Music and Interaction Development Using the Countess Quanta Robot

Brad Pitney

Yin Shi

Chal McCollough

ECE 478/578

Perkowski

December 12, 2013

Table of Contents

[Introduction 3](#_Toc374990213)

[Section 1: IGA-based Motion Generation 3](#_Toc374990214)

[Existing Functionality and Limitations 4](#_Toc374990215)

[Move Representation 4](#_Toc374990216)

[IGA Structure 5](#_Toc374990217)

[Fitness Evaluation Process 6](#_Toc374990218)

[Parent Selection, Crossover, and Mutation 6](#_Toc374990219)

[Move Pruning 6](#_Toc374990220)

[Song Rating System 7](#_Toc374990221)

[IGA Performance Comparison 10](#_Toc374990222)

[Improvement of IGA Analysis 13](#_Toc374990223)

[Future Development 14](#_Toc374990224)

[Section 2: Person Tracking with Kinect 15](#_Toc374990225)

[Purpose 15](#_Toc374990226)

[Design Process 15](#_Toc374990227)

[Simulated Robot and Target 16](#_Toc374990228)

[Mapping to Servo Position 17](#_Toc374990229)

[Example Output 18](#_Toc374990230)

[Future Development 19](#_Toc374990231)

[Section 3: Right Arm Kinematics Modeling 20](#_Toc374990232)

[Conclusion 21](#_Toc374990233)

[Appendix 23](#_Toc374990234)

[IGA Trial Data 23](#_Toc374990235)

[Servo Range Data 25](#_Toc374990236)

[Right Arm Length Measurements 27](#_Toc374990237)

[Source Code for IGA Motion Generation 28](#_Toc374990238)

[main.cpp Modifications 28](#_Toc374990239)

[RhythmGenerator.h 29](#_Toc374990240)

[RhythmGenerator.cpp 31](#_Toc374990241)

[Source Code for Kinect Person Tracking 37](#_Toc374990242)

[MainWindow.xaml.cs Modifications 37](#_Toc374990243)

[RobotDisplay.xaml.cs 38](#_Toc374990244)

# Introduction

This project involved development in several areas related to improving the music generation and human-interaction capabilities of the Countess Quanta robot. The robot’s existing ability for playing the attached harp instrument was extended with new types of motion, and an interactive genetic algorithm was used to generate motion sequences based on user feedback. To support improved human-interaction, person tracking using a Microsoft Kinect was developed with a simulated Countess Quanta robot. Length measurements and servo range and orientation data was collected for the right arm of the Countess Quanta robot, for use in kinematics modeling.

This document is divided into three sections to address these main areas of improvement.

* Section 1: IGA-based Motion Generation
* Section 2: Person Tracking with Kinect
* Section 3: Right Arm Kinematics Modeling

# Section 1: IGA-based Motion Generation

An interactive genetic algorithm (IGA) was created to evolve new motion sequences to be used by the Countess Quanta robot in playing the existing harp instrument. The IGA chromosome is used to encode a sequence of hand motions, which defines a “song” that can be played by the robot. The available hand motions include the ability to lift and reposition the hand, which offers an improvement over the existing rhythm files. During fitness evaluation, the robot performs each song for a human user, who then rates the relative quality of each song. Along with the IGA development process, this section includes a review of the song rating system and a performance analysis of the selected GA parameters.

## Existing Functionality and Limitations

The existing software for the Countess Quanta robot allows the robot to play from a selection of seven rhythm files. Each of these rhythm files is relatively simple, consisting of a series of 3 to 5 moves using only the wrist servo (Servo 0). One of the main limitations of the existing rhythms is that they only utilize a single servo. The robot moves the arm into position with the hand pressed against the instrument strings, and each rhythm file moves the hand left and right in different patterns to strum the strings. This places some large restrictions on what kind of sounds the robot is able to create compared to what a human would be able to do playing the same instrument. For instance, when a human strums an instrument, they might lift their hand at the end of the motion to change the sound. They might also position their hand at different locations above the strings, before lowering their hand against the strings and strumming. To capture this extra complexity, an additional degree of freedom was included, to let the robot lift its hand while playing.

## Move Representation

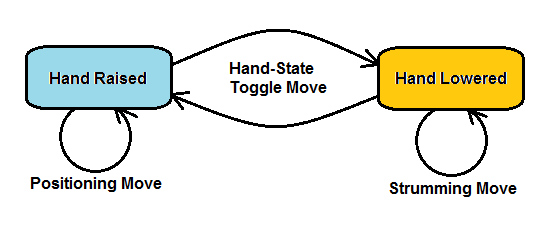
To capture this behavior, four types of moves were defined:

1. Robot lifts hand off of the strings.
2. Robot lowers hand onto the strings.
3. Robot moves raised hand above the strings, to a new position.
4. Robot moves lowered hand across the strings, strumming the instrument.

The robot can only perform move types 1 and 4 while the hand is in the ‘lowered’ state, and can only perform moves 2 and 3 while the hand is in the ‘raised’ state. Additionally, the actual servo motions for move types 3 and 4 can be identical, and only change functionally depending on whether the hand is currently raised or lowered. Because of these features, the moves can be simplified to two types:

1. A hand-state toggling move, which raises the hand off of the strings if it is currently lowered, or lowers the hand onto the strings if it is currently raised.
2. A positioning/strumming move, which strums the hand across the strings if the hand is currently lowered, or repositions the hand above the strings if the hand is currently raised.

The figure below is a simple state machine showing this behavior:



## IGA Structure

To structure the GA, a chromosome was defined consisting of ten genes, with each gene representing a single movement in a sequence. Each gene contains a ‘toggleHandState’ Boolean, to store whether this is a move to raise/lower hand, or whether this will move the hand side-to-side. A ‘wristPosition’ integer within each gene stores the new side-to-side hand position, which is used if ‘toggleHandState’ is false. When a gene is created or mutated, the ‘toggleHandState’ value is randomized with a 50% chance of either state. If this value is false (i.e. ‘wristPosition’ will be used), then the ‘wristPosition’ value is set to a random integer within the allowed 600 to 1300 wrist position range.

The actual servo motions corresponding to a single movement were determined by experimenting with the Countess Quanta robot. Before running any motions, the existing motion script file called ‘MoveArm’ is executed, to move the robot’s right arm into instrument playing orientation. Once in position, the hand is raised and lowered by changing the position of ‘servo::Elbow\_Extension’ (Servo 2). Moving this servo to position 720 raises the hand off of the instrument strings, and setting this servo to position 884 lowers the hand so that it touches the strings. The strumming and repositioning moves are executed by changing the position of servo::Wrist\_right (Servo 0). Integer values in the range 600 to 1300 are allowed, since this range keeps the robot’s hand on or above the strings of the instrument. Prior to executing a move sequence, the hand is always raised and the wrist position is set to 950, for consistency.

## Fitness Evaluation Process

During execution, the GA first randomly generates a population of ten chromosomes. The fitness of each individual in the population is then evaluated by converting each gene in the chromosome into a servo move command. The sequence of move commands is sent to the robot servo controller with 100ms intervals between each motion. The resulting robot performance is then evaluated by a human viewer, who rates the quality of the song on a scale from 1 to 9. The viewer inputs their rating value into the program terminal window, and this value is stored as the fitness of the corresponding individual.

To save time during the evaluation process, the system automatically skips re-evaluation of any individuals that had already been evaluated in a previous generation. That is, if a particular move sequence was rated by the user in Generation 1 and this sequence happens to be passed on unchanged into Generation 2, then we simply use the previous fitness value for this individual during the Generation 2 evaluations, rather than requiring the user to rate this song again. This also reduces error due to the user possibly rating identical songs differently in each generation.

## Parent Selection, Crossover, and Mutation

Once the viewer has rated all ten individuals, the GA uses roulette wheel selection to select ten parents for the next generation. For each pair of parents, there’s a 60% chance that 2-point crossover is then applied to create two offspring. The remaining 40% of the time, the parents are passed along as offspring, unmodified. After the crossover process, each gene in the offspring population is subjected to a 1% mutation chance. If a gene is selected for mutation, then it is replaced with a new, randomly generated move (i.e. new random ‘toggleHandState’ and ‘wristPosition’ values). The evaluation process then repeats for this new population, and the GA progresses until the specified number of generations is reached. The selected 60% crossover and 1% mutation rates represent typical values that had been used successfully in previous projects.

## Move Pruning

One question that arose early on was whether to blindly perform all moves described in a chromosome, or to include some logic for pruning moves that don’t affect the actual strumming of the instrument. For instance, the move list might include a move to raise the hand above the strings, followed by a series of multiple wrist moves that just wiggle the hand in the air, followed by a move to lower the hand onto the strings. In this case, only the last wrist move actually matters, since this move affects the position that the hand will be in when it is finally lowered onto the strings – every other wrist move prior to this is superfluous. To prevent this situation, logic was added to ignore all but the last wrist move prior to lowering the hand.

Another scenario where move pruning was considered is the case where multiple ‘hand-toggle’ moves occur in series. For instance, the move list might include a move to lift the hand off of the strings, immediately followed by a move to lower the hand back onto the strings. In this case, the hand hasn’t been repositioned, so it would seem that this move was pointless. However, in this case, the move sequence does cause the hand to impact the strings, which would have an effect on the song. Similarly, a human playing a stringed instrument might strum the instrument and then touch the strings again to dampen the sound. Because of this, it was decided to preserve this ‘raise then lower’ move sequence, since it might allow for interesting behavior to appear.

## Song Rating System

One of the challenges of this project was in determining how to rate the quality of a given motion sequence. This required experimental testing with the robot, in order to get a sense for what kinds of sounds this system allows. Once some experience was acquired with the range of sound quality that the instrument and robot was able to produce, it was possible to provide a basic rating of how pleasant or unpleasant a song was, compared to others that were being produced. Below are some examples of generated motion sequences that resulted in songs that were found to be either ‘good’ or ‘bad’. The sounds produced by a motion sequence are not obvious from reading the servo coordinates, so descriptions of the performance and explanations of the perceived quality are included.

**Good Song 1:**

Repositioning hand to 1174.

Lowering hand onto strings.

Strumming strings to 1113.

Strumming strings to 1288.

Strumming strings to 740.

Strumming strings to 1201.

Strumming strings to 685.

Raising hand off of strings.

Lowering hand onto strings.

Strumming strings to 806.

When playing this song, the robot makes several large strumming motions in sequence, lifting its hand at the end. It then lowers its hand and makes one last strumming motion. The vigorous strumming seems to provide a sense of enthusiasm. Lifting the hand before the last strum added variety, and seemed more like the kind of motion a human would make if they were playing the instrument.

**Good Song 2:**

Lowering hand onto strings.

Strumming strings to 1251.

Raising hand off of strings.

Skipping superfluous move to 1074.

Skipping superfluous move to 1211.

Skipping superfluous move to 769.

Skipping superfluous move to 1151.

Repositioning hand to 775.

Lowering hand onto strings.

Strumming strings to 1088.

In this song, the robot makes two large strumming motions from different locations. The strumming sounded very deliberate, and simulated how a human might use the instrument.

**Bad Song 1:**

Lowering hand onto strings.

Raising hand off of strings.

Repositioning hand to 1154.

Lowering hand onto strings.

Raising hand off of strings.

Lowering hand onto strings.

Raising hand off of strings.

Repositioning hand to 1052.

Lowering hand onto strings.

Strumming strings to 1136.

In this song, the robot pats the strings repeatedly, and strums a small distance at the end. This kind of motion might work better for a drum than a stringed instrument.

**Bad Song 2:**

Lowering hand onto strings.

Raising hand off of strings.

Skipping superfluous move to 1214.

Skipping superfluous move to 632.

Skipping superfluous move to 1168.

Skipping superfluous move to 671.

Skipping superfluous move to 1146.

Repositioning hand to 1015.

Lowering hand onto strings.

Strumming strings to 763.

This song appeared while evolving decent songs with many strumming motions. It shows an example of a potentially good song that was greatly reduced in quality by the placement of an extra hand state toggle move just prior to the strumming sequence. The hand is raised off of the strings before the sequence, so the large strumming motions are skipped entirely. The resulting song is a single strumming motion, which sounded very boring compared to the other rapid strumming songs that the GA had been evolving.

After viewing and evaluating many songs, some specific features could be identified, which often determined song quality. Of course, this rating system is very subjective, and another user might have a very different set of criteria, depending on what kind of song they are trying to evolve. Here are some of the criteria that were identified:

**Features of a ‘good’ song:**

* The song involves enthusiastic strumming, due to sequences of wrist moves with large changes in wrist position.
* The song involves a sequence of strumming, raising and repositioning the hand, and then strumming again.

**Features of a ‘bad’ song:**

* The song involves very few moves, due to much of the sequence being superfluous positioning moves.
* The song involves little motion, due to strumming moves having little position change between wrist coordinates.
* The song consists mostly of ‘patting’ the strings, due to sequences of hand state toggle moves.

## IGA Performance Comparison

From experimenting with the IGA over several trials, it was found that songs from later generations tended to sound similar to the highly rated songs of earlier generations. From this experience, it seemed that the IGA was generally successful in evolving a particular kind of song. To perform a more quantitative analysis of the IGA performance, song rating data was gathered from three trials. Each trial consisted of evolving a population of ten individuals over a period of five generations, using the “Standard GA” properties described earlier (60% crossover rate, 1% mutation rate).

Note that this displayed data for each generation excludes any individuals that were brought over unmodified from the previous generation. The displayed ratings within each generation (after the first generation, since this population is random) are based only on new songs that were created from the GA’s crossover and mutation routines. The logic is that the new ratings in each generation better represents the GA’s capability for evolving improved individuals based on the user’s feedback. To be clear, the IGA itself is still using the unmodified individuals in its evolution process, they are just omitted from the displayed data below.

The figure below shows a Microsoft Excel plot of the average song rating of each of the five generations, for the three trials. This chart includes best-fit linear trend lines for the data of each trial.

From the trend lines, we can see that each trial showed some increase in average song rating as the generations progressed. Trial 3 showed the least increase, but it also happened to start with the highest average rating of the initial populations. From the individual data points, we can see that the average rating sometimes varied greatly between generations, as in the case of the Trial 2 and Trial 3 value at Generation 4, which are both below the adjacent generations. This variation is likely an impact of the relatively small GA population, which results in collecting a small number of data points within each generation.

To get a better idea of how well these “Standard GA” parameters performed compared to other potential IGA settings, two other configurations were tested and trial data was collected for each of these. A “Mutation GA” was defined by setting the crossover rate to 0% and raising the mutation rate to 30%. This configuration relies entirely on occasional mutations of individual genes to generate new motion sequences. A “Random Motions” configuration was also defined by setting the crossover rate to 0% and the mutation rate to 100%. With this configuration, every generation consists entirely of new randomly generated individuals. This random configuration acts as a kind of control group, and should show some baseline song rating with no expected change over each generation.

The figure below shows the results of these new trials. The “Standard GA” plot consists of an average of the three trials discussed above. The “Mutation GA” and “Random Motions” plots are each the results of similar trials, with populations of ten individuals evolved over five generations. The best-fit linear trend lines for each data set are also included.

From the trend lines, we can see that the “Standard GA” parameters tended to result in the best overall increase in average song rating across the generations, compared to the other methods. The “Mutation GA” also showed an increase in song rating over the generations, although this wasn’t as pronounced as with the “Standard GA”. The “Mutation GA” did appear to generate a greater variety of songs in later generations, compared to the “Standard GA” which tended to focus in on a single type of song. The IGA program includes the option to save songs to a file at any point, so the “Mutation GA” might be useful for exploring different types of songs and saving off any that are appealing.

The “Random Motion” configuration shows a roughly horizontal line, as we would expect from a method that doesn’t adapt to user feedback. These “Random Motion” results also indicate that the average rating for a completely random song within this framework tended to be around 3. It’s interesting to note that the “Random Motion” average ratings also showed high variability, ranging from an average rating of 2.0 for Generation 3 to a value of 4.2 for Generation 4. Since all generations should ideally show the same average with this method, this range reflects the kind of variation that we would expect to see in the other trial data, most likely due to the small number of individuals in the population.

## Improvement of IGA Analysis

There are several limitations of this IGA analysis that could be improved with future work. One significant issue is the small population size that was used during testing. This issue is obvious from looking at how much the average ratings vary between each generation when using the “Random Motion” GA parameters, since this configuration is expected to show a consistent average rating at each generation. Increasing the number of chromosomes in the population would result in a more accurate average rating value for each generation, and the “Random Motion” output should more clearly show a horizontal line with little variance between generations.

Similarly, the small number of generations limits our evaluation of the GA learning curve. As the GA adapts over multiple generations, we would expect to see the average song rating increase and eventually plateau. To get an accurate assessment of the GA performance, we would need to increase both the population size, to increase the average rating accuracy, and the number of generations, to see how the GA performs over many generations. Note that both of these values affect the number of songs per trial that must be manually evaluated, so total evaluation time would also need to be considered.

More accurate analysis of the GA learning curve would allow for better comparison between different GA parameters. This project considered only one crossover-based GA, with 60% crossover and 1% mutation rates. It could be useful to compare this GA with those using different combinations of crossover and mutation rates, to see if other parameters result in better performance for this application. Similarly, different parent selection schemes and crossover/mutation mechanics could be examined.

Other limitations of this IGA analysis include psychological effects on the rating process. For instance, knowledge of the particular GA parameters or the current generation number could introduce bias in the user who is rating the songs. The user might assign higher ratings to songs in later generations due to the expectation that the GA should be providing higher quality songs in later generations. The effects of bias could be reduced by using blind experimentation techniques, such as randomizing and hiding from the user the type of GA that is being evaluated in a particular trial. To reduce bias from the user knowing the relative generation throughout a trial, songs from several trials could be evaluated in parallel, while hiding the trial and generation information from the user. Random songs might also be inserted into the rating process, and changes in the ratings from these random songs might be used to evaluate and compensate for the psychological impact on the ratings.

## Future Development

In addition to the IGA analysis improvements, several other possible enhancements for this project were identified:

* A new ‘delay’ move type could be added to introduce a pause in the move sequence. This could be implemented as a parameter in the gene, along with a random integer specifying the time delay.
* A new ‘repeat’ move type could be used to repeat some number of moves in the sequence. For instance, processing a gene with this parameter might specify that the next five moves following this one should be repeated three times before continuing on to the sixth move. The number of moves to repeat and the number of repetitions could be randomized within some range.
* The initial IGA population could be seeded with some known ‘good’ songs, rather than being entirely randomly generated.
* Other input schemes could be used to collect rating data for songs. For instance, instead of entering the rating in a terminal window, a Microsoft Kinect could be used to extract rating data from the gestures or facial expressions of a viewed audience.

# Section 2: Person Tracking with Kinect

For this section, Microsoft Kinect hardware was used along with Microsoft’s ‘Kinect for Windows SDK v1.8’ to implement person tracking for a simulated Countess Quanta robot. The sample program ‘SkeletonBasics-WPF’ included within the ‘Kinect for Windows Developer Toolkit v1.8’ was extended to support this simulation. Skeleton positioning data was used to control the servo 10 neck rotation of the simulated robot, so that it would turn to face the tracked person. A simulated robot was used in place of the actual Countess Quanta robot, due to issues encountered while trying to install and utilize the Kinect hardware under the Ubuntu 12.04 operating system, which hosts the existing Countess Quanta motion control software.

## Purpose

The intent of this project was to gain experience with using the Kinect system and add basic Kinect support to the Countess Quanta robot that could be extended in the future. Person tracking was selected because it complements the existing voice synthesis and music playing functionality that has already been developed for the Countess Quanta robot. For instance, the person tracking feature could be used during interaction with a human by having the robot turn to face the person before speaking. Person tracking can also be used as a starting point to develop other functionality, such as gesture recognition, since the same underlying skeleton tracking methods in the SDK could be used.

## Design Process

The original intention was to implement this person tracking feature on an Ubuntu 12.04 virtual machine, to allow for control the Countess Quanta robot hardware through the existing software, which had been developed under Ubuntu. However, the process of establishing communication with the Kinect hardware was found to be far more complex and time consuming under Ubuntu than it had been under Windows 7. This is due in part to having to integrate separate hardware drivers, communication libraries, and skeleton tracking software from different developers, while ensuring that each component is of a version that is compatible with the operating system and with each other component. For example, many online guides to running Kinect under Ubuntu 12.04 indicate that the latest version of OpenNI should be installed, since this is a primary SDK used in communicating with the Kinect. However, after much research and experimentation, it was found that support for the Kinect hardware under Linux has been removed in the 2.x version of OpenNI (latest version is 2.2.0.33, at the time of writing). Due to these issues, integration with the robot hardware was postponed in favor of simulated robot development under Windows 7.

In Windows 7, both the ‘Kinect for Windows SDK v1.8’ and the ‘Kinect for Windows Developer Toolkit v1.8’ were downloaded from Microsoft’s website and installed. The included ‘Developer Toolkit Browser v1.8.0’ software displays a list of many sample programs for working with different aspects of the Kinect hardware. From here, the ‘Skeleton Basics-WPF’ sample program was executed, and immediately recognized the Kinect hardware and began displaying skeleton outlines of anyone within the Kinect’s visual range. From the Toolkit Browser, the ‘Skeleton Basics-WPF’ source code was downloaded and modified to simulate the intended behavior of the Countess Quanta robot.

## Simulated Robot and Target

To simulate the robot, a new ‘RobotDisplay’ form was added to the ‘Skeleton Basics-WPF’ project. Code was added to automatically display this form on startup, so that both the existing skeleton display and the simulated robot display would be visible. In the RobotDisplay class, code was added to display a circle at the top of the form to represent a simplified top-down view of the Countess Quanta robot. This circle also represents the location of the Kinect sensor oriented towards the bottom of the form, since the Kinect will eventually be mounted to the front of the actual Countess Quanta robot. To represent the target person in front of the simulated robot, a second circle was added to the form, below the first circle. A red line was drawn from the robot to the target, to indicate the current facing of the robot’s head.

After reviewing the existing sample code and performing some tests with the Kinect, it was found that the Kinect SDK provides three-dimensional Cartesian positioning data (in meters) for each skeleton joint. Each SkeletonJoint object contains an X coordinate reflecting the horizontal position of the joint, with the very center of the Kinect image being zero, the right side of the image being positive values, and the left side being negative. Similarly, the Y coordinate represents placement on the vertical axis. The Z coordinate represents the joint’s distance in the direction away from the front of the sensor, with the Kinect sensor being the zero point.

Since it makes the most sense for the robot to look at the target’s face, only the location of the skeleton’s ‘Head’ joint needs to be tracked. To modify the target’s location, an UpdateTargetPosition method was added to the RobotDisplay class, and this method was called from the existing DrawBone method, which processes the skeleton joint data in the sample application. When the DrawBone method processes a ‘Head’ joint, it calls UpdateTargetPosition and passes the position data to this method. UpdateTargetPosition then displays the latest XYZ coordinates of the joint on the RobotDisplay form. To update the target’s top-down position relative to the robot, the X and Z coordinates are scaled and oriented to match the display on the form.

## Mapping to Servo Position

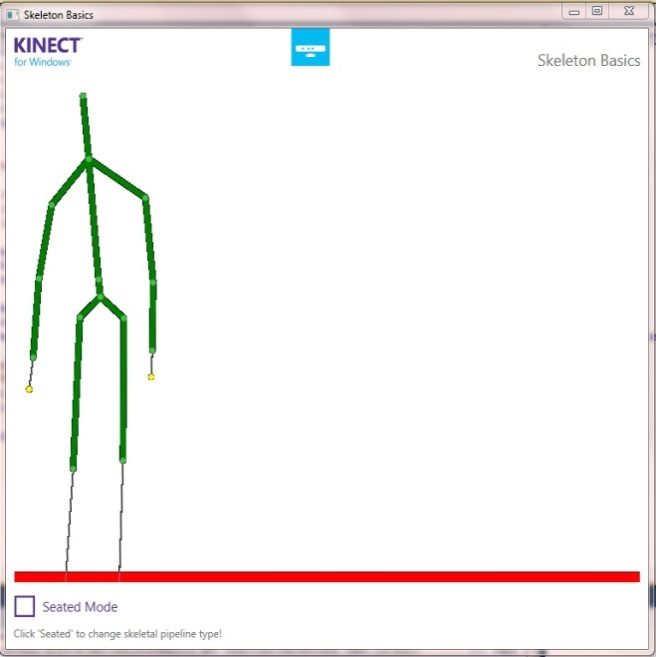
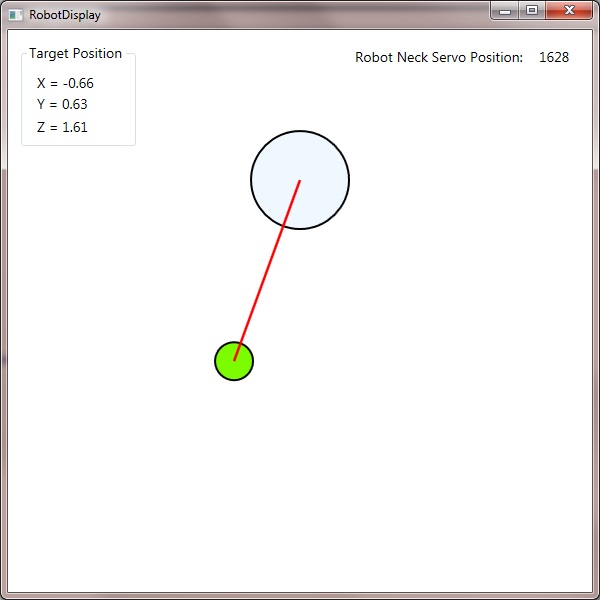
To determine the required position of the neck rotation servo for the robot to correctly face the target, experiments were performed with the robot to collect range data on this servo. The allowed physical range of this servo was found to extend from position value 1350 at the farthest left rotation, to position value 1750 at the farthest right. At value 1550, the robot appears to be facing straight forward.

To map the Kinect data to these servo values, code was added to calculate the rotation angle off of center of the simulated robot’s facing direction, using arctangent and the targets X and Z Kinect location. This was then scaled by an approximated ‘servo increments per radian’ value and the 1550 servo position offset was added. For the ‘servo increments per radian’, an estimated value of 200 was used, which kept the neck rotation within the 1350-1750 limits while the target is within the Kinect area:

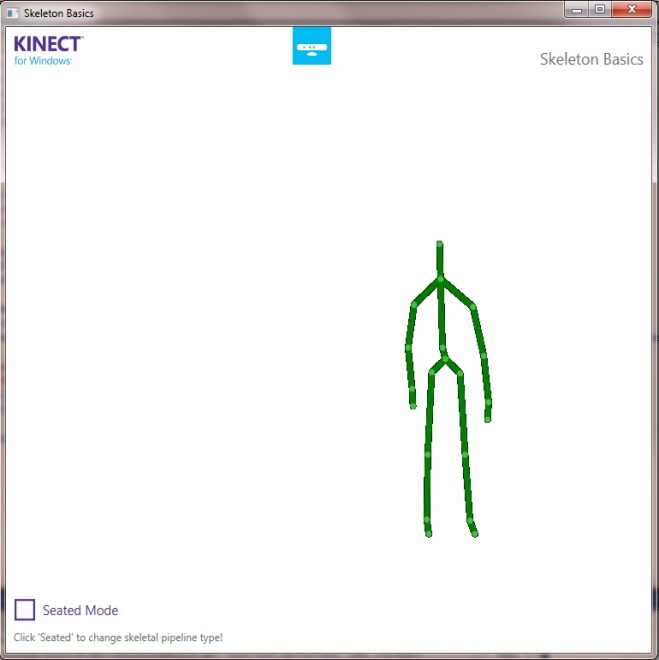
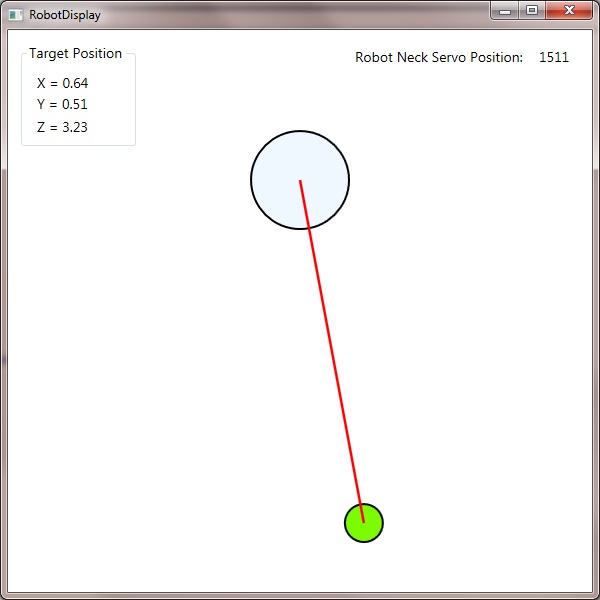
This ‘servo increments per radian’ value will need to be revised once this logic is applied to the actual robot hardware. For now, the final servo position is displayed at the top of the RobotDisplay form. The expectation is that, if a motion control path existed (i.e. by either porting this Kinect functionality to Ubuntu, or moving the robot motion code to Windows), this value could be sent to the robot hardware and would result in the correct physical motion, after some minor calibrations.

## Example Output

The figures below are screenshots from the application, taken while the Kinect was tracking a person standing in different locations. The window on the left is the ‘Skeleton Basics’ sample application, which displays the skeleton data of a viewed person. The window on the right is the new RobotDisplay form, which shows the top-down view of the simulated robot (large blue circle) and the target person (small green circle). Note that the skeleton display is actually mirrored compared to what the Kinect actually sees, which matches the top-down view shown in the RobotDisplay form. Since the simulated robot is oriented towards the bottom of the form, it has to turn its head to the right to face the target. The displayed neck servo position of 1628 reflects this head rotation.

The figures below show the target in another location. In this case, the target has stepped back, away from the robot, and has stepped to the right of the robot (i.e. to the robot’s left). To track the target, the robot turns its head slightly left, to servo position 1511.

## Future Development

During development of the person tracking feature, many ideas arose for how this, and other Kinect functionality, could be extended in the future, to improve the human interaction capabilities of the Countess Quanta robot. Listed below are a few of these ideas:

* Integrate the person tracking feature with the Countess Quanta motion control software, to allow the robot to rotate its head and track the target.
* Add support for vertical tracking, allowing the robot to tilt its head up and down to look at the target’s face.
* Add support for multiple targets, such as periodically turning to face each detected target.
* Add support for recognizing and responding to gestures from the target.
* Integrate existing robot eye motion and base wheel motion to give the robot additional options in how it faces the target. For instance, the robot might face the target by turning its body, rotating its head, moving its eyes, or any combination of the three.
* Incorporate servo velocity control to make motions appear more natural. This could include modifications to reflect some emotional state, such as turning quickly to face the target if the robot is alert or turning slowly after some delay if the robot is tired.
* Use the target facing to display attentiveness of the robot. For instance, the robot might watch the target closely if the robot is focused, or it might periodically look around the room if the robot is distracted.

# Section 3: Right Arm Kinematics Modeling

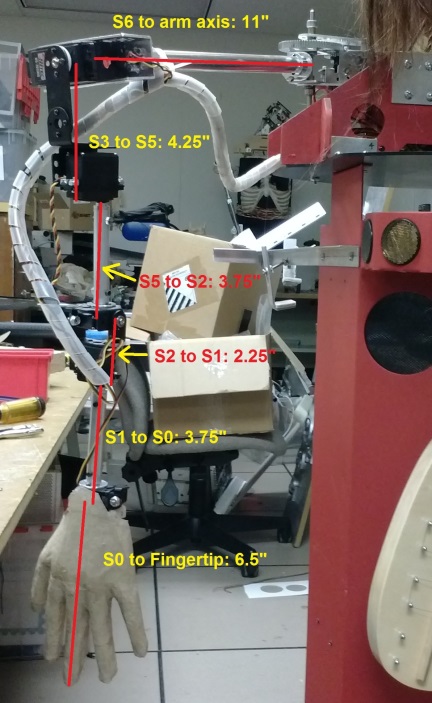
The right arm of the Countess Quanta robot is controlled by six servo motors placed at joints along the arm. To support improved modeling of the arm's kinematics, measurements of the arm's structure were collected. Range data was collected for each servo in the robot by moving each servo to its physical limit and recording the value just prior to reaching this point. That is, moves to positions beyond these points result in high stress on the servo without additional motion. Measurements were also taken of the lengths of each arm segment between the servo joints.

The intent was for these measurements to be used to create a kinematics model of the right arm. This model could then be used to display the arm in a 3D renderer, and to support inverse kinematics calculations for moving the end effector in Cartesian coordinates. Some software work has been done, but this feature is still in development. The collection data is presented in this report to support future work in this area.

The figures below display the right arm and are annotated showing the motion of each of the six servos and min/max directions used by the servo controller. Additional descriptions, servo position data, and length measurements can be found in the Appendix.

The figure below shows the length measurements of the arm segments between each servo joint.



# Conclusion

In this project, several areas relating to the music generation and human-interaction capabilities of the Countess Quanta robot were improved. Section 1 discussed the development of an IGA to evolve new motion sequences for playing the existing harp instrument. Moves types were defined to allow the robot to lift and lower its hand, to support strumming motions across the strings, and repositioning moves above the strings. IGA fitness evaluation was implemented by performing songs for the user and collecting song quality rating data from the user. The song rating system was reviewed and the IGA performance was evaluated and compared to GAs with alternate parameters. Section 2 discussed the use of Microsoft Kinect hardware to implement person tracking in a simulated Countess Quanta robot. The ‘SkeletonBasics-WPF’ sample program from the ‘Kinect for Windows Developer Toolkit v1.8’ was extended to support the robot simulation display. Spatial positioning data from the Kinect was used to update the location of the target which was then used to calculate the approximate neck servo position that would be required for the physical robot to face the target. Section 3 presented measurement data that was collected to support kinematics modeling of the robot’s right arm. Ideas for application of this data are also discussed.

# Appendix

## IGA Trial Data

This section contains the data that was collected during testing of the motion generation IGA. Note that this displayed data for each generation excludes any individuals that were brought over unmodified from the previous generation. The displayed ratings within each generation (after the first generation, since this population is random) are based only on new songs that were created from the GA’s crossover and mutation routines. The logic is that the new ratings in each generation better represents the GA’s capability for evolving improved individuals based on the user’s feedback. To be clear, the IGA itself is still using the unmodified individuals in its evolution process, they are just omitted from the analysis.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Standard GA Trial 1** | |  |  |  |  |
| **Generation:** | **1** | **2** | **3** | **4** | **5** |
|  | 2 | 3 | 6 | 2 | 5 |
|  | 2 | 3 | 2 | 5 | 5 |
|  | 3 | 4 | 2 | 1 | 5 |
|  | 5 | 2 | 5 | 7 | 3 |
| **New** | 6 | 1 |  | 5 | 6 |
| **Ratings:** | 2 | 2 |  | 5 | 6 |
|  | 3 | 5 |  | 6 | 6 |
|  | 2 |  |  |  |  |
|  | 1 |  |  |  |  |
|  | 5 |  |  |  |  |
| **Average:** | 3.10 | 2.86 | 3.75 | 4.43 | 5.14 |
|  |  |  |  |  |  |
| **Standard GA Trial 2** | |  |  |  |  |
| **Generation:** | **1** | **2** | **3** | **4** | **5** |
|  | 5 | 7 | 8 | 2 | 8 |
|  | 4 | 2 | 3 | 8 | 7 |
|  | 2 | 4 | 2 | 8 | 7 |
|  | 8 | 4 | 8 | 1 | 8 |
| **New** | 3 | 3 | 8 | 2 | 7 |
| **Ratings:** | 3 | 7 | 2 | 2 | 8 |
|  | 6 | 7 |  |  |  |
|  | 4 | 3 |  |  |  |
|  | 2 |  |  |  |  |
|  | 5 |  |  |  |  |
| **Average:** | 4.20 | 4.63 | 5.17 | 3.83 | 7.50 |
|  |  |  |  |  |  |
| **Standard GA Trial 3** | |  |  |  |  |
| **Generation:** | **1** | **2** | **3** | **4** | **5** |
|  | 4 | 6 | 7 | 1 | 5 |
|  | 7 | 4 | 5 | 2 | 8 |
|  | 5 | 4 | 5 | 5 | 5 |
|  | 4 |  | 7 | 6 | 8 |
| **New** | 8 |  |  |  | 7 |
| **Ratings:** | 5 |  |  |  | 1 |
|  | 7 |  |  |  | 6 |
|  | 3 |  |  |  | 3 |
|  | 3 |  |  |  | 6 |
|  | 1 |  |  |  |  |
| **Average:** | 4.70 | 4.67 | 6.00 | 3.50 | 5.44 |
|  |  |  |  |  |  |
| **Mutation-based GA** | |  |  |  |  |
| **Generation:** | **1** | **2** | **3** | **4** | **5** |
|  | 2 | 3 | 4 | 8 | 4 |
|  | 7 | 5 | 7 | 2 | 8 |
|  | 2 | 3 | 1 | 4 | 3 |
|  | 2 | 3 | 3 | 6 | 2 |
| **New** | 7 | 4 | 6 | 1 | 3 |
| **Ratings:** | 3 | 3 | 3 | 3 | 6 |
|  | 6 | 7 | 5 | 4 | 1 |
|  | 3 | 6 | 4 | 2 | 7 |
|  | 3 | 3 | 1 |  | 7 |
|  | 3 | 2 | 4 |  |  |
| **Average:** | 3.80 | 3.90 | 3.80 | 3.75 | 4.56 |
|  |  |  |  |  |  |
| **Random Motions** | |  |  |  |  |
| **Generation:** | **1** | **2** | **3** | **4** | **5** |
|  | 3 | 2 | 2 | 4 | 3 |
|  | 5 | 6 | 1 | 2 | 3 |
|  | 4 | 2 | 1 | 5 | 2 |
|  | 3 | 3 | 1 | 4 | 5 |
| **New** | 2 | 4 | 4 | 1 | 2 |
| **Ratings:** | 3 | 4 | 3 | 7 | 3 |
|  | 3 | 2 | 4 | 5 | 1 |
|  | 4 | 2 | 1 | 4 | 2 |
|  | 2 | 4 | 1 | 7 | 1 |
|  | 3 | 3 | 2 | 3 | 3 |
| **Average:** | 3.20 | 3.20 | 2.00 | 4.20 | 2.50 |

## Servo Range Data

This section list the effective movement range of each servo on the Countess Quanta robot, along with descriptions of what each movement represents. The Min and Max values indicate the point just prior to the mechanism reaching the physical limit. Commanding the servo to a position past this point results in high stress on the servo without additional motion. The Mid values represent some meaningful neutral position, for particular servos. Note that all directions are stated with respect to the robot itself (i.e. from the robot's perspective, or someone standing behind the robot). Also note that references to a 'human thumb' or 'human pinky' specify the direction of the right hand, if this hand was oriented as a human hand. The current right hand is actually implemented with cardboard 'left hand', which can make things confusing.

Servo 0:

Part: Right Arm Wrist left/right

Min: 700 (toward human thumb)

Mid: 1400 (hand straight)

Max: 2300 (toward human pinky)

Servo 1:

Part: Right Arm Wrist rotation

Min: 800 (toward human pinky)

Mid: 1350 (wrist aligned with shoulder)

Max: 2400 (toward human thumb)

Servo 2:

Part: Right Arm Wrist Up/down

Min: 700 (wrist straight along arm)

Mid: 750 (wrist straight along arm)

Max: 2100 (wrist folds towards body)

Servo 3:

Part: Right Arm Up/down

Min: 700 (arm swings in front)

Mid: 1400 (arm hangs straight down)

Max: 2650 (arm swings behind)

Servo 4: not used

Servo 5:

Part: Right Arm rotation

Min: 550 (toward human pinky)

Mid: 1500 (wrist aligned with shoulder)

Max: 2400 (toward human thumb)

Servo 6:

Part: Right Arm Shoulder left/right

Min: 300 (into body, straight forward)

Max: 2250 (away from body, straight right from body)

Servo 7: not used

Servo 8:

Part: Eyes left/right

Min: 1000 (eyes look right)

Mid: 1600 (eyes look center)

Max: 2000 (eyes look left)

Servo 9:

Part: Head up/down

Min: 1000 (head down)

Mid: 1500 (head center)

Max: 2000 (head up)

Servo 10:

Part: Head left/right

Min: 1350 (head turns left)

Mid: 1550 (head in center)

Max: 1750 (head turns right)

Servo 11:

Part: Mouth open/close

Min: 1000 (mouth close)

Max: variable, due to collision with neck: 1350 with Servo 9 at 1500, 1550 with Servo 9 at 2000 (mouth open)

Servo 12:

Part: Left Arm Ring finger

Min: 1350 (extended)

Max: 2000 (retracted)

Servo 13:

Part: Left Arm Middle finger

Min: 1250 (extended)

Max: 1800 (retracted)

Servo 14:

Part: Left Arm Pinky finger

Min: 1200 (extended)

Max: 1750 (retracted)

Servo 15:

Part: Left Arm

Min: 1250 (towards body)

Max: 1600 (away from body; falls at ~1700)

Servo 16:

Part: Left Arm Thumb

Min: 1500 (extended)

Max: 1700 (retracted)

Servo 17:

Part: Left Arm Index finger

Min: 1000 (retracted, opposite of other fingers)

Max: 1500 (extended, opposite of other fingers)

## Right Arm Length Measurements

Wrist (servo 0) to tip of middle finger: 6.5 inches

Wrist rotating axis (servo 1) to wrist (servo 0): 3.75

Wrist bending axis (servo 2) to wrist rotating axis (servo 1): 2.25

Arm rotating axis (servo 5) to wrist bending axis (servo 2): 3.75

Arm raising axis (servo 3) to arm rotating axis (servo 5): 4.25

Shoulder (servo 6) to arm axis: 11

## Source Code for IGA Motion Generation

The code below was added to the existing ‘GuidebotupperPlayMusic’ project to support the new IGA-based music generation. This includes a new (temporary) main() function, used for testing, and new RhythmGenerator.h and RhythmGenerator.cpp files, which contain the new RhythmGenerator class.

### main.cpp Modifications

// This is a new main() function, for testing the RhythmGenerator IGA.

int main()

{

srand(time(NULL));

speech \*newspeak = new speech;

Gestures \*thisgestures = new Gestures;

thisgestures->ServoSetup();

//move arm to right position and ready for playing

Readfile \*newfilereader = new Readfile("script/MoveArm");

newfilereader->process(thisgestures,newspeak);

RhythmGenerator generator(thisgestures);

generator.RunIGA();

return 0;

}

### RhythmGenerator.h

// Brad Pitney

// ECE 578

// Fall 2013

// The RhythmGenerator class implements an interactive genetic algorithm, which is used to evolve

// movement sequences for playing a stringed instrument attached to the front of the Countess Quanta

// robot.

#ifndef RHYTHMGENERATOR\_H

#define RHYTHMGENERATOR\_H

#include <ctime>

#include <cstdlib>

#include <iostream>

#include <fstream>

#include <string>

#include "gestures.h"

using namespace std;

// IGA parameters

////////////////////////////////////////////////////////

// The size of the move sequence stored in each individual of the population.

const int numberOfMoves = 10;

// The number of individuals in the population. Note: this value must be an even number.

const int populationSize = 10;

// Total GA iterations.

const int totalGenerations = 5;

// Chance of applying crossover to each pair of parents.

const double recombinationProbability = 0.6;

// Chance of applying mutation to each gene.

const double mutationProbability = 0.01;

// Standard GA parameters

//const double recombinationProbability = 0.6;

//const double mutationProbability = 0.01;

// Mutation-based GA parameters

//const double recombinationProbability = 0;

//const double mutationProbability = 0.30;

// Random generation parameters

//const double recombinationProbability = 0;

//const double mutationProbability = 1.0;

////////////////////////////////////////////////////////

// Motion parameters

////////////////////////////////////////////////////////

// Range of allowed positions for servo::Wrist\_right.

const int minAllowedWristPosition = 600;

const int maxAllowedWristPosition = 1300;

// Starting location of servo::Wrist\_right, prior to evaluating a move sequence.

const int defaultWristPosition = 950;

// Positions of servo::Elbow\_Extension which result in raising and lowering the hand onto the intrument strings.

const int handRaisedPosition = 720;

const int handLoweredPosition = 884;

// The delay in microseconds between each move in a sequence, during evaluation.

// This value should be in increments of 100000, since this is the increment used when storing moves to a file.

const int usecMoveDelay = 100000;

////////////////////////////////////////////////////////

// Move sequences that are saved are stored in this file.

const string saveFileName = "script/IGA\_Rhythms.txt";

// Enables automatic storage of the rating values that the user selects when evaluating each individual.

const bool storeRatings = true;

const string ratingListFileName = "script/IGA\_RatingList.txt";

class RhythmGenerator

{

public:

RhythmGenerator(Gestures \*objGestures);

virtual ~RhythmGenerator();

void RunIGA();

private:

struct Gene

{

bool toggleHandState;

int wristPosition;

};

struct Chromosome

{

Gene gene[numberOfMoves];

int fitness;

int rouletteValue;

};

bool m\_handRaised;

Chromosome m\_population[populationSize];

Gestures \*m\_objGestures;

ofstream m\_saveFile;

int m\_totalMoveDelay;

void GenerateRandomMove(Gene &move);

void PerformMove(servo::servo\_name servoID, unsigned short position, bool writeToFile);

void PlaySong(int individual, bool writeToFile);

int EvaluateFitness(int individual, bool reviewMode);

};

#endif

### RhythmGenerator.cpp

// Brad Pitney

// ECE 578

// Fall 2013

// The RhythmGenerator class implements an interactive genetic algorithm, which is used to evolve

// movement sequences for playing a stringed instrument attached to the front of the Countess Quanta

// robot.

#include "RhythmGenerator.h"

RhythmGenerator::RhythmGenerator(Gestures \*objGestures)

{

m\_objGestures = objGestures;

m\_handRaised = true;

}

RhythmGenerator::~RhythmGenerator()

{

}

// This method randomly generates either a hand state toggle move or a wrist move, with a 50% chance of each.

// If a wrist move is selected, then the new wrist position is randomly generated within the allowed wrist range.

void RhythmGenerator::GenerateRandomMove(Gene &move)

{

bool toggleState = ((rand() % 2) == 1);

move.toggleHandState = toggleState;

if (!toggleState)

{

move.wristPosition = minAllowedWristPosition + (rand() % (maxAllowedWristPosition - minAllowedWristPosition + 1));

}

else

{

move.wristPosition = 0;

}

}

// This method makes the call to actually move a servo to the specified position. If writeToFile is set,

// it instead writes the move description to the save file.

void RhythmGenerator::PerformMove(servo::servo\_name servoID, unsigned short position, bool writeToFile)

{

if (writeToFile)

{

m\_saveFile << "%" << servoID << "\_" << position << "\_" << m\_totalMoveDelay << "%";

m\_totalMoveDelay += (usecMoveDelay / 100000);

}

else

{

m\_objGestures->SetServo(servoID, position\*4); // SetServo positions are 4x those displayed on the Maestro Control Center.

usleep(usecMoveDelay);

}

}

// This method executes the move sequence specified by the selected individual (i.e. chromosome).

void RhythmGenerator::PlaySong(int individual, bool writeToFile = false)

{

if (writeToFile)

{

cout << "Saving song to file '" << saveFileName.c\_str() << "'.\n";

m\_saveFile.open(saveFileName.c\_str(), std::ofstream::app);

m\_totalMoveDelay = 0;

}

else

{

cout << "Playing song number " << individual << ".\n";

usleep(usecMoveDelay);

}

// Move to initial position.

PerformMove(servo::Elbow\_Extension, handRaisedPosition, writeToFile);

PerformMove(servo::Wrist\_right, defaultWristPosition, writeToFile);

m\_handRaised = true;

// Step through the chromosome, performing each move.

for (int j = 0; j < numberOfMoves; j++)

{

if (m\_population[individual].gene[j].toggleHandState)

{

// Raise or lower hand.

if (m\_handRaised)

{

// Lower hand onto strings.

cout << "Lowering hand onto strings.\n";

PerformMove(servo::Elbow\_Extension, handLoweredPosition, writeToFile);

}

else

{

// Raise hand off of strings.

cout << "Raising hand off of strings.\n";

PerformMove(servo::Elbow\_Extension, handRaisedPosition, writeToFile);

}

m\_handRaised = !m\_handRaised;

}

else

{

// Move hand left or right.

if (m\_handRaised)

{

// If hand is above strings, check the next move to ensure that we're not wasting time moving the hand multiple times

// without touching the strings. Skip this move if the next move will just move the hand to another coordinate.

if ((j+1 >= numberOfMoves) || !m\_population[individual].gene[j+1].toggleHandState)

{

cout << "Skipping superfluous move to " << m\_population[individual].gene[j].wristPosition << "\n";

}

else

{

// Move hand.

cout << "Repositioning hand to " << m\_population[individual].gene[j].wristPosition << "\n";

PerformMove(servo::Wrist\_right, m\_population[individual].gene[j].wristPosition, writeToFile);

}

}

else

{

// If hand is on the strings, always move hand.

cout << "Strumming strings to " << m\_population[individual].gene[j].wristPosition << "\n";

PerformMove(servo::Wrist\_right, m\_population[individual].gene[j].wristPosition, writeToFile);

}

}

}

if (writeToFile)

{

m\_saveFile << "\n";

m\_saveFile.close();

}

}

// This method evaluates the fitness of an individual by first playing the song, and then prompting the user to

// rate the performance. Options to repeat the song or to save the song to a file are also presented. if reviewMode

// is set, then the method just plays the song and offers to repeat/save, without requesting a rating.

int RhythmGenerator::EvaluateFitness(int individual, bool reviewMode = false)

{

bool writeFile = false;

// If this individual's fitness was already evaluated during a previous generation, then just return this fitness.

if (m\_population[individual].fitness != 0 && !reviewMode)

{

return m\_population[individual].fitness;

}

while (true)

{

PlaySong(individual, writeFile);

writeFile = false;

bool repeatPrompt = false;

do

{

repeatPrompt = false;

if (reviewMode)

{

cout << "Type 'c' to continue to the next song.\n";

}

else

{

cout << "Please type 1-9 to rate the song (1 = very bad song, 9 = very good song):\n";

}

cout << "Type 'r' to repeat the song.\n";

cout << "Type 's' to save the song to the '" << saveFileName << "' file.\n";

char inputChar;

cin >> inputChar;

int inputValue = atoi(&inputChar);

if (inputChar == 'r' || inputChar == 'R')

{

// Repeat the song.

continue;

}

else if (inputChar == 's' || inputChar == 'S')

{

// Save the song to the file.

writeFile = true;

continue;

}

else if (inputValue >= 1 && inputValue <= 9)

{

if (reviewMode)

{

// Ignore ratings while in review mode.

repeatPrompt = true;

}

else

{

// Store the selected rating, if this option is enabled.

if (storeRatings)

{

ofstream ratingListFile;

ratingListFile.open(ratingListFileName.c\_str(), std::ofstream::app);

ratingListFile << inputValue << "\n";

ratingListFile.close();

}

// Return the selected rating as the fitness evaluation.

return inputValue;

}

}

else if (inputChar == 'c' || inputChar == 'C')

{

// 'continue' is only used in review mode.

if (reviewMode)

{

return 0;

}

else

{

repeatPrompt = true;

}

}

else

{

repeatPrompt = true;

}

} while(repeatPrompt);

}

}

// This method manages the interactive genetic algorithm process by creating a random population and evolving

// the population for the selected number of generations. During each generation, the fitness of each individual

// in the population is first determined by performing each move sequence and gathering fitness ratings from the

// user. These user-selected fitness values are then used in roulette wheel parent selection. Recombination and

// mutatation are applied to create the population for the next generation.

void RhythmGenerator::RunIGA()

{

// Generate random population.

for (int i = 0; i < populationSize; i++)

{

for (int j = 0; j < numberOfMoves; j++)

{

GenerateRandomMove(m\_population[i].gene[j]);

m\_population[i].fitness = 0;

}

}

// Evolve the population for the selected number of generations.

for (int numGen = 0; numGen < totalGenerations ; numGen++)

{

cout << "\n\n\*\*\* Beginning evaluation of Generation " << numGen << " \*\*\*\n\n";

if (storeRatings)

{

ofstream ratingListFile;

ratingListFile.open(ratingListFileName.c\_str(), std::ofstream::app);

ratingListFile << "\nRatings for Generation " << numGen << ":\n";

ratingListFile.close();

}

// Step through population, evaluating the fitness of each individual.

for (int i = 0; i < populationSize; i++)

{

m\_population[i].fitness = EvaluateFitness(i);

}

cout << "\n\*\*\* Creating new songs \*\*\*\n\n";

// Select parents for next generation.

// Create roulette wheel.

int totalFitness = 0;

for (int i = 0; i < populationSize; i++)

{

totalFitness += m\_population[i].fitness;

m\_population[i].rouletteValue = totalFitness;

}

// Create parent list.

int parentList[populationSize];

for (int i = 0; i < populationSize; i++)

{

int randValue = (rand() % (totalFitness + 1));

for (int j = 0; j < populationSize; j++)

{

if (randValue <= m\_population[j].rouletteValue)

{

parentList[i] = j;

break;

}

}

}

// Apply 2-point crossover.

Chromosome newPopulation[populationSize];

// Step through parent pairs.

for (int i = 0; i < populationSize; i += 2)

{

if (((double)rand() / RAND\_MAX) < recombinationProbability)

{

// If recombination chance succeeds, then apply crossover.

// Select 2 random crossover points in the chromosome.

int point1 = rand() % numberOfMoves;

int point2 = rand() % numberOfMoves;

// Reorder these points, if necessary, so that point1 comes first.

if (point2 < point1)

{

int temp = point1;

point1 = point2;

point2 = temp;

}

// Apply crossover to create offspring.

for (int j = 0; j < numberOfMoves; j++)

{

if (j < point1 || j > point2)

{

newPopulation[i].gene[j] = m\_population[parentList[i]].gene[j];

newPopulation[i+1].gene[j] = m\_population[parentList[i+1]].gene[j];

}

else

{

newPopulation[i+1].gene[j] = m\_population[parentList[i]].gene[j];

newPopulation[i].gene[j] = m\_population[parentList[i+1]].gene[j];

}

}

// Set fitness to 0 to mark these individuals for re-evaluation.

newPopulation[i].fitness = 0;

newPopulation[i+1].fitness = 0;

}

else

{

// If recombination chance fails, then copy over parents unchanged.

for (int j = 0; j < numberOfMoves; j++)

{

newPopulation[i].gene[j] = m\_population[parentList[i]].gene[j];

newPopulation[i+1].gene[j] = m\_population[parentList[i+1]].gene[j];

}

// Individuals are unchanged, so use the same fitness as before.

newPopulation[i].fitness = m\_population[parentList[i]].fitness;

newPopulation[i+1].fitness = m\_population[parentList[i+1]].fitness;

}

}

// Apply mutation.

for (int i = 0; i < populationSize; i++)

{

for (int j = 0; j < numberOfMoves; j++)

{

if (((double)rand() / RAND\_MAX) < mutationProbability)

{

GenerateRandomMove(newPopulation[i].gene[j]);

// Set fitness to 0 to mark this individual for re-evaluation.

newPopulation[i].fitness = 0;

}

}

}

// Copy newPopulation to population.

for (int i = 0; i < populationSize; i++)

{

for (int j = 0; j < numberOfMoves; j++)

{

m\_population[i].gene[j] = newPopulation[i].gene[j];

m\_population[i].fitness = newPopulation[i].fitness;

}

}

}

// Review final population.

cout << "\n\n\*\*\* Finished Evolving \*\*\*\n";

cout << "\*\*\* Now reviewing final population. Last chance to save songs! \*\*\*\n";

for (int i = 0; i < populationSize; i++)

{

EvaluateFitness(i, true);

}

}

## Source Code for Kinect Person Tracking

The code below was added to the existing ‘SkeletonBasics-WPF’ project, which is sample code included in ‘Kinect for Windows Developer Toolkit v1.8’. Code was added to ‘MainWindow.xaml.cs’ to create and update the new RobotDisplay form. The ‘RobotDisplay.xaml.cs’ contain the new RobotDisplay class, which implements the simulated robot and person tracking.

### MainWindow.xaml.cs Modifications

private void WindowLoaded(object sender, RoutedEventArgs e)

{

[Existing code omitted]

robotDisplayWindow = new RobotDisplay();

robotDisplayWindow.Show();

}

private void DrawBone(Skeleton skeleton, DrawingContext drawingContext, JointType jointType0, JointType jointType1)

{

[Existing code omitted]

// If the Head joint is updated, update the target position on the RobotDisplay form.

if (jointType0 == JointType.Head)

{

robotDisplayWindow.UpdateTargetPosition(joint0.Position);

}

}

### RobotDisplay.xaml.cs

// Brad Pitney

// ECE 578

// Fall 2013

// The RobotDisplay class implements a person tracking with a simulated Countess Quanta

// robot. Skeleton joint data from the Kinect hardware is used to update the location of

// the target person. This is used to calculate the new neck servo position that would

// cause the head to turn and face the target, if this were value were set on the physical

// robot. A top-down representation of the robot and target are displayed on the form.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Shapes;

using Microsoft.Kinect;

namespace Microsoft.Samples.Kinect.SkeletonBasics

{

/// <summary>

/// Interaction logic for RobotDisplay.xaml

/// </summary>

public partial class RobotDisplay : Window

{

public RobotDisplay()

{

InitializeComponent();

}

// Radius of the circle representing the Countess Quanta robot.

const double robotRadius = 50;

// Distance from the top of the form to the robot circle.

const double robotFromTop = 100;

// Radius of the circle representing the target person.

const double targetRadius = 20;

// Position of the robot neck servo (servo 10) at which the robot faces straight forward.

const double robotNeckServoCenterPostion = 1550;

// Number of servo position increments per radian of rotation for the robot's neck servo.

const double servoIncrementsPerRadian = 200;

Ellipse target;

Line targetLine;

// This method initializes the circles representing the robot and target, as well

// as the line indicating the direction the robot is facing.

private void WindowLoaded(object sender, RoutedEventArgs e)

{

const double defaultTargetDistance = 200;

// Circle representing the robot.

Ellipse robot = new Ellipse();

robot.Fill = Brushes.AliceBlue;

robot.StrokeThickness = 2;

robot.Stroke = Brushes.Black;

robot.Width = robotRadius \* 2;

robot.Height = robotRadius \* 2;

Canvas.SetTop(robot, robotFromTop);

Canvas.SetLeft(robot, RobotDisplayCanvas.ActualWidth / 2 - robotRadius);

RobotDisplayCanvas.Children.Add(robot);

// Circle representing the target person.

target = new Ellipse();

target.Fill = Brushes.LawnGreen;

target.StrokeThickness = 2;

target.Stroke = Brushes.Black;

target.Width = targetRadius \* 2;

target.Height = targetRadius \* 2;

Canvas.SetTop(target, robotFromTop + defaultTargetDistance);

Canvas.SetLeft(target, RobotDisplayCanvas.ActualWidth / 2 - targetRadius);

RobotDisplayCanvas.Children.Add(target);

// Line representing the facing direction of the robot.

targetLine = new Line();

targetLine.Stroke = System.Windows.Media.Brushes.Red;

targetLine.StrokeThickness = 2.5;

targetLine.X1 = RobotDisplayCanvas.ActualWidth / 2;

targetLine.Y1 = robotFromTop + robotRadius;

targetLine.X2 = RobotDisplayCanvas.ActualWidth / 2;

targetLine.Y2 = robotFromTop + defaultTargetDistance + targetRadius;

RobotDisplayCanvas.Children.Add(targetLine);

}

// The UpdateTargetPosition method is called from the DrawBone method in the sample

// 'Skeleton Basics' program. It updates the position of the target and calculates

// the required position of the neck servo to track the target on the actual robot.

public void UpdateTargetPosition(SkeletonPoint targetPosition)

{

// Update coordinates of the target joint.

targetXLabel.Content = "X = " + targetPosition.X.ToString("F2");

targetYLabel.Content = "Y = " + targetPosition.Y.ToString("F2");

targetZLabel.Content = "Z = " + targetPosition.Z.ToString("F2");

// Calculate the new target position, based on the Kinect data.

double displayScale = 100;

double targetCenterTop = robotFromTop + robotRadius + targetRadius + targetPosition.Z \* displayScale;

double targetCenterLeft = RobotDisplayCanvas.ActualWidth / 2 + targetPosition.X \* displayScale;

// Update the location of the target circle.

Canvas.SetTop(target, targetCenterTop - targetRadius);

Canvas.SetLeft(target, targetCenterLeft - targetRadius);

targetLine.X2 = targetCenterLeft;

targetLine.Y2 = targetCenterTop;

// Update the position of the Countess Quanta neck servo, to track the target.

double robotNeckServoPosition = robotNeckServoCenterPostion - Math.Atan(targetPosition.X / targetPosition.Z) \* servoIncrementsPerRadian;

neckServoLabel.Content = robotNeckServoPosition.ToString("F0");

}

}

}