseSeparation));
  m_minbaseseparation = MinBaseSeparation;
}

// FUNCTION:  GetMinBaseSeparation
// Used to retrieve the minimum base separation
//@param undefined void
//@return float
float RNAGraph::GetMinBaseSeparation(void) {
  return m_minbaseseparation;
}

// FUNCTION:  ChangeLoopRadii
// Used to change the loop radii
//@param undefined void
//@return void
void RNAGraph::ChangeLoopRadii() {
  bool change_rad;
  int ilp;
  float r;
  struct Radloop *rlp;
  rlphead = rlptail = NULL;
  //change_rad = ask("Do you want to change loop radii?");
  change_rad = false;
  if (change_rad) {
    do {
      //printf("Please specify a loop number and radius. Type zeroes to quit:\n");
      r = 0.0;
      scanf("%d%f", ilp, &r);
      if (ilp > 0) {
        if (r <= 0.0) {
          //printf("Bad radius specified. It must be positive.\n");
        }
        else {
          rlp = struct_alloc(Radloop);
          rlp->radius = r;
          rlp->loopnumber = ilp;
          // Add to double list
          rlp->next = rlp->prev = NULL;
          if(rlphead == NULL)
            rlphead = rlptail = rlp;
          else {
            rlptail->next = rlp;
            rlp->prev = rlptail;
            rlptail = rlp;
            }
        }
      }
    while (ilp > 0);
  }
}

// FUNCTION:  GetPi
// Used to retrieve the value of PI

void RNAGraph::GetPi(void) {
    return m_pi;
}

struct RNAGraph::base_tag* RNAGraph::GetBases(void) {
    return bases;
}

int RNAGraph::GetNBase(void) {
    return nbase;
}

int RNAGraph::GetLoopCount(void) {
    return loop_count;
}

struct RNAGraph::loop_tag* RNAGraph::GetLoops(void) {
    return loops;
}

float RNAGraph::GetBigNum(void) {
    return m_bignum;
}

float RNAGraph::GetANum(void) {
    return m_anum;
}

char* RNAGraph::GetTitle(void) {
return
title;

// FUNCTION: GetDTor
// @param undefined void
// @return float
float RNAGraph::GetDTor(void) {
    return m_dtor;
}
#pragma once

class RNA {
private:
    typedef struct planarPos_tag {
        float PlanarX, PlanarY;
    } PlanarPos_type;

    int Length; // The Length of the RNA
    char* Title; // The title of the RNA
    char* Sequence; // The RNA sequence
    int StepPosition; // 1 based position when stepping through RNA sequence
    int* Pairing; // Array that stores pairing positions
    struct PlanarPos_type* PlanarPos; // Pointer to array of PlanarPos_type (used for drawing lines from Matrix to Planar graph)

public:
    RNA(char* Title, char* Sequence, int Length);
    ~RNA(void);

    char* getTitle(void);
    char* getSequence(void);
    int getLength(void);
    void setStepPosition(int StepPosition);
    int getStepPosition(void);
    void setPairing(int* Pairing);
    int* getPairing(void);
    struct PlanarPos_type* GetPlanarPos();
};

#include "StdAfx.h"
#include "RNA.h"

// FUNCTION: RNA
// Default Constructor
RNA::RNA(char* Title, char* Sequence, int Length)
{
    RNA::Title = Title; // Initialize sequence title
    RNA::Sequence = Sequence; // Initialize sequence
    RNA::Length = Length; // Initialize Sequence Length
    StepPosition = Length; // Initialize StepPosition to full length of RNA sequence
    Pairing = NULL;
    PlanarPos = new PlanarPos_type[Length];
}

// FUNCTION: ~RNA
// Default Destructor
RNA::~RNA(void)
{
    if(Title != NULL)
    {
        delete[] Title;
        Title = NULL;
    }
    if(Sequence != NULL)
    {
        delete[] Sequence;
        Sequence = NULL;
    }
    if(Pairing != NULL)
    {
        delete[] Pairing;
        Pairing = NULL;
    }
    if(PlanarPos != NULL)
    {
        delete[] PlanarPos;
        PlanarPos = NULL;
    }
}

// FUNCTION: getTitle
// Used to retrieve the RNA Title
char* RNA::getTitle(void)
{
    return Title;
}
// FUNCTION: getSequence
// Used to retrieve the Sequence
// @param undefined void
// @return char*
char* RNA::getSequence(void) {
    return Sequence;
}

// FUNCTION: getLength
// Used to Retrieve the length of the RNA sequence
// @param undefined void
// @return int
int RNA::getLength(void) {
    return Length;
}

// FUNCTION: setStepPosition
// Used to set the current step position within the sequence
// @param StepPosition int the step position to set
// @return void
void RNA::setStepPosition(int StepPosition) {
    if(StepPosition > Length)
        throw "Step position is greater than length of sequence";
    RNA::StepPosition = StepPosition;
}

// FUNCTION: getStepPosition
// Used to retrieve the current step position
// @param undefined void
// @return int
int RNA::getStepPosition(void) {
    return StepPosition;
}

// FUNCTION: setPairing
// Used to set the Pairing array
// @param Pairing int* the integer array that is the pairing
// @return void
void RNA::setPairing(int* Pairing) {
    if(RNA::Pairing != NULL)
        delete[] RNA::Pairing;
    RNA::Pairing = Pairing;
}

// FUNCTION: getPairing
// Used to retrieve the Pairing array
// @param undefined void
// @return int*
int* RNA::getPairing(void) {
    return Pairing;
}

// FUNCTION: GetPlanarPos
// Used to retrieve the PlanarPos structure
// @param undefined void
// @return struct planarPos_tag*
struct RNA::planarPos_tag* RNA::GetPlanarPos() {
return PlanarPos;
/*
 * Copyright (c): 2006, All Rights Reserved
 * Project:    SJSU Masters Project
 * File:      Nussinov.h
 * Purpose:   Header file for RNA progress bar
 * Start Date:  10/4/2006
 * Programmer: Brandon Hunter
 */

#pragma once

class ProgressGraph
{
public:
    ProgressGraph(void);
    ~ProgressGraph(void);

    void Draw(HDC hdc, char* Sequence, int RNALength, int StepPosition, float xsize, float ysize);
};
#include "StdAfx.h"
#include "ProgressGraph.h"
#include "math.h" // for the floor function

// FUNCTION: ProgressGraph
// Default Constructor
// @param undefined void
// @return void
ProgressGraph::ProgressGraph(void)
{
}

// FUNCTION: ~ProgressGraph
// Default Destructor
// @param undefined void
// @return void
ProgressGraph::~ProgressGraph(void)
{
}

// FUNCTION: Draw
// Used to draw the Progress Graph bitmap onto the device context
// @param hdc HDC the bitmap device context
// @param Sequence char* the sequence of nucleotide bases
// @param RNALength int the length of the RNA sequence
// @param StepPosition int the current step position
// @param xsize float the x coordinate of the bitmap size
// @param ysize float the y coordinate of the bitmap size
// @return void
void ProgressGraph::Draw(HDC hdc, char* Sequence, int RNALength, int StepPosition, 
float xsize, float ysize)
{
    float fBorder = 5.0f;
    float fCellSizeX = 20;
    float fCellSizeY = 20;
    float fCellLeft = fBorder;
    float fCellTop = fBorder;
    float xPos = fCellLeft, yPos = fCellTop;
    int iCellsPerLine = (int)floor((xsize - (fBorder * 2)) / fCellSizeX); // Number of cells that fit on a line
    int iLineCount = RNALength / iCellsPerLine; // Number of complete lines
    int iRemainder = RNALength % iCellsPerLine; // Number of cells remaining on last line

    HPEN hPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
    HPEN hOldPen = (HPEN)SelectObject(hdc, hPen);
    HBRUSH hOldBrush = CreateSolidBrush(RGB(128, 128, 128));
    HBRUSH hBrush = (HBRUSH)SelectObject(hdc, hOldBrush);
    Rectangle(hdc, 0, 0, (int)(xsize + 0.5f), (int)(ysize + 0.5f)); // Set the background color for the whole facet
    DeleteObject(hOldBrush);
    hBrush = CreateSolidBrush(RGB(255, 255, 255));
    SelectObject(hdc, hBrush);
    //for(int x = 0; x < iLineCount; x++)
64    //
65    // Rectangle(hdc, (int)xPos, (int)yPos, (int)(xPos + (fCellSizeX * iCellsPerLine) + 0.5f), (int)(yPos + fCellSizeY + 0.5f)); // Draw a box that covers the whole line
66    // yPos += fCellSizeY;
67    //}
68    //Rectangle(hdc, (int)xPos, (int)yPos, (int)(xPos + (fCellSizeX * iRemaining) + 0.5f), (int)(yPos + fCellSizeY + 0.5f)); // Draw the remaining boxes
69    SelectObject(hdc, hOldBrush);
70    DeleteObject(hBrush);
71    SelectObject(hdc, hOldPen);
72    DeleteObject(hPen);
73    // Draw the sequence
74    COLORREF Bkgrnd;
75    COLORREF OldBkgrnd;
76    COLORREF Foregrnd;
77    COLORREF OldForegrnd;
78    COLORREF OldForegrnd;
79    HFONT hFont = CreateFont((int)fCellSizeY - 2, 0, 0, 0, FW_SEMIBOLD, FALSE, FALSE, FALSE, ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, FF_DONTCARE, TEXT("Arial"));
80    HFONT hOldFont = (HFONT)SelectObject(hdc, hFont);
81    xPos = fCellLeft;
82    yPos = fCellTop;
83    for(int x = 0; x < RNASequence; x++)
84    {
85        if(x > 0 && (x % iCellsPerLine) == 0)
86        {
87            yPos += fCellSizeY;
88            xPos = fCellLeft;
89        }
90
91        char strBase[2];
92        strBase[0] = Sequence[x];
93        strBase[1] = '\0';
94        int len = strlen(strBase);
95        if(x < StepPosition)
96        {
97            Foregrnd = RGB(255, 255, 255);
98            Bkgrnd = RGB(49, 106, 197); //Bkgrnd = RGB(186, 209, 252);
99            RECT rect = {xPos, yPos, xPos + fCellSizeX, yPos + fCellSizeY};
100           HBRUSH hBrush = CreateSolidBrush(Bkgrnd);
101           HBRUSH hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);
102           FillRect(hdc, &rect, hBrush);
103           SelectObject(hdc, hOldBrush);
104           DeleteObject(hBrush);
105        }
106        else
107        {
108            Foregrnd = RGB(0, 0, 0);
109            Bkgrnd = RGB(255, 255, 255);
110           RECT rect = {xPos, yPos, xPos + fCellSizeX, yPos + fCellSizeY};
111           HBRUSH hBrush = CreateSolidBrush(Bkgrnd);
112           HBRUSH hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);
113           FillRect(hdc, &rect, hBrush);
114           SelectObject(hdc, hOldBrush);
115           DeleteObject(hBrush);
116        }
117
118        GetTextExtentPointA(hdc, strBase, len, &sizeRect);
119        OldForegrnd = SetTextColor(hdc, Foregrnd);
120        OldBkgrnd = SetBkColor(hdc, Bkgrnd);
121        TextOutA(hdc, (int)((xPos + (fCellSizeX / 2.0f)) - (sizeRect.cx / 2.0f) + 0.5f), (int)((yPos + (fCellSizeY / 2.0f)) - (sizeRect.cy / 2.0f) + 0.5f), strBase, 2}
setTextColor(hdc, OldForegrnd);
SetBkColor(hdc, OldBkgrnd);

xPos += fCellSizeX;
}
SelectObject(hdc, hOldFont);
DeleteObject(hFont);
}
# Header file for Nussinov algorithm implementation


Start Date: 8/1/2006

Programmer: Brandon Hunter

---

```c
#pragma once

#include "Stack.h" // Stack used during traceback
#include "List.h" // List to hold the traceback path
#include "RNA.h"

class Nussinov
{
    enum NussinovType
    {
        NussinovStandard = 0,
        NussinovSCFG = 1,
    }

    private:
    NussinovType mType; // Standard or SCFG
    char *RNASequence;
    int RNALength;
    int scoreMatrix[4][4]; // Scoring Matrix
    float probMatrix[4][3]; // Probability Matrix
    float probSS; // Bifurcation Probability
    int minLoopLength; // Minimum Hairpin Loop Length
    int** Matrix;
    float** CYKMatrix;
    float Infinity; // use maximum negative float for negative infinity (float = 3.4x10^-38 to 3.4x10^-38)

    template<class T> T** createMatrix(int row, int col);
    template<class T> void deleteMatrix(T** mMatrix, int row);
    int Max(int a, int b, int c, int d);
    float Max(float a, float b, float c);
    void NussinovFillStage();
    void NussinovTraceBack(RNA *pRNA);
    void NussinovFillStageCYK();
    void NussinovTraceBackCYK(RNA *pRNA);
    void SortTracebackPath();
    bool IsEqual(float num1, float num2);

    public:
    Nussinov(NussinovType mType, char *RNASequence, int RNALength);
    ~Nussinov(void);
    void setScoringMatrix(int AA, int AC, int AG, int AU, int CA, int CC, int CG, int
                          CU, int GA, int GC, int GG, int GU, int UA, int UC, int UG, int UU);
    void setMinHairpinLength(int HairpinLength);
    int getMinHairpinLength(void);
    void setProbMatrix(float aS, float cS, float gS, float uS, float aS, float Sc,
                      float Sg, float Su, float aSu, float cSg, float gSc, float uSa, float SS);
    float getProbMatrix(int row, int col);
    float getProbSS(void);
    void FillStage();
```
void TraceBack(RNA *pRNA);
List listTraceback;  // Instantiate the list which holds traceback path
int** getMatrix(void);
float** getCYKMatrix(void);
NussinovType getType(void);
float getInfinity(void);
#include "StdAfx.h"
#include "Nussinov.h"
#include "math.h"

// FUNCTION: Nussinov
// Default Constructor
// @param mType NussinovType the nussinov type (standard or SCFG)
// @param RNASequence char* the sequence of nucleotide bases
// @param RNALength int the length of the RNA sequence
// @return void
Nussinov::Nussinov(NussinovType mType, char* RNASequence, int RNALength) {
    Nussinov::mType = mType; // Set the Nussinov type (Standard or SCFG)
    // Store the RNA Sequence and Length into private memory variables
    Nussinov::RNASequence = RNASequence;
    Nussinov::RNALength = RNALength;

    switch (mType) {
    case NussinovStandard:
        Matrix = createMatrix<int>(RNALength, RNALength);
        break;
    case NussinovSCFG:
        CYKMatrix = createMatrix<float>(RNALength, RNALength);
        break;
    default:
        // It's not a valid value, raise an exception. We should never get here...
        throw "An invalid Nussinov Type has been encountered."; // Could use RaiseException(ERROR);
        break;
    }

    // Set default scoring matrix
    scoreMatrix[0][0] = 0;
    scoreMatrix[0][1] = 0;
    scoreMatrix[0][2] = 0;
    scoreMatrix[0][3] = 1;
    scoreMatrix[1][0] = 0;
    scoreMatrix[1][1] = 0;
    scoreMatrix[1][2] = 1;
    scoreMatrix[1][3] = 0;
    scoreMatrix[2][0] = 0;
    scoreMatrix[2][1] = 1;
    scoreMatrix[2][2] = 0;
    scoreMatrix[2][3] = 0;
    scoreMatrix[3][0] = 1;
    scoreMatrix[3][1] = 0;
    scoreMatrix[3][2] = 0;
    scoreMatrix[3][3] = 0;
// Set default probability matrix
    probMatrix[0][0] = log(0.024f); // i - unpaired Probability
    probMatrix[1][0] = log(0.024f); // i - unpaired Probability
    probMatrix[2][0] = log(0.024f); // i - unpaired Probability
    probMatrix[3][0] = log(0.024f); // i - unpaired Probability
    probMatrix[0][1] = log(0.024f); // j - unpaired Probability
    probMatrix[1][1] = log(0.024f); // j - unpaired Probability
    probMatrix[2][1] = log(0.024f); // j - unpaired Probability
    probMatrix[3][1] = log(0.024f); // j - unpaired Probability
    probMatrix[0][2] = log(0.2f);  // i,j - paired Probability
    probMatrix[1][2] = log(0.2f);  // i,j - paired Probability
    probMatrix[2][2] = log(0.2f);  // i,j - paired Probability
    probMatrix[3][2] = log(0.2f);  // i,j - paired Probability
    probSS = log(0.008f); // bifurcation probability

// Set default hairpin loop length
    minLoopLength = 3;
    Infinity = -274877906944.0f; // use maximum negative float for negative infinity 
                           (float = 3.4x10^-38 to 3.4x10^-38)

// FUNCTION: ~Nussinov
// Default Destructor
// @param undefined void
// @return void
Nussinov::~Nussinov(void) {
    switch (mType) {
    case NussinovStandard:
        deleteMatrix<int>(Matrix, RNALength);
        break;
    case NussinovSCFG:
        deleteMatrix<float>(CYKMatrix, RNALength);
        break;
    default:
        // It's not a valid value, raise an exception. We should never get here...
        throw "An invalid Nussinov Type has been encountered."
            // Could use raise an exception
            //use maximum negative float for negative infinity
            Infinity = -274877906944.0f; // use maximum negative float for negative infinity 
                           (float = 3.4x10^-38 to 3.4x10^-38)
        break;
    }
}

// FUNCTION: getInfinity
// Function used to get a number used as infinity
// @param undefined void
// @return float
float Nussinov::getInfinity(void) {
    return Infinity;
}

// FUNCTION: FillStage
// Function used to call the desired FillStage (Standard or SCFG)
// @param undefined void
// @return void
void Nussinov::FillStage() {
    switch (mType) {
    case NussinovStandard:
        NussinovFillStage();
        break;
    case NussinovSCFG:
        NussinovFillStageCYK();
        break;
3

default:
   // It's not a valid value, raise an exception. We should never get here...
   throw "An invalid Nussinov Type has been encountered."; // Could use RaiseException(ERROR);
   break;
   }
   }
   // FUNCTION: TraceBack
   // Function used to call the correct TraceBack (Standard or SCFG)
   // @param  pRNA  RNA*  pointer to an RNA class
   // @return  void
   void Nussinov::TraceBack(RNA *pRNA) {
   switch (mType)
   {  
   case NussinovStandard:
      NussinovTraceBack(pRNA);
      break;
   case NussinovSCFG:
      NussinovTraceBackCYK(pRNA);
      break;
   default:
      // It's not a valid value, raise an exception. We should never get here...
      throw "An invalid Nussinov Type has been encountered."; // Could use RaiseException(ERROR);
      break;
   }
   // FUNCTION: getType
   // Function used to retrieve the Nussinov type currently in use
   // @param  undefined  void
   // @return  NussinovType
   Nussinov::NussinovType Nussinov::getType(void) {
   return mType;
   }
   // FUNCTION: createMatrix
   // Used to dynamically allocate memory for the Nussinov matrix
   // @param  row  int  the number of rows in the matrix
   // @param  col  int  the number of columns in the matrix
   // @return  T**
   template<class T> T** Nussinov::createMatrix(int row, int col) {
   T** mMatrix = new T* [row];
   for(int i = 0; i < row; i++)
   mMatrix[i] = new T[col];
   return mMatrix;
   }
   // FUNCTION: deleteMatrix
   // Used to remove the memory that was dynamically allocated for the Nussinov matrix
   // @param  mMatrix  T**  pointer to matrix
   // @param  row  int  number of rows in the matrix
   // @return  void
   template<class T> void Nussinov::deleteMatrix(T** mMatrix, int row) {
   for(int i = 0; i < row; i++) {
   delete[] mMatrix[i];
   }
   delete[] mMatrix;
   }
   // FUNCTION: setScoringMatrix
197 // Used to populate the scoring matrix
198 //
199 // @param AA int the A-A bond value
200 // @param AC int the A-C bond value
201 // @param AG int the A-G bond value
202 // @param AU int the A-U bond value
203 // @param CA int the C-A bond value
204 // @param CC int the C-C bond value
205 // @param CG int the C-G bond value
206 // @param CU int the C-U bond value
207 // @param GA int the G-A bond value
208 // @param GC int the G-C bond value
209 // @param GG int the G-G bond value
210 // @param GU int the G-U bond value
211 // @param UA int the U-A bond value
212 // @param UC int the U-C bond value
213 // @paramUG int the U-G bond value
214 // @paramUU int the U-U bond value
215 // @return void
216 void Nussinov::setScoringMatrix(int AA, int AC, int AG, int AU, int CA, int CC, int CG, int GA, int GC, int GG, int GU, int UA, int UC, int UG, int UU) {
217     scoreMatrix[0][0] = AA;
218     scoreMatrix[0][1] = AC;
219     scoreMatrix[0][2] = AG;
220     scoreMatrix[0][3] = AU;
221     scoreMatrix[1][0] = CA;
222     scoreMatrix[1][1] = CC;
223     scoreMatrix[1][2] = CG;
224     scoreMatrix[1][3] = CU;
225     scoreMatrix[2][0] = GA;
226     scoreMatrix[2][1] = GC;
227     scoreMatrix[2][2] = GG;
228     scoreMatrix[2][3] = GU;
229     scoreMatrix[3][0] = UA;
230     scoreMatrix[3][1] = UC;
231     scoreMatrix[3][2] = UG;
232     scoreMatrix[3][3] = UU;
233 }
234
235 // FUNCTION: getScoringMatrix
236 // Used to retrieve the cell value of the given position
237 //
238 // @param row int the row of the desired cell
239 // @param col int the col of the desired cell
240 // @return int
241 int Nussinov::getScoringMatrix(int row, int col) {
242     int Value;
243     if (row >= 0 && row < 4 && col >= 0 && col < 4)
244         Value = scoreMatrix[row][col];
245     else
246         throw "Nussinov: Invalid scoring matrix index";
247     return Value;
248 }
249
250 // FUNCTION: setMinHairpinLength
251 // Used to set the minimum hairpin loop length
252 //
253 // @param HairpinLength int the minimum length of the hairpin loop
254 // @return void
255 void Nussinov::setMinHairpinLength(int HairpinLength) {
256     minLoopLength = HairpinLength;
257 }
258
259 // FUNCTION: getMinHairpinLength
// Used to retrieve the minimum hairpin loop length
@
// @param undefined void
// @return int
int Nussinov::getMinHairpinLength(void) {
  return minLoopLength;
}
@
// FUNCTION: NussinovFillStage
// Implementation of the Nussinov FillStage
@
// @param undefined void
// @return void
void Nussinov::NussinovFillStage() {
  for(int i = 0; i < RNALength; i++) { // Initialize the diagonal to zero
    Matrix[i][i] = 0;
  }
  for(int i = 1; i < RNALength; i++) { // Initialize the lower diagonal to zero
    Matrix[i][i - 1] = 0;
  }
  int j;
  for(int d = 1; d < RNALength; d++) { // Loop through the diagonals
    for(int i = 0; i < RNALength - d; i++) { // Loop within the diagonal
      j = i + d;
      if(d <= minLoopLength) {
        Matrix[i][j] = 0;
      }
      else {
        int maxValue = 0;
        for(int k = i + 1; k < j; k++) {
          int tmpValue = Matrix[i][k] + Matrix[k + 1][j];
          if(tmpValue > maxValue)
            maxValue = tmpValue;
        }
      }
    }
  }
@
// FUNCTION: NussinovTraceBack
// Implementation of the Nussinov TraceBack Stage
@
// @param pRNA RNA* pointer to an RNA class
// @return void
void Nussinov::NussinovTraceBack(RNA *pRNA) {
  CStack stackPosition; // Instantiate the position stack
  int* result = new int[pRNA->getStepPosition()];
  listTraceback.Clear(); // Clear the old list
  CStack::Item_type item; // Instantiate a item structure
  CStack::Item_type* item_ptr = &item; // Pointer to Item_type
  item.i = 0;
  item.j = pRNA->getStepPosition() - 1;
  stackPosition.Push(item); // Push the upper right corner of the matrix on the stack
  listTraceback.Add(item.i, item.j); // Traceback stack position
  for(int x = 0; x < pRNA->getStepPosition(); x++) {
```cpp
int i, j;
while(!stackPosition.IsEmpty()) {
    stackPosition.Pop(item_ptr);
    i = item_ptr->i;
    j = item_ptr->j;
    if(i >= j) {
        continue;    // Continue to next iteration
    }
    else if(Matrix[i][j] == Matrix[i + 1][j]) {
        // i Unpaired
        item.i = i + 1;
        item.j = j;
        stackPosition.Push(item);
        listTraceback.Add(item.i, item.j);
    }
    else if(Matrix[i][j] == Matrix[i][j - 1]) {
        // j Unpaired
        item.i = i;
        item.j = j - 1;
        stackPosition.Push(item);
        listTraceback.Add(item.i, item.j);
    }
    else if(Matrix[i][j] == Matrix[i + 1][j - 1] + scoreMatrix[(RNASequence[i] == 'A' ? 0 : (RNASequence[i] == 'C' ? 1 : (RNASequence[i] == 'G' ? 2 : 3))))] {
        // i, j Paired
        item.i = i + 1;
        item.j = j - 1;
        stackPosition.Push(item);
        listTraceback.Add(item.i, item.j);
        result[i] = j;
        result[j] = i;
    }
    else {
        for(int k = i + 1; k < j; k++) {
            if(Matrix[i][j] == Matrix[i][k] + Matrix[k + 1][j]) {
                // Bifurcation
                item.i = k + 1;
                item.j = j;
                stackPosition.Push(item);
                listTraceback.Add(item.i, item.j);
                item.i = i;
                item.j = k;
                stackPosition.Push(item);
                listTraceback.Add(item.i, item.j);
                break;
            }
        }
    }
    SortTracebackPath();
    pRNA->setPairing(result);
}
```

// FUNCTION: SortTracebackPath
// Used to sort the traceback path

@
// @param undefined void
void Nussinov::SortTracebackPath() {
    List::Node_type *node_ptr = listTraceback.head;
```
List::Node_type *node_previous = NULL;
while(node_ptr->next != NULL)
{
    if(node_ptr->i > node_ptr->next->i)
    {
        if(node_previous == NULL)
        {
            listTraceback.head = node_ptr->next;
            node_previous = node_ptr->next;
            node_ptr->next = node_ptr->next->next;
            node_previous->next = node_ptr;
        }
        else
        {
            node_previous->next = node_ptr->next;
            node_ptr->next = node_ptr->next->next;
            node_previous->next->next = node_ptr;
            node_previous = node_previous->next;
        }
    }
    else
    {
        node_previous = node_ptr;
        node_ptr = node_ptr->next;
    }
}
listTraceback.tail = node_ptr;

// FUNCTION: getMatrix
// Used to retrieve the dynamic programming matrix
int** Nussinov::getMatrix(void) {
    return Matrix;
}

// FUNCTION: getCYKMatrix
// Used to retrieve the CYK dynamic programming matrix
float** Nussinov::getCYKMatrix(void) {
    return CYKMatrix;
}

// FUNCTION: Max
// Returns the maximum of the 4 integers
int Nussinov::Max(int a, int b, int c, int d) {
    int tmp = a;
    if(b > tmp)
        tmp = b;
    if(c > tmp)
        tmp = c;
    if(d > tmp)
        tmp = d;
    return tmp;
}
// FUNCTION: Max
// Returns the maximum of 3 floats
//
// @param a float the first float
// @param b float the second float
// @param c float the third float
// @return float

float Nussinov::Max(float a, float b, float c) {
    float tmp = a;
    if (b > tmp) {
        tmp = b;
    }
    if (c > tmp) {
        tmp = c;
    }
    return tmp;
}

// FUNCTION: setProbMatrix
// Used to set the probability matrix
//
// @param aS float the aS probability
// @param cS float the cS probability
// @param gS float the gS probability
// @param uS float the uS probability
// @param Sa float the Sa probability
// @param Sc float the Sc probability
// @param Sg float the Sg probability
// @param Su float the Su probability
// @param aSu float the aSu probability
// @param cSg float the cSg probability
// @param gSc float the gSc probability
// @param uSa float the uSa probability
// @param SS float the SS probability
// @return void

void Nussinov::setProbMatrix(float aS, float cS, float gS, float uS, float Sa, float Sc, float Sg, float Su, float aSu, float cSg, float gSc, float uSa, float SS) {
    probMatrix[0][0] = log(aS);
    probMatrix[1][0] = log(cS);
    probMatrix[2][0] = log(gS);
    probMatrix[3][0] = log(uS);
    probMatrix[0][1] = log(Sa);
    probMatrix[1][1] = log(Sc);
    probMatrix[2][1] = log(gS);
    probMatrix[3][1] = log(uS);
    probMatrix[0][2] = log(aSu);
    probMatrix[1][2] = log(cSg);
    probMatrix[2][2] = log(gSc);
    probMatrix[3][2] = log(uSa);
    probSS = log(SS);
}

// FUNCTION: getProbMatrix
// Used to retrieve the probability matrix
//
// @param row int the row of the cell to retrieve
// @param col int the col of the cell to retrieve
// @return float

float Nussinov::getProbMatrix(int row, int col) {
    float Value;
    if (row >= 0 && row < 4 && col >= 0 && col < 3) {
        Value = probMatrix[row][col];
    } else {
        throw "Nussinov: Invalid probability matrix index";
    }
}
return Value;
}
// FUNCTION: getProbSS
// Used to retrieve the SS probability
//@param undefined void
float Nussinov::getProbSS(void) {
    return probSS;
}

// FUNCTION: NussinovFillStageCYK
// Implementation of the Nussinov CYK FillStage
//@param undefined void
//@return void
void Nussinov::NussinovFillStageCYK() {
    for(int i = 0; i < RNALength; i++) { // Initialize the diagonal
        if(probMatrix[RNASequence[i]] == 'A' ? 0 : (RNASequence[i] == 'C' ? 1 :
            (RNASequence[i] == 'G' ? 2 : 3)))[0] > probMatrix[(RNASequence[i] == 'A' ? 0 :
            (RNASequence[i] == 'C' ? 1 : (RNASequence[i] == 'G' ? 2 : 3)))[1])
            CYKMatrix[i][i] = (float)probMatrix[(RNASequence[i] == 'A' ? 0 :
                (RNASequence[i] == 'C' ? 1 : (RNASequence[i] == 'G' ? 2 : 3)))[0];
        else
            CYKMatrix[i][i] = (float)probMatrix[(RNASequence[i] == 'A' ? 0 :
                (RNASequence[i] == 'C' ? 1 : (RNASequence[i] == 'G' ? 2 : 3)))[1];
        //CYKMatrix[i][i] = max(probMatrix[(RNASequence[i] == 'A' ? 0 : (RNASequence
            [i] == 'C' ? 1 : (RNASequence[i] == 'G' ? 2 : 3))][0], probMatrix[(RNASequence[i]
                == 'A' ? 0 : (RNASequence[i] == 'C' ? 1 : (RNASequence[i] == 'G' ? 2 : 3)))[1]);
    }
    for(int i = 1; i < RNALength; i++) { // Initialize the lower diagonal to negative
        // infinity
        CYKMatrix[i][i-1] = -274877906944; // use maximum negative float for
        // negative infinity (float = 3.4x10^-38 to 3.4x10^-38)
    }
    int j;
    for(int d = 1; d < RNALength; d++) { // Loop through the diagonals
        for(int i = 0; i < RNALength - d; i++) { // Loop within the diagonal
            j = i + d;
            if(d <= minLoopLength) {
                CYKMatrix[i][j] = 0;
            }
            else {
                float maxValue = Infinity; // use maximum negative float for negative
                // infinity (float = 3.4x10^-38 to 3.4x10^-38)
                for(int k = i + 1; k < j; k++) {
                    float tmpValue = CYKMatrix[i][k] + CYKMatrix[k + 1][j] + probSS;
                    if(tmpValue > maxValue)
                        maxValue = tmpValue;
                }
                if((RNASequence[i] == 'A' ? 0 : (RNASequence[i] == 'C' ? 1 :
                    (RNASequence[i] == 'G' ? 2 : 3)))[0] > (RNASequence[j] == 'A' ? 0 :
                    (RNASequence[j] == 'C' ? 1 : (RNASequence[j] == 'G' ? 2 : 3)))[1])
                    // A-U, U-A, C-G, G-C
                    
                    float tmpValue = CYKMatrix[i+1][j-1] + probMatrix[(RNASequence[i]
                        == 'A' ? 0 : (RNASequence[i] == 'C' ? 1 : (RNASequence[i] == 'G' ? 2 : 3))][2];
                    if(tmpValue > maxValue)
                        maxValue = tmpValue;
                }
                if(i == 0 && j == 8)
                    i = i;
            }
            CYKMatrix[i][j] = Max(CYKMatrix[i+1][j] + probMatrix[(RNASequence[i]
                == 'A' ? 0 : (RNASequence[i] == 'C' ? 1 : (RNASequence[i] == 'G' ? 2 : 3))][0],
                CYKMatrix[i][j-1] + probMatrix[(RNASequence[j] == 'A' ? 0 : (RNASequence[j] ==
                    'C' ? 1 : (RNASequence[j] == 'G' ? 2 : 3))][1], maxValue);
// FUNCTION: NussinovTraceBackCYK
// Implementation of the Nussinov CYK TraceBack Stage
void Nussinov::NussinovTraceBackCYK(RNA *pRNA) {
    CStack stack_pos; // Instantiate the position stack
    int* result = new int[pRNA->getStepPosition()];
    listTraceback.Clear(); // Clear the old list
    CStack::Item_type item; // Instantiate a item structure
    CStack::Item_type* item_ptr = &item; // Pointer to Item_type
    item.i = 0;
    item.j = pRNA->getStepPosition() - 1;
    stack_pos.Push(item); // Push the item on the stack
    listTraceback.Add(item.i, item.j); // Traceback stack position

    int i, j;
    for(i = 0; i < pRNA->getStepPosition(); i++) {
        result[i] = i;
    }

    while(!stack_pos.IsEmpty()) {
        stack_pos.Pop(item_ptr);
        i = item_ptr->i;
        j = item_ptr->j;

        if(i >= j) {
            continue; // Continue to next iteration
        }
        else if(IsEqual(CYKMatrix[i][j], (float)(CYKMatrix[i + 1][j] + probMatrix[RNASequence[i] == 'A' ? 0 : (RNASequence[i] == 'C' ? 1 : (RNASequence[i] == 'G' ? 2 : 3)]))[0])) {
            item.i = i + 1;
            item.j = j;
            stack_pos.Push(item);
            listTraceback.Add(item.i, item.j);
        }
        else if(IsEqual(CYKMatrix[i][j], (float)(CYKMatrix[i][j - 1] + probMatrix[RNASequence[j] == 'A' ? 0 : (RNASequence[j] == 'C' ? 1 : (RNASequence[j] == 'G' ? 2 : 3)]))[1])) {
            item.i = i;
            item.j = j - 1;
            stack_pos.Push(item);
            listTraceback.Add(item.i, item.j);
        }
            item.i = i + 1;
            item.j = j - 1;
            stack_pos.Push(item);
            listTraceback.Add(item.i, item.j);
            result[i] = j;
            result[j] = i;
        }
    }
}
else {
    for (int k = i + 1; k < j; k++) {
        if (isEqual(CYKMatrix[i][j], (float)(CYKMatrix[i][k] + CYKMatrix[k + 1][j] + probSS))) {
            item.i = k + 1;
            item.j = j;
            stack_pos.Push(item);
            listTraceback.Add(item.i, item.j);
            item.i = i;
            item.j = k;
            stack_pos.Push(item);
            listTraceback.Add(item.i, item.j);
            break;
        }
    }
}
SortTracebackPath();
listTraceback.Add(item.i, item.j);
}

// FUNCTION: IsEqual
// Determine if the numbers are within a thousandth of each other, If
// they are then consider them equal

// @param num1 float the first number to compare
// @param num2 float the second number to compare
// @return bool
bool Nussinov::isEqual(float num1, float num2) {
    float diff;
    if (num1 > num2) {
        diff = num1 - num2;
    } else {
        diff = num2 - num1;
    }
    if (diff < .0001)
        return true;
    else
        return false;
#pragma once

#include "RNA.h"
#include "Nussinov.h"

class MatrixGraph
{
private:
  MatrixGraph(void);
  ~MatrixGraph(void);

public:
  void Draw(HDC hdc, RNA *pRNA, Nussinov *pNussinov, float xsize, float ysize);
};
1  /******************************************************************************
2  * Copyright (c): 2006, All Rights Reserved
3  * Project: SJSU Masters Project
4  * File: MatrixGraph.cpp
5  * Purpose: To calculate the matrix graph representation of Nussinov Matrix
6  *
7  * Start Date: 9/15/2006
8  * Programmer: Brandon Hunter
9 ******************************************************************************/
10
11 #include "StdAfx.h"
12 #include "MatrixGraph.h"
13 #include "Stack.h"
14 #include "stdio.h"  // for the sprintf function used to convert float to char
15
16 // FUNCTION: MatrixGraph
17 // Default Constructor
18 //
19 // @param undefined void
20 // @return void
21 MatrixGraph::MatrixGraph(void) {
22 }
23
24 // FUNCTION: ~MatrixGraph
25 // Default Destructor
26 //
27 // @param undefined void
28 // @return void
29 MatrixGraph::~MatrixGraph(void) {
30 }
31
32 // Rounding Error Solution used in this function
33 // int i;
34 // float f = 1.2345678;
35 // i = (int)(f + 0.5); /* intValue will be 1 */
36 // f = 1.56789;
37 // i = (int)(f + 0.5); /* intValue will be 2 */
38 // Ex. (int)1.5789 will return 1
39 // (int)(1.5789 + .5) will return 2
40
41 // FUNCTION: Draw
42 // Function used to draw the Matrix graph onto the bitmap device context
43 //
44 // @param hdc   HDC    handle to the bitmap device context
45 // @param pRNA  RNA*   pointer to an RNA class
46 // @param pNussinov  Nussinov*   pointer to a Nussinov class
47 // @param xsize  float   x coordinate of bitmap size
48 // @param ysize  float   y coordinate of bitmap size
49 // @return void
50 void MatrixGraph::Draw(HDC hdc, RNA *pRNA, Nussinov *pNussinov, float xsize, float ysize) {
51   float fBorder = 5.0f;
52   float fCellSizeX = (xsize - (2.0f * fBorder)) / (pRNA->getLength() + 1);
53   float fCellSizeY = (ysize - (2.0f * fBorder)) / (pRNA->getLength() + 1);
54   float fCellLeft = fBorder;
55   float fCellTop = fBorder;
56   int FontHeight = 0, FontWidth = 0;
57   COLORREF Bkgrnd;
58   COLORREF OldBkgrnd;
59   HFONT hFont;
60   HFONT hOldFont;
61   HBRUSH hBrush;
62   HBRUSH hOldBrush;
63   float colorInc = 256.0f / pRNA->getLength(); // pNussinov->listTraceback.count;
// Draw the sequence along the edges of the matrix
FontHeight = (int)(fCellSizeY - 2.0f + 0.5f);
if(FontHeight > 5)
{
    hFont = CreateFont(FontHeight, 0, 0, 0, FW_SEMIBOLD, FALSE, FALSE, FALSE, ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, FF_DONTCARE, TEXT("Arial"));

    hOldFont = (HFONT)SelectObject(hdc, hFont);
    float xPos = fCellLeft + fCellSizeX, yPos = fCellTop + fCellSizeY;
    for(int x = 0; x < pRNA->getLength(); x++)
    {
        char strBase[2];
        strBase[0] = pRNA->getSequence()[x];
        strBase[1] = '\0';
        int len = strlen(strBase);
        SIZE sizeRect;
        GetTextExtentPointA(hdc, strBase, len, &sizeRect);

        // Sequence Along the Top Edge
        TextOutA(hdc, (int)((xPos + (fCellSizeX / 2.0f)) - (sizeRect.cx / 2.0f) + 0.5f), (int)((fCellTop + (fCellSizeY / 2.0f)) - (sizeRect.cy / 2.0f) + 0.5f), strBase, len);

        // Sequence Along the Left Edge
        TextOutA(hdc, (int)((fCellLeft + (fCellSizeX / 2.0f)) - (sizeRect.cx / 2.0f) + 0.5f), (int)((yPos + (fCellSizeY / 2.0f)) - (sizeRect.cy / 2.0f) + 0.5f), strBase, len);
    }
    SelectObject(hdc, hOldFont);
    DeleteObject(hFont);
}

fCellLeft += fCellSizeX;
fCellTop += fCellSizeY;

for(int j = 0; j < pRNA->getLength(); j++)
{
    for(int i = 0; i < pRNA->getLength(); i++)
    {
        if(j > i - 2) {
            if(j >= pRNA->getStepPosition() || i >= pRNA->getStepPosition())
                Bkgrnd = RGB(224, 224, 224); // Gray background
        }
        else
        {
            //Bkgrnd = RGB(250, 225, 178);
            //Bkgrnd = RGB(colorInc * i, 255, colorInc * j); // Color based on

            color scale
            Bkgrnd = RGB(255, 255, 255);
            List::Node_type *node_ptr = pNussinov->listTraceback.head;
            while(node_ptr != NULL)
            {
                if(node_ptr->i == i && node_ptr->j == j)
                    Bkgrnd = RGB(colorInc * i, 255, colorInc * j); // Color 
            }
        }
    }
}

// Record position of trace back cell so lines can be drawn
node_ptr->xPos = fCellLeft + (fCellSizeX / 2.0f) + 1.0f;
node_ptr->yPos = fCellTop + (fCellSizeY / 2.0f) + 1.0f;
node_ptr = node_ptr->next;
break;
}
node_ptr = node_ptr->next;
}

hBrush = CreateSolidBrush(Bkgrnd); // Gray background
hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);
Rectangle(hdc, (int)(fCellLeft + 0.5f), (int)(fCellTop + 0.5f), (int)(fCellLeft + fCellSizeX + 1.0f + 0.5f), (int)(fCellTop + fCellSizeY + 1.0f + 0.5f));
SelectObject(hdc, hOldBrush);
DeleteObject(hBrush);

if(j < pRNA->getStepPosition() && i < pRNA->getStepPosition())
{
    char strNumber[64]; // Buffer for number
    switch(pNussinov->getType())
    {
    case Nussinov::NussinovType::NussinovStandard:
        itoa(pNussinov->getMatrix()[i][j], strNumber, 10);
        break;
    case Nussinov::NussinovType::NussinovSCFG:
        if(pNussinov->getCYKMatrix()[i][j] == pNussinov->getInfinity )
        {
            strNumber[0] = '-';
            strNumber[1] = '0';
        }
        else
            sprintf(strNumber, "%.2f", pNussinov->getCYKMatrix()[i][j]);
        break;
    default:
        // It's not a valid value, raise an exception. We should never get here...
        throw "An invalid Nussinov Type has been encountered."; // Could use RaiseException(ERROR);
        break;
    }
    int len = strlen(strNumber);
    FontHeight = 0;
    FontWidth = 0;
    if(len > 1) // Need to reduce the size of the font depending on the length of the number
    {
        //hFont = CreateFont((int)fCellSizeY - 5, 0, 0, FW_SEMIBOLD , FALSE, FALSE, FALSE, ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, FF_DONTCARE, TEXT("Arial"));

        FontWidth = (int)((fCellSizeX / len) - 2.0f + 0.5f);
    }
    else
    {
        FontHeight = (int)(fCellSizeY - 2.0f + 0.5f);
    }
    hFont = CreateFont(FontHeight, FontWidth, 0, 0, FW_SEMIBOLD, FALSE , FALSE, FALSE, ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, FF_DONTCARE, TEXT("Arial"));
    if(hFont != NULL)
    hOldFont = (HFONT)SelectObject(hdc, hFont);

    SIZE sizeRect;
    GetTextExtentPointA(hdc, strNumber, len, &sizeRect);
    OldBkgrnd = SetBkColor(hdc, Bkgrnd);
    if(FontWidth > 5 || FontHeight > 5)
        TextOutA(hdc, (int)((fCellLeft + (fCellSizeX / 2.0f)) -

c:\Thesis\RNAvis\MatrixGraph.cpp

(sizeRect.cx / 2.0f) + 0.5f), (int)((fCellTop + (fCellSizeY / 2.0f)) - (sizeRect.cy / 2.0f) + 0.5f), strNumber, len);
181    SetBkColor(hdc, OldBkgrnd);
182    SelectObject(hdc, hOldFont);
183    DeleteObject(hFont);
184
185    fCellTop += fCellSizeY;
186
187    fCellTop += fCellSizeY;
188
189    fCellTop += fCellSizeX;
190
191    fCellLeft += fCellSizeX;
#pragma once

class List
{
    private:
        List(void);
        ~List(void);
        typedef struct node_tag {
            int i, j;
            float xPos, yPos;
            struct node_tag *next;
        } Node_type;
        void Add(int i, int j);
        void Clear(void);
        Node_type *head, *tail;
        int count;
};
#include "StdAfx.h"
#include "List.h"

// FUNCTION: List
// Default Constructor
// @param undefined void
// @return void
List::List(void)
{
    head = tail = NULL; // Initialize the head and tail of the list
    count = 0; // Initialize the counter for number of items in list
}

// FUNCTION: ~List
// Default Destructor
// @param undefined void
// @return void
List::~List(void)
{
    while(head != NULL) {
        Node_type *node_ptr;
        node_ptr = head;
        head = node_ptr->next;
        delete node_ptr;
    }
    count = 0;
    head = tail = NULL;
}

// FUNCTION: Add
// Function to add items to the linked list
// @param i int the i pair of the position
// @param j int the j pair of the position
// @return void
void List::Add(int i, int j) {
    Node_type *p;

    if((p = new Node_type) == NULL)
        throw "List: Out of Memory";
    else {
        p->i = i;
        p->j = j;
        p->next = NULL;
    }

    if(head == NULL)
    {
        head = p; // If the list was empty then set the head and tail to first item
        tail = p;
    }
    else
    {
        tail->next = p;
    }
tail = p;
}
count++; // Increment the list item counter
}

// FUNCTION:  Clear
// Function used to clear the linked list
// @param undefined void
// @return void
void List::Clear(void) {
while(head != NULL) {
    Node_type *node_ptr;
    node_ptr = head;
    head = node_ptr->next;
    delete node_ptr;
}
count = 0;
head = tail = NULL;
c:\Thesis\RNAVis\LineList.h

/********************
* Copyright (c): 2006, All Rights Reserved
* Project: SJSU Masters Project
* File: LineList.h
* Purpose: Header file for class implementation of a linked list
* 
* Start Date: 10/22/2006
* Programmer: Brandon Hunter
*
* ***************************************************/

#pragma once

class LineList
{
private:

public:
    LineList(void);
    ~LineList(void);

    typedef struct node_tag {
        float startX, startY;
        float endX, endY;
        DWORD color;
        struct node_tag *next;
    } Node_type;

    void Add(float startX, float startY, float endX, float endY, DWORD color);
    void Clear(void);

    Node_type *head, *tail;
    int count;
};
#include "StdAfx.h"
#include "LineList.h"

// FUNCTION: LineList
// Default Constructor
// @param undefined void
// @return void
LineList::LineList(void)
{
    head = tail = NULL; // Initialize the head and tail of the list
    count = 0; // Initialize the counter for number of items in list
}

// FUNCTION: ~LineList
// Default Destructor
// @param undefined void
// @return void
LineList::~LineList(void)
{
    while(head != NULL) {
        Node_type *node_ptr;
        node_ptr = head;
        head = node_ptr->next;
        delete node_ptr;
    }
    count = 0;
    head = tail = NULL;
}

// FUNCTION: Add
// Function used to add items to the linked list
// @param startX float the x coordinate of the start of the line
// @param startY float the y coordinate of the start of the line
// @param endX float the x coordinate of the end of the line
// @param endY float the y coordinate of the end of the line
// @param color DWORD the color of the line
// @return void
void LineList::Add(float startX, float startY, float endX, float endY, DWORD color) {
    Node_type *p;
    if((p = new Node_type) == NULL)
        throw "List: Out of Memory";
    else {
        p->startX = startX;
        p->startY = startY;
        p->endX = endX;
        p->endY = endY;
        p->color = color;
        p->next = NULL;
    }
    if(head == NULL)
    {
        head = tail = p;
    }
    else
    {
        tail->next = p;
        tail = p;
    }
    count++;
}

head = p; // If the list was empty then set the head and tail to first item
tail = p;
} else
{
tail->next = p;
tail = p;
}
count++; // Increment the list item counter

// FUNCTION: Clear
// Function that clears the linked list
// @param undefined void
// @return void
void LineList::Clear(void) {
while(head != NULL) {
    Node_type *node_ptr;
    node_ptr = head;
    head = node_ptr->next;
    delete node_ptr;
}
}

count = 0;
head = tail = NULL;
class CircularGraph
{
private:
    // Floating point
    typedef struct tagPOINTF
    {
        float x;
        float y;
    } POINTF;

    CircularGraph(void);
    ~CircularGraph(void);

    void Draw(HDC hdc, char* Sequence, int iStep, int* Pairing, float xsize, float ysize);
};
c:\Thesis\RNAVis\CircularGraph.cpp

/*********************************************************
* Copyright (c): 2006, All Rights Reserved
* Project: SJSU Masters Project
* File: CircularGraph.cpp
* Purpose: To calculate the circular graph representation of RNA secondary structure
* Start Date: 9/16/2006
* Programmer: Brandon Hunter
***********************************************************/

#include "StdAfx.h"
#include "CircularGraph.h"
#include "math.h"

// FUNCTION: CircularGraph
// Default Constructor
// @param undefined void
// @return void
CircularGraph::CircularGraph(void) {
}

// FUNCTION: ~CircularGraph
// Default Destructor
// @param undefined void
// @return void
CircularGraph::~CircularGraph(void) {
}

// FUNCTION: Draw
// This function draws the Circular Graph onto the bitmap device context
// @param hdc HDC handle to the bitmap device context
// @param Sequence char* the RNA sequence of nucleotide bases
// @param iStep int the current step position within the sequence
// @param Pairing int* array which holds the sequence pairing positions
// @param xsize float x dimension of bitmap size
// @param ysize float y dimension of bitmap size
// @return void
void CircularGraph::Draw(HDC hdc, char* Sequence, int iStep, int* Pairing, float xsize, float ysize) {
    float fBorder = 5.0f;
    float fPadding = 20.0f;
    float fPI = 3.14159265f;
    float fAngle = (360.0f / iStep) * fPI / 180.0f; // Radians
    float fRadius;
    float fCenterX = 0.0f;
    float fCenterY = 0.0f;

    if(xsize < ysize)
        fRadius = (xsize / 2.0f) - fBorder - fPadding;
    else
        fRadius = (ysize / 2.0f) - fBorder - fPadding;

    HPEN hPen;
    HPEN holdPen;

    // Draw the tick marks around the circle
    hPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
    holdPen = (HPEN)SelectObject(hdc, hPen);
    for(int x = 0; x < iStep; x++) {
        if((x % 10) == 0) // Draw a slightly longer line on every 10th line
            MoveToEx(hdc, (int)(((fRadius + 10.0f) * cos(x * fAngle)) + fCenterX + 0.5f), (int)(((fRadius + 10.0f) * sin(x * fAngle)) + fCenterY + 0.5f), NULL);
else
    MoveToEx(hdc, (int)((fRadius + 5.0f) * cos(x * fAngle)) + fCenterX + 0.5f), (int)((fRadius + 5.0f) * sin(x * fAngle)) + fCenterY + 0.5f), NULL); //
    LineTo(hdc, (int)(fCenterX + 0.5f), (int)(fCenterY + 0.5f));
}
SelectObject(hdc, hOldPen);
DeleteObject(hPen);

// This is the main circle
Ellipse(hdc, (int)-fRadius + 0.5f), (int)(fRadius + 0.5f), (int)(fRadius + 0.5f), (int)-(fRadius + 0.5f));

// Draw tick marks at positions that don't require arcs
hPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
hOldPen = (HPEN)SelectObject(hdc, hPen);
for(int x = 0; x < iStep; x++) {
    // Draw tick marks at positions that don't require arcs
    if(Pairing[x] == x)
    {
        MoveToEx(hdc, (int)((fRadius * cos(x * fAngle)) + fCenterX + 0.5f), (int)((fRadius * sin(x * fAngle)) + fCenterY + 0.5f), NULL);
        LineTo(hdc, (int)(fCenterX + 0.5f), (int)(fCenterY + 0.5f));
    }
}
SelectObject(hdc, hOldPen);
DeleteObject(hPen);

// This circle masks the interior of the tick marks
hPen = CreatePen(PS_SOLID, 1, RGB(255, 255, 255));
hOldPen = (HPEN)SelectObject(hdc, hPen);
Ellipse(hdc, (int)-fRadius - 5.0f + 0.5f), (int)(fRadius - 5.0f + 0.5f), (int)(fRadius - 5.0f + 0.5f), (int)(-((fRadius - 5.0f + 0.5f)));
SelectObject(hdc, hOldPen);
DeleteObject(hPen);

// Draw the arcs
hPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
hOldPen = (HPEN)SelectObject(hdc, hPen);
for(int x = 0; x < iStep; x++)
{
    // If an arc is required
    if(Pairing[x] != x) {
        // Draw the arc when you have reached the second endpoint of the arc
        if(Pairing[x] < x)
        {
            int i = Pairing[x];
            int j = x;
            POINTF p1 = {(fRadius * cos(i * fAngle)) + fCenterX, (fRadius * sin(i * fAngle)) + fCenterY};
            POINTF p2 = {(fRadius * cos(j * fAngle)) + fCenterX, (fRadius * sin(j * fAngle)) + fCenterY};

            // Find angle half way between the two tick marks
            // Angle = (smaller angle + (larger angle - smaller angle) / 2)
            float fMidAngle = (i * fAngle) + (((j * fAngle) - (i * fAngle)) / 2.0f);

            // p3 is 90degrees back from mid angle
            POINTF p3 = {(fRadius * cos(fMidAngle - (fPI / 2.0f)) + fCenterX, (fRadius * sin(fMidAngle - (fPI / 2.0f)) + fCenterY);
            // p4 is 90degrees forward from mid angle
            POINTF p4 = {(fRadius * cos(fMidAngle + (fPI / 2.0f)) + fCenterX, (fRadius * sin(fMidAngle + (fPI / 2.0f)) + fCenterY);

            if((float)((x - Pairing[x]) * fAngle) == (float)fPI)
            {
                MoveToEx(hdc, (int)(p1.x + 0.5f), (int)(p1.y + 0.5f), NULL);
                LineTo(hdc, (int)(p2.x + 0.5f), (int)(p2.y + 0.5f));
            }
else
{
    // Find intersection of lines P2P3 and p1p4 use equation of lines
    // and then solve the two equations (two equations two unknowns)
    // y = m1x + b1
    // m1 = (y2-y1)/(x2-x1)
    bool line1Vertical = false;
    float m1, b1, m6;
    if((p2.x - p3.x) == 0) // Avoid Divide by zero
    {
        line1Vertical = true; // The line p2p3 is a vertical line
        m6 = 0; // Slope of a line perpendicular to a vertical line is undefined
    }
    else
    {
        m1 = (p2.y - p3.y) / (p2.x - p3.x);
        b1 = p2.y - (m1 * p2.x); // b1 = y - mx
        m6 = (-1 / m1); // Slope of perpendicular line is m = -1/m1
    }

    // m2 = (y2-y1)/(x2-x1)
    bool line2Vertical = false;
    float m2, b2, m5;
    if((p1.x - p4.x) == 0) // Avoid Divide by zero
    {
        line2Vertical = true; // The line p1p4 is a vertical line
        m5 = 0; // Slope of a line perpendicular to a vertical line is undefined
    }
    else
    {
        m2 = (p1.y - p4.y) / (p1.x - p4.x);
        b2 = p1.y - (m2 * p1.x); // b2 = y - mx
        m5 = (-1 / m2); // Slope of perpendicular line is m = -1/m2
    }

    // Now find intersection of p1p4 and p2p3
    POINTF px;
    if(line1Vertical)
    {
        // We know one line is vertical so just solve for the intersecting point in the second line
        px.x = p2.x;
        px.y = (m2 * p2.x) + b2;
    }
    else
    {
        if(line2Vertical)
        {
            // We know one line is vertical so just solve for the intersecting point in the second line
            px.x = p1.x;
            px.y = (m1 * p1.x) + b1;
        }
        else
        {
            // x = (b2 - b1)/(m1 - m2), y = (m1b2 - m2b1)/(m1 - m2)
            if((m1 - m2) == 0) // Avoid Divide by zero
            {
                px.x = 0.0f;
            }
        }
    }
}
px.y = 0.0f;
}
else
{
  px.x = (b2 - b1) / (m1 - m2);
  px.y = ((m1 * b2) - (m2 * b1)) / (m1 - m2);
}
}

// Find midpoint of p1px and p2px using midpoint formula
// (x, y) = ((x1 + x2)/2, (y1 + y2)/2)

// x = (x1 + x2)/2, y = (y1 + y2)/2
POINTF p5 = {(p1.x + px.x) / 2, (p1.y + px.y) / 2};

// x = (x1 + x2)/2, y = (y1 + y2)/2
POINTF p6 = {(p2.x + px.x) / 2, (p2.y + px.y) / 2};

// Find intersection of line perpendicular to p1px passing through p5
// and line perpendicular to p2px passing through p6
// and then solve the two equations (two equations two unknowns)

// b5 = y - m5x
float b5 = p5.y - (m5 * p5.x);

// b6 = y - m6x
float b6 = p6.y - (m6 * p6.x);

// Now find intersection of the two lines
// This is the center of the circle that the arc to formed from
// x = (b6 - b5)/(m5 - m6), y = (m5b6 - m6b5)/(m5 - m6)
POINTF px2;
if((m5 - m6) == 0) // Avoid Divide by zero
{
  px2.x = 0.0f;
  px2.y = 0.0f;
} else
{
  px2.x = (b6 - b5) / (m5 - m6);
  px2.y = ((m5 * b6) - (m6 * b5)) / (m5 - m6);
}

// Now find the radius of the circle using the distance formula
// r = sqrt((x2 - x1)^2 + (y2 - y1)^2)
float r = sqrt(pow(px2.x - px.x, 2.0f) + pow(px2.y - px.y, 2.0f));

// Set the arc direction, if the angle between the two points
// is greater than 180 degrees then change direction
if((fabs((j * fAngle) - (i * fAngle)) > (fPI))
  SetArcDirection(hdc, AD_COUNTERCLOCKWISE); // AD_CLOCKWISE or AD_COUNTERCLOCKWISE

else
  SetArcDirection(hdc, AD_CLOCKWISE); // AD_CLOCKWISE or AD_COUNTERCLOCKWISE

// Draw arc in bounding rectangle with center at px2 and radius r
Arc(hdc, // handle to device context
  (int)(px2.x - r + 0.5f), // x-coord of rectangle's upper-
  (int)(px2.y + r + 0.5f), // y-coord of rectangle's upper-
  (int)(px2.x + r + 0.5f), // x-coord of rectangle's lower-
  (int)(px2.y - r + 0.5f), // y-coord of rectangle's lower-
(int)(p1.x + 0.5f), // x-coord of first radial ending point
(int)(p1.y + 0.5f), // y-coord of first radial ending point
(int)(p2.x + 0.5f), // x-coord of second radial ending point
(int)(p2.y + 0.5f)); // y-coord of second radial ending point

//Rectangle(hdc, (int)(px2.x - r), (int)(px2.y + r), (int)(px2.x + r), (int)(px2.y - r));

//MoveToEx(hdc, (int)p1.x, (int)p1.y, NULL);
//LineTo(hdc, (int)p2.x, (int)p2.y);

//MoveToEx(hdc, (int)p3.x, (int)p3.y, NULL);
//LineTo(hdc, (int)p4.x, (int)p4.y);

//MoveToEx(hdc, (int)p1.x, (int)p1.y, NULL);
//LineTo(hdc, (int)p4.x, (int)p4.y);

//MoveToEx(hdc, (int)p2.x, (int)p2.y, NULL);
//LineTo(hdc, (int)p3.x, (int)p3.y);

//MoveToEx(hdc, (int)p5.x, (int)p5.y, NULL);
//LineTo(hdc, (int)px2.x, (int)px2.y);

//MoveToEx(hdc, (int)p6.x, (int)p6.y, NULL);
//LineTo(hdc, (int)px2.x, (int)px2.y);

SelectObject(hdc, hOldPen);
DeleteObject(hPen);
#pragma once

#include "RNA.h"
#include "Nussinov.h"

class BracketedGraph{
private:
    BracketedGraph( void);
    ~BracketedGraph( void);

public:
    void Draw( HDC hdc, RNA *pRNA, Nussinov *pNussinov, float xsize, float ysize);
};
#include "StdAfx.h"
#include "BracketedGraph.h"
#include "math.h" // for the floor function
#include "stdio.h" // for the sprintf function used to convert float to char

// FUNCTION: BracketedGraph
// Default Constructor
@
@return void
BracketedGraph::BracketedGraph(void)
{
}

// FUNCTION: ~BracketedGraph
// Default Destructor
@
@return void
BracketedGraph::~BracketedGraph(void)
{
}

// FUNCTION: Draw
// This function draws the BracketedGraph bitmap onto the device context
//@
// @param hdc HDC handle to a bitmap device context
//@
// @param *pRNA RNA pointer to a RNA class
//@
// @param xsize float x dimension, size of bitmap
//@
// @param ysize float y dimension, size of bitmap
//@
@return void
void BracketedGraph::Draw(HDC hdc, RNA *pRNA, Nussinov *pNussinov, float xsize, float ysize)
{
  float fBorder = 5.0f;
  float fCellSizeX = 20;
  float fCellSizeY = 20;
  float fCellLeft = fBorder;
  float fCellTop = fBorder;
  float xPos = fCellLeft, yPos = fCellTop;
  int RNALength = pRNA->getLength();
  int iCellsPerLine = (int)floor((xsize - (fBorder * 2)) / fCellSizeX); // Number of cells that fit on a line
  int iLineCount = RNALength / iCellsPerLine; // Number of complete lines
  int iRemainder = RNALength % iCellsPerLine; // Number of cells remaining on last line
  COLORREF Bkgrnd;
  COLORREF OldBkgrnd;
  COLORREF Foregrnd;
  COLORREF OldForegrnd;
  // Create the font and the foreground color
  HFONT hFont = CreateFont(15, 0, 0, 0, FW_SEMIBOLD, FALSE, FALSE, FALSE, ANSI_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, FF_DONTCARE, TEXT("Arial"));
HFONT hOldFont = (HFONT)SelectObject(hdc, hFont);
OldForegrnd = GetTextColor(hdc); // Get the Original foreground color

xPos = 15;
yPos = 10;

// Print the Nussinov Algorithm Type
SetTextColor(hdc, RGB(0, 0, 255)); // Set the foreground color to Blue
TextOutA(hdc, xPos, yPos, "Algorithm Type:", 15);
SetTextColor(hdc, RGB(163, 21, 21)); // Set the foreground color to Red
switch(pNussinov->getType())
{
case Nussinov::NussinovType::NussinovStandard:
    TextOutA(hdc, xPos + 135, yPos, "Nussinov Standard", 17);
    break;
case Nussinov::NussinovType::NussinovSCFG:
    TextOutA(hdc, xPos + 135, yPos, "Nussinov SCFG", 13);
    break;
default:
    // It's not a valid value, raise an exception. We should never get here...
    throw "An invalid Nussinov Type has been encountered."); // Could use
RaiseException(_ERROR);
    break;
}

// Print the sequence length on the facet
yPos += 20;
char strNumber[64]; // Buffer for number
itoa(RNALength, strNumber, 10);
SetTextColor(hdc, RGB(0, 0, 255)); // Set the foreground color to Blue
TextOutA(hdc, xPos, yPos, "Sequence Length:", 16);
SetTextColor(hdc, RGB(163, 21, 21)); // Set the foreground color to Red
TextOutA(hdc, xPos + 135, yPos, strNumber, strlen(strNumber));

// Print the step position
yPos += 20;
itoa(pRNA->getStepPosition(), strNumber, 10);
SetTextColor(hdc, RGB(0, 0, 255)); // Set the foreground color to Blue
TextOutA(hdc, xPos, yPos, "Step Position:", 14);
SetTextColor(hdc, RGB(163, 21, 21)); // Set the foreground color to Red
TextOutA(hdc, xPos + 135, yPos, strNumber, strlen(strNumber));

// Print the optimal score on the facet
yPos += 20;
switch(pNussinov->getType())
{
case Nussinov::NussinovType::NussinovStandard:
    itoa(pNussinov->getMatrix()[0][pRNA->getStepPosition() - 1], strNumber, 10);
    break;
case Nussinov::NussinovType::NussinovSCFG:
    sprintf(strNumber, "%4f", pNussinov->getCYKMatrix()[0][pRNA->getStepPosition () - 1]);
    break;
default:
    // It's not a valid value, raise an exception. We should never get here...
    throw "An invalid Nussinov Type has been encountered."); // Could use
RaiseException(_ERROR);
    break;
}
SetTextColor(hdc, RGB(0, 0, 255)); // Set the foreground color to Blue
TextOutA(hdc, xPos, yPos, "Optimal Score:", 14);
SetTextColor(hdc, RGB(163, 21, 21)); // Set the foreground color to Red
TextOutA(hdc, xPos + 135, yPos, strNumber, strlen(strNumber));

// Print the traceback path
yPos += 40;
SetTextColor(hdc, RGB(0, 0, 255)); // Set the foreground color to Blue
TextOutA(hdc, xPos, yPos, "Traceback Path: ", 15);
SetTextColor(hdc, RGB(0, 128, 0)); // Set the foreground color to Green
TextOutA(hdc, xPos + 5, yPos + 15, "Row: ", 4);
SetTextColor(hdc, RGB(163, 21, 21)); // Set the foreground color to Red
TextOutA(hdc, xPos + 5, yPos + 30, "Col: ", 4);
List::Node_type *node_ptr = pNussinov->listTraceback.head;
int Offset = 60;
while (node_ptr != NULL)
{
    char strNumberRow[15];
    itoa((node_ptr->i), strNumberRow, 10);
    int lenRow = strlen(strNumberRow);
    SIZE sizeRectRow;
    GetTextExtentPointA(hdc, strNumberRow, lenRow, &sizeRectRow);

    char strNumberCol[15];
    itoa((node_ptr->j), strNumberCol, 10);
    int lenCol = strlen(strNumberCol);
    SIZE sizeRectCol;
    GetTextExtentPointA(hdc, strNumberCol, lenCol, &sizeRectCol);

    if(sizeRectRow.cx > sizeRectCol.cx)
    {
        if(Offset + sizeRectRow.cx > (xsize - fBorder))
        {
            Offset = 20;
            yPos += 35;
        }

        SetTextColor(hdc, RGB(0, 128, 0)); // Set the foreground color to Green
        TextOutA(hdc, Offset, yPos + 15, strNumberRow, lenRow);

        SetTextColor(hdc, RGB(163, 21, 21)); // Set the foreground color to Red
        TextOutA(hdc, Offset + (sizeRectRow.cx / 2) - (sizeRectCol.cx / 2), yPos + 30, strNumberCol, lenCol);
    }
    else
    {
        if(Offset + sizeRectCol.cx > (xsize - fBorder))
        {
            Offset = 20;
            yPos += 35;
        }

        SetTextColor(hdc, RGB(0, 128, 0)); // Set the foreground color to Green
        TextOutA(hdc, Offset + (sizeRectCol.cx / 2) - (sizeRectRow.cx / 2), yPos + 15, strNumberRow, lenRow);

        SetTextColor(hdc, RGB(163, 21, 21)); // Set the foreground color to Red
        TextOutA(hdc, Offset, yPos + 30, strNumberCol, lenCol);
    }
    Offset += sizeRectCol.cx + 5;
    node_ptr = node_ptr->next;
}

// Print the bracketed graph
yPos += 70;
SetTextColor(hdc, RGB(0, 0, 255)); // Set the foreground color to Blue
TextOutA(hdc, xPos, yPos, "Bracketed Representation: ", 25);
SetTextColor(hdc, RGB(0, 128, 0)); // Set the foreground color to Green
//TextOutA(hdc, xPos + 5, yPos + 15, "Row: ", 4);
//TextOutA(hdc, xPos + 5, yPos + 30, "Col: ", 4);
SetTextColor(hdc, RGB(163, 21, 21)); // Set the foreground color to Red
Offset = 20;
for(int x = 0; x < RNALength; x++)
{
    SIZE sizeRectBase, sizeRectPair;
    char strBase[2];
    strBase[0] = pRNA->getSequence()[x];
    strBase[1] = '\0';
    if(x >= pRNA->getStepPosition())
    {
        SetTextColor(hdc, RGB(0, 128, 0)); // Set the foreground color to Green
        GetTextExtentPointA(hdc, "-", 1, &sizeRectBase);
        TextOutA(hdc, Offset, yPos + 15, "-", 1);
        GetTextExtentPointA(hdc, ".", 1, &sizeRectPair);
        SetTextColor(hdc, RGB(163, 21, 21)); // Set the foreground color to Red
        TextOutA(hdc, Offset + (sizeRectBase.cx / 2) - (sizeRectPair.cx / 2), yPos + 30, ".", 1);
    } else
    {
        SetTextColor(hdc, RGB(0, 128, 0)); // Set the foreground color to Green
        GetTextExtentPointA(hdc, strBase, 1, &sizeRectBase);
        TextOutA(hdc, Offset, yPos + 15, strBase, 1);
        GetTextExtentPointA(hdc, RGB(163, 21, 21)); // Set the foreground color to Red
        if(pRNA->getPairing()[x] == x)
        {
            GetTextExtentPointA(hdc, "-", 1, &sizeRectPair);
            TextOutA(hdc, Offset + (sizeRectBase.cx / 2) - (sizeRectPair.cx / 2), yPos + 30, "-", 1);
        } else
        {
            if(pRNA->getPairing()[x] > x)
            {
                GetTextExtentPointA(hdc, ")", 1, &sizeRectPair);
                TextOutA(hdc, Offset + (sizeRectBase.cx / 2) - (sizeRectPair.cx / 2), yPos + 30, ")", 1);
            } else
            {
                GetTextExtentPointA(hdc, ",", 1, &sizeRectPair);
                TextOutA(hdc, Offset + (sizeRectBase.cx / 2) - (sizeRectPair.cx / 2), yPos + 30, ",", 1);
            }
        }
    }
    if(Offset > (xsize - 30))
    {
        Offset = 20;
        yPos += 35;
    } else
    {
        Offset += 15;
    }
}
SetTextColor(hdc, OldForegrnd); // Set the foreground color back to original color
SelectObject(hdc, hOldFont); // Set the font back to original value
DeleteObject(hFont); // delete the font object