Musical Agents

Thesis

of

Joseph Rukshan Fonseka

12275727

Abstract

A *Musical Agent* is an application that consists of a general-purpose music script handler that is flexible enough to support many different functions an artist might require for writing algorithmic compositions. A musical agent is part of a distributed application that communicates with other similar agents over a network to create an algorithmic musical performance.

Experimental musicians around the world can make use of musical agents to perform a musical composition, each agent contributing to the performance. The group's performance can be heard by any participating agent. Each agent can be considered as a member of a choir. Unlike a true choir, the agents can be distributed across a network of computers.

In the real world, air is the medium in which sound travels, the agents instead use network connections as their communication medium i.e. the network latency between agents is used to provide a measure of virtual distances over the Internet to recreate sound loss through a physical medium.

The agent application is written in Java and makes use of the multicast Internet protocol to communicate over the Internet. MIDI instruments provide the sound output which allows for the implementation of additional voices.

This report also researches different methods of creating algorithmic compositions. The website for this project is at http://www.csse.monash.edu.au/~jfonseka

Table of Contents

Abstract 2

Table of Contents 3

List of Figures 5

Acknowledgments 6

1. Introduction 7

1.1 Aim of the Project 7

1.2 Intended Audience 8

1.3 Definitions 8

2. Related Work 9

2.1 D.A.S.E 9

2.2 Chaosynth 10

2.3 CAMUS 3D 12

3. Algorithmic Music 14

3.1 Comparison to Traditional Music 14

3.2 Stochastic Processes 15

3.3 Markov Chains 17

3.4 Cellular Automata 21

3.5 Computers in Algorithmic Compositions 24

4. Experimental Music 26

4.1 *Paragraph 7* 26

4.2 Relevance to the Project 28

5. Agent Implementation 29

5.1 Programming Language 29

5.2 Internet Protocol 29

5.3 Agent's Voice 32

5.4 Scripting Language 34

5.5 Obeying Natural Laws 35

6. Agent Design 36

6.1 Design Overview 36

6.2 Remote Switch 38

6.3 Are You Still There? 39

6.4 MIDI Notes 39

6.5 Intensity of Sound 41

7. Results and Discussions ___ ***42***

7.1 Testing Environment ___ **42**

7.2 Testing the Musical Agent ___ **42**

7.3 *Paragraph 7* ___ **43**

7.4 *Instrument Rush* ___ **46**

7.5 Miscellaneous Remarks ___ **48**

8. Future Work ___ ***49***

8.1 Implement Extended MIDI Functionality ___ **49**

8.2 Simplify the Script Language Syntax ___ **50**

8.3 Auto-Download Music Scripts ___ **50**

8.4 Permanent Multicast IP Address ___ **51**

Conclusion ___ ***52***

Bibliography ___ ***54***

Appendix 1 - Useful Resources ___ ***55***

Appendix 2 - Using the Agent ___ ***56***

Appendix 3 – Instrument Rush Script ___ ***61***

Appendix 4 – Source Code ___ ***63***

Appendix 5 – JavaDocs for the Source Code ___ ***82***

List of Figures

Fig 1: Example use of *DASE*. 9
Fig 2: Many granules form a sound event. 10
Fig 3: The Chaosynth system, a grid of 196 nerve cells connected to 4 oscillators. 11
Fig 4: Calculation of the four-note chord from the *Game of Life* automata. 12
Fig 5: Traditional musical notation. 14
Fig 6: Sample probability table for a stochastic process (not to scale). 15
Fig 7: Sample cumulative distribution (not to scale). 16
Fig 8: Examples of some distributions that can be used to generate the probability table. 17
Fig 9: Example of a musical state machine and its Markov's transitional probability matrix. 18
Fig 10: Probabilities of being in one of the three states after two time units. 18
Fig 11: An alive cell and its neighbours in the cellular space. 22
Fig 12: Stationary objects in Life. 22
Fig 13: Oscillatory objects in Life. 23
Fig 14: A glider: an object that moves. 23
Fig 15: Generating notes using Life. 24
Fig 16: Paragraph 7 script. 28
Fig 17: IP Addresses. 30
Fig 18: Datagram routing using unicast.
Fig 19: Datagram routing using multicast. 30
Fig 20: Single point of failure in client/server system. 31
Fig 21: Playing a MIDI note. 33
Fig 22: Musical agents send packets to a black box, which distributes the data. 36
Fig 23: Pictorial view of the data structure. 37
Fig 24: Definition of round trip time. 37
Fig 25: Format of messages. 37
Fig 26: Class diagram of MIDI. 40
Fig 27: Output from MidiNotate of the test music script. 43
Fig 28: The first 10 seconds of *Paragraph 7*. 43
Fig 29: Six seconds of *Paragraph 7* with random instruments assigned. 46
Fig 30: First 7 seconds of *Instrument Rush*. 47

Acknowledgments

Thanks must go to Dr. Alan Dorin and Mr. Jon McCormack for their generous support throughout the project. I am ever so grateful for their suggestions and ideas that made my project a reality.

I would also like to thank Mr. Ian Kaminskyj for being my second reader. Kudos to my friends for their encouragement and continuous interest in my project and my sister for her patience while I made use of her computer during the testing phase.

No thanks must go to my only enemy, time. With whom I can live without.

1. Introduction

Composed by the British composer Cornelius Cardew in 1967, the experimental musical work *The Great Learning* consists of seven parts or paragraphs. The music is described using text, symbols and instructions, unlike traditional music that uses well-defined musical notation [8]. In *Paragraph 7* of the *Great Learning*, each performer is provided with the same script. When signalled by a leader, each performer must begin singing. Initially, all the singers may choose random notes to sing. However, at the end of each line a singer must pick a note his or her neighbour is currently singing. Although the music is created by these simple rules the outcome is very complex.

Algorithmic music is becoming more popular in the music industry, as composers are becoming computer programmers so as to make maximum use of the computer as a viable instrument to generate music that is too complex to create otherwise [10, pp. 153-9].

1.1 Aim of the Project

The aim of this project is to develop algorithmic music composition software that is capable of producing compositions such as *Paragraph 7*. The application is to be distributed among several computers within a network. It must communicate via an Internet protocol to enable musicians around the world to compose together. The system must have a general-purpose scripting language that is flexible enough for a musician to write different algorithmic compositions. MIDI instruments will be used to generate the sound output.

Due to the design strategy of the agent, a stand-alone agent (not distributed) can be used simply as a device to experiment with algorithmic compositions. The agent can be considered as a tape deck and the algorithmic composition (script) as the tape media. It is important to understand that the musical agent itself cannot generate music but needs a musical script to produce music. Thus, the quality of the music generated is determined by the algorithm used in the musical script.

1.2 Intended Audience

A musical agent would be useful to musicians, especially those who have an interest in experimental or algorithmic composition. The artist must have some programming knowledge to take advantage of the in-built scripting language to write interesting compositions.

1.3 Definitions

Throughout this report, the terms *agent*, *client* and *member* are used interchangeably to refer to the musical agent (the application). A composition refers to the set of instructions that describes a musical work. The term *distributed* refers to the sharing of a process by more than one processor.

2. Related Work

2.1 D.A.S.E

DASE [6] is a Distributed Audio SEquencer developed by K. Cheung in 2000. It allows electronic musicians all over the world to make music together (jam) over the Internet. The system is made up of two components namely the *peasant* and the *castle*. The peasant is a client application that enables musicians to plot beats and load audio samples to be played. The castle is a server and an audio engine that outputs the music the peasants are playing. Every musician uses a peasant to join a castle and contribute a music loop to the performance. Using the peasant, a musician can for example load a drum audio file and specify the audio sample to be played at beats one and four in a bar. The musician can run the castle application if they intend to hear the complete performance. A castle can also be used to join another castle if the musician wishes to simply listen to the performance.

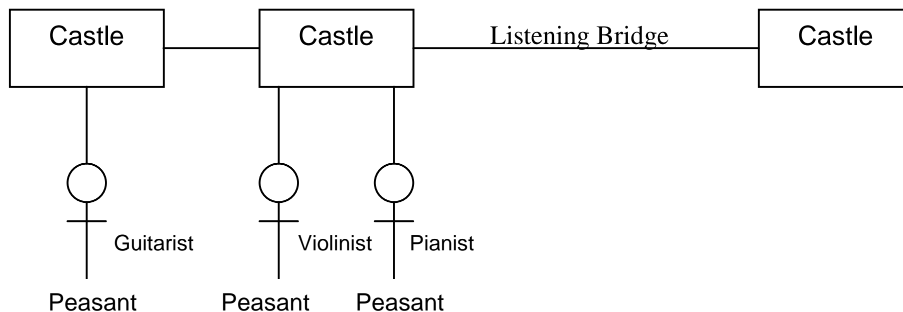

Fig 1: Example use of *DASE*.

The peasants and castles communicate with each other using the TCP Internet protocol. Once a peasant uploads an audio sample to a castle, it will be played along with the other samples on the specified beat. It must be noted that once the sample and beats are uploaded, there is no further communication between the peasant and castle except for chat messages. If a post (audio data and beat information) to the castle occurs while in the middle of the music loop, that audio sample will not be

introduced to the performance until the beginning of the next loop. In this system, all beats are synchronised in a castle. However, this synchronisation does not extend to other peasants or castles.

2.2 Chaosynth

Chaosynth [15] [16] is a cellular automata based granular synthesizer. It was created at the *Edinburgh Parallel Computing Center* between 1993 and 1994. The original system was developed from hardware and since has been ported to *Linux* as well as to the Windows operating system. Chaosynth has been used to create the sounds for a number of prize-winning electroacoustic pieces. This system is useful for film and computer game composers to produce science fiction sound effects. It is also possible to generate sounds such as flowing water and birdcalls. The sound events are generated by short sound bursts of 35 milliseconds long known as *granules*. The creation of a sound event by many granules is analogous to the creation of animation with a sequence of still images. The granules are formed in quick successions so that it seems like one long sound to the human ear. In the figure below, each ellipse is a granule of sound. Like animation, these non-continuos bursts of sounds can be played very quickly to create a sound event that is seemingly continuous.

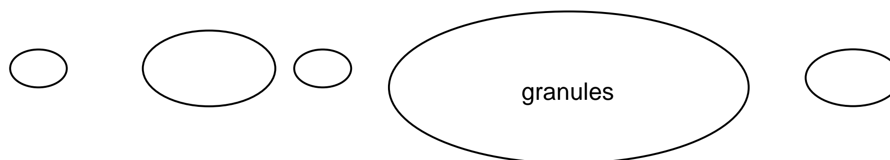

Fig 2: Many granules form a sound event.

Such systems have existed since the 1950s, however early systems used very complex mathematical formulas to control the production of the granules. Chaosynth uses a different approach, it makes use of ChaOS, a CHemical OScillator cellular automaton. Chaosynth consists of an array of electronic nerve cells. Each cell has three different states that are described by the potential differences. The states of these cells are dependent on the state of neighbouring nerve cells. The cells represent the frequency of sound, which is then fed into a number of oscillators. However, the amplitude of the sound is user-specified. The frequency at a given time is the mean of the frequencies within the cells. Each oscillator is responsible for a single component of a granule, which is then fed through a waveform adder to produce the sound granule. This method of creating sound resembles the functioning of some acoustic instruments such as the guitar [15].

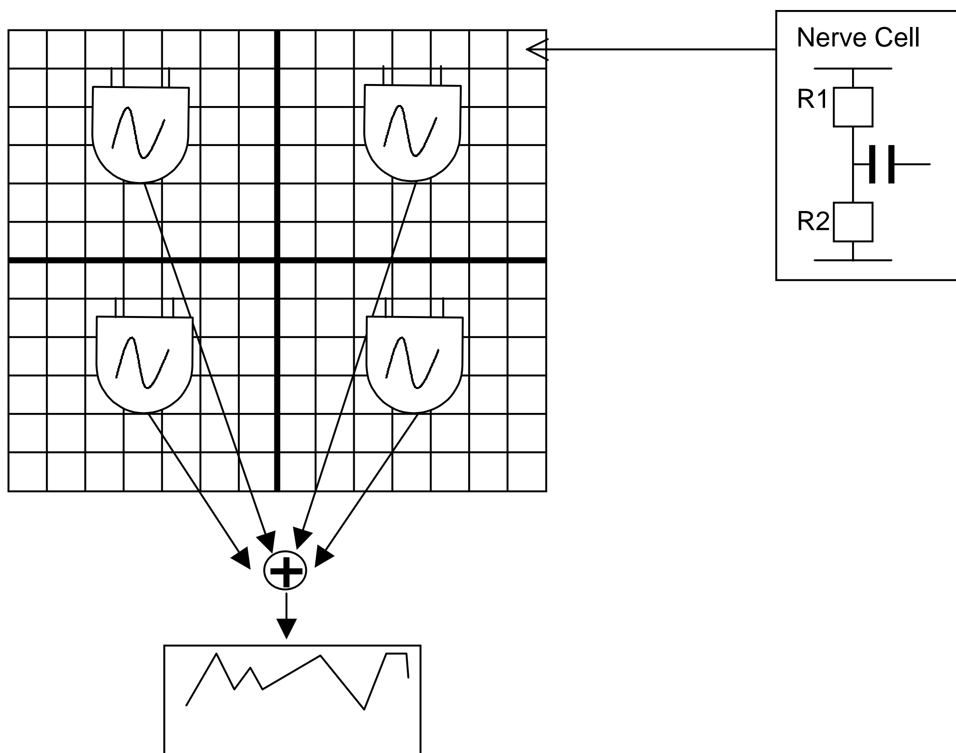

Fig 3: The Chaosynth system, a grid of 196 nerve cells connected to 4 oscillators.

An award-winning software version of Chaosynth is now commercialised by nyrsound.

2.3 CAMUS 3D

CAMUS 3D [13, pp. 24-9] is a Cellular Automaton based MUSic generator, an improved version of the original CAMUS [17] developed by Miranda. CAMUS 3D is the result of a research project funded by *Carnegie Trust for the Universities* in Scotland. Miranda used his system to compose a piece for the chamber orchestra that was premiered in March 1995, performed by *The Chamber Group of Scotland.* This system creates music using a combination of *Game of Life* [12, pp. 49-58] and *Demon Cyclic Space* [13, pp. 24] automatons that are stacked upon each other to create a 3D system. Initially, the *Game of Life* automaton is run with a specified starting cell configuration while the *Demon Cyclic Space* is started with a random state. At each time interval, the cells determine a four-note chord to be played at a MIDI channel specified by the state of the corresponding cell of the *Demon Cyclic Space* automaton. A first-order Markov chain is used within the system to calculate the note duration. The composition is performed in real time and there is an option to save the performance in a MIDI file.

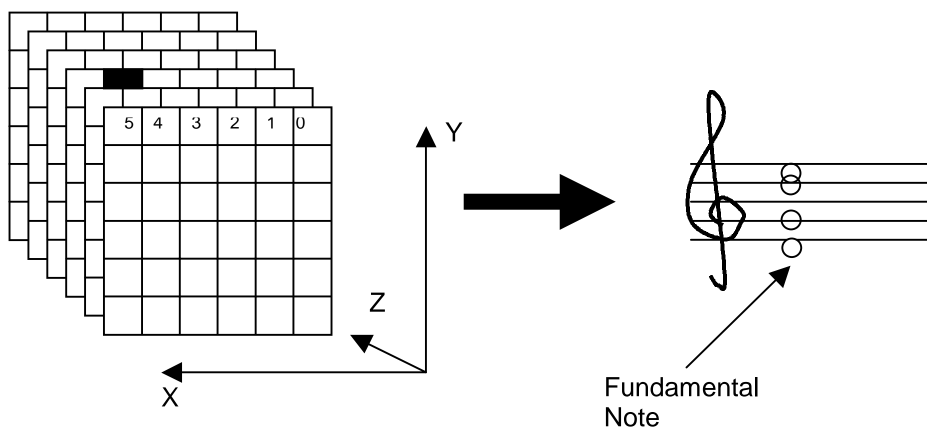

Fig 4: Calculation of the four-note chord from the *Game of Life* automata.

The four-note chord is calculated using the position of the cell being analysed. In the above example, a cell (coloured black) is at (4,5,2). A semitone interval from the fundamental pitch to the second lowest pitch of the chord is determined by the X cell position. The Y cell position defines a semitone interval from second lowest pitch to the second highest pitch in the chord. The semitone interval from the second highest to the highest pitch is determined by the Z position. The fundamental note of the chord can be determined by a user-specified list or by a stochastic selection routine [13, pp. 25-6].

There are several other systems similar to the Musical Agent, it would be impossible to include them all in this report. A small list of useful resources is provided in the appendices that may be of interest to the reader.

There is no known work of a general-purpose distributed algorithmic composer as described in the introduction and so there is nothing to compare the performance of this system with.

3. Algorithmic Music

Algorithm: "A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer"

The New Oxford Dictionary of English, 1998.

3.1 Comparison to Traditional Music

Algorithmic music refers to compositions where the rules or instructions are laid down and it is up to the performer to interpret and execute these rules. The rules can be compared to a mathematical function and the sound output the result of the manipulation. In fact, it has been proved that virtually any mathematical formula can be turned into music [19, pp. 825]. Algorithmic music is a result of considering music as being procedural and very much like a programming language. This is different from traditional music where a composer would describe the music in detail such that another musician can duplicate the performance. There are relatively few uncertainties and the performer is limited in the decisions they can make that will alter the outcome of the performance.

Fig 5: Traditional musical notation.

In traditional music, the tempo, duration of the beats, notes and artistic impression are all specified by the composer. This music is thus considered "static", in the sense that it does not evolve during the performance.

Most of the 'algorithms' used in algorithmic compositions are borrowed from the world of science. The most popular algorithms consist of Stochastic processes, Markov chains and Cellular Automata. There are many others techniques of generating music including Fractals, Chaos generators and Neural networks. However, this paper will concentrate on the first three methods of generating algorithmic music.

3.2 Stochastic Processes

If the behaviour of an algorithm is determinable by predicting its output given the current state and input, it is known as *deterministic*. A process whose behaviour is controlled by some random or probabilistic procedure, and thus whose outcome is unpredictable, is known as a *stochastic process*. Most algorithmic compositions tend to be based on some form of a stochastic process as this is the easiest algorithm to generate music [3]. A simple stochastic process is one where a probability table is used to determine the pitch of the note. In this system, the probability of a note being played is either specified by the musician or generated from a mathematical function and is normalised such that the values add up to 1.

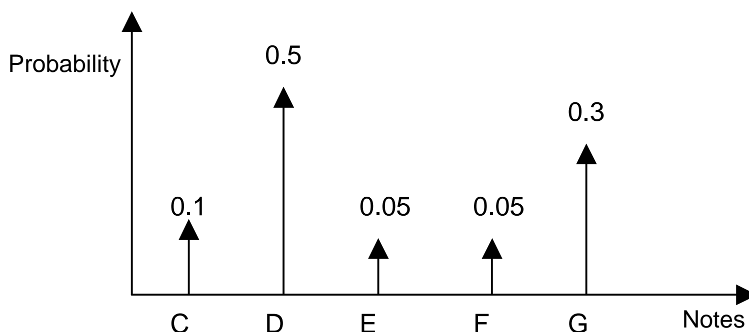

Fig 6: Sample probability table for a stochastic process (not to scale).

A random number generator is used to pick a number between 0 and 1. The number is then compared with the probabilities in the probability table. If the number is less than the number in the table, then that pitch will be selected for the next note. Using

the above table, if 0.4 was generated, the note D will be played. The algorithm for this system can be described as:

```
for(i=C; i<=G; i++)
{
     if(rnd(seed) < ptable[i])
     {
          play(i);
     }
}
```

This system however does suffer from a minor problem. If a number greater than 0.5 is generated, no note will be selected. Also the note F will never be played if the number 0.04 is generated because E is chosen instead. Thus the cumulative distribution is often used instead.

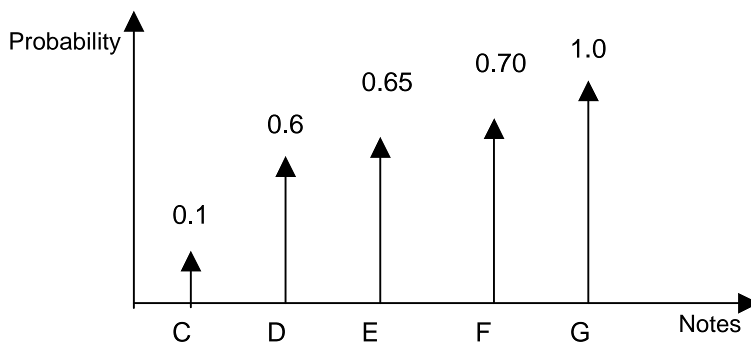

Fig 7: Sample cumulative distribution (not to scale).

If using the cumulative distribution, a generated number of 0.5 will play the note C. If a note has the probability of 0, it is considered that this note will never by played and thus will not appear in the cumulative distribution. In computing, there is no true random number generator. Most random generators are deterministic as they are generated from a mathematical function that is computed many times. These random generators are thus called *pseudo-random* generators. It is essential that a random seed be provided to the generator such as the current time to make this system stochastic.

The probability table as explained previously can be generated from a mathematical formula or specified by the musician. Below are examples of some mathematical distributions that can be used to generate the probability table.

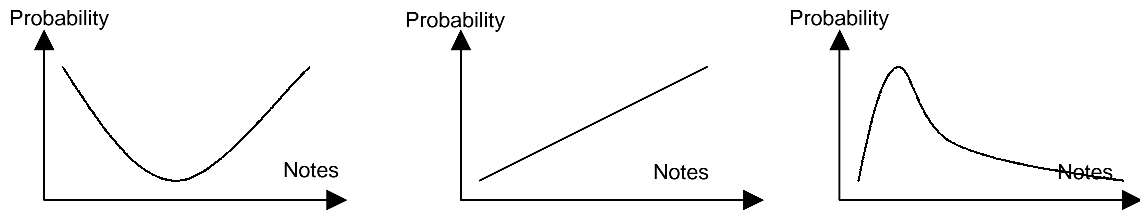

Fig 8: Examples of some distributions that can be used to generate the probability table.

In this system, the musician has no control over the output of the stochastic system except for the seed and the probability of the notes to be played. The musical output is quite random and of low quality [3]. That is, it does not sound much like traditional music where a sequence of notes (usually the chorus) is repeated several times within the composition. The above probability table system is explained in detail in the *Computer Music Tutorial* [19, pp. 868-78] and *Making Music with Algorithms* [13, pp.20-2].

3.3 Markov Chains

The Russian mathematician A. Markov described Markov chains in 1907. There are a time-dependent stochastic process whose future behaviour is determined by the current state and not by the previous history of the system. Thus, Markov chains are memory-less and are 1st order Markov processes. Their behaviour is *discrete*, meaning that the system will always be in one of the states in the finite state machine.

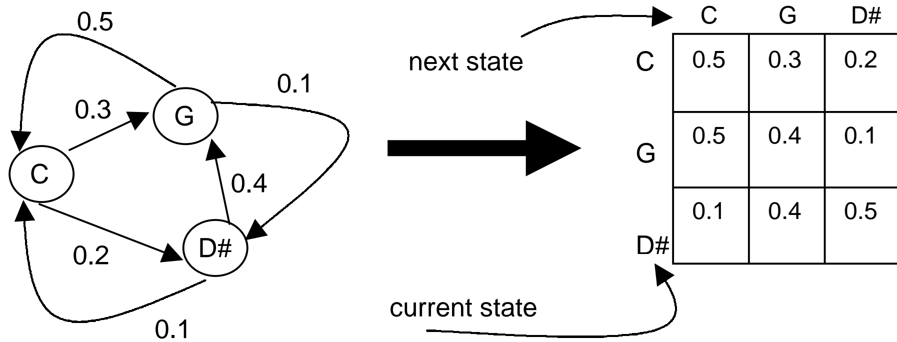

Fig 9: Example of a musical state machine and its Markov's transitional probability matrix.

Above is a state diagram that describes the probability of a note being played at the next time step after a particular note is played at the current time step. If C was the last note to be played, the probability that note G will be played next is 0.3 while the chance of note D# being played is 0.2. The chances that note C is again played are 0.5.

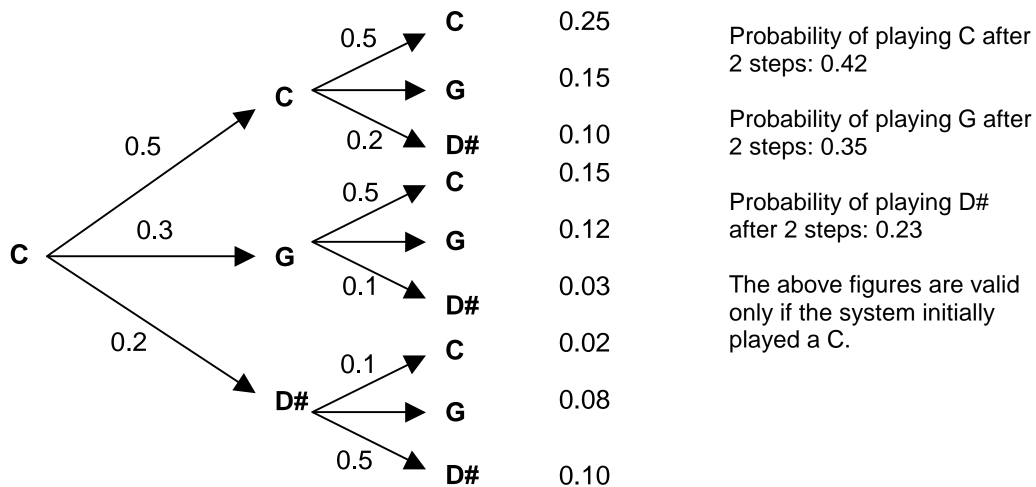

Fig 10: Probabilities of being in one of the three states after two time units.

If the current note being played is C, the probability that it will be played again in the next two steps is 0.42 (0.25 + 0.15 + 0.02) as shown in the probability tree. The more states that exist, the more computations and memory that are required to calculate the probability of a state occurring. This is also true when calculating the probabilities for the nth step, where n is very large using a probability tree.

Markov chains provide a simple method that can be used to determine the probabilities of being in a state at a particular time. The probabilities in the state diagram are represented in a transition probability matrix (P) as shown in fig. 9. The rows represent the probabilities of switching to one of the states and so always add up to 1.

$$\begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{bmatrix} \longrightarrow \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.5 & 0.4 & 0.1 \\ 0.1 & 0.4 & 0.5 \end{bmatrix}$$

P_{11} represents the probability that the state 1 will be chosen after state 1. The initial conditions P(0) are represented by a 1xN matrix.

$$P(0) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

C G D#

The probability of C is 1 because we are making the assumption in this example that the system starts in state C. The probability information after n steps can be calculated using the state vectors as follows.

$$P(1) = P(0)\,P$$
$$P(2) = P(1)\,P = P(0)\,P^2$$
$$P(n) = P(0)\,P^n$$

Thus the same information calculated using the probability tree could be calculated using Markov chain as shown below.

$$P(1) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.5 & 0.4 & 0.1 \\ 0.1 & 0.4 & 0.5 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.3 & 0.2 \end{bmatrix}$$

$$P(2) = \begin{bmatrix} 0.5 & 0.3 & 0.2 \end{bmatrix} \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.5 & 0.4 & 0.1 \\ 0.1 & 0.4 & 0.5 \end{bmatrix} = \begin{bmatrix} 0.42 & 0.35 & 0.23 \end{bmatrix}$$

Thus Markov chain predicts that C will have a higher chance of being played in the second step with a probability of 0.42 which agrees with the results calculated using the probability tree (see fig.10). This simple example demonstrates that Markov chains does accurately calculate the probability of state transitions and also provides a simple algorithm that can be programmed.

A musician may specify the initial state and probabilities of state transitions, and using Markov chains, the probabilities of the next state transitions will then be calculated. These probabilities could be used to generate a cumulative distribution and using a random number generator, select the next note to be played as described in the stochastic process. The state chosen is then fed back as the initial state into the Markov process and the procedure repeated over again. This is a simple method of generating music.

The application *mother*, is an example of a score generator that uses Markov chain [9]. Instead of using notes to represent the states as in the above example, a musician can represent the state with a chord or short tune to create a more interesting composition. Markov chains can also be used in a hierarchy system where one Markov chain selects the tempo of the composition, another the artistic style while the atomic Markov chain selects which notes are to be played [19, pp.880]. Roads states that the higher the order of the Markov process (that is the more previous states that are taken into account in deciding the next state) the better the music composition will be [19, pp.878]. This is because the generated music will be built of juxtaposed phrases, a part of one section of the composition is

crudely repeated in another section. This creates repeated sections within the composition similar to that found in traditional music.

3.4 Cellular Automata

Like all the other methods described above, using cellular automata in music compositions is not new. The cellular automata is an invention of Von Neumann in his quest to create a universal self-productive automaton [2, pp.26]. This kind of automaton system is based upon a two dimensional array of cells that are aligned next to each other such that each cell has neighbouring cells.

The cellular automata developed by Von Neumann had 29 states that each cell could be in and was relatively complex. John Conway in 1968 simplified the 29 states to simply 2 states. Each cell could be either dead or alive. He called this automaton *The Game of Life*, as it resembles changing societies of living organisms, which can grow, move and occasionally die out [14, pp. 45]. The rules for this game are very simple. If a cell is currently alive and has two or three of its neighbouring cells also alive, it will be alive in the next iteration. If a cell is currently alive and there are more than three neighbouring cells alive, it will die because of overcrowding. A cell will die of exposure in the next iteration if there are less than two neighbouring cells. A cell will become alive if it is not currently alive in the next iteration when exactly three of its neighbouring cells are alive [14, pp. 45].

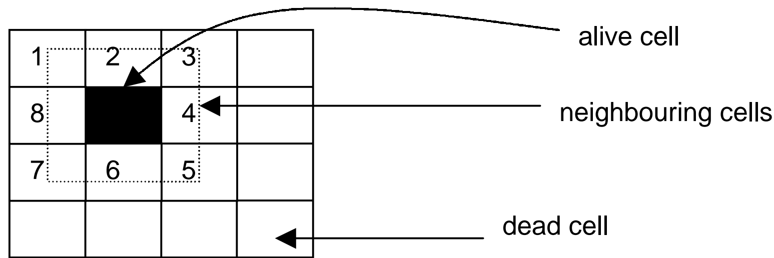

Fig 11: An alive cell and its neighbours in the cellular space.

These simple local rules that govern life on the cellular space or checkerboard create some complex and interesting global behaviour that might not be anticipated. In fact, John Conway proclaimed that on grid of a million cells, this automaton could well yield a one-celled animal [12, pp. 58]. There are some interesting patterns that have been discovered by individuals and Conway's group. Patterns discovered include stationary objects, oscillatory objects that repeat within a cycle, objects that move, objects that reproduce to create other objects and *eater* objects that consume debris on their path. These patterns are well documented in *Life with your Computer* including a count of all known objects that fall within the above categories [14, pp.45-67]. This paper will present a small subset of objects for the interest of the reader.

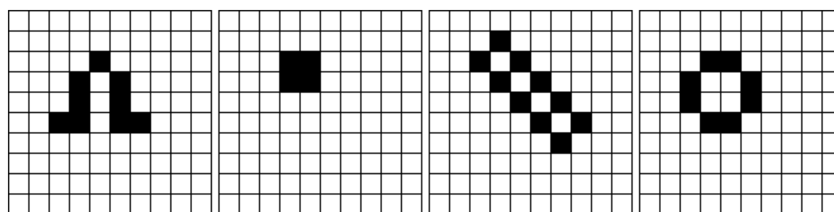

Fig 12: Stationary objects in Life.

Above are a few of the many stationary objects that exist in the *Game of Life*. These objects are called stationary because the living cells remain for all future time.

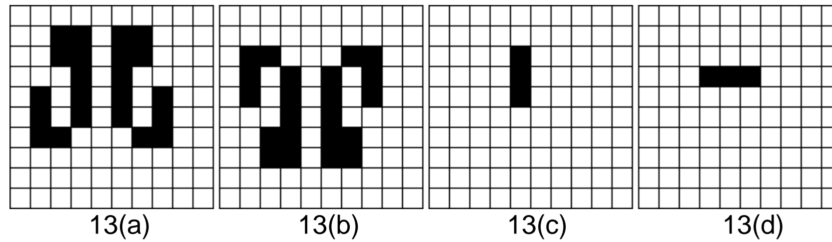

Fig 13: Oscillatory objects in Life.

Oscillatory objects are patterns that repeat themselves after a number of generations. The tumbler (fig.13a) repeats itself every 14 generations and after every seven generations it is upside down (fig.13b). The most common and simplest of the oscillatory objects has to be the blinker. This pattern has a period of 2 cycles. It will oscillate from a horizontal bar to a vertical bar and back (fig.13c & 13d).

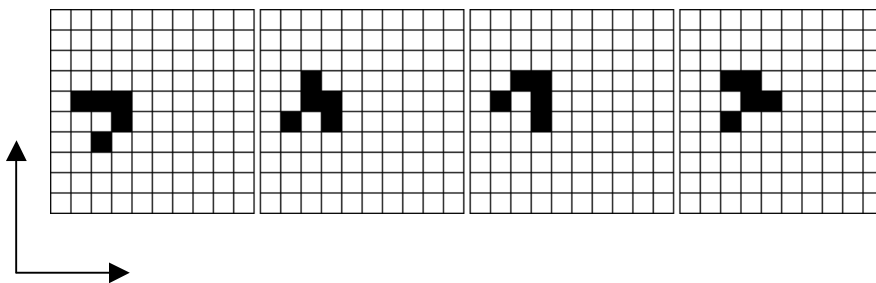

Fig 14: A glider: an object that moves.

The glider is the simplest object that moves across the grid. As shown in fig. 14, this object has a period of 4 cycles and in just 2 cycles, the object is repeated but oriented in a different position. The glider moves one cell diagonally every 4 cycles. In the above case the glider started at cell (2,6) and so in 4 cycles will have moved to (3,7).

The cells in the automata can be used to represent musical notes. The states ‘dead’ and ‘alive’ could be reinterpreted to ‘not playing a note’ and ‘playing a note’. The rules of Life will then generate the music. The outcome of the music will depend on the initial cell configuration of the cellular automata. As described above, certain

configurations lead to interesting complex behaviours. For instance a tumbler within the configuration would generate a cyclic rhythm within the composition that is pleasant to the human ear. In fig.15, a blinker is used to generate music. The cells are predefined to represent musical notes and the cellular automaton is left to generate the music.

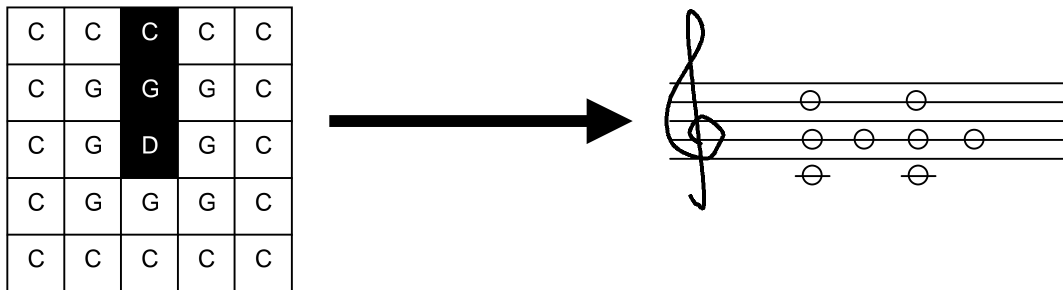

Fig 15: Generating notes using Life.

The *Game of Life* cellular automata described above is deterministic. A deterministic automaton is one that has a single initial state and a unique next state for each cell. Non-determinist cellular automaton on the other hand, may have more than one initial state and its transition function yields a set of possible next states for that cell [4, pp. 8]. The transition function is some probabilistic function.

3.5 Computers in Algorithmic Compositions

The computer is an important instrument when generating algorithmic music. The computer can make quicker decisions than real performers and can handle much more detail. Since algorithmic music is procedural and can be written or described in a programming language, the computer is a good choice for generating such music. However in most cases, the computer will be used as a tool to handle the tiny details of the music and it is left to the composer to worry about the higher level of abstraction [19, pp. 908]. Most of the above algorithmic techniques may well compute similar music but in a different manner. A composer decides which

technique to use in his or her composition depending on how much control the method provides to the musician. For example if a composer wishes to have total control of the pitch of the notes, a deterministic algorithm (perhaps cellular automata) would be a better choice than Markov Chains. Parts of generated music may well be combined with other algorithmic music or even traditional music to create a larger interesting piece. There is no limit to the number of ways a composer can compose algorithmic music, it is left to their creativity.

4. Experimental Music

Experimental Music: "A work of art or an artistic technique involving a radically new and innovative style"

The New Oxford Dictionary of English, 1998.

The composition *The Illiac Suite for String Quartet* by Hiller and Isaacson in 1956 is the first computer composed composition. Hiller followed this with other compositions including *The Computer Cantana* (1964) and *HPSCHD* (1968), a collaboration with John Cage [19, pp.831]. This was the beginning of the concept of *experimental music*. This kind of music evolves during the performance and the performers decide how the music should flow rather than simply phonograph the composition. In fact John Cage described a performance of experimental music as being unique. That is, it can never be repeated exactly, as the decisions made in the performance are unique. Experimental music is quite different from traditional music, it does not need to have rhythm or sound pleasant. "Pleasantness" of course, is in the ear of the beholder. John Cage sums up experimental music with this statement, "Wherever we are, what we hear is mostly noise. When we ignore it, it disturbs us. When we listen to it, we find it interesting" [11, pp. 128].

4.1 *Paragraph 7*

This composition is a popular piece of work from the British composer Cardew. This piece of work is one of the seven parts of the *Great Learning* that was composed in 1967. All performers all provided with the script shown in fig.16. When signalled by the leader, all performers select a random note to begin singing the first line. Each performer tries to keep within that note until the end of the line. The line is sung until the performer can no longer hold his or her breath. Thus the

lines of the script become interleaved as some performers will be able to hold their breath for longer periods than other performers. When a performer has finished singing their line, before beginning the next line, they will listen to their members and pick a note they can hear to sing on the next line. *Paragraph 7* is a process that allows the performers to move through the given material at their own speed [18, pp. 6].

→	sing 8	IF
	sing 5	THE ROOT
	sing 13 (f3)	BE IN CONFUSION
	sing 6	NOTHING
	sing 5 (f1)	WILL
	sing 8	BE
	sing 8	WELL
	sing 7	GOVERNED
	hum 7	
→	sing 8	THE SOLID
	sing 8	CANNOT BE
	sing 9 (f2)	SWEPT AWAY
	sing 8	AS
	sing 17 (f1)	TRIVIAL
	sing 6	AND
	sing 8	NOR
	sing 8	CAN
	sing 17 (f1)	TRASH
	sing 8	BE ESTABLISHED AS
	sing 9 (f2)	SOLID
	sing 5 (f1)	IT JUST
	sing 4	DOES NOT
	sing 6 (f1)	HAPPEN
	hum 3 (f2)	
	speak 1	MISTAKE NOT CLIFF FOR MORASS AND TREACHEROUS BRAMBLE

Notation:
"=>" The leader gives a signal and all enter concertedly at the same moment. The second of these signals are optional; those wishing to observe it should gather to the leader and choose a new note and enter just as at the beginning.

"Sing 9(f2) SWEPT AWAY" means sing the words "swept away" on a length-of-breath note (syllabus freely disposed) nine times; the same note each time; of the nine notes, two (any two) should be loud, the rest soft. After each note take in breath and sing again.

"Hum 7" means hum a length-of-a-breath note seven times; the same note each time; all soft.

"Speak 1" means speak the given word in steady tempo all together in a low voice, once (follow the leader).

Procedure:
Each chorus member chooses his own note (silently) for the first line (IF eight times). All enter together on the leader's signal. For each sub sequent line choose a note that you can hear being

sung by a colleague. It may be necessary to move to within an earshot of certain notes. The note once chosen, must be carefully retained. Time may be taken over the choice. If there is no note, or only the note you have just been singing, or only 2 note or notes that you are unable to sing, choose your note for the next line freely. Do not sing the same note on two consecutive lines. Each singer progresses through the text at his own speed. Remain stationary for the duration of a line; move around only between lines. All must have completed "hum 3(f2)" before the signal for the last line is given. At the leader's discretion this last line may be omitted.

C. Cardew, “Scratch Music”, MIT Press, Cambridge, 1969.

Fig 16: Paragraph 7 script.

Initially, the performance begins with a complex chord as all performers have selected random notes. The number of notes over the course of the performance gradually converges to one or two notes as each performer begins selecting neighbouring notes [18, pp.125].

The concept of *Paragraph 7* is very similar to that of cellular automata. Each member in the performance, like a cell in a cellular space, obeys a local rule. In this case each member listens to his or her neighbours at the end of each line and picks a note he or she can hear. This simple local rule, also like in the cellular automata, generates complex global behaviour. The members follow the same rules, except each member does so slightly differently. Nymann describes this as “unity becoming multiplicity” [18, pp.6].

4.2 Relevance to the Project

The development of the musical agent is inspired by the unity becoming multiplicity concept of *Paragraph 7*. A musical agent will replace each choir member. However, the musical agent will be general-purpose so that any other compositions a composer can dream up can be performed.

5. Agent Implementation

Some may find this chapter appearing before the design unusual. The reason for this is some of the implementation decisions made, lead to the design. If another choice were made in the implementation, the design would have also been affected.

5.1 Programming Language

It was decided to implement the musical agent as a Java application for several reasons. Firstly, Java is a cross-platform language and there is no restriction on the operating system or hardware an agent can run on. This is true as long as the operating system can support the *Java Virtual Machine* (JVM). Java is available freely and comes with a large library of useful functions. Included in this library, are networking functions and with the recent release of *Java 2 SDK V1.3.0*, MIDI support. This allows the programmer to concentrate on developing the agent instead of writing code to duplicate these functions. Though not a big issue, Java applications tend to be smaller than the source code used to generate them. This is a plus for musicians when downloading the agent from the Internet.

5.2 Internet Protocol

After implementing partial code in TCP/IP and UDP, it was decided that multicast IP was the best choice for a musical agent. Multicast is an extension to the standard IP network-level protocol [7]. It is specifically designed for broadcasting video and audio over the Internet, where unicast and broadcast techniques are not appropriate for this kind of data. This protocol is beneficial for applications such as the Musical Agent where the same datagram is to be sent out to many destinations. All agents

must join a multicast group with a class D address and are then eligible to receive messages sent to this group. A class D address is an IP address within the range 224.0.0.0 and 239.255.255.255 as shown in fig.17. Any IP address within this range is assumed to be a multicast IP address by routers.

Bit 0

						Address Range
0					Class A address	0.0.0.0 – 127.255.255.255
1	0				Class B address	128.0.0.0 – 191.255.255.255
1	1	0			Class C address	192.0.0.0 – 223.255.255.255
1	1	1	0		Class D address	224.0.0.0 – 239.255.255.255
1	1	1	1	0	Reserved	240.0.0.0 – 247.255.255.255

Fig 17: IP Addresses [7].

This protocol is very efficient compared to UDP where the source must send individual copies out to all recipients. Thus, multicast improves network performance and conserves bandwidth. Because only one copy of the datagram is sent out by the source, it is left to the routers within the network to duplicate the datagram wherever necessary. Below is a diagram illustrating this point where agent ‘A’ wishes to send a datagram out to all other agents.

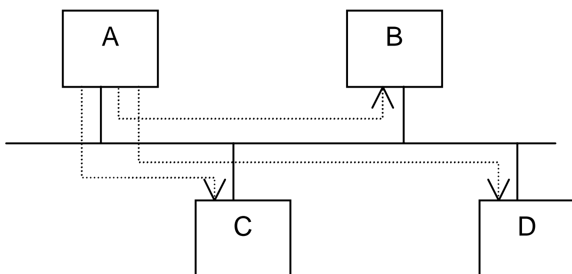

Fig 18: Datagram routing using unicast.

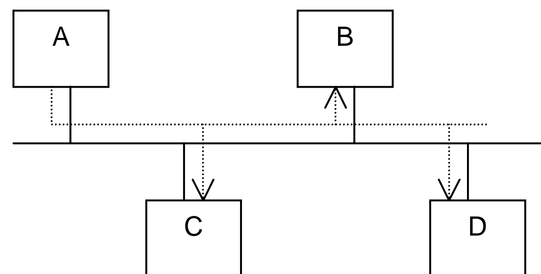

Fig 19: Datagram routing using multicast.

By making use of multicast, there is no server/client relationship. Instead, any agent may act as a server for a short period of time while sending data. This eliminates the single point of failure that is present if the agent instead used unicast datagrams,

where the bottleneck lies at the server. This is especially true when the number of clients connected to the server is large (fig. 20).

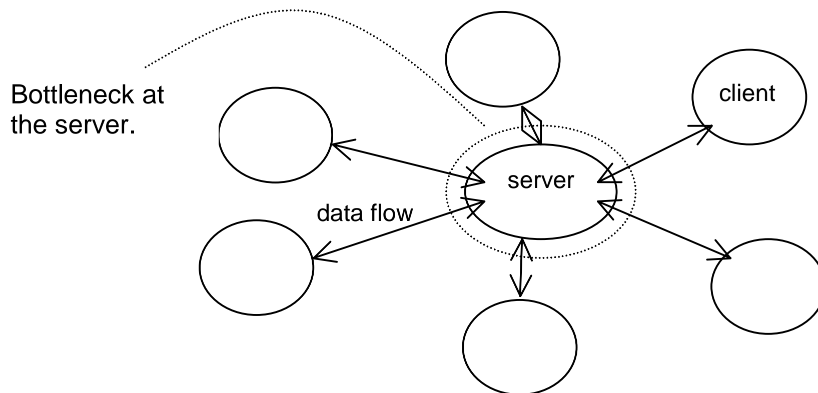

Fig 20: Single point of failure in client/server system.

Using multicast does have some minor setbacks. In the IPv4 Internet protocol standard, it was not mandatory for routers to support multicast. Multicast routing was introduced in IPv6 and it is now a requirement for routers to support this. However, there are still some routers in use on the Internet that do not route multicast datagrams and hence this creates *multicast islands*. Multicast island is a network of computers that can route multicast datagrams. Within the island, multicast datagrams are routed properly, but they cannot be transmitted or received from island to island, unless the network is connected to MBONE. Multicast backBONE is a hack that tunnels multicast packets across islands by hiding them as unicast packets so that unicast routers can handle the information.

Multicast transmissions using datagrams is not a reliable service. There isn't the same guarantee as TCP that a packet will arrive at its destination. This is not problematic if the agents are viewed as humans. Humans tend not to hear all sounds at a particular moment and so this loss of data packets is seen as a benefit rather than a problem in our musical application. There is also the tendency for packets to

appear at the destination out of sequence and so datagram packets will have to be labelled sequentially to allow the receiving agent to distinguish between old and new messages.

There is a noticeable delay of about 30 - 900ms and in the worst case about 2 seconds after a datagram is received at the destination after leaving the source. If the agents are all connected within a local area network, this latency is approximately 10ms or less, which is negligible for the purpose of this project. However, this delay is unavoidable and cannot be controlled especially in an uncontrollable environment as the Internet. There exists some protocols such as the RTP (RealTime Protocol) that tries to minimise this delay in audio and video broadcasts by measuring the quality of the service/connection and then dropping or increasing the frame rate as required. This system will not work with the musical agents as the data one agent receives is being distributed by many agents asynchronously.

The reason why Multicast was preferred over TCP is because it is not reliable and is therefore considerably faster than TCP. It does not need open virtual connections across the Internet. Lastly, the system does not suffer from the single point of failure that is present in both TCP and UDP systems.

5.3 Agent's Voice

The sound output is implemented using MIDI. Musical Instrument Digital Interface is a protocol designed to control a music device in realtime. In MIDI, the notes to be played are sent to an instrument using "note/on" and "note/off" messages. Once a note/on message is sent to the MIDI device, it will play the note specified (this is

equivalent to a key hold on a keyboard) until a note/off message is sent. The velocity is the intensity of the note (equivalent to how hard the key is pressed on a keyboard). To play many notes simultaneously, many note/ons are sent to the MIDI device one after another.

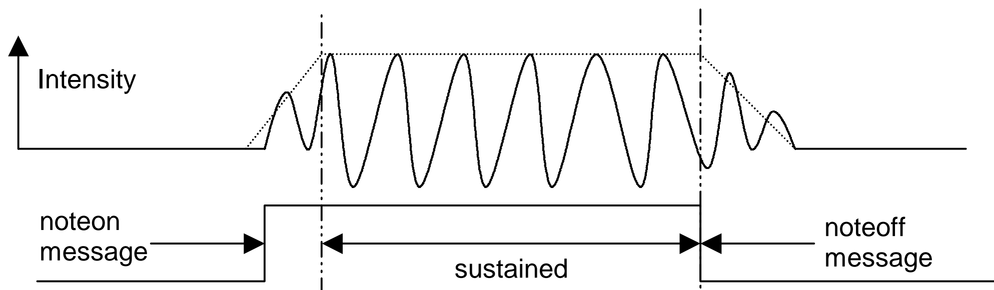

Fig 21: Playing a MIDI note.

The MIDI protocol supports 16 channels that can be assigned to an instrument. “A channel is like an electronic address label that identifies a packet of digital information, specifying its ultimate destination” [19, pp.983].

General MIDI is a standard formed by the MIDI Manufacturers Association (MMA) in an attempt to standardise the protocol to enable devices from different manufacturers to communicate with each other. General MIDI specifies a standard set of 128 instruments that can be used with a guarantee that another MIDI device will also support these same instruments. It is also possible to load a sound bank (sampled instrument) onto a MIDI device including voices. This is the real reason for using MIDI in the musical agent. By using MIDI for this project, there is much scope for further improving the sound output by introducing sampled human voices into the performance. This can be achieved without modifying the source code as this is a configuration of the MIDI device.

The limitations on the MIDI bandwidth will affect the performance if there are too many agents contributing. The amount of data that can be transmitted over a MIDI cable is limited to 31,250 bits per second and it takes approximately 320 milliseconds to transmit one word. When all sixteen channels are in use, messages can be transmitted at about 50 – 150 per second [19, pp.1007]. This is dependent on the speed of the MIDI device. The benefits of using MIDI however, are its ease of use and extensibility. These outweigh MIDI's limitations.

5.4 Scripting Language

Initially, it was decided to write a small scripting language that the musician could use to write his or her music compositions. Due to the lack of time, an open source public domain scripting language known as BeanShell [1], is used to parse and execute the script file. However, a small library of useful functions will be provided that the musician can use within their compositions.

BeanShell is a lightweight embeddable Java source interpreter that is written in Java and is quite comprehensive. It executes standard Java statements and expressions and also supports a weakly typed form of the Java language such as the Perl-like objects that do not need to be declared before use. Many open-source developers contribute to the BeanShell project and there is good documentation and support, making this a good front end for the musical agent scripting language. By making use of BeanShell, the composer will be provided with the same flexibility the Java language provides.

5.5 Obeying Natural Laws

Imagine there are twenty performers in a hall creating a performance of *Paragraph 7*. All the performers are singing halfway through the script. It is time for one performer to now start a new line. She listens to the other choir members to make a decision as to choosing her next note. In reality at a given moment the performer will not be able to hear all twenty performers. In the case of the musical agent, there is a similar effect due to loss of packets through the network caused by the unreliability of multicast datagrams.

The performer in the real performance would also find that her neighbours sound much louder than performers further away, as the intensity of sound decreases with distance. This will no doubt affect her decision on her next note. Since there is no measure of distance between computers over a network, it has been decided to use the round-trip time of a packet (connection latency) to estimate which computers are closer to which others. This is not a true indication of distance, but it is an interesting virtual counterpart. The intensity of a note an agent hears will be the result of a function of latency and the intensity of the note being sung by the other agent.

Imagine the performers are very far apart such that the distance between performers makes the sound delay from one singer to another noticeable. This is similarly true in a performance created by musical agents. In the case of the agents, it is the delay of the network connection that mimics the physical sound delay.

6. Agent Design

6.1 Design Overview

As described in the implementation, the multicast Internet protocol will be used. This simplifies the design of the agent as only a message need be sent and all agents will receive that message. Thus, the multicast IP address can be considered as a black box that will automatically inform all the other agents of a message.

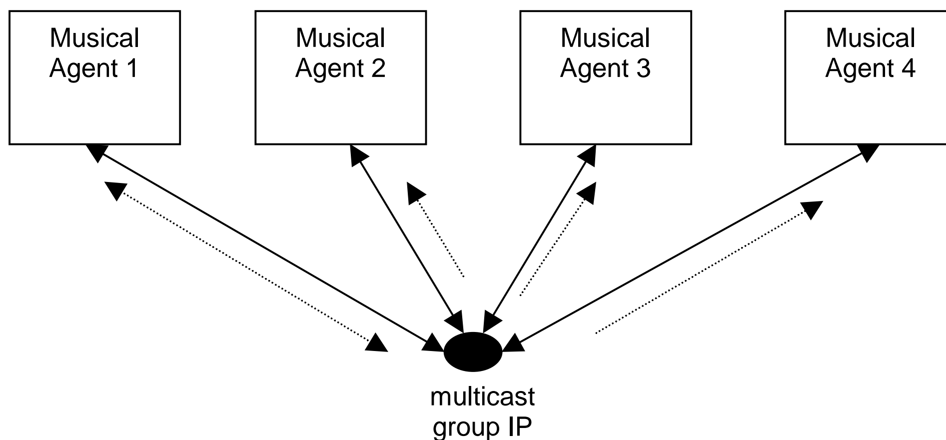

Fig 22: Musical agents send packets to a black box, which distributes the data.

A musical agent consists of a musical script file and a data structure that stores information of the agent and other Y number of agents. The data structure is constantly updated as messages are received from other agents.

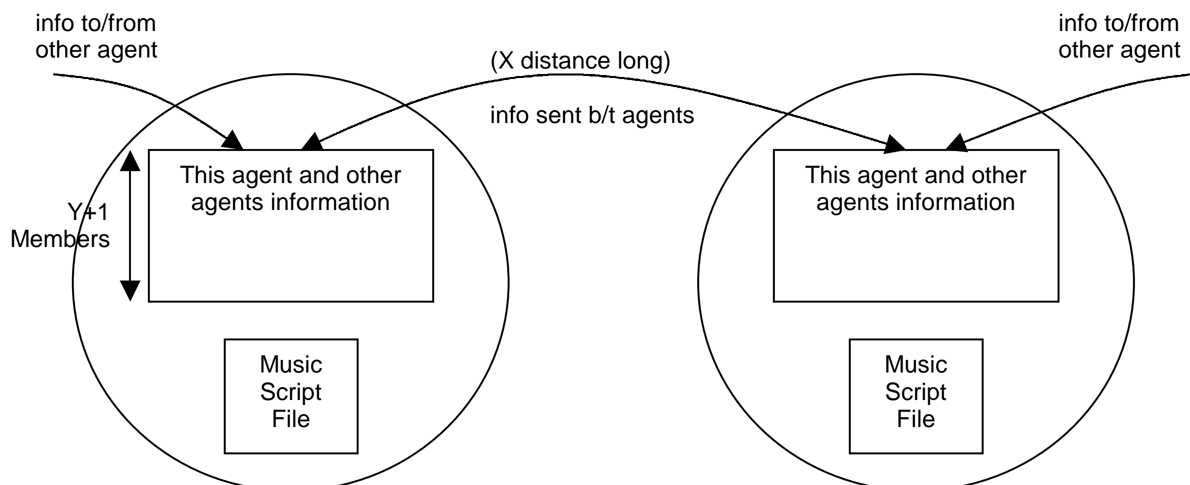

Fig 23: Abstract view of a musical agent.

The data structure that stores the information of all agents is illustrated in fig.23. The first entry is the information of the agent itself (grey line) and the others, information of its Y members.

Y+1 members

Agent ID	NoteSung	LineAt	Breath	Volume	Latency	TimeStamp	PacketSequence	Timeout	Intensity	Duration	Instrument

Fig 23: Pictorial view of the data structure.

The fields, latency and intensity are not directly extracted from a message sent from an agent but rather calculated. The intensity variable represents the pseudo intensity of a note sung by another agent. This is a function of latency. Latency is calculated half the round trip time of the message. The round trip time is defined as the time for the message to travel from the source to the destination.

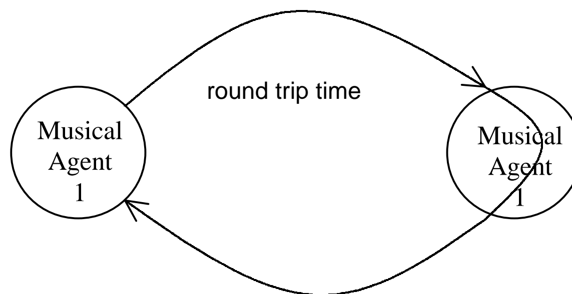

Fig 24: Definition of round trip time.

There are three different types of messages an agent may send out to its members. They are the chat message, ping message and note message.

chat message

100	Text Message	ID of source

ping message

125	ID of dest	ID of source	timestamp received	Packet Seq

120	ID of source	Note Information	timestamp	Packet Seq	Mode

Fig 25: Format of messages.

Throughout the life of the application, there are several threads continuously running. The main thread executes the music script and spawns the other threads. There is a listening thread that continuously listens on the socket waiting for new messages to arrive. This thread is asleep until a new message arrives. It is then woken up to process the message. There are also the timeout and midi note threads that are described below.

6.2 Remote Switch

On execution the agent will automatically start in the pinging mode. In this mode, the agent is continuously sending note messages every ½ second All agents will receive this datagram and will add the agent to its members list, only if the source agent is not known to the agent. The agent will then send a ping message to the source agent. Included in this ping message is the original timestamp sent by the source agent extracted from the note message, this is required by the source agent to calculate the round trip time of the message and hence determine the latency of the connection to that agent.

As soon as one of the composers press the leader signal button, the agent will switch from pinging to singing mode. The mode flag of the note message (see fig.25) will switch from 0 to 1. When an agent receives a note message, it checks the mode flag to see if it should switch to singing mode, if it does find it is 1, all its note messages will also have mode=1. This is the fastest method of informing all agents to make the switch to singing mode as the more agents that make the switch, the more messages there are informing the rest. Thus, even though there may be a slow

connection between the leader and an agent, this is not problematic because another agent will inform it to make the switch if it hasn't already.

6.3 Are You Still There?

As with all Internet applications, it is necessary for the agents to determine whether its members are still online. The musical agent achieves this by using the timeout field of each member. When any message is received from an agent, its record is updated in the data structure and the timeout field is set to the time at which the message was received. A timeout thread is always running at the background and every 10 seconds, this thread wakes up and goes through the members' data structure comparing the current time with the timeout field. If the difference between the current time and if the timeout is greater than 20 seconds, then that record is deleted from the data structure. It is assumed that the agent does no longer exist. However, if that agent does eventually send a note message, it will be added onto the data structure as a new member.

6.4 MIDI Notes

The playNote(int note,int duration,int velocity,int instrument) member of the MIDI class is the only function that is provided to generate MIDI notes. This function will automatically look through the 16 MIDI channels to see if the instrument requested is already assigned to one of the channels. If it is, the note will be played on that channel. If there are no channels with that instrument, the first free channel will be assigned that instrument. If all the channels are assigned to an instrument and the composer requests a new instrument, that note will be played on the last channel regardless of which instrument is assigned to it.

Fig 26: Class diagram of MIDI.

The playNote() function creates an instance of the Note class. The Note class inherits from the java.lang.Thread class and thus has a member function run() which overrides the run() method inherited. The Note thread will immediately send a note/on message to the MIDI device and then will sleep for the duration of the note. After this duration has passed, the Note thread will wake up and immediately send a note/off message. It will also set the free flag on the channel it played on if no more notes are being played on this channel. This puts the channel back into the free channel pool, allowing other instruments to be assigned.

6.5 Intensity of Sound

Once the latency is calculated using the round trip time, the intensity of the sound, as the agent would hear it, is also calculated. Remembering the intensity of sound decreases over distance, in this project the latency is a method used to represent virtual distances. The volume of the note sung by an agent is sent to another agent using the note messages. This volume is processed by a simple function, intensity=(Volume*Math.exp(-0.005*latency)), which gives the pseudo-intensity of that note. The above equation was obtained by taking into account the average latency of the network and tries to generate suitable differences in intensities before and after the mathematical manipulation.

7. Results and Discussions

Refer to the musical agent documentation found in appendix 2 on how to write musical script files and use the agent in a performance. Recordings for all the compositions presented in this section can be found on the website[1].

7.1 Testing Environment

The musical agents were run on five to seven Pentium III processors that were connected to a LAN via coaxial cables with a maximum bandwidth of 10 Mbits/second. The average latency of the network was measured using the musical agent to be approximately 1-2ms. All musical agents were run on Redhat Linux except for the spy (listening) agent, which was run on Windows NT.

7.2 Testing the Musical Agent

To determine whether the musical agent is working correctly, the following script file was written. This script was run on a single agent and the performance recorded.

```
cmd.WaitForLeader();

for(i=60;i<=65;i++)
{
      cmd.PlayNote(i,1000,127,40);
      cmd.Sleep(1000);
}
```

The first line, which is always required (see appendix 2), simply keeps the agent in pinging mode until the leader's signal button is pressed. The agent will then play six notes starting from middle C. The music generated from the above script was recorded using the recording functionality in the musical agent. The MIDI file was then process by MidiNotate, a shareware application by Notation Software that

[1] Recordings can be found at http://www.csse.monash.edu.au/~jfonseka/download.html

transcribes standard MIDI files back into traditional music notation. The output from MidiNotate is shown below.

Fig 27: Output from MidiNotate of the test music script.

This output agrees with the expected result of the script file. The script file plays six notes starting from middle C (note 60) on the violin (instrument 40), this is clearly illustrated in fig. 27. This proves that the musical agent and the MidiNotate software, which will be used to illustrate the music generated, are both working as expected.

7.3 *Paragraph 7*

The following script file was used to perform *Paragraph 7*.

```
// Initialise parameters
song=cmd.ReadIntoArray("paragraph7");
velocity=cmd.Random(70,127);
note=cmd.Random(27,70);
duration=cmd.Random(3000,5000);
instrument=0;
cmd.WaitForLeader();

// for each line of paragraph 7
for(i=0;i<song.length;i++)
{
   lyrics=song[i][0];
   repeat_tmp=(Double) song[i][1];
   repeat=repeat_tmp.intValue();
   loudat_tmp=(Double) song[i][2];
   loudat=loudat_tmp.intValue();

   // if the current line must be sung loud a certain number
   // of times, work out when to sing the line loud
   playHighAt=cmd.GenerateRandomNums(0,repeat,loudat);

   // sing the line j times
   for(j=0;j<repeat;j++)
   {
      // sing at normal intensity otherwise loud
      if(cmd.NumIn(playHighAt,j))
         cmd.PlayNote(note,duration,127,instrument);
      else
         cmd.PlayNote(note,duration,velocity,instrument);

      // inform the other agents about the note this agent is
      // singing
      cmd.Inform();
```

```
            cmd.Print("Singing: " + lyrics + " Note: " + note);

            cmd.Sleep(duration);

            // pick a note according to the rules of paragraph 7
            currentnote=cmd.currentNote();
            note=cmd.LoudestNote(currentnote);
            while(note==0 || note==currentnote)
            {
               note=cmd.Random(27,70);
            }
            duration=cmd.Random(1000,4000);
         }
      }
```

The first line cmd.ReadIntoArray("paragraph7"), reads in a text file that contains the text of the *paragraph 7* to be sung. Below is a subset of this text file.

```
/ You must specify the exact dimensions of the 2 dimensional
/ array to be created
/ All column data for a particular row must exist in one line

23,3

"THE ROOT"              ,   5       ,   0
"BE IN CONFUSION"       ,   13      ,   3
"NOTHING"               ,   6       ,   0
"WILL"                  ,   5       ,   1
"BE"                    ,   8       ,   0
"WELL"                  ,   8       ,   0
"GOVERNED"              ,   7       ,   0
```

The first column is the text of *Paragraph 7*, but this text is represented as a note instead in this composition. "THE ROOT" would thus, correspond to a single MIDI note. The second column represents the number of times to sing the text and the third column is the number of times to sing the text much louder than average. This is an example of a distributed performance. There were seven agents running on separate computers, each executing the above script file.

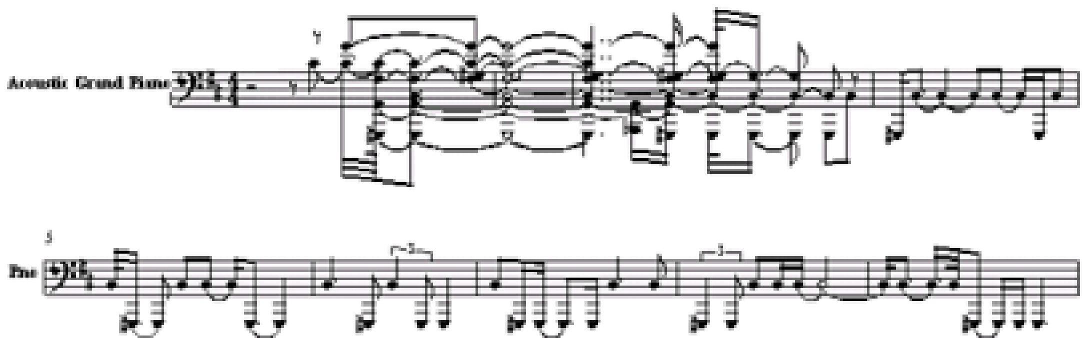

Fig. 28: The first 10 seconds of *Paragraph 7.*

As soon as the leader's signal button was pressed, all the agents started singing almost instantly. Fig.28 shows that the leader's agent produced the first note. This was expected because this agent will respond to the singing mode switch much faster compared with the other agents. However, after the first bar, all agents are participating in the performance as shown by the chord consisting of seven notes. The agents initially choose random notes and this forms a complex unpleasant chord. This chord is unpleasant because it consists of random notes played together and usually these notes do not sound good together. In the next few seconds, there were approximately three unique notes being played by the agents and the numbers dropped throughout the performance. The number of notes converged into only two unique notes that were repeated over and over again as shown in fig.28. If an agent choose $G^{\#}$, it would in the next line choose B and again in the next line $G^{\#}$. The two notes hardly changed until after a considerable amount of time, unexpectedly, one of the notes was replaced with a different pitch. This pattern was repeated continuously until the end of the performance. Because some agents can hold their breath for a longer time period, that is play a note for a longer duration, some agents finish the performance before others. This creates a sense of 'ending the performance', as one agent is left by itself to finish the performance.

The music started off from an unpleasant chord to pleasant repetitious cycle of two notes. This was the overall behaviour throughout the performance. This agrees with Nymann's statement that the notes finally converge into one or two notes as each performer begins selecting neighbouring notes [18, pp.125].

It must be noted that the pitch range was restricted from 27 to 70. By trial and error, these pitches were found to create a more pleasant performance. In the original *Paragraph 7* script, there were no restrictions on the notes the performers selected, however the human voice cannot generate notes with pitches in the extremes anyway. The performers of the original *Paragraph 7* only used one instrument, that is the human voice. The *Paragraph 7* script was modified to allow the musical agents to randomly select an instrument to play before starting the performance. This is the result obtained when each agent picks a random instrument. Note that in this case, all the six agents are not playing a unique instrument.

Fig 29: Six seconds of *Paragraph 7* with random instruments assigned.

It is interesting to note that regardless of how the script is performed, the overall result is the same, the notes always converge. In the first six seconds the notes converged to E and B^b.

<u>7.4 *Instrument Rush*</u>

This is a composition written by me. In this composition, there are seven performers in a room as well as five instruments on the floor. When the leader gives the signal, each performer must quickly grab an instrument from the floor and play the C note for a random duration. Once that note has been played, the performer must put down the instrument and try to grab another to play. If there are no instruments on

the floor, then the performer must wait patiently until one is put down. This process is continued until the performers are tired of playing. This is another example of a distributed performance. The script file for this composition can be found in appendix 3.

Fig 30: First 7 seconds of *Instrument Rush*.

The music generated from this composition seems to be generated from a random function. It is extremely difficult to associate the composition with the performance. However, by looking at fig.30, it can be seen that the rules of the composition were followed. There are only five instruments in the room and there are no chords within this composition that are built from more than five notes. By looking at fig. 30, it is extremely difficult to estimate the number of performers in the room, this is because those performers without an instrument will be silent while they wait for an instrument to become available.

7.5 Miscellaneous Remarks

A musical agent by itself does not generate interesting music but just a series of beeps (musical notes). It takes five or more agents to generate something complex and interesting from the *Paragraph 7* composition. Generally five is the least number of agents that should be used to perform any distributed composition. The more agents that are participating, the more interesting the performance will be.

It was also found during testing that a Musical Agent can handle more than ten computers without any performance loss. Because the amount of computation that occurs within an agent is minimum, the only factor that will determine how many agents can participate is the bandwidth of the Internet bandwidth. However, it must be said that if more than one agent is run on the same computer, this may have unpredictable results as each agent has a number of threads continuously running. The processor will be 'hogged up' by some of the agents not providing the rest with a time slice.

An agent that is listening to the performance may not be able to play all the notes being sung by all the agents as it will be restricted by the MIDI bandwidth. Since it is not feasible to have hundreds of computers with a musical agent running on each one, these restrictions cannot be tested. However, it must be said that these restrictions will most likely never be reached in most performances.

8. Future Work

Like most other projects, this project can have no end. Viable extensions to the current system are suggested below.

8.1 Implement Extended MIDI Functionality

In this project, only the basic MIDI functions were implemented. These are the note/on, note/off and program/change events. The musical agent is also capable of capturing these MIDI events and storing them into a MIDI file for later playback using any MIDI synthesiser. However, MIDI has some other interesting functions that would be of benefit to the composer. For example the *sound attack time* allows the musician to control how long it takes for a sound to fade in, that is, control the time it takes for the note to reach its sustain level. The *aftertouch* function could be used to change the pressure of the note while it is playing. This is analogous to a key being pressed on a keyboard and the musician gently releasing and applying pressure to it. The note is played when the key is pressed down, but there are further messages to the MIDI device notifying it of the pressure being applied at an instant. These are a small number of many MIDI functions that are available.

As explained previously, a sound bank of sampled voices could be loaded to the MIDI device and assigned to the 16 channels. This way, instead of generating beeps, the texts of a script like that of *Paragraph 7* could be sung.

8.2 Simplify the Script Language Syntax

The scripting language used in the musical agent is the one provided by Beanshell as explained previously. Since Beanshell uses the same language syntax as Java (with a few exceptions), the syntax may be difficult for novice programmers to learn. Ideally, the language should be simple so that the composer can quickly grasp it and begin writing music scripts. Making the scripting language syntax simpler however, usually means loss of flexibility. Currently, the musical agent is very powerful as any functionality can be programmed quickly using the Beanshell scripting language. In a nutshell, the simplicity of the language syntax should be compromised with the flexibility required.

8.3 Auto-Download Music Scripts

The whole purpose of the musical agent is to have many composers come together and play their compositions. These compositions are different from normal compositions because they are distributed that is to say, you need more than one musical agent to perform the composition. Currently, it is difficult for many composers to organise a performance, as the music script file will need to be downloaded by each composer by other means such as e-mail or ftp. Implementing an auto script download into the musical agent so that the music script is downloaded to all participating agents while they are waiting for the leader to give the signal will greatly improve the useability of the musical agent.

8.4 Permanent Multicast IP Address

For a composer to participate in a performance, they need to know the multicast IP address and port number. If these two parameters were made permanent so that all composers used these same settings, it would create a community of online composers. The chat feature in the musical agent was implemented so that members of this community can meet online and arrange a time for a performance or discuss interesting features of their compositions.

Conclusion

The goals of the project were fulfilled. The *Musical Agent* was developed in the Java programming language. A Java scripting language known as Beanshell was integrated to the application to handle the parsing and execution of the music script. Thus the script handler is very powerful and flexible enough for composers to write algorithmic compositions.

The musical agent was used to perform the *Paragraph 7* and *Instrument Rush* algorithmic compositions. These are distributed compositions and thus many agents were run on separate computers, each agent contributing to the performance. The music generated was saved to MIDI files using the musical agent's own recording facility.

References

1. "Beanshell – Lightweight scripting for Java", 2000. http://www.beanshell.org (22nd Sep. 2000)
2. Adami C., "Introduction to Artificial Life", Telos, 1998.
3. Beckert D., "Algorithmic Composition" [Thesis], University of Reykjavik, 1997.
4. Burks W., "Essays on Cellular Automata", University of Illinois Press, 1970.
5. Cardew C., "Scratch Music", MIT Press, Cambridge, 1969.
6. Cheung K., "DASE realtime net jamming" [Thesis], University of Technology, 2000.
7. Deering S., "Host Extensions for IP Multicasting", RFC 1112, 1989.
8. Dorin A. & McCormack J., CEMA Honours Projects, 2000. http://www.csse.monash.edu.au/~cema/projects/hons2000/hons.html (17th Sep 2000)
9. Hanna A., "MOTHER: a Csound Score Generator" [Thesis], Australian National University, 1999.
10. Harry H., "A Computational Perspective on Twenty-First Century Music", Contemporary Music Review, Vol. 14, No. 3, 1995.
11. Holmes T. B., "Electronic and Experimental Music", Charles Scribner's Sons, 1985.
12. Levy S., "Artificial Life – The Quest for a New Creation", Penguin, 1993.
13. McAlpine K., Miranda E. R. and Hoggar S., "Making Music with Algorithms: A Case-Study System", Computer Music Journal, Vol 23, No.2, Summer 1999.
14. Milliun J., Reardon J. & Smart P., "Life with your Computer", Byte, Vol. 3, No. 12, Dec. 1978.
15. Miranda E. R., "Chaosynth a Cellular Automata-based Granular Synthesiser", 1998. http://website.lineone.net/~edandalex/chaosynt.htm (14th Sep 2000)
16. Miranda E. R. & Wright J., "The Cellular Automata Granular Synthesis Technique of Chaosynth", http://www.nyrsound.com/downloads/whitepaper.pdf (16th Sep 2000)
17. Miranda E. R., "CAMUS – A Cellular Automata MUSic Generator", 1998. http://website.lineone.net/~edandalex/camus.htm (14th Sep 2000)
18. Nymann M., "Experimental Music – Cage and Beyond", Cambridge University Press, 1999.
19. Roads C., "The Computer Music Tutorial", MIT Press, 1996.
20. Trubitt D., "Making Music with Your Computer", EM Books, 1993.

Bibliography

1. Berg C. F., “Advanced Java 2: Development for Enterprise Applications”, 2nd Edition, Sun Microsystems Press, 2000.
2. Berg C. F., “Parsing Expressions in Java”, Dr. Dobbs Journal, Vol. 24, No. 1, Jan.1999, pp. 50, 52-3, 56-8.
3. Courtois T., “Java Networking and Communications”, Prentice Hall, 1997.
4. Harold E. R., “Java Network Programming”, O’Reilly, 1997.
5. Lemay L. & Cadenhead R., “Teach Yourself Java 2 in 21 Days”, SAMS, 1999.
6. Nelkon & Parker, “Advanced Level Physics”, 6th Edition, Heinemann International, 1987, pp. 584-5.
7. Oaks S. & Wong H., “Java Threads”, O’Reilly, 1999.
8. Shankel J., “Little Languages with Lex, Yacc and MFC”, Dr. Dobb’s Journal, Vol. 24, No.1, Jan 1999, pp 28,30,32-3.

Appendix 1 - Useful Resources

Algorithmic Music

1. This website has a large amount of CA music implemented that is available in MIDI, Real Audio and MP3 format. The algorithms for the cellular automata are also provided. http://jmge.net/camusic.htm

2. The homepage of the Dynamic System and Research Group. Included in this website is an introduction to cellular automata music and links to papers on algorithmic compositions. The CAMUS and Chaosynth applications can be downloaded from here. http://www.maths.gla.ac.uk/~km/dsysmus.htm

3. Visit the Fractal Music Project for papers on music created by the result of a recursive algorithm that is applied several times on the previous output. http://www-ks.rus.uni-stuttgart.de/people/schulz/fmusic/

4. This site has a lot of software for musicians including algorithmic composers. http://www.bright.net/~dlphilp/linuxsound/

5. The website for *mother*, a score generator that uses Markov processors. This website gives a detailed explanation of the Markov process and the *mother* application can also be downloaded from here. http://www.geocities.com/SiliconValley/Peaks/3346/

Cellular Automata

1. An excellent article about one dimensional cellular automata including several Java simulations. http://cgi.student.nada.kth.se/cgi-bin/d95-aeh/get/lifeeng

2. Cellsprings, a general cellular automaton explorer written in Java. This application has included a vast amount of different CAs. http://jmge.net/java/csprings/

Appendix 2 - Using the Agent

1. Requirements

The Musical Agent application requires the *Java 2 V1.3 runtime* environment be installed on the computer. The agent uses multicast IP to communicate with other musical agents and so the computer must also be setup to allow for this. The author has tested the application on a Win 98, Win NT and Linux environment. All the recent versions of these operating systems come configured with multicast support straight out of the box. The network must be also properly configured to route multicast datagrams. If the agent is to participate in a group performance with other musical agents over the Internet, there must be routers between the computers that properly route multicast datagrams. Since there are some old routers still in use that are not IPv6 compliant, this creates multicast islands where multicast datagrams cannot be transmitted from one island to another. A MIDI device is required if the user wishes to hear the performance. However, the agent can still contribute to a performance if a MIDI device is not available.

2. Downloading an Agent

The agent can be downloaded from http://www.csse.monash.edu.au/~jfonseka

3. Running the Agent

Assuming that the Java runtime environment directory is within the search path, typing the below command on the prompt will start the agent.

java -jar MusicClient.jar

On a Windows system, double clicking on the MusicClient.jar file will automatically execute the application.

4. Configuring your Agent

The 'character.ini' file is used to configure the agent as required. Below is a table of configurable parameters and some examples.

Parameter	Description	Example
breath	Specifies how long this agent can hold its breath.	6
id	A unique identifier that distinguishes this agent from others. This must be one word.	ruki
agentis	If the agent is specified as a spy, you can listen to the whole performance. This parameter may be 'spy' or 'nospy'.	spy
multicastip	Specify the multicast IP address to use.	239.1.2.3
port	Specify the port number to use.	1234
ttl	Specify how far the datagrams should travel. ttl = 0 Restricted to host tt1 =1 Restricted to same subnet tt1 < 32 Restricted to same department	1

	ttl < 128 Restricted to same continent ttl < 255 Unrestricted.	
script	Specifies the music script file to be performed. All script files must be in the 'scripts' directory.	paragraph7.bsh
splash	Specifies whether the splash screen should be displayed during start up. This parameter can either be 'yes' or 'no'.	no
/	This is a comment and will be ignored.	/ this is a comment

5. Using the Agent

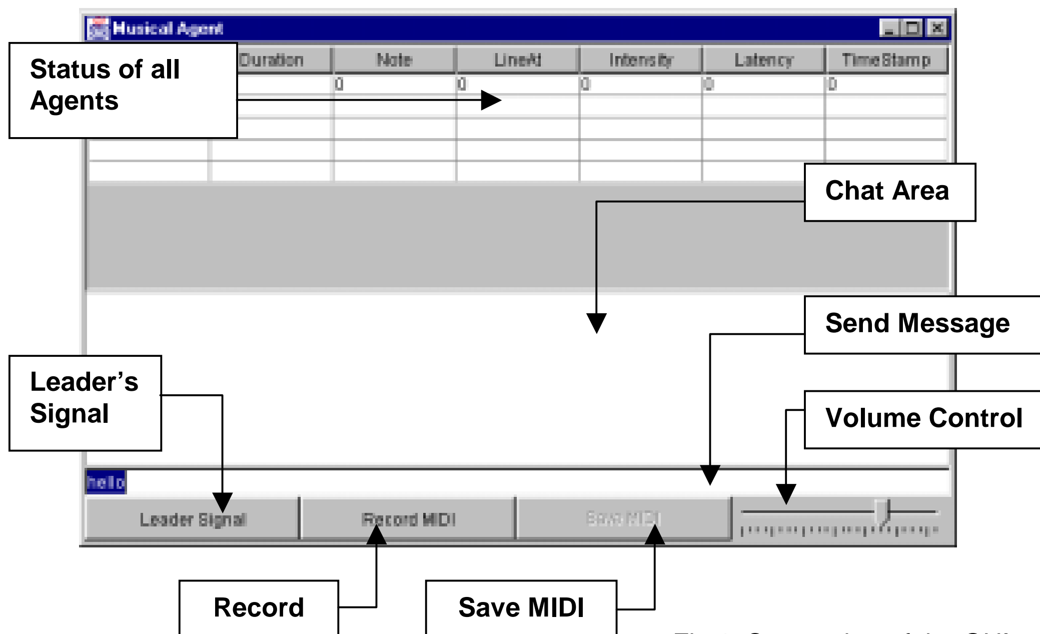

Fig 1: Screenshot of the GUI

Once the splash screen has been displayed, the above window will appear. On the status area will be displayed all the agents participating in the performance. Once all agents are registered as part of the performance, that is they all appear on the status area distinguishable by their MemberID, the nominated leader can press click the 'start singing' button. This will switch all agents from pinging to singing mode. If your agent is configured as a spy (see configuring your agent) you will hear the performance.

Sending Chat Messages

To send a text message to all agents, write the message in the textbox and then press the 'enter' key. Your message will appear all agents' chat area.

Recording a Performance

This feature is only available if there is a MIDI device on the host computer. Pressing the 'record MIDI' button will begin capturing all MIDI events. Pressing on the button again, which will have been renamed to 'stop', will stop recording. To save the recording to a MIDI file, press the 'Save MIDI' button and enter a filename.

6. Writing Music Scripts

All script files must be placed in the 'scripts' directory. To specify a script to be performed by the agent, see 'configuring your agent'. All script files must end with '.bsh'. The script files are written in Java syntax with the exception of some rules.

- It is not required to declare variables
- Variables are perl-like. They change their type automatically depending on the object they are storing. (x=5 makes x an integer while x=5.5 make x a float)
- No need to write a class with a main function. Any code within the script file will be executed.

To make maximum use of the script language, refer to the BeanShell document at http://www.beanshell.org/docs.html. Below is the set of instructions available to the user that can be used within the script files.

Function	Description
`cmd.Inform()`	Informs all other musical agents the status of this agent. (that is its note, duration, instrument, breath and volume)
`cmd.WaitForLeader()`	Put the agents in pinging mode until one agent from the performance is instructed to start singing. This line must always be included if your composition is distributed and requires other agents to create the performance.
`cmd.Print(String str)`	Prints the string out to the console.
`cmd.Sleep(int duration)`	This is a blocking function that stops all processing for the duration specified.
`cmd.PlayNote(int note, int duration, int velocity, int instrument)`	Plays a note on the MIDI device. If all 16 channels are already allocated to an instrument and a new instrument is specified, that note will be played on the 16th channel.
`int cmd.Breath()`	Returns an integer, the breath parameter specified in the character.ini file.
`int cmd.Random(int min, int max)`	Returns a random integer that is between the range min and max (both inclusive).
`int [] cmd.GenerateRandomNums(int min, int max, int num)`	Returns an array of random integers, all between the range min and max (both inclusive). All numbers in the array will be unique unless the range is not sufficient enough to generate unique numbers.
`boolean cmd.NumIn(int [] nums, int val)`	Checks whether the number val is in the array of integers nums. Returns true if it is otherwise false.
`int cmd.LoudestNote(int ignoteNote)`	This function returns an integer that is the pitch of the note being sung by an agent. The user can specify a note or notes to ignore. If all the notes being sung at the moment by the agents are in the ignore list, 0 will be returned.
`int cmd.LoudestNote(int [] ignoreNotes)`	
`int cmd.SoftestNote(int ignoreNote)`	
`int cmd.SoftestNote(int [] ignoreNotes)`	

`int cmd.RandomNote(int ignoreNote)`	
`int cmd.RandomNote(int [] ignoreNote)`	
`int cmd.CurrentNote()`	This will return an integer, the pitch of the note being sung by the agent.
`Object [][] cmd.ReadIntoArray(String filename)`	Loads in a text file from the filename specified and returns a 2-dimensional array of Objects. This function will interpret the data as either a Double or a String. The file format of the text file is as follows: 2,3 “french fries”, 5, 5.95 “pizza”, 2, 1.95

The below functions all deal with the following MemberList class structure.

```
public class MemberList
{
   public String ID;
   public int NoteSung;
   public int LineAt;
   public int Breath;
   public short Volume;
   public short Duration;
   public short Instrument;
}
```

`int cmd.MembersNum()`	Returns the number of members this agent has.
`MemberList cmd.ReadInfo(int index)`	Returns a member’s information (see MemberList structure above) at the index specified. Information of the members are stored in an array. Use the cmd.MembersNum() function to first work out the number of members in the array.

Example music script

The sample music script demonstrates some of the above functions. Included in the script directory can be found more sample music script files.

```
velocity=127;
note=20;
instrument1=cmd.Random(30,60);
instrument2=cmd.Random(100,127);
cmd.WaitForLeader();
duration=cmd.Random(1000,2000);
increase=true;

while(true)
{
   cmd.Inform();
   cmd.PlayNote(note,duration,velocity,instrument1);
   cmd.Sleep(duration/2);
   cmd.PlayNote(note,duration,velocity,instrument2);
   cmd.Sleep(duration/2);
```

```
   if(increase)
      note=(cmd.LoudestNote(0) + 1) % 127;
   else
      note=(cmd.SoftestNote(0) - 1) % 127;
   velocity=cmd.Random(120,127);

   if(note>=60)
      increase=false;

   if(note<=15)
      break;
}
```

Appendix 3 – Instrument Rush Script

```
// initialise all parameters
instruments=new int[5];
instruments[0]=60;
instruments[1]=13;
instruments[2]=34;
instruments[3]=44;
instruments[4]=95;

currentlybeingplayed=new int[5];


cmd.WaitForLeader();

while(true)
{
	// clear currentlybeingplayed array
	for(i=0;i<instruments.length;i++)
		currentlybeingplayed[i]=0;

	// check to see which instruments are being used
	for(i=0;i<cmd.MembersNum();i++)
	{
		member=cmd.ReadInfo(i);
		for(j=0;j<instruments.length;j++)
		{
			if(instruments[j]==member.Instrument)
				currentlybeingplayed[j]=1;
		}
	}

	// work out how many instruments are free
	notbeingplayed=0;
	for(i=0;i<instruments.length;i++)
	{
		if(currentlybeingplayed[i]==0)
			notbeingplayed++;
	}


	// play a note only if you are holding an instrument
	if(notbeingplayed>0)
	{

		// pick a random instrument
		pick=cmd.Random(0,notbeingplayed);
		free=0;
		for(i=0;i<instruments.length;i++)
		{
			if(currentlybeingplayed[i]==0)
			{
			if(free==pick)
				{
					pickedinstrument=instruments[i];
					break;
				}
				free++;
```

```
                }
            }


            // now play it
            cmd.Inform();
            cmd.Sleep(1000);
        }
}
```