

Applications of Dynamical Systems to Music Composition

Kenneth B. McAlpine, A.L.C.M., B.Sc.

Thesis submitted for the Degree of Doctor of Philosophy in the
Faculty of Science, University of Glasgow

Department of Mathematics
University of Glasgow
August 1999

ProQuest Number: 13834243

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 13834243

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

GLASGOW
UNIVERSITY
LIBRARY

11938 (copy 2)

For mum and dad.



Table of Contents

TABLE OF CONTENTS	III
LIST OF FIGURES.....	VIII
LIST OF TABLES.....	XIII
LIST OF ALGORITHMS	XV
ACKNOWLEDGEMENTS.....	XVI
DECLARATION.....	XVIII
PREFACE.....	XIX
ABSTRACT	XXI
1. ALGORITHMIC COMPOSITION – A DEFINITION.....	1
1.0 INTRODUCTION.....	1
1.1 WHAT IS ALGORITHMIC COMPOSITION?.....	2
1.1.1 <i>Algorithmic processes</i>	2
1.1.2 <i>Musical ‘algorithms’</i>	6
1.1.3 <i>Musical composition</i>	7
1.2 COMPOSING ALGORITHMICALLY	8
1.2.1 <i>A step-by-step guide to making music</i>	8
1.2.2 <i>Algorithmic composition as a model for inspiration</i>	9
1.3 A GENERAL OUTLINE OF THE PROBLEM	11
1.3.1 <i>Music as pattern propagation</i>	11
1.3.2 <i>Research goals</i>	12
2. A BRIEF HISTORY OF ALGORITHMIC COMPOSITION.....	13
2.0 INTRODUCTION.....	13
2.1.1 <i>What are stochastic algorithms?</i>	13
2.1.2 <i>Probability Lookup Tables</i>	14
2.1.3 <i>Using probability tables</i>	20
2.1.4 <i>Markov Chains</i>	24
2.1.5 <i>Quality of Results</i>	28
2.2 FORMAL GRAMMARS AND AUTOMATA	30
2.2.1 <i>Music as language</i>	30
2.2.2 <i>The use of formal grammars in music</i>	31

2.2.3 Cellular Automata.....	32
2.2.4 The Game of Life.....	33
2.2.5 Musical Applications of Cellular Automata.....	36
2.3 ITERATIVE ALGORITHMS.....	37
2.3.1 What is an iterative process?.....	37
2.3.2 Implementing iterative processes musically.....	40
2.3.3 Fractal Algorithms.....	41
2.3.4 Musical fractals.....	45
2.3.5 1/f Noise.....	46
2.3.6 The Mandelbrot Set.....	47
2.4 EVOLUTIONARY ALGORITHMS.....	50
2.4.1 Artificial neural networks.....	50
2.4.2 Modelling neural networks.....	50
2.4.3 Types of activation function.....	52
2.4.4 Graphical representation of a neural network.....	54
2.4.5 Learning in a neural network.....	55
2.4.6 Neural networks and music.....	56
2.4.7 Genetic Algorithms.....	57
2.4.8 Musical applications of GAs.....	59
2.5 SERIAL ALGORITHMS.....	60
2.5.1 Serialism.....	60
2.5.2 Implementing a serial algorithm.....	61
2.5.3 Quality of results.....	64
2.6 RULE-BASE ALGORITHMS.....	65
2.6.1 What is a rule-base?.....	65
2.6.2 Knowledge representation in an expert system.....	66
2.6.3 Some benefits and drawbacks of rule-base systems.....	67
2.7 TURING SYSTEMS: A NEW MEANS OF CLASSIFICATION.....	68
2.8 SUMMARY.....	70
3. CAMUS – A CELLULAR AUTOMATA BASED MUSIC ALGORITHM.....	72
3.0 INTRODUCTION.....	72
3.1 CELLULAR AUTOMATA MUSIC.....	72
3.1.1 The Game of Life.....	72
3.1.2 The Demon Cyclic Space.....	73
3.1.3 The Automata Space.....	74
3.1.4 The CAMUS algorithm.....	75
3.1.5 Chords in CAMUS.....	76
3.1.6 From automata to music.....	77

3.2 COMPOSITION EXAMPLE	89
3.2.1 <i>The interface</i>	89
3.2.2 <i>Changing the Demon Cyclic Space rules</i>	93
3.2.3 <i>Changing the Game of Life rules</i>	95
3.2.4 <i>Setting the instrumentation</i>	96
3.2.5 <i>Setting the articulations</i>	97
3.2.6 <i>Initialising the Game of Life</i>	102
3.2.7 <i>Generating the composition</i>	103
3.3 TOWARDS AN EFFICIENT WORKING PROCEDURE IN CAMUS.....	106
3.3.1 <i>Two important considerations</i>	107
3.3.2 <i>Some terminology</i>	107
3.3.3 <i>Initial states of the game of Life</i>	107
3.3.4 <i>Doomed cells and their musical legacy</i>	108
3.3.5 <i>Gliders</i>	111
3.3.6 <i>Cyclic configurations</i>	114
3.3.7 <i>The process of 'reverse engineering'</i>	125
4. DEVELOPING THE SYSTEM.....	128
4.0 INTRODUCTION.....	128
4.1 LIMITATIONS WITH THE PRESENT SYSTEM.....	128
4.1.1 <i>Chord complexity</i>	128
4.1.2 <i>New dimensions of complexity</i>	129
4.1.3 <i>Rules for life in a 3-dimensional universe</i>	133
4.1.4 <i>Needles in haystacks</i>	137
4.1.5 <i>A serial composer</i>	140
4.1.6 <i>Dynamical systems for a dynamic system</i>	142
4.1.7 <i>Alternative mappings</i>	148
5. CAMUS 3D.....	150
5.0 INTRODUCTION.....	150
5.1 THE DEVELOPMENT OF THE SYSTEM	150
5.1.1 <i>Into the third dimension</i>	150
5.1.2 <i>The 3-dimensional Game of Life</i>	150
5.1.3 <i>Candidates for rules in the 3-dimensional Game of Life</i>	151
5.1.4 <i>The rule $R = (4, 5, 5, 5)$</i>	151
5.1.5 <i>The rule $R = (5, 7, 6, 6)$</i>	155
5.1.6 <i>Conway's Life in 3 dimensions</i>	159
5.1.7 <i>The Demon Cyclic Space in three dimensions</i>	162
5.1.8 <i>The emergent behaviour of the 3-dimensional Demon Cyclic Space</i>	162
5.2 MAPPING THE CONTROL SYSTEM TO MUSIC	163

5.2.1	<i>The mapping</i>	163
5.2.2	<i>Data entry in the third dimension</i>	167
5.2.3	<i>The display</i>	174
5.2.4	<i>Altering the evolution rules</i>	177
5.2.5	<i>Choosing rhythms in CAMUS 3D</i>	180
5.2.6	<i>Pitch generation</i>	183
5.2.7	<i>Instrumentation</i>	186
5.3	THE CAMUS 3D ALGORITHM AS A FLOWCHART	190
5.4	COMPOSITION EXAMPLE	202
5.5	COMPARISONS BETWEEN THE 2 AND 3-DIMENSIONAL SYSTEMS	218
5.5.1	<i>Peas in a pod?</i>	219
5.5.2	<i>Key differences between the systems</i>	219
5.5.3	<i>Differences in interface</i>	223
5.5.4	<i>Differences in working techniques</i>	223
6.	WORK IN PROGRESS	224
6.0	INTRODUCTION	224
6.1	AN AUTOMATIC CHORD CLASSIFIER	224
6.1.1	<i>The dissonance function</i>	224
6.1.2	<i>The order-of-preference function</i>	230
6.1.3	<i>The dissonance network</i>	240
6.1.4	<i>Constructing the network</i>	241
6.1.5	<i>Validity of the index of dissonance</i>	246
6.1.6	<i>Extending the dissonance network</i>	248
6.1.7	<i>Integrating the dissonance network with CAMUS 3D</i>	253
6.2	FURTHER DEVELOPMENT OF THE SYSTEM	256
6.2.1	<i>Developing long term trends</i>	256
6.2.2	<i>Fractal orbits</i>	258
6.2.3	<i>The rhythm of Life</i>	259
7.	CONCLUSION	263
7.1	RESEARCH GOALS	263
7.1.1	<i>Producing new musical algorithms</i>	263
7.1.2	<i>what mappings produce useful results in sound?</i>	264
7.1.3	<i>Producing new tools for music composition</i>	265
7.1.4	<i>Producing new compositions</i>	265
7.2	AUTHORSHIP OF ALGORITHMIC COMPOSITIONS	267
APPENDIX A.		271
	USING THE CD	271

APPENDIX B	274
SOME COMMON CONWAY OBJECTS	274
APPENDIX C	277
MIDI.....	277
<i>What is MIDI?</i>	277
<i>The history of MIDI</i>	278
<i>MIDI transmission</i>	279
<i>MIDI pitch representation</i>	281
<i>MIDI modes</i>	282
<i>The MIDI file format</i>	283
<i>Connectivity</i>	292
<i>General MIDI and beyond</i>	294
APPENDIX D	298
AN ALTERNATIVE APPROACH TO ALGORITHMIC COMPOSITION.....	298
<i>Chaosynth</i>	298
<i>Musinum</i>	304
<i>KeyKit</i>	310
<i>The Well-Tempered Fractal</i>	314
<i>Band-in-a-Box</i>	317
APPENDIX E	321
PUBLICATIONS AND PAPERS	321
<i>Papers published in the proceedings of a refereed conference</i>	321
<i>Papers published in a refereed journal</i>	321
<i>Other publications</i>	321
BIBLIOGRAPHY	323

List of figures

FIGURE 1.1.1 – TWO NUMERIC TABLES FROM A MUSICAL DICE GAME BY MOZART.....	4
FIGURE 2.1.1 – CARE MUST BE TAKEN WHEN CREATING A DISCRETE PROBABILITY TABLE FROM A CONTINUOUS DISTRIBUTION.....	16
FIGURE 2.1.2 – THE NORMALISED EXPONENTIAL DISTRIBUTION ON THE POINTS $x = 0, 0.5, 1$	17
FIGURE 2.1.3 – UNIFORM PROBABILITY DISTRIBUTION.....	17
FIGURE 2.1.4 – LINEAR PROBABILITY DISTRIBUTION.....	18
FIGURE 2.1.5 – PIECEWISE-LINEAR PROBABILITY DISTRIBUTION.....	19
FIGURE 2.1.6 – EXPONENTIAL PROBABILITY DISTRIBUTION.....	19
FIGURE 2.1.7 – BELL-SHAPED PROBABILITY DISTRIBUTION.....	20
FIGURE 2.1.8 – U-SHAPED PROBABILITY DISTRIBUTION.....	20
FIGURE 2.1.9 – CUMULATIVE DISTRIBUTION METHOD FOR NON-UNIFORM DISTRIBUTIONS.....	22
FIGURE 2.1.10 – STATE-TRANSITION MATRIX AND GRAPHICAL REPRESENTATION OF A (FIRST ORDER) MARKOV CHAIN.....	25
FIGURE 2.3.1 – SCHEMATIC OF AN ITERATIVE PROCESS, F	37
FIGURE 2.3.2 – GRAPHICAL REPRESENTATION OF THE ORBIT THAT ARISES FROM $x_{N+1} = 3.1x_N(1 - x_N)$ WITH $x_0 = 0.5$	38
FIGURE 2.3.3 – GRAPHICAL REPRESENTATION OF THE ORBITS THAT ARISE FROM $x_{N+1} = 4x_N(1 - x_N)$ WITH $x_0 = 0.3$ AND $x_0 = 0.301$	39
FIGURE 2.3.4 A – THE COMPLETE MANDELBROT SET AND TWO SUCCESSIVE MAGNIFICATIONS.....	43
FIGURE 2.3.4 B – GRAPH OF $\sin x$ AND MAGNIFICATION.....	44
FIGURE 2.3.5 – $1/f$ NOISE AND MAGNIFICATION.....	45
FIGURE 2.3.6 – SIERPINSKI GASKET AND MAGNIFICATION.....	45
FIGURE 2.3.7 – FRACTAL SPIRAL AND DETAIL.....	45
FIGURE 2.4.1 – MODEL OF AN ARTIFICIAL NEURON.....	51
FIGURE 2.4.2 – THRESHOLD FUNCTION.....	52
FIGURE 2.4.3 – PIECEWISE LINEAR FUNCTION.....	53
FIGURE 2.4.4 – SIGMOID FUNCTION.....	54
FIGURE 2.4.5 – A SIMPLE NEURAL NETWORK.....	54
FIGURE 2.6.1 – SIMPLISTIC EXPERT SYSTEM THAT REPEATS THE EXPERTS' RESPONSES EXACTLY.....	68
FIGURE 3.1.1 – SIX SUCCESSIVE TIMESTEPS OF THE GAME OF LIFE.....	73
FIGURE 3.1.2 – TWO TIMESTEPS OF THE DEMON CYCLIC SPACE.....	74
FIGURE 3.1.3 – MAPPING A 2-DIMENSIONAL AUTOMATON SPACE ONTO A TORUS.....	75
FIGURE 3.1.4 – CONFIGURATION FOR A TYPICAL TIMESTEP OF THE CELLULAR AUTOMATA MUSIC ALGORITHM USED IN CAMUS.....	78
FIGURE 3.1.5 – TEN DIFFERENT TEMPORAL CODES.....	82
FIGURE 3.1.6 – THE CAMUS ALGORITHM.....	83

FIGURE 3.1.7 – ALGORITHM FOR CALCULATING THE NOTES OF THE TRIAD DEFINED BY CELL (X, Y) AND FUNDAMENTAL PITCH P_0 .	84
FIGURE 3.1.8A – INITIALISATION OF PARAMETERS FOR CALCULATING THE NOTE ORDERINGS OF THE TRIAD DEFINED BY (X, Y) IN A TOROIDAL GAME OF LIFE AUTOMATON OF SIZE (HSIZE, VSIZE).	85
FIGURE 3.1.8B – DECISION ROUTINE BASED ON THE TGG AND DUR PARAMETERS.	86
FIGURE 3.1.8C – DECISION ROUTINE BASED ON THE TGG AND DUR PARAMETERS.	87
FIGURE 3.1.8D – DECISION ROUTINE BASED ON THE TGG AND DUR PARAMETERS.	88
FIGURE 3.1.8E – DECISION ROUTINE BASED ON THE TGG AND DUR PARAMETERS.	89
FIGURE 3.2.1 – THE CAMUS TOOLBAR.	90
FIGURE 3.2.2 – THE GAME OF LIFE WINDOW.	91
FIGURE 3.2.3 – THE DEMON CYCLIC SPACE WINDOW.	92
FIGURE 3.2.4 – THE ALTER DEMON CYCLIC SPACE RULES WINDOW.	93
FIGURE 3.2.5 – ALTERING THE NUMBER OF INSTRUMENTS BY LIMITING THE NUMBER OF POSSIBLE STATES IN THE DEMON CYCLIC SPACE.	94
FIGURE 3.2.6 – THE ALTER GAME OF LIFE RULES WINDOW.	95
FIGURE 3.2.7 – THE MIDI INSTRUMENTATION SETUP WINDOW.	96
FIGURE 3.2.8 – CHANGING THE MIDI INSTRUMENTS OF THE FIRST FOUR MIDI CHANNELS.	96
FIGURE 3.2.9 – THE CHANGE COMPOSITION SETTINGS WINDOW.	97
FIGURE 3.2.10 – ALTERING THE TWELVE-PITCH SEQUENCE OF AN ARTICULATION.	98
FIGURE 3.2.11 – SETTING THE SPEED AND RUBATO PARAMETERS.	99
FIGURE 3.2.12 – SETTING THE DYNAMIC LEVEL AND RANGE.	99
FIGURE 3.2.13 – RANDOM NUMBER CONTROL.	99
FIGURE 3.2.14 – ORDERING THE ARTICULATIONS.	100
FIGURE 3.2.15 – CHANGING THE NUMBER OF LOOPS.	101
FIGURE 3.2.16 – ALTERING THE ARTICULATION ORDER.	101
FIGURE 3.2.17 – SETTINGS FOR THE SECOND ARTICULATION.	102
FIGURE 3.2.18 – INITIAL CELL CONFIGURATION FOR THE GAME OF LIFE.	102
FIGURE 3.2.19 – C3(4, 3) CHORD GENERATED BY THE FIRST LIVE CELL OF THE CONFIGURATION OF FIGURE 3.2.18.	103
FIGURE 3.2.20 – THE RESULTING CHORD OBTAINED FROM LIVE CELL (4, 3) IN THE GAME OF LIFE.	105
FIGURE 3.2.21 – MUSIC GENERATED BY PERFORMING TWO STEPS OF THE CAMUS ALGORITHM.	106
FIGURE 3.3.1 – SINGLE LIVE CELL.	108
FIGURE 3.3.2 – ADJACENT PAIR OF LIVE CELLS.	109
FIGURE 3.3.3 – CELLULAR EVOLUTION FOR THE GLIDER CONFIGURATION.	111
FIGURE 3.3.4 – CELLULAR EVOLUTION FOR THE BROKEN CROSS.	115
FIGURE 3.3.5 – CELLULAR EVOLUTION FOR THE THREE-QUARTER CROSS CONFIGURATION.	117
FIGURE 3.3.6 – CELLULAR EVOLUTION FOR THE BOW TIE CONFIGURATION.	121
FIGURE 4.1.1 – CONSTRUCTING AN EMBEDDING BY PROJECTING A PLANE FROM THE 3-DIMENSIONAL GAME OF LIFE TO A ROW (OR COLUMN) OF CELLS IN A 2-DIMENSIONAL ARRAY.	131

FIGURE 4.1.2A -- THE USER CLICKS ON THE POINT (3,5) IN THE X-Y PLANE, FIXING THE LINES $x = 3$ AND $y = 4$ IN THE OTHER TWO EMBEDDINGS.....	132
FIGURE 4.1.2B --THE USER CLICKS ON THE POINT (3,4) IN THE X-Z PLANE, FIXING THE UNIQUE POINT (3, 5, 4) IN THE THREE-DIMENSIONAL SPACE.	132
FIGURE 4.1.3 -- A DEAD CELL WHICH LIES NEXT TO A PLANE OF LIVE CELLS HAS AT MOST NINE LIVE NEIGHBOURS.....	134
FIGURE 4.1.4 -- THE RULE OF THE FORM (3, 3, 4, 5) LEADS TO UNLIMITED GROWTH.	136
FIGURE 4.1.5 -- A TYPICAL RHYTHMIC FIGURE PRODUCED BY CAMUS'S RHYTHM ENGINE.....	138
FIGURE 4.1.6 -- A PHRASE FROM BACH'S FUGUE NO. 2 FROM THE WELL-TEMPERED CLAVIER.	138
FIGURE 4.1.7 - INTERVAL SEQUENCE FOR AN $(M \times M)$ ARRAY.	140
FIGURE 4.1.8 -- CELL NEIGHBOURHOOD OF SIZE 2.	141
FIGURE 4.1.9 -- THE MANDELBROT SET AND TWO JULIA SETS WITH RESPECTIVE REGIONS.	145
FIGURE 4.1.10 - AN ALTERNATIVE TWO-DIMENSIONAL SPACE FOR USE IN MAPPING DYNAMICAL SYSTEMS TO MUSIC.	148
FIGURE 4.1.11 - A CELL AT POSITION (2 , 4) DETERMINES A PROBABILITY TABLE ASSOCIATED WITH THAT NOTE.....	149
FIGURE 5.1.1 -- EXPANDING 2-DIMENSIONAL OBJECTS TO 3-DIMENSIONS	151
FIGURE 5.1.2 -- 3-DIMENSIONAL ANALOGUE OF A STABLE 4-CELL OBJECT UNDER $R = (4, 5, 5, 5)$	155
FIGURE 5.1.3 -- 3-DIMENSIONAL ANALOGUE OF THE CONWAY GLIDER UNDER $R = (5, 7, 6, 6)$	158
FIGURE 5.1.4 -- 3-DIMENSIONAL ANALOGUE OF A THREE-CELL OSCILLATOR UNDER $R = (5, 7, 6, 6)$	158
FIGURE 5.1.5 -- 3-DIMENSIONAL ANALOGUE OF A FOUR-CELL STABLE OBJECT UNDER $R = (5, 7, 6, 6)$	158
FIGURE 5.1.6 -- A 3-DIMENSIONAL GAME OF LIFE AS A SERIES OF PARALLEL STACKED 2-DIMENSIONAL GAMES	160
FIGURE 5.1.7 -- A SECTION OF A TIME-SPACE BARRIER.....	160
FIGURE 5.1.8 - TWO INFINITELY LARGE PARALLEL TIME-SPACE BARRIERS ARE CONSTRUCTED AND PLACED IN THE UNIVERSE SO THAT THEY ARE SEPARATED BY FOUR PLANES	161
FIGURE 5.1.9 -- SELF-ORGANISING BEHAVIOUR IN THE X-Y PLANE OF THE 3-DIMENSIONAL DEMON CYCLIC SPACE.....	163
FIGURE 5.2.1 -- CONFIGURATION OF A TYPICAL TIMESTEP IN CAMUS 3D.....	165
FIGURE 5.2.2 -- THE NOTE ORDERINGS DIALOG BOX.....	166
FIGURE 5.2.3 -- THE INVERT CELL DIALOG BOX.	168
FIGURE 5.2.4 -- THE LINE DIALOG BOX.....	169
FIGURE 5.2.5 -- THE CUBE DIALOG BOX.	171
FIGURE 5.2.6 -- THE EXTEND DIALOG BOX.	174
FIGURE 5.2.7 -- CELLS IN THE 3-DIMENSIONAL DEMON CYCLIC SPACE VERY QUICKLY BECOME OBSCURED BY THOSE IN FRONT.....	175
FIGURE 5.2.8 -- THE ISOMETRIC VIEW OF THE 3-DIMENSIONAL GAME OF LIFE.	176
FIGURE 5.2.9 -- THE X-Y PLANE VIEW OF THE 3-DIMENSIONAL GAME OF LIFE.	176
FIGURE 5.2.10 -- THE X-Z PLANE VIEW OF THE 3-DIMENSIONAL GAME OF LIFE.....	177

FIGURE 5.2.11 – THE Y-Z PLANE VIEW OF THE 3-DIMENSIONAL GAME OF LIFE.....	177
FIGURE 5.2.12 – THE ALTER GAME OF LIFE RULES DIALOG BOX.....	178
FIGURE 5.2.13 – THE ALTER DEMON CYCLIC SPACE RULES DIALOG BOX.....	179
FIGURE 5.2.14 – THE NOTE LENGTHS DIALOG BOX.....	181
FIGURE 5.2.15 – THE STANDARD NOTE LENGTHS DIALOG BOX.....	183
FIGURE 5.2.16 – THE PITCHES DIALOG BOX.....	185
FIGURE 5.2.17 – THE INSTRUMENTATION DIALOG BOX.....	187
FIGURE 5.3.1 – THE CAMUS 3D ALGORITHM.....	190
FIGURE 5.3.2 – ALGORITHM FOR CALCULATING THE NOTES OF THE FOUR-NOTE CHORD DEFINED BY CELL (X, Y, Z).....	191
FIGURE 5.3.3 – ALGORITHM FOR DETERMINING THE INSERTION OF RESTS.....	192
FIGURE 5.3.4 – ALGORITHM FOR DETERMINING CHORD SHAPE.....	193
FIGURE 5.3.5A – DECISION ALGORITHM FOR GENERATING CHORDS.....	194
FIGURE 5.3.5B – ALGORITHM FOR PERFORMING FOUR-NOTE CHORDS.....	195
FIGURE 5.3.5C – ALGORITHM FOR PERFORMING THREE-NOTE CHORDS.....	195
FIGURE 5.3.5D – ALGORITHM FOR PERFORMING TWO-NOTE CHORDS.....	196
FIGURE 5.3.5D.1 – ALGORITHM FOR PERFORMING TWO-NOTE CHORDS.....	197
FIGURE 5.3.5D.2 – ALGORITHM FOR PERFORMING TWO-NOTE CHORDS.....	198
FIGURE 5.3.5D.3 – ALGORITHM FOR PERFORMING TWO-NOTE CHORDS.....	199
FIGURE 5.3.5D.4 – ALGORITHM FOR PERFORMING TWO-NOTE CHORDS.....	200
FIGURE 5.3.5E – ALGORITHM FOR PERFORMING ONE-NOTE CHORDS.....	201
FIGURE 5.3.6 – ALGORITHM FOR DETERMINING NOTE VALUES USING A FIRST ORDER MARKOV CHAIN WITH TRANSITION MATRIX P(X, Y).....	202
FIGURE 5.4.1 – THE CAMUS 3D TOOLBAR.....	203
FIGURE 5.4.2 – UPDATED SETTINGS FOR THE 3-DIMENSIONAL GAME OF LIFE AUTOMATON.....	206
FIGURE 5.4.3 – UPDATED SETTINGS FOR THE 3-DIMENSIONAL DEMON CYCLIC SPACE AUTOMATON....	207
FIGURE 5.4.4A – A TYPICAL CONFIGURATION OF THE 3-DIMENSIONAL GAME OF LIFE.....	208
FIGURE 5.4.4B – A TYPICAL CONFIGURATION OF THE 3-DIMENSIONAL GAME OF LIFE VIEWED AS AN XY-PLANE PROJECTION.....	208
FIGURE 5.4.4C – A TYPICAL CONFIGURATION OF THE 3-DIMENSIONAL GAME OF LIFE VIEWED AS AN XZ-PLANE PROJECTION.....	209
FIGURE 5.4.4D – A TYPICAL CONFIGURATION OF THE 3-DIMENSIONAL GAME OF LIFE VIEWED AS A YZ-PLANE PROJECTION.....	209
FIGURE 5.4.5 – UPDATED SETTINGS FOR THE MIDI INSTRUMENTATION SETUP DIALOG.....	210
FIGURE 5.4.6A – UPDATED SETTINGS FOR PITCHES DIALOG.....	211
FIGURE 5.4.6B – NORMALISED SETTINGS FOR PITCHES DIALOG.....	212
FIGURE 5.4.7 – TYPICAL SETTINGS FOR THE NOTE ORDERINGS DIALOG.....	213
FIGURE 5.4.8A – TYPICAL RESPONSE FOR THE ADVANCED NOTE LENGTHS DIALOG.....	214
FIGURE 5.4.8B – TYPICAL RESPONSE FOR THE STANDARD NOTE LENGTHS DIALOG.....	215

FIGURE 5.4.9 – LABELLED DIRECTED GRAPH CORRESPONDING TO THE PROBABILITY SETTINGS OF FIGURE 5.4.8.	216
FIGURE 5.4.10 – MUSIC GENERATED USING THE CAMUS 3D ALGORITHM.	218
FIGURE 6.1.1 – PART OF THE TWELVE EQUIVALENCE CLASSES FORMED BY THE RELATION ρ	229
FIGURE 6.1.2A – FIRST PAGE OF THE HARMONIC PREFERENCE RESPONSE FORM.	232
FIGURE 6.1.2B – SECOND PAGE OF THE HARMONIC PREFERENCE RESPONSE FORM.	233
FIGURE 6.1.3 – RANK FOR EACH OF THE TWELVE SIMPLE INTERVALS.	237
FIGURE 6.1.4 – NUMBER OF VOTES CAST IN EACH CATEGORY FOR THE TWELVE SIMPLE CHORDS.	238
FIGURE 6.1.5 – THE CHORD CLASSIFICATION STAGE OF THE DISSONANCE NETWORK.	242
FIGURE 6.1.6 – DISSONANCE NETWORK FOR TWO SINGLE-NOTE CHORDS.	243
FIGURE 6.1.7 – MAIN OPERATING WINDOW OF THE DISSONANCE NETWORK.	245
FIGURE 6.1.8 – NETWORK PARAMETER SETTINGS DIALOG BOX.	246
FIGURE 6.1.9 – AUTOMATED NETWORK FOR CALCULATING THE COMBINED INDEX OF DISSONANCE GIVEN TWO 4-NOTE CHORDS, C_1 AND C_2	253
FIGURE 6.1.10 – AN EXAMPLE USAGE OF THE DISSONANCE NETWORK	254
FIGURE 6.1.11 – AN ALTERNATIVE EXAMPLE USAGE OF THE DISSONANCE NETWORK.	255
FIGURE 6.2.1 – THE TRANSFORMER MODULE.	258
FIGURE 6.2.2 – THE LORENZ ATTRACTOR.	259
FIGURE C1 – SIMPLE MIDI CONNECTION	277
FIGURE C2 – ONE BAR OF MUSIC TRANSLATED TO A MIDI FILE.	291
FIGURE C3 – SIMPLE BI-DIRECTIONAL MIDI CONFIGURATION FOR A KEYBOARD AND A COMPUTER RUNNING SEQUENCER SOFTWARE.	292
FIGURE C4 – SEVERAL MIDI INSTRUMENTS DAISY-CHAINED USING THRU CONNECTIONS.	293
FIGURE D1 – THE MAIN CHAOSYNTH WINDOW.	301
FIGURE D2 – THE GRANULE SIZE GRAPH.	302
FIGURE D3 – THE OSCILLATORS GRAPH.	303
FIGURE D4 – THE MAIN OPERATING WINDOW OF MUSINUM.	307
FIGURE D5 – THE MUSINUM GRAPHICS DIALOG.	308
FIGURE D6 – THE MUSINUM SCRIPT DIALOG.	308
FIGURE D7 – 12 BARS OF MUSIC GENERATED USING MUSINUM’S DEFAULT SETTINGS.	309
FIGURE D8 – THE MAIN KEYKIT WINDOW.	311
FIGURE D9 – THE KEYKIT MENU.	311
FIGURE D10 – THE BLOCKS, CHORD PALETTE, CONTROLLER AND MARKOV MAKER TOOLS.	313
FIGURE D11 – THE WELL-TEMPERED FRACTAL SCREEN.	315

List of tables

TABLE 2.1.1 – DISCRETE PROBABILITY TABLE, P , BASED ON THE EXPONENTIAL DISTRIBUTION, $P(x) = e^{-x}$, FOR $x \geq 0$.	16
TABLE 2.2.1 – GLOSSARY OF CONWAY OBJECTS.	35
TABLE 2.4.1 – TYPICAL PARAMETER DEVELOPMENT STEP FOR A GA.	58
TABLE 2.8.1 – SUMMARY OF THE SIX MAIN ALGORITHM TYPES.	71
TABLE 3.1.1 – SOME COMMON 3-NOTE CHORD TYPES IN DUPE NOTATION.	77
TABLE 3.1.2 – THE AND CODEWORDS AND THEIR RESPECTIVE NUMERICAL VALUES.	81
TABLE 3.2.1 – CELL ORDERING FOR THE CONFIGURATION OF FIGURE 3.2.18.	103
TABLE 3.2.2 – AND CODE PARAMETER VALUES.	104
TABLE 3.2.3 – NOTE PARAMETERS FOR EACH NOTE OF THE CHORD GENERATED BY LIVE CELL (4, 3) ..	105
TABLE 3.3.1 – TRIGGER AND DURATION CODEWORDS FOR EACH OF THE FOUR POSSIBLE CELL PAIR CONFIGURATIONS.	110
TABLE 3.3.2 – SEQUENCE OF LIVE CELLS GENERATED BY A GLIDER.	112
TABLE 3.3.3 – ONE POSSIBLE CHORDING OF THE GLIDER STARTING CONFIGURATION.	114
TABLE 3.3.4 – SEQUENCE OF LIVE CELLS GENERATED BY THE BROKEN CROSS.	115
TABLE 3.3.5 – SEQUENCE OF LIVE CELLS GENERATED BY THE THREE-QUARTER CROSS.	118
TABLE 3.3.6 – ONE POSSIBLE CHORDING OF THE THREE-QUARTER CROSS STARTING CONFIGURATION.	120
TABLE 3.3.7 – SEQUENCE OF LIVE CELLS GENERATED BY THE THREE-QUARTER CROSS.	125
TABLE 4.1.1 – A TYPICAL PARAMETER DEVELOPMENT STEP.	147
TABLE 5.1.1 – COMPARISON OF THE NUMBER OF NEIGHBOURS AND THE STATUS FOR LIVE CONWAY CELLS AND 3-DIMENSIONAL LIFE (4, 5, 5, 5) CELLS.	153
TABLE 5.1.2 – COMPARISON OF THE NUMBER OF NEIGHBOURS AND THE STATUS FOR DEAD CONWAY CELLS AND 3-DIMENSIONAL LIFE (4, 5, 5, 5) CELLS.	154
TABLE 5.1.3 – COMPARISON OF THE NUMBER OF NEIGHBOURS AND THE STATUS FOR LIVE CONWAY CELLS AND 3-DIMENSIONAL LIFE (5, 7, 6, 6) CELLS.	156
TABLE 5.1.4 – COMPARISON OF THE NUMBER OF NEIGHBOURS AND THE STATUS FOR DEAD CONWAY CELLS AND 3-DIMENSIONAL LIFE (5, 7, 6, 6) CELLS.	157
TABLE 5.4.1 – DESCRIPTION AND FUNCTIONALITY OF THE CAMUS 3D TOOLBAR ICONS.	204
TABLE 5.5.1 – COMPARING THE CAMUS AND CAMUS 3D ALGORITHMS.	222
TABLE 6.1.1 – THE TWELVE BASIC INTERVAL CLASSES.	228
TABLE 6.1.2 – RANK FOR EACH OF THE TWELVE SIMPLE INTERVALS.	235
TABLE 6.1.3 – NUMBER OF VOTES CAST IN EACH CATEGORY FOR THE TWELVE SIMPLE CHORDS.	236
TABLE 6.1.4 – A TYPICAL ORDER-OF-PREFERENCE FUNCTION.	240
TABLE 6.1.5 – COMPONENT INTERVALS OF THE MAJOR TRIAD WITH OCTAVE.	250
TABLE 6.1.6 – COMPONENT INTERVALS OF THE FOUR CHORD TYPES LISTED ABOVE.	250
TABLE A1 – LIST OF ITEMS OF SOFTWARE AND THEIR INSTALLATION FILES.	272
TABLE A2 – AUDIO TRACK LIST.	273

TABLE C1 – SOME COMMON CHANNEL MESSAGES.....	280
TABLE C2 – SOME COMMON SYSTEM MESSAGES	281
TABLE C3 – EXAMPLE OF A MIDI HEADER BLOCK.....	285
TABLE C4 – EXAMPLE OF A MIDI TRACK BLOCK.....	286
TABLE C5 – THE GM SOUND SET.....	296
TABLE C6 – THE GM PERCUSSION SET.....	297
TABLE D1 – THE MAPPING FROM BINARY NUMBERS TO PITCH USED BY MUSINUM.....	305
TABLE D2 – EXAMPLES OF THE MAPPING EMPLOYED BY MUSINUM.....	306

List of algorithms

ALGORITHM 2.1.1 - UNIFORM PROBABILITY SELECTION ROUTINE.	21
ALGORITHM 2.1.2 - NON-UNIFORM PROBABILITY SELECTION ROUTINE.	22
ALGORITHM 2.1.3 - ROUTINE FOR ASSIGNING PROBABILITIES ACCORDING TO A LINEAR DISTRIBUTION.	23
ALGORITHM 2.1.4 - PROBABILITY NORMALISATION ROUTINE.	24
ALGORITHM 2.1.5 - SIMPLE MARKOV COMPOSITION ROUTINE.	28
ALGORITHM 2.3.1 - SIMPLISTIC FRACTAL MUSIC GENERATOR.	49
ALGORITHM 2.5.1 - SERIAL COMPOSITION ALGORITHM.	62
ALGORITHM 2.5.2 - A SIMPLE TONE ROW GENERATOR.	63
ALGORITHM 2.5.3 - ALGORITHM FOR TRANSPOSING A TONE ROW.	63
ALGORITHM 2.5.4 - ALGORITHM FOR CALCULATING A RETROGRADE TONE ROW.	64
ALGORITHM 2.5.5 - ALGORITHM FOR INVERTING A TONE ROW.	64
ALGORITHM 4.1.1 - STANDARD CELL CHECKING ROUTINE FOR THE CAMUS ALGORITHM.	141
ALGORITHM 4.1.2 - PARALLEL CELL CHECKING FOR A NEIGHBOURHOOD OF SIZE 1.	141
ALGORITHM 5.2.1 - INVERT CELL.	169
ALGORITHM 5.2.2 - THE 3-DIMENSIONAL LINE ALGORITHM.	171
ALGORITHM 5.2.3 - THE CUBE ALGORITHM.	172
ALGORITHM 5.2.4 - THE RANDOMISE GAME OF LIFE CELL ALGORITHM.	173
ALGORITHM 5.2.5 - THE RANDOMISE DEMON CYCLIC SPACE CELL ALGORITHM.	173
ALGORITHM 5.2.6 - THE NOTE GENERATION ALGORITHM EMPLOYED BY CAMUS 3D.	182
ALGORITHM 5.2.7 - THE NOTE GENERATION ALGORITHM EMPLOYED BY CAMUS 3D.	185
ALGORITHM 5.2.8 - THE OCTAVE GENERATION ALGORITHM EMPLOYED BY CAMUS 3D.	186

Acknowledgements

I would like to extend my heartfelt thanks to a number of people and organisations, without whom, this research would not have been possible.

Firstly, I express my gratitude to *the Carnegie Trust for the Universities in Scotland*, whose generous funding provided equipment and a comfortable standard of living throughout my research tenure. Their continued support undoubtedly helped to keep me going during those times when progress was slow.

Thanks must also go to the *Departments of Mathematics and Music* in the *University of Glasgow*, and to the University itself for the resources necessary to complete this research. In particular, thanks must go to the secretaries of both departments for providing invaluable help with administration, and to *Dr. Marjorie Rycroft*, the Head of Music for dealing promptly and effectively with my occasional queries. Also, to *Professors Brown and Fearn*, successive Heads of Mathematics, I offer thanks for finding the funds to send me to conferences, both overseas and at home.

I am thoroughly indebted to my supervisors, *Doctors Eduardo Miranda and Stuart Hoggar*, who, from the first, provided me with the support and guidance necessary to complete the project. I am particularly grateful to Dr. Miranda for continuing to offer advice after leaving academia to pursue a career in industry, and also for providing me with a platform on which to base my research. Special thanks must also be extended to Dr. Hoggar, who has regularly worked beyond the call of duty to ensure my welfare. He has, on occasions, fed me, reassured me, censured me where necessary and displayed unabated enthusiasm throughout this project. He has also taken the time to proof-read this document several times, and for his ceaseless vigilance with regard to grammatical and typographical errors I am also extremely grateful.

My thanks must also go to the sixteen people who participated in my harmonic preference survey. Their willingness to spare a few minutes of their time helped a great deal.

I am also extremely grateful to all those who contributed to the accompanying compact disc. They are *Chris Sansom*, who very generously allowed me to use a number of algorithmically-generated compositions to provide contrast to my own, and who entered into continued correspondence with me; *Joe Wright*, who collaborated with Dr. Miranda on the coding of *Chaosynth* for Windows, and who granted permission for me to provide the full commercial package on CD; *Jason H. Moore*, who very kindly granted me permission to include his software, *GAMusic*; *Lars Kindermann* and *Robert Greenhouse*, who, although they were not contactable, released their composition systems, *MusiNum* and *The Well-Tempered Fractal* to the public domain, thus allowing for their inclusion on the CD; *Campbell McLean* from Largs in Ayrshire, who provided me with several constructive comments and a short composition, and, of course, *Dr. Eduardo Miranda*, whose compositions, *Jazz 3*, and *Entre l'Absurde et le Mystère* demonstrate just what is possible with the system.

My thanks must also go to all of those who helped me by testing the software, but in particular to *Neil McDonald*, *Fiona McIlwraith* and *Susan Hutchison* who spent a great deal of time looking for bugs and providing comment.

Finally, I would like to extend my gratitude to all my family and friends whose support has helped keep me going. I would like to pay particular tribute to my *mother* and *father*, whose love and support throughout was unfaltering.

Thank you all.

Declaration

I declare that this thesis describes work carried out by me, except for those matters mentioned specifically in the acknowledgements. It has not been submitted in any form for another degree or professional qualification.

Kenneth B. McAlpine

Preface

Genius is one percent inspiration and ninety-nine percent perspiration

Edison

The above statement is one with which the author heartily concurs, although he is at pains to point out that his own research in no way implies genius!

However, the general sentiment of Edison's statement is certainly borne out by this work.

At the core of this research lies a small group of very simple ideas. However, these ideas, despite their simplicity, have generated a multitude of non-trivial problems, the solutions to which have required a great deal of thought, practical consideration and complex software engineering.

It should perhaps be pointed out that simplicity of ideas is not necessarily a bad thing. On the contrary, since this research is aimed primarily at musicians who have an interest in technology but who may not have an in-depth knowledge of mathematics, simplicity of ideas is a positive asset. The author believes that if the reader looks beyond this, he or she will recognise the ninety-nine percent that has been duly perspired.

In addition to the hurdles thrown up in developing the system, the author has also had practical problems to overcome.

For example, the author taught himself to program competently in C++ alongside developing his own knowledge of music technology and working on the algorithmic techniques described herein. Thus, the computer code used to test the techniques is certainly not as neat nor as efficient as it could be.

Also, because the author was implementing these developments on his own, a large amount of time which could otherwise have been devoted to system development was used to code and test the algorithmic composition software described in this thesis.

It is unfortunate that time constraints have dictated that several promising developmental ideas could not be implemented. However, the author hopes to continue this research after his research tenure at the University of Glasgow has ended.

Finally, the author would like to point out this research is extremely practical in nature and the software described herein is an integral part of it.

This thesis should not be judged in isolation but in conjunction with the music software provided on the accompanying CD. It is strongly recommended that the reader spends a little time familiarising his or herself with both versions of CAMUS so that he or she might better appreciate the problems and solutions detailed in these pages.

Also, the measure of the success of the project must lie, at least in part, with the compositions it has inspired. The CD also contains a number of compositions created with the help of CAMUS.

In order to aid the reader in forming a judgement and so that like compositions may be compared, a number of CAMUS-inspired compositions are presented alongside compositions created using other algorithmic composition systems. It is important that the reader's lays aside his or her own musical tastes for the moment in order to appreciate the compositions on their own merit. We believe that in so doing, he or she will recognise the merits of the system and will agree that CAMUS can truly be used to create music.

Abstract

Mathematics and music have long enjoyed a close working relationship: mathematicians have frequently taken an interest in the organisational principles used in music, while musicians often utilise mathematical formalisms and structures in their works. This relationship has thrived in recent years, particularly since the advent of the computer, which has allowed mathematicians and musicians alike to explore the creative aspects of various mathematical structures quickly and easily.

One class of mathematical structure that is of particular interest to the technologically-minded musician is the class of dynamical systems – those that change some feature with time. This class includes fractal zooms, evolutionary computing techniques and cellular automata, each of which holds some potential as the basis of a composition algorithm.

The studies that comprise this thesis were undertaken in order to further examine the relationship between mathematics and music. In particular we explore the notion that music can essentially be thought of as a type of pattern propagation: we begin with initial themes and motifs – the musical patterns – which, during the course of the composition, are subjected to certain transformations and developments according to the rules dictated by the composer or the musical form. This is exactly analogous to the process which occurs within a cellular automaton: initial configurations of cells are transformed and developed according to a set of evolution rules.

We begin our study by describing the development of the CAMUS v2.0 composition software, which was based on an earlier system by Dr. Eduardo Miranda, and discuss how best to use the system to compose new musical works.

The next step in our study is concerned with highlighting the limitations of CAMUS as it currently stands, and suggesting techniques for improving the capabilities of the system.

We then chart the development of CAMUS 3D. At each stage we justify the changes made to the system using both aesthetic and technical arguments. We also provide a

composition example, which illustrates not only the changes in operation, but also in interface. The system is then re-evaluated, and further developments are suggested.

1. Algorithmic composition – a definition

1.0 Introduction

The worlds of art and science seem, at first glance, to be poles apart. For example, few people would associate particle physics or mathematics with visual art, and yet both scientific disciplines have spawned whole series of works – one only has to look at the poster prints of particle trails in cloud chambers, or of fractal spirals that are available in any high street art store for evidence of this.

Art too has made inroads to the scientific community, with image processing ([Schalkoff, 1989]), physical modelling of acoustic instruments ([Sullivan, 1990]) and even automated painting ([Holtzman, 1994]) attracting considerable research attention.

Musicians, perhaps more so than any other group of artists, have always been quick to embrace technology in all its forms. From early attempts at synthesis with the Telharmonium ([Roads, 1996]) to the latest digital audio workstations (see, for example, [Lehrman, 1997], [Jones, 1996]), musicians have looked to science to provide them with new and challenging ways of working. Indeed, in many instances, new technology has provided us with sounds ([Roads, 1988]) and even compositions ([Xenakis, 1971]) which, to all intents and purposes, would have been impossible otherwise – surely a case of ‘science begat art’.

One outcome of the acceptance of scientific techniques by the musical community has been the flourishing of algorithmic composition techniques.

Algorithmic composition is by no means new, but the advent of affordable and powerful computing resources has meant that more and more people, armed with no more than a little programming knowledge and some musical ideas, have been able to realise their composition algorithms from the comfort of their own desktops.

In this section, we introduce the term *algorithmic composition*. We define algorithmic processes in the strict sense, then loosen the definition slightly to allow the inclusion

in our discussion of systems that are generally referred to by musicians as algorithmic, even though they do not necessarily meet the chosen criteria.

We must also state what we mean by the act of musical composition. This is not as easy as it first seems, since the working techniques of composers have changed considerably over the years. Today, a composer is as likely to be found performing sonic manipulations behind the controls of a digital editing suite as at a desk writing individual instrument lines on manuscript.

The compositions themselves have also changed a great deal as technology has developed. New composition tools have given composers greater sonic palettes with which to work and more complex tools to transform the sounds. This has resulted in a new type of composition that explores sound itself (see [Sutherland, 1994]).

1.1 What is algorithmic composition?

Before we can appreciate fully the development of algorithmic composition, we must first form a clear notion of what the phrase actually means. The obvious answer is, of course, that algorithmic composition is the technique of applying algorithmic processes to compose musical works. This, however, begs the question: What is an algorithmic process?

1.1.1 Algorithmic processes

A simplistic notion, though one which, for most practical purposes is perfectly adequate, is that an algorithm is a collection of rules or a sequence of operations for accomplishing some task or solving some problem ([Loy, 1989]). With this in mind, it is apparent that algorithmic composition is by no means new – composers have rigidly adhered to formal rule systems almost since the dawn of music itself ([Roads, 1996]). Indeed, formal techniques for composing melodies to texts date back to at least 1026, when Guido d'Arezzo proposed a scheme that assigned different pitches to vowel sounds in liturgical texts ([Loy, 1989]).

Another famous historical example of rule-based composition is the game of *Musikalisches Würfelspiel*, or *musical dice* (see Figure 1.1.1), which was played, by amongst others, W. A. Mozart, J. P. Kirnberger and F. J. Haydn ([Cope, 1991]).

To indulge in the game of musical dice, the composer writes a number of pieces of music, designed to slot together to form a complete work. The game proceeds by rolling dice, the outcome of which is used to select one of the pieces, which is then positioned in the score. A second roll of the dice is then used to select a piece to follow the first. The process continues until the component pieces are exhausted.

The results of the musical dice game will depend, of course, on the quality of the component pieces and how well they interact with one another. Indeed, part of the reason for playing this game was to demonstrate the composer's skill in writing musical passages that would work well together irrespective of the final composition's combinatorial makeup.

As we have seen, many traditional composers have been utilising rule-based composition without being considered as using algorithmic techniques. Indeed, during the Classical period, rigid adherence to form was all, and composers had to wrestle with countless formal constraints when developing a work. It was not until Beethoven arrived to challenge this view and usher in the Romantic period that a more individualistic style of composition emerged ([Abraham, 1982], [Abraham, 1990]).

However, despite his fondness for parlour games and the rigidity of the formal constraints to which he worked, one would not generally consider either the works or the working techniques of Mozart as epitomising algorithmic composition.

Clearly, then, by an algorithmic process, we mean a process that exhibits more than simply rule-following.

Zahlentafel

1. Walzerteil

	<i>I</i>	<i>II</i>	<i>III</i>	<i>IV</i>	<i>V</i>	<i>VI</i>	<i>VII</i>	<i>VIII</i>
2	98	22	141	41	105	122	11	30
3	32	6	128	63	146	46	134	81
4	69	95	158	13	153	55	110	24
5	40	17	113	85	161	2	159	100
6	148	74	163	45	80	97	36	107
7	104	157	27	167	154	68	118	91
8	152	60	171	53	99	133	21	127
9	119	84	114	50	140	86	169	94
10	98	142	42	156	75	129	62	123
11	3	87	165	61	135	47	147	33
12	54	130	10	103	28	37	106	5

2. Walzerteil

	<i>I</i>	<i>II</i>	<i>III</i>	<i>IV</i>	<i>V</i>	<i>VI</i>	<i>VII</i>	<i>VIII</i>
2	70	121	26	9	112	49	109	14
3	117	39	126	56	174	18	116	83
4	66	139	15	132	73	58	145	79
5	90	176	7	34	67	160	52	170
6	25	143	64	125	76	136	1	93
7	138	71	150	29	101	162	23	151
8	16	155	57	175	43	168	89	172
9	120	88	48	166	51	115	72	111
10	65	77	19	82	137	38	149	8
11	102	4	31	164	144	59	173	78
12	35	20	108	92	12	124	44	131

Figure 1.1.1 – Two numeric tables from a musical dice game by Mozart. The columns numbered I – VIII denote the eight parts of the waltz. The rows numbered 2 – 12 indicate the possible values of two dice. The table values refer to the bar numbers of four pages of musical fragments. [Roads 1996]

Today, most, if not all, algorithmic compositions are produced by computer. The composer converts his or her algorithm into machine code and uses the computer to perform the menial composition tasks, such as ensuring that the musical phrases adhere to the composition rules, or the writing of score files, leaving the composer free to assume a more supervisory role. Thus, many algorithmic composition systems are *automated* composition systems.

Is this sufficient to constitute an algorithmic process? Unfortunately not – automated processes need not be algorithmic.

For example, the Æolian harp, a stringed instrument with strings of different thickness all tuned to the same note, was used in ancient Greece to generate automatic compositions ([Cope, 1991]). The instrument was placed outdoors, so that the wind, when it caught the strings, caused them to vibrate with various harmonics, depending on the wind speed and direction. This is not an algorithmic process, although compositional algorithms that simulate the effects of the wind on strings or wind chimes certainly do exist ([Syntrillium, 1996]).

Similarly, an algorithmic process need not be automated – all of the calculations and transformations applied by computer could, if so desired, be done manually, although it may take a considerable time so to do ([Knuth, 1973]).

It has also become increasingly common to use the terms ‘algorithmic composition’ and ‘computer-aided composition’ synonymously. While it is true that algorithmic composition can be a form of computer-aided composition, there are a great many ways in which a computer can aid the composition process without resorting to composition algorithms.

For example, using a notation package to produce a score is a form of computer aided composition, as is the use of a word processor for making programme notes. It is extremely important that one says precisely what one means, so as to avoid unnecessary ambiguity.

How then do we formulate a concrete definition of an algorithm? In his book, ‘The Art of Computer Programming’ [Knuth, 1973], Donald E. Knuth gives five criteria that should be fulfilled if a process is to be considered algorithmic:

- (i) *Each process must be finite.* Thus, an algorithmic process must terminate after some finite number of steps.
- (ii) *Each step must be clearly defined.* In other words, there must be no ambiguity surrounding any of the stages. Each must give a clear instruction as to the operation that is to be performed, or of the rule that is to be applied. Consequently, a human operator should be able to perform each stage of the process manually if so desired.
- (iii) *The process may have input.* Parameters may or may not be required to solve a particular problem. For a composition algorithm input could take the form of a chord progression, a MIDI file, or live input from a musician.
- (iv) *The process must have output.* The algorithm must, when concluded, return a result. The process is guaranteed to terminate by (i). The output will depend on the nature of the algorithm, but for compositional purposes, it is likely to be a musical score, or a MIDI or audio file.
- (v) *The process must be effective.* By effective, we mean that the process must solve the problem in a “sufficiently basic” manner, by which we mean that the process is, in some sense, irreducible, and cannot be broken down into further sub-processes.

With these criteria, it is easy to define reasonably closely what we mean by an algorithmic process – it is simply any process that satisfies (i) – (v) above.

1.1.2 Musical ‘algorithms’

We now relax the definition of algorithm slightly, to allow for processes that do not strictly adhere to the five criteria presented above.

For example, although the game of musical dice is generally considered by musicians to be algorithmic, it can be argued that the use of dice to control the development of the composition contradicts the second of our five criteria – *each step must be clearly defined*. The non-determinism of the dice roll means that the process is not clearly defined, since it depends on the outcome of a random event.

It can, however, also be argued that the outcome of the dice roll merely provides input to the system, and so it is truly algorithmic.

In any case, as was noted at the beginning of Section 1.2.2, a reasonable rule-of-thumb guide is that an algorithm is a collection of rules or a sequence of operations for accomplishing some task or solving some problem, and this is exactly what the game of musical dice is. The problem here is the construction of a complete piece of music from a number of musical fragments. The game provides us with a simple set of rules that incorporate an element of chance.

For the remainder of our discussion, then, we will use the term *algorithm* to refer to any finite formal system of rules that can be applied to solve the problem at hand.

1.1.3 Musical composition

We must also indicate broadly what we mean by a musical composition. For example, many modern compositions are concerned more with tonal structure and development than with themes and melodic motifs (see, for example, [Sutherland, 1994] for a brief discussion of *musique concrète*). The instruments of such compositions are often sound snapshots recorded from the real world or completely synthetic tones, while the development may consist of the splicing together of tapes or the application of digital signal processors to sampled sounds.

In compositions such as these, the sounds that constitute the composition are often more important than the notes that are played, if, indeed we can speak in such terms. Extrapolating this idea, it becomes apparent that programming languages, such as *CSound* or *Pascal* with suitable add-ons, can be considered as algorithmic composition systems, since we may use these languages to implement algorithms for creating and manipulating sound samples on computer.

We must ask ourselves *at what point do we draw the line?*

In the discussion that follows, we lean towards the more traditional notions of musical composition – those works that are composed of melodic and harmonic instrument lines that are developed as the composition progresses.

Whilst we do not focus on electroacoustic composition and musique concrète, nor do we exclude them from consideration. A good example is Eduardo Miranda's *Chaosynth* system (see Appendix E), which uses cellular automata to generate *granular sounds*¹.

Although this system cannot be said to produce complete compositions, the sounds it creates can be used to form the basis of an electroacoustic composition. It is also possible to use the system as a simple melody generator by altering the granular synthesis parameters.

1.2 Composing algorithmically

Now, having examined what constitutes an algorithmic process, we must examine how they can be utilised in a musical way.

We begin by considering the processes behind the construction of a new musical composition.

1.2.1 A step-by-step guide to making music

The construction of a new musical work, whether algorithmically or manually, can be broken down into three stages:

- (i) *Generating musical ideas.*
- (ii) *Creating a rough sketch of the composition.*
- (iii) *Creating the final composition.*

Steps (i) and (ii) above are the most likely candidates for the use of an algorithmic process. For example, at step (i) an algorithmic composition system could be used to generate a number of short melodic lines or rhythmic figures, which the composer would select and subject to steps (ii) and (iii) according to his or her own aesthetic judgement.

¹ For a full discussion on granular synthesis see [Roads, 1978], [Roads, 1988], [Roads 1996], [Jones & Parks, 1988] and [Truax, 1988].

Alternatively, a composer may already have a very specific idea as to the form of a composition. An algorithm could then be specified and used to generate a rough sketch of a work, which could then be fine-tuned by the composer.

The amount of work needed to produce the final composition – step (iii) of the composition process – will depend on how closely the sketch conforms to the composer's ideal. It may range from none at all to a complete rewrite, although in practice, it is usually somewhere between these two extremes.

In the subsequent discussion of algorithmic composition systems in general and of CAMUS in particular we provide several examples of how various algorithmic composers can be used in stages (i) and (ii) of the composition process. We also give a particular example of a composition system that produces complete musical works from scratch. That is, we present an algorithmic composition system that can effectively realise stages (i) – (iii) of the composition process.

1.2.2 Algorithmic composition as a model for inspiration

Inspiration is a phenomenon that is as difficult to define as it is to obtain. It is far from being understood let alone effectively formalised.

Part of the problem may lie with the fact that inspiration is a highly subjective phenomenon. Another source of difficulty surely lies with the fact that inspiration exhibits itself in such a wide variety of forms; from the germ of an idea which after successive refinements becomes a complete work, to a moment of clarity when entire symphonies suddenly unfold before the recipient's ears.

For example, W. A. Mozart is quoted in *the Life of Mozart Including his Correspondence* ([Holmes, 1878]) as writing of inspiration and his method of working:

“All this fires my soul, and provided I am not disturbed, my subject enlarges itself, becomes methodised and defined, and the whole, though it be long, stands almost complete and finished in my mind, so that I can survey it like a picture or a beautiful statue at a glance. Nor do I hear in my imagination the parts successively, but I hear them, as it were, all at once. What a delight this is I cannot tell.”

Algorithmic composition systems are often utilised as ‘idea machines’. In other words, they provide moments of musical inspiration that the composer can then seize and build on. When viewed as such, it is clear that algorithmic composition systems provide us with a model for inspiration.

That is not to say, however, that algorithmic composition systems closely mimic or even come close to replicating the processes at work in the creative human brain. Rather, algorithmic composition may be viewed as an artificial inspiration model in much the same way as computers that exhibit artificial intelligence can be said to model intelligent beings without necessarily mimicking the low level processes at work in their brains.

In spite, or perhaps because of this, algorithmic systems are often seen as being something of a cheat – a cop-out for composers who lack creativity. The author, however, considers this to be more than a little unfair. Why should composers be denied a source of inspiration because it comes from a machine? Many composers and other artists have been inspired by machinery (see, for example, [Lewis, 1935], [Honegger, 1994] and [Chaplin, 1936]).

Indeed, it is the author’s opinion that the creative aspects of composition are much more relevant when developing initial ideas and sketches into a complete work than in generating those ideas in the first place. Thus, an algorithmic composition system that provides a starting point for the composer’s own imagination may help to develop the composer’s natural creative instincts.

On the other hand, an algorithmic composition system that could reliably perform all three stages in the composition system would totally remove the need for human intervention. This certainly has applications in the interactive entertainment industry. Computer games and virtual environments often require audio that will react to the game player or observer in order to create a truly immersive environment (see, for example [Microsoft, 1999]). Without the ability to generate such music in real time, short segments of sound must be composed beforehand and some sort of selection routine employed in order to play the correct segment at the correct time.

As well as the issue of the need for interactivity with human operators, this raises important points on the authorship of algorithmic music. We discuss this in Section 7.

1.3 A general outline of the problem

And so we come to define the problem with which this research is concerned. Below we present the main driving force behind our work on algorithmic composition and conclude by suggesting the direction we will take.

1.3.1 Music as pattern propagation

Initially, this research project was concerned with dynamical systems and their applications to music composition. It was the ultimate aim of the project to explore how many different types of dynamical system, such as cellular automata, fractals and neural networks could be utilised in the composition process.

However, as the project developed, it was found that the particular case of cellular automata provided us with a rich source of material. With each new development came new problems and challenges to be overcome.

This had the effect of shifting the emphasis of the research from:

How can we make music with many different dynamical systems?

to:

How can we develop and improve this particular cellular automata system to produce interesting and natural sounding music?

In effect, the research has focused on one particular aspect of algorithmic music composition and explored its possibilities as a viable source of new musical material.

The research focuses on modelling the composition process as a form of pattern propagation – each theme in a composition may be viewed as a separate pattern. As the composition progresses, the patterns are subjected to certain transformations (such as straight repetition, transposition, inversion, augmentation and so on) according to the formal structure that the composer has chosen for the work. This structure can be rigidly adhered to or used as a general guiding principle, but so long as certain design constructs are in place to guide the temporal development of the composition, we can

say that we have a system of pattern propagation according to some predetermined constraints.

Traditionally, composers have intuitively employed pattern propagation when composing, but algorithmic composition techniques, such as those described in the following chapters, allow the pattern propagation to be formalised, albeit at a much higher level. Here, the composer does not, in general, apply specific transformations to a particular pattern. Instead, all of the musical patterns evolve according to the rules and constraints that have been specified at the design stage.

We shall see that any common stylistic musical features that emerge from the use of cellular automata as music generators can be said to be a sonification of the *emergent behaviour* of the automaton – the temporal development arises as a result of the local evolution rules and is not fully specified in advance.

1.3.2 Research goals

In the proposal for this research project the following goals were set:

- i.) Produce new algorithms for mapping mathematical structures to music.
- ii.) Gain knowledge as to what kinds of mathematical structures and mappings produce useful results in sound.
- iii.) Produce new tools for music composition.
- iv.) Produce new compositions.

We shall discuss whether or not these goals have been met in Chapter 7.

2. A brief History of Algorithmic Composition

2.0 Introduction

In this section we present a brief outline of the history of algorithmic composition systems. We then describe seven main algorithm types used in composition. These are *stochastic algorithms*; *formal grammars and automata*; *iterative algorithms*; *fractal algorithms*; *evolutionary algorithms*; *serial algorithms* and *rule-base algorithms*. We focus on those algorithm classes that have the most significance to our research and conclude by proposing a new method of grouping together algorithms that generate and perform music in a specific manner.

In this, and subsequent chapters, we provide the reader with short sections of pseudo-code under the headings *Algorithm x*. These ‘algorithms’ are written midway between computer code and English and are intended to complement the main body of the text.

2.1 Stochastic Algorithms

2.1.1 What are stochastic algorithms?

Of all the compositional algorithms, stochastic algorithms are undoubtedly the easiest, both to comprehend and to implement.

A stochastic algorithm is one that is intrinsically dependent on the laws of probability, making it impossible to predict the precise outcome of the process at any point in the future ([Roads, 1996]). Stochastic processes may be natural, such as the decay of radioactive isotopes, or artificial (that is, man-made), as is the case for certain types of dithering employed by many computer graphics and sound synthesis packages (see, for example, [Hearn & Baker, 1994] and [Roads, 1996]).

It is partly due to the accessibility of stochastic algorithms that they have become so extensively employed for compositional purposes. Most, if not all, composers engaged in algorithmic composition have used stochastic processes in one form or another, whether as the basis for an entire composition algorithm or as an incidental decision-making routine.

2.1.2 Probability Lookup Tables

As mentioned above, the outcome of a stochastic process depends wholly on certain underlying *probability distributions*, a term used to describe the way that the probabilities are divided amongst each of the possible outcomes.

The simplest, and most widely utilised method of modelling and implementing these distributions is that of *probability lookup tables*. These are discrete, finite tables of values that correspond to the likelihood of occurrence of one or more events. The table entries range in value from 0, which indicates that the corresponding event will never occur, to 1, which indicates that the event will occur with absolute certainty². A value of 0.25, for example, would equate to a one-in-four chance of occurrence in common parlance.

If the sum of all the entries in the probability table is equal to 1, then we say that the table is *normalised*. This is a very desirable state of affairs, because it implies that when a decision-making routine is called upon, we can say with absolute certainty that there will be some output, since the probability of one of the outcomes being chosen (that is, the sum of the probabilities of the individual outcomes) is, by definition, equal to 1.

The events will depend on the application of the probability table, but for compositional purposes, they are likely to be note events or pre-recorded sequences of notes.

We mentioned above that probability tables are discrete and finite, and depend on some underlying probability distribution, after which, by convention, we name the table. However, a probability distribution may be continuous, and some care must be taken when using such a distribution as the basis for a discrete table of values.

²In fact this definition is not strictly correct. It is possible for a single outcome to have probability 0 and still occur. However, for the (discrete) stochastic selection routines given below, it is certainly the case that a probability of 0 indicates that an event will never take place. It is for this reason we have presented the definition as above. For a fuller discussion on the nature of probability, the interested reader is referred to, for example, [Murphy, 1991].

For a general continuous distribution, p , the probability that x lies between the values a and b is defined as:

$$\int_a^b p(x)dx.$$

Thus, the probability that $x = a$ is

$$\int_a^a p(x)dx = 0.$$

We can see that in order to specify the probabilities of a discrete distribution, we cannot simply assign the probabilities of the corresponding values of the relevant continuous distribution since these are 0. Instead, we assign the values of the distribution itself. This, however, leads to a further problem.

Take, for example, the exponential probability distribution, given by:

$$p(x) = e^{-x},$$

for $x \geq 0$.

Clearly, this distribution is normalised, since

$$\int_0^{\infty} e^{-x} = 1.$$

Suppose we now wish to construct the discrete probability table, P , using the above distribution for the three values, 0, 0.5 and 1. We may do this by letting $P(x) = p(x)$ for $x = 0, 0.5, 1$ as illustrated in Table 2.1.1 below:

x	0	0.5	1
$P(x)$	1	$e^{-0.5}$	e^{-1}

Table 2.1.1 – Discrete probability table, P , based on the exponential distribution, $p(x) = e^{-x}$, for $x \geq 0$.

However, if we now examine the probability values of P , we see that the sum is

$$1 + e^{-0.5} + e^{-1} > 1.$$

Thus, P is not normalised. The reason for this is that on discretisation we introduce a sampling error because the area under the curve between 0 and 0.5 is not be the same as the sum of the function values at those points. This is illustrated in Figure 2.1.1 below.

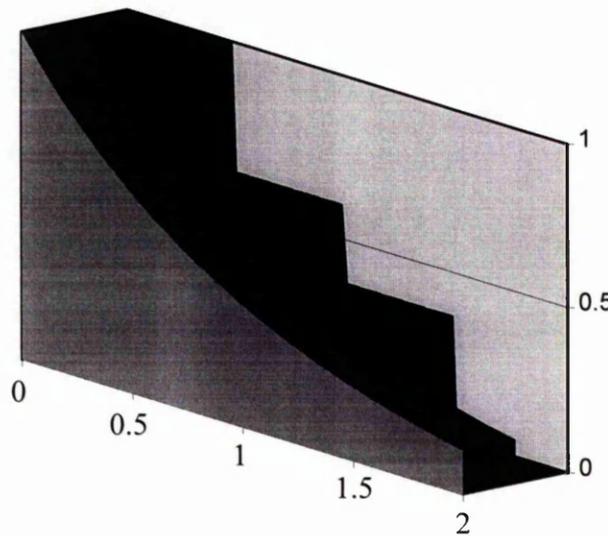


Figure 2.1.1 – Care must be taken when creating a discrete probability table from a continuous distribution. The lighter region in the foreground represents part of the probability distribution given by $p(x) = e^{-x}$ for $x \geq 0$. The darker region in the background represents the discrete probability table for $x = 0, 0.5, 1$ created using this distribution. The area under the curve $p(x)$ does not equal the sum of the three discrete probabilities, leading to a sampling error.

In order to compensate for this, we scale the probabilities in the discrete table so that it is normalised. A normalised discrete table is presented in Figure 2.1.2. Here, the continuous distribution is in the background of the graph.

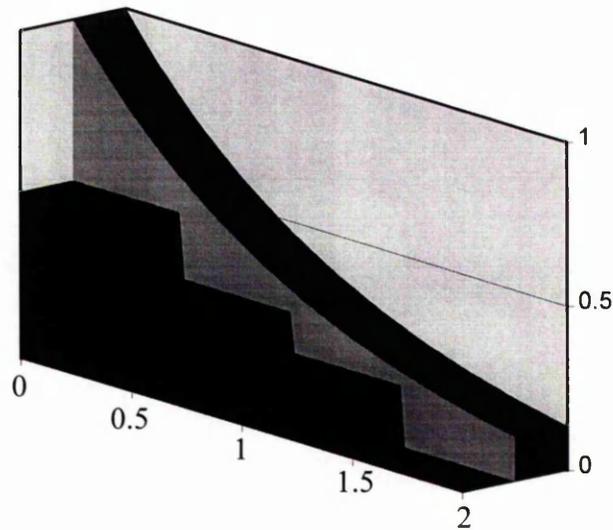


Figure 2.1.2 – The normalised exponential distribution on the points $x = 0, 0.5, 1$.

The simplest type of probability distribution is the *uniform distribution* (see Figure 2.1.3). Here, each possible outcome is assigned an equal probability of success. Since a normalised table requires the sum of the probabilities of the possible outcomes to be equal to 1, this means that if we have N equiprobable outcomes, the probability of each will be $1/N$.

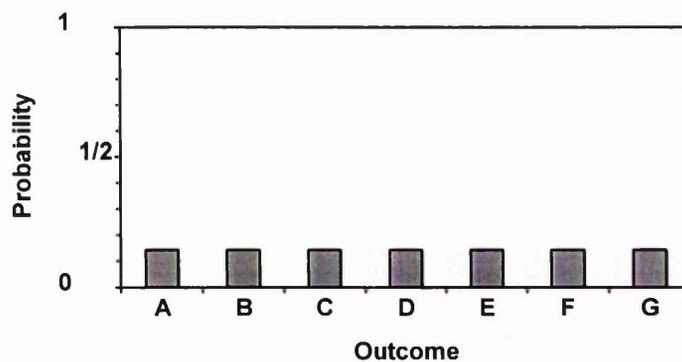


Figure 2.1.3 – Uniform probability distribution.

The uniform distribution is used, for example, when we wish to select an equally weighted 'yes' or 'no' answer – this corresponds exactly to a uniform distribution between two events. However, it is likely that a composer who is using probability tables for note selection would wish to use distributions that favour certain outcomes over others. There are many alternative distributions that can be used.

Linear distributions (see Figure 2.1.4) give a line of fixed, increasing or decreasing probability between two limits. Clearly, the uniform distribution is a special case of the linear distribution in which the line of probability has zero gradient and the limit points are equal.

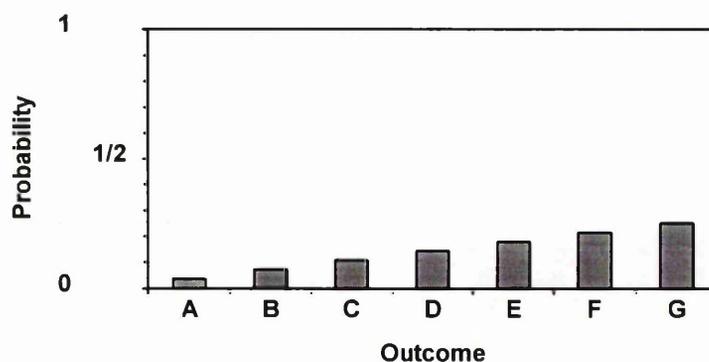


Figure 2.1.4 – Linear probability distribution.

The linear distribution can also be extended to the *piecewise-linear distribution* (see Figure 2.1.5). Here, the distribution can be divided into a number of distinct regions, each of which is linear. Note that the distribution may or may not also be piecewise-continuous (that is, the endpoint of the i^{th} region is the startpoint of the $i + 1^{th}$ region). However, since the distributions described here are all discrete, we can consider each of the data points as being the end or startpoint of the implied linear segment that lies between it and the next data point. Thus, we may consider *any* discrete probability distribution as being an extreme case of a piecewise-linear distribution.

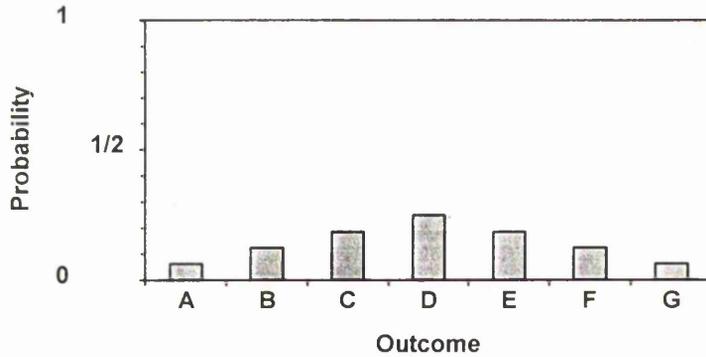


Figure 2.1.5 – Piecewise-linear probability distribution.

Exponential distributions (see Figure 2.1.6) give an exponential curve between two endpoints.

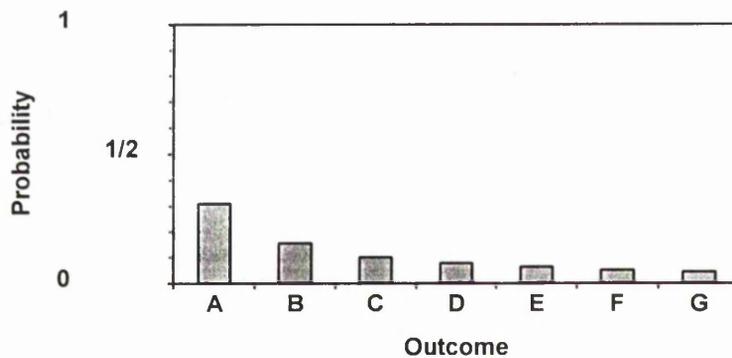


Figure 2.1.6 – Exponential probability distribution.

As with the linear distribution, the exponential distribution can be used to construct a piecewise-exponential distribution with some number of distinct regions, each of which is an exponential distribution.

Bell-shaped distributions (see Figure 2.1.7) are one of the most common naturally-occurring distributions. Traditionally, the term bell-shaped was used to describe the normal distribution. Here, however, we use the term to denote any distribution which exhibits a distinctive n-shaped curve. Bell-shaped distributions may or may not be symmetrical and have their minima at the endpoints.

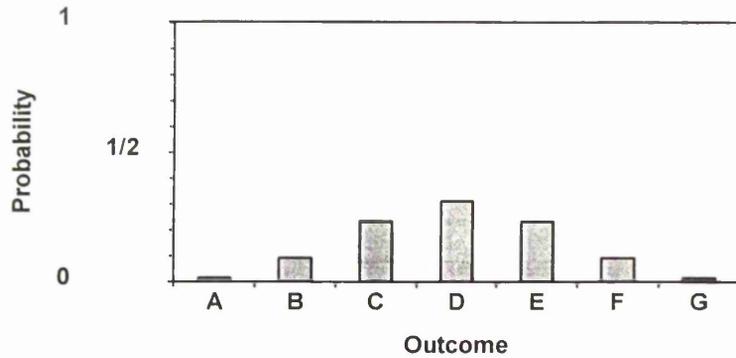


Figure 2.1.7 – Bell-shaped probability distribution.

U-shaped distributions (see Figure 2.1.8) are essentially upturned bell-shaped distributions. They peak at the endpoints, and need not be symmetrical. The arcsine and beta distributions are U-shaped.

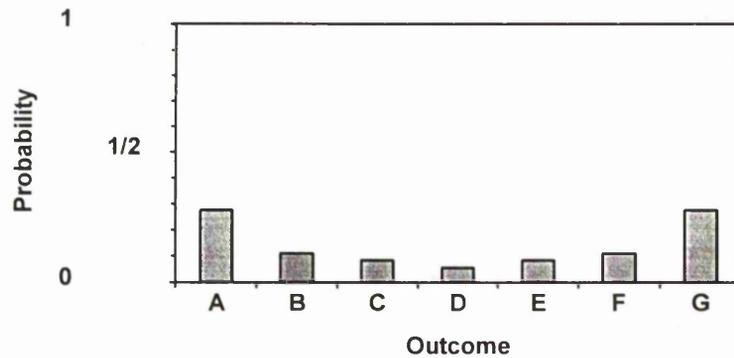


Figure 2.1.8 – U-shaped probability distribution.

2.1.3 Using probability tables

Having now seen what a probability table is, we must devise a method of choosing outcomes according to the probability values stored within. This is a relatively simple task in the case of the uniform distribution.

The task of choosing one equiprobable outcome from N is equivalent to the task of choosing an integer from $0, 1, \dots, N - 1$. Thus we have the following:

```

procedure Uniform_Distribution()
{
    select a random integer, i, between 0 and N - 1
    execute ith outcome
}

```

Algorithm 2.1.1 - Uniform probability selection routine.

This algorithm assumes that

- (i) N , the number of outcomes is known in advance.
- (ii) Each of the N outcomes has associated with it a unique procedure, *ith outcome*, for performing the corresponding operations required.

This algorithm is fairly intuitive, but how do we go about extending it to cope with non-uniform distributions? For this, we need the notion of *cumulative distributions* ([Lorrain, 1980]).

The idea behind this is as follows: Suppose we wish to choose one of N outcomes, labelled x_1, x_2, \dots, x_N . Then, since $0 \leq P(x_i) \leq 1$, and assuming that we are dealing with a normalised distribution, we have that

$$0 \leq P(x_1) \leq P(x_1) + P(x_2) \leq \dots \leq P(x_1) + P(x_2) + \dots + P(x_N) = 1.$$

Thus, we have an increasing sequence of non-negative real numbers that lie in the range $[0, 1]$. We can view this as a set of points that divide the real segment $[0, 1]$ into smaller segments whose widths correspond to the different probabilities of the individual outcomes, as shown in Figure 2.1.9. Thus, in order to select an outcome, we randomly choose a real number from $[0, 1]$ and observe into which segment it falls.

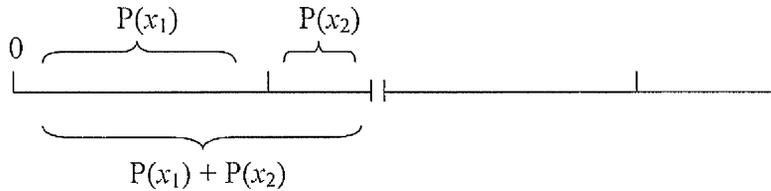


Figure 2.1.9 – Cumulative distribution method for non-uniform distributions.

Cumulative distributions can be implemented as follows:

```

void NonUniform_Distribution()
{
    Generate a random real number, x, between 0 and 1
    interval = P[0]
    i = 0
    while (x < interval)
    {
        interval = interval + P[i]
        i = i + 1
    }
    execute ith outcome
}

```

Algorithm 2.1.2 – Non-Uniform probability selection routine.

Again, we assume that N is known previously, and that each outcome has associated with it a unique procedure. We have also assumed that the probabilities are stored in a constant global array, $P[]$.

It should also be noted at this point that if a probability is defined to be 0 here, the implication is that the corresponding outcome will never take place. Thus, it is necessary to assign extremely improbable outcomes a small but finite probability.

Now that we have seen an algorithm for selecting non-equiprobable events, we need only concern ourselves with assigning probabilities to outcomes according to the appropriate probability distribution. We illustrate with the linear distribution.

Suppose we wish to assign probabilities to events so that the resulting distribution is linear between two endpoints, $(a, P(a))$ and $(b, P(b))$. This essentially reduces the problem to one of linear interpolation. This can be achieved by using the following algorithm. We begin by calculating the gradient and y-intercept of the line between the points $(a, P(a))$ and $(b, P(b))$, before cycling through each of the intermediate table entries, and loading with the corresponding point on the line.

```
procedure Linear_Distribution(a, Pa, b, Pb)
{
    gradient = (Pb - Pa)/(b - a)
    for i = a to b
        P[i] = Pa + gradient * (i - a)
    next i
    normalise probability array
}
```

Algorithm 2.1.3 – Routine for assigning probabilities according to a linear distribution.

A procedure whose task is to normalise the probability table can be specified very simply.

Recall that a normalised distribution is one in which the sum of the probabilities is equal to 1. In order to achieve this, our normalisation routine should calculate the sum of the probabilities then divide each of the probabilities in the array by this normalisation factor.

```

procedure Normalise()
{
    Normalisation_Factor = 0
    for i = 0 to N - 1
        Normalisation_Factor = Normalisation_Factor + P[i]
    next i
    for i = 0 to N - 1
        P[i] = P[i] / Normalisation_Factor
    next i
}

```

Algorithm 2.1.4 – Probability normalisation routine.

This technique can be extended to fill lookup tables with non-linear probabilities. All that is required is some method of interpolating points with respect to the appropriate distribution.

We now introduce a stochastic process that is, in effect, one complexity level higher than the lookup table, that of *Markov Chains*.

2.1.4 Markov Chains

A Markov Chain is a sequence of trials of the same experiment whose outcomes, X_1, X_2, \dots , satisfy the following ([Lipschutz, 1974]):

- i.) We associate with each trial a particular moment in discrete time, and call the outcome of the n^{th} trial the *state* of the system at timestep n .
- ii.) The outcome of any trial depends only on the outcome of the immediately preceding trial. With each pair of states, (X, Y) is associated the probability p_{XY} , which is the probability that state Y immediately follows state X . The p_{XY} are called *transition probabilities*.

Throughout this discussion we consider only the case of a finite number of states.

In fact, we can extend the above definition slightly to allow for Markov Chains that retain memory of additional past events in order to influence the outcome of future events. The number of past events that are taken into consideration at each stage is known as the *order* of the chain. Thus, a Markov Chain in the strict sense is first

order, while a chain that considers an event's predecessor and its predecessor's predecessor is second order and so on. In this thesis we concentrate on zero³ and first order Markov Chains.

In general, an N^{th} order Markov Chain can be represented by a state-transition matrix – an $N + 1$ -dimensional probability table. The state-transition matrix gives us information on the likelihood of a particular outcome given the previous N states. Figure 2.1.10 shows a possible state transition matrix for a first order Markov Chain.

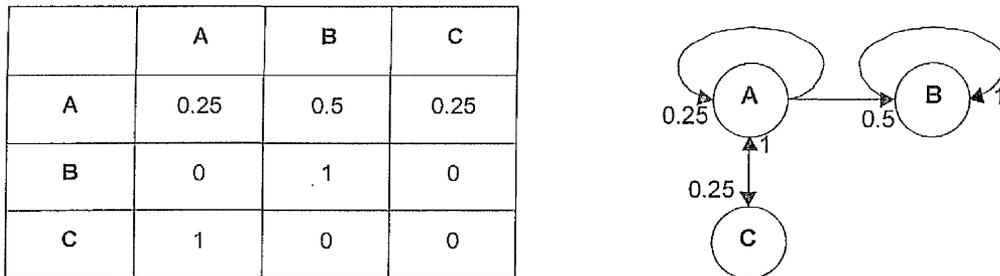


Figure 2.1.10 – State-transition matrix and graphical representation of a (first order) Markov Chain.

By convention the previous states are listed vertically, and the transition states are listed horizontally. Thus, if we wish to find, for example, the probability of state B occurring immediately after state A, we simply find state A in the first column (the one headed by a blank), and then move horizontally along to the 'B' column. The entry here is 0.5, so there is a 1 in 2 chance of this transition occurring. This can be extended to calculating the probability of an event occurring on the n^{th} transition using matrix multiplication.

For example, the state-transition matrix of Figure 2.1.10 can be expressed as a real 3×3 matrix, M , whose rows correspond to the initial states and whose columns correspond to the final states.

Suppose that we are currently in state A, and we wish to calculate the probability that we will be in state B on the next-but-one transition. This is the probability that we will move to state j on the next step and then from state j to state B the step after for all

³ A Markov Chain of zero order is one in which the outcome of each trial depends on no previous states. That is, the system is *independent* of the influence of the outcome of previous trials.

possible choices of j . That is, the element in column j of the first row of the matrix (the A row) multiplied by the element in the second column (the B column) of row j as j ranges through the states A, B, C.

More formally, we have

$$\begin{aligned} P(\text{A to B in 2 steps}) &= \sum_j M_{Aj} M_{jB}, j = A, B, C \\ &= (M^2)_{AB} \end{aligned}$$

where M_{ij} denotes the entry in row i , column j of matrix M , and M^2 denotes the matrix $M \cdot M$. With the probability values as given in Figure 2.1.10, this turns out to have value 0.625.

This is an immensely useful property of Markov Chains, since it allows us to make calculations of probabilities, which can rapidly become confusing, using the more familiar and well-understood methods of matrix multiplication. This technique can easily be extended to any number of steps – the probability of arriving in state j after n steps, given that we are currently in state i is $(M^n)_{ij}$ – and so by using computers, it is possible to quickly and reliably calculate the probabilities of particular outcomes as far into the future as we desire with a minimum of effort.

It should also be apparent that the probability lookup table technique presented in the previous section is simply a Markov Chain of order zero, that is, a probability system that depends on no previous states.

For any Markov Chain, a state, X , is said to be *reachable* from a state, Y , if it is possible to reach state Y from state X after a finite number of steps. If state Y is reachable from state X and state X is reachable from state Y , then the two states are said to *communicate*.

For example, in Figure 2.1.8 above, states A and C communicate, since clearly C is reachable from A and A is reachable from C. However, neither A and B, nor B and C communicate, since, although B is reachable from A (and thus also from C), neither state is reachable from B, which is always followed by itself.

It can be shown fairly easily that the communication relation on a Markov chain is an *equivalence*. That is, the communication relation is *reflexive*, since a state always communicates with itself; *symmetric*, since if a state X communicates with a state Y, then clearly, Y communicates with X, and *transitive*, since if a state X communicates with a state Y, and state Y communicates with a state Z, then state X also communicates with state Z.

Grouping communicating states together, we can partition the states of the chain into equivalence classes of communicating states.

Those states that are certain⁴ to occur again once they have been reached by the chain are called *recurrent* and the equivalence class they belong to is known as a *recurrent class*. States which may never occur again (i.e. those that are not recurrent) are called *transient*, and the class they belong to is known as a *transient class* (for a proof of the fact that recurrence and transience is a class property see [Freedman, 1971]).

It can further be shown (see, for example, [Freedman, 1971]) that every Markov chain consists of at least one recurrent class and some number (possibly none) of transient classes.

In Figure 2.1.8 above, these equivalence classes are

$$t = \{A, C\}$$

and

$$r = \{B\}.$$

The class r is recurrent, since the state B is always followed by itself. Thus once we arrive in r , we can never leave. The remaining class, t is transient – we may visit states A and C a number of times, but once we have reached state B, we can never return to either of the other two states.

⁴ That is, those that occur again with probability 1 as $t \rightarrow \infty$.

We now illustrate the usage of Markov Chains with an algorithm that is designed to play back one of five pre-recorded sequences of notes depending on the outcome of a first order state-transition matrix (*c.f.* [Jones, 1980]).

The algorithm begins by selecting at random one of the five sequences as a starting configuration⁵. From here, the appropriate sequence subroutine is called before a new sequence is selected, using the relevant row in the state-transition matrix. This process repeats until a user interrupt is received.

```
procedure MarkovChain()
{
    select an integer, i, between 0 and 4
    repeat
    {
        play sequence i
        select a new integer, i,
            using the probabilities in the ith row of the
                state-transition matrix
        until user interrupts
    }
}
```

Algorithm 2.1.5 – Simple Markov composition routine.

Here, we have assumed that the state-transition matrix is known *a priori* and remains constant throughout the composition.

2.1.5 Quality of Results

As mentioned earlier, stochastic algorithms are amongst the easiest to implement. This is reflected in the sheer number of systems that incorporate stochastic processes. Yet despite, or perhaps because of this, there have been very few stochastically-oriented commercial packages. Those that do offer extensive tools for stochastic composition are generally composition environments, more akin to programming languages than to one-stop music systems.

⁵ Thus the initial distribution of the Markov Chain is uniform – each of the states has the same probability of being the initial state of the system.

Pekka Tolonen's *Symbolic Composer* ([Tolonen, 1987]) and IRCAM's *Patchwork* ([IRCAM, 1993]) are two such packages. These systems are very much more than mere composition systems. They offer modules and a working environment that allows composers to explore their own compositional algorithms and mappings without having their working techniques dictated to them by the way that a programmer has implemented a complete system.

One of the main drawbacks to using composition environments as opposed to ready-made composition systems is that very much more effort must be expended by the composer to get similar results. This may take the form of learning the composition language or spending time designing new composition algorithms. For sheer flexibility, however, the effort is often more than worthwhile.

It is undoubtedly also a result of the ease of use and comprehension of stochastic algorithms that they were the first to be explored as a method of generating music. We saw in the previous chapter, for example, the parlour game of *musical dice*. The similarities between this game and the Markov Chain Algorithm 2.1.5 presented above will no doubt be immediately apparent: In musical dice, each of the n sections of the piece has an equal probability (in fact, $1/n$) of exposition at the beginning. However, once the first section has been selected, say section i , we can consider the composition as having moved into the particular state with section i at the beginning. The probability that section i will occur next is 0, since it has already been used, and each of the remaining states will occur with probability $1/n-1$. Thus, we have a probability system featuring n trials of the same experiment in which the outcome of the second trial depends solely on the outcome of the first – exactly analogous to our Markov Chain. Now, however, the analogy breaks down slightly, because the outcome of the k^{th} trial depends on the outcomes of the first $k - 1$ trials.

The first comprehensive study of stochastic musical processes was due to Iannis Xenakis. In his book, *Formalized Music* ([Xenakis, 1971]), he describes in considerable detail the stochastic processes employed by him in his compositions, and provides lengthy discourse on the aesthetic implications of using stochastics in music.

The book arrived some years after the unleashing on the world of the first wave of a new mathematical music, characterised by works such as Xenakis' *Metastasis* (1954)

and *Pithoprakta* (1956) and the *Illiatic Suite for String Quartet* (1956) by Lejaren Hiller and Leonard Isaacson, which depended on stochastic processes for the very notes from which it was composed.

This new music was very different from anything that had gone before. It was often written entirely for synthetic instruments, with many contrasting themes sounding simultaneously. Clusters of activity and long periods of silence combined with no musical clues as to where the piece was heading all contrived to make the music very difficult to listen to.

This does highlight one of the main difficulties of stochastic algorithms – the inherent randomness. Although the use of probabilities does lead to the emergence of long-term behaviour, the ‘melodies’ produced by stochastic processes often exhibit a lack of coherence, seemingly wandering aimlessly from note to note. This can be countered to some extent by employing Markov Chains of high order, or by using self-adapting probability tables, but these, in turn, raise new problems, namely defining the state-transition matrices and the rules by which the tables evolve.

This means that stochastic algorithms are, on the whole, unsuited for generating music in a particular style⁶, and it is useful to have a number of different techniques ready to be used should the user wish to compose non-random music.

2.2 Formal Grammars and Automata

2.2.1 Music as language

Music is often thought of as a hierarchical structure. At the lowest level we have notes that make up the composition. These notes form phrases, which in turn form melodies and themes, movements and finally the entire composition. There is a striking analogy here with linguistics – the notes correspond to individual letters, the phrases words,

⁶ Throughout this thesis, when talking about musical style, we refer to the taxonomic classes, such as Classical or Baroque, into which we group compositions with particular musical characteristics. The author is aware of the difficulties that can arise when attempting to pigeonhole music in this way, but feels that a discussion on the nature of musical style lies outside the scope of this thesis.

melodies sentences and so on. This correspondence has led to the use of *language theory* in music.

Language theory is concerned with the makeup of *words*, each of which belongs to some *language*. A language is defined to be a subset of the (infinite) set formed by taking all possible combinations of a set of symbols.

For musical purposes, the symbols could correspond to, for example, notes and rests. The set of all combinations of notes would then give the universal set of all possible musical compositions. A musical language can therefore be thought of as being a collection of musical pieces in one particular style, such as the set of all baroque compositions. The motivation behind the theory is to discover the relationships between words, how they are formed, and how they can be categorised.

2.2.2 The use of formal grammars in music

There are essentially two ways – most easily thought of as analysis versus synthesis – that the theory of languages can be utilised musically:

- (i) The analysis of the syntactic structure of music. By examining the structure of musical input, we can hope to recognise and classify it.
- (ii) Generation of music that conforms to predefined syntactic structures. Here, the composer specifies the syntactic makeup of the musical language that he or she wishes to use and generates note data that adhere to this specification.

Although the latter seems to be the more relevant usage of formal grammars, the former also has its place, particularly amongst composition systems that aim to compose *replicatively*, that is, in a particular musical style, or in the style of a given composer.

One of the most successful applications of formal grammars to music generation is undoubtedly David Cope's *EMI* system ([Cope, 1991]). This particular system combines formal grammars with a device known as an *advanced transition network* to produce pastiche. The system takes as input at least two works in the style that the composer wishes to replicate. The system scans the input looking for *musical signatures* – short musical phrases that are a direct result of and thus indicative of a

particular composer's style. The system then produces new phrases incorporating variations on the signatures, building up a complete composition piece by piece, whilst ensuring that it fits the desired grammatical design.

The system has been used to produce several compositions in a wide range of musical styles, from Bach-like inventions to a Joplin-esque piano rag. The results have been varied but are generally very convincing. Some example scores are provided in Cope's book, *Computers and Musical Style* ([Cope, 1991]).

2.2.3 Cellular Automata

Cellular automata are very important to scientists as a modelling and simulation tool. They have found applications in many different disciplines, from image processing ([Preston & Duff, 1984]) and cryptography ([Wolfram, 1985]) to ecology [Hogeweg, 1988]) and biology ([Ermentrout & Edelstein-Keshet, 1993]).

Cellular automata are *dynamical systems*, that is, they change some feature with time. A Cellular automaton is discrete over both space and time and all quantities take on discrete values.

We often view a cellular automaton as an array of elements, referred to as *cells*, to which we apply some evolution rule that determines how the automaton develops in time. The cells are updated simultaneously, so that the state of the automaton as a whole advances in discrete timesteps.

Each cell can exist in one of p possible states, represented by the integers $0, 1, 2, \dots, p - 1$. We often refer to such an automaton as a *p-state cellular automaton*.

In order to specify fully and run a cellular automaton, we need one further piece of information – an initial cell configuration. When this is specified, the automaton can be set to run and the cellular evolution observed.

Any long-term global trends that arise in the automaton's development are an example of *emergent behaviour*, in the sense that the evolution rules in a cellular

feels that a discussion on the nature of musical style lies outside the scope of this thesis.

automaton are concerned only with local neighbourhoods around the cell under consideration – global trends are not coded explicitly beforehand.

2.2.4 The Game of Life

The Game of Life ([Gardner, 1971]) is a 2-dimensional cellular automaton that attempts to model a colony of simple organisms.

Theoretically, the automaton is defined on an infinite square lattice. However, for practical purposes we define Life as consisting of a finite $m \times n$ array of cells, each of which can exist in two states – alive, represented by 1, or dead, represented by 0.

The states of the cells as time progresses are determined by the states of neighbouring cells. There essentially four situations that can arise. These are:

- i.) *Birth*. A cell that is dead at time t becomes live at time $t + 1$ if and only if exactly 3 of its neighbours are alive at time t .
- ii.) *Death by overcrowding*. A cell that is alive at time t will die at time $t + 1$ if 4 or more of its neighbours are alive at time t .
- iii.) *Death by exposure*. A cell that is alive at time t will die at time $t + 1$ if it has 1 or fewer live neighbours at time t .
- iv.) *Survival*. A cell that is alive at time t will remain live at time $t + 1$ if and only if it has either 2 or 3 live neighbours at time t .

Note that rule (iv) holds if and only if rules (ii) and (iii) do not.

These four rules can be expressed as follows:

The *environment*, E , is defined as the number of living neighbours that surround a particular live cell. The *fertility*, F , is defined as the number of living neighbours that surround a particular dead cell. Note that both the environment and fertility vary as we move from cell to cell.

For Conway's Life, the life of a currently living cell is preserved whenever it has either 2 or 3 living neighbours. In other words, whenever $2 \leq E \leq 3$. Similarly, a

$$2 \leq E \leq 3$$

and

$$3 \leq F \leq 3.$$

Clearly, a number of alternative rule sets exist. The general form for such rule sets is

$$(E_{min}, E_{max}, F_{min}, F_{max}),$$

where

$$E_{min} \leq E \leq E_{max}$$

and

$$F_{min} \leq F \leq F_{max}.$$

However, although alternative rule sets to $R = (2, 3, 3, 3)$ exist, not all produce emergent behaviour that is as worthy of note as Conway's original.

In the Game of Life characterised by the rule set R above, there are a number of interesting initial cell configurations that give rise to notable cell dynamics. Some of these are presented in Appendix B.

Table 2.2.1 below describes the main classifications of Conway object.

Object class	Description
Oscillator	Any pattern that reappears in the same position after a certain number of generations.
Still Life	<i>Oscillators</i> of period 1, that is stable patterns.
Ship	Any pattern that translates across space.
Spaceship	Any pattern that moves, leaving no trail.
Billiard table	Any <i>oscillator</i> that is built inside a stable border.
Blinker	An <i>oscillator</i> of period two consisting of three live cells arranged in either a horizontal or vertical line.
Breeder	Any pattern that grows quadratically by creating multiple copies of a second object, each of which creates multiple copies of a third object.
Eater	Any <i>still life</i> that can repair itself from some attacks.
Flipper	Any <i>oscillator</i> or <i>spaceship</i> that forms its mirror image at half its period.
Glider	Any <i>ship</i> that travels diagonally across space.
Glider gun	Any pattern that grows forever by emitting <i>gliders</i> .
Primordial soup	Any random initial configuration of cells.

Table 2.2.1 – Glossary of Conway objects.

In order for a rule set, R to be considered worthy of the name Life, we demand that the following two conditions are met:

- i.) A glider must exist and occur “naturally” if we apply R repeatedly to primordial soup configurations.
- ii.) All primordial soup configurations when subjected to R must exhibit bounded growth.

For reasons of space we do not pursue this further, although we pick up on these ideas again in Section 4.1.3. Instead the interested reader is referred to [Dewdney, 1987].

2.2.5 Musical Applications of Cellular Automata

Cellular automata were originally introduced as a means of modelling crystalline growth, and as such, may be considered as a system of pattern propagation – the initial pattern (i.e. cell configuration) is transformed and developed according to a set of evolution rules. There is a direct analogy here to the processes involved with composition – an initial theme or motif is developed according to a set of composition rules specified by the composer or dictated by the musical style. In a sense, cellular automata can be considered as a model of the composition process itself.

It should come as little surprise, then, that cellular automata have been successfully employed in a number of musical applications.

For example [Miranda 1993], [Miranda, 1994] and [Beyls, 1997] all describe how different types of cellular automata can be harnessed as the basis for algorithmic composition systems.

The results of these systems vary and depend on the way that the automata are utilised. For example, Eduardo Miranda’s CAMUS system (see Chapter 3) generates compositions that are free from formal constraints (other than those imposed by the MIDI specification) and yet all share elements of a characteristic style, albeit one which is new and quite alien.

Peter Beyls’ system on the other hand imposes more pre-defined structure on the compositions, resulting in music that conforms more readily to existing musical styles. Automata have also been used in sound synthesis (see, for example, [Chareyron, 1990] and [Miranda, 1995]).

2.3 Iterative Algorithms

We now approach the problem of melody generation from a different angle, and see how iterative formulae and fractal techniques can be of use to the mathematically minded composer.

2.3.1 What is an iterative process?

One way of constructing musical melodies is to utilise an *iterative algorithm*. Iteration is the process of repeating an action, such as applying a mathematical function, over and over again, feeding the output of the action back into the input of the process. Figure 2.3.1 demonstrates graphically the process of iteration.

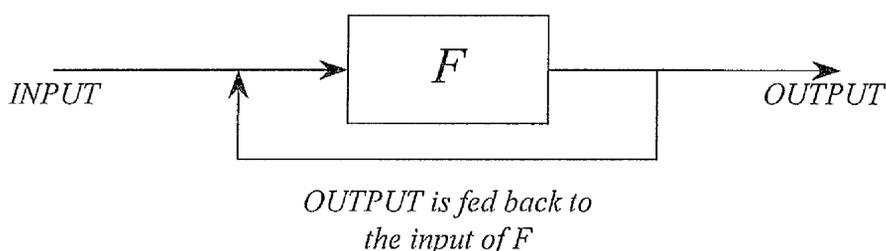


Figure 2.3.1 – Schematic of an iterative process, F .

An iterative process is defined by specifying the action that is to be repeatedly applied, and some initial value, x_0 , which we are free to choose.

The set of output values that arises from an iterative process is known as an *orbit* of the process. Note that an iterative process may have many orbits depending on the initial value, x_0 , that is fed into the process. We denote the orbit of the iterative process F that arises as a result of the initial value, x_0 by $\text{orb}_F(x_0)$.

There are essentially three classes of behaviour that the orbits of an iterative system can fall into⁷. These are:

- i.) The points in the orbit tend towards a stable⁸ fixed point.

⁷ Here we consider only those orbits that are bounded – those in which the orbit points remain finite.

⁸ That is, a point which is left unchanged by the process.

For example, consider the iterative function $x_{n+1} = x_n/2$. This will generate orbits of the form $\{x_0, x_0/2, x_0/4, \dots\}$. In general, the n^{th} iterate is of the form $x_0/2^n$. Clearly, as $n \rightarrow \infty$, the n^{th} iterate will tend towards 0, no matter what (finite) value we choose for x_0 . Thus, every finite orbit of this system tends towards the stable fixed point, 0.

ii.) The points in the orbit oscillate between elements of a finite set of points.

For example, consider the system defined by the function $x_{n+1} = ax_n(1 - x_n)$ (see, for example, [Peitgen & Saupe, 1988] for a discussion of this, the *logistic equation*) with $a = 3.1$ and $x_n = 0.5$ (see Figure 2.3.2). This results in the orbit $\{0.5, 0.775, 0.540, 0.770, 0.549, 0.768, 0.553, 0.766, 0.555, 0.766, 0.556, 0.765, 0.557, 0.765, 0.557, 0.765, \dots\}$. One can see that after an initial settling period, the orbit settles into oscillatory behaviour between the values 0.765 and 0.557.

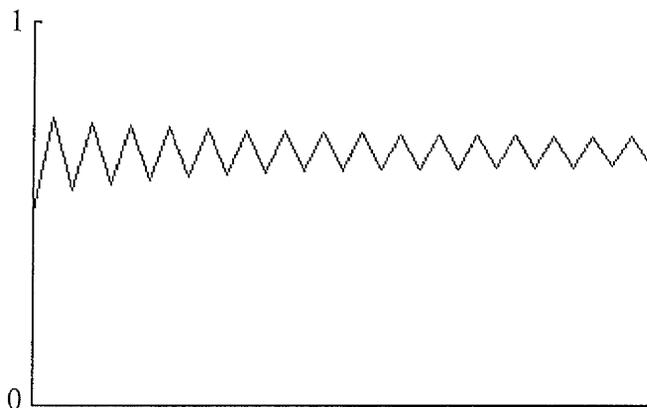


Figure 2.3.2 – Graphical representation of the orbit that arises from $x_{n+1} = 3.1x_n(1 - x_n)$ with $x_0 = 0.5$. Note that the orbit quickly settles into oscillatory behaviour.

iii.) The orbit points behave in a ‘chaotic’ manner. Sometimes, they may seem to behave randomly, never visiting the same point twice. At other times, they may seem to exhibit near-oscillatory behaviour with sets of almost identical points arising one after the other.

It is impossible to state precisely what we mean by the term *chaos*, due to the fact that there is no universally accepted definition. There are, however, a number of texts which offer precise, but slightly different definitions. The interested reader is referred to, for example [Peitgen & Richter, 1986] and [Peitgen, Jurgens & Saupe, 1992].

For the purposes of this discussion, we consider chaos as referring to a system that has gone ‘out of control’ in the sense that although each orbit is deterministic and specified in full by its initial value, we cannot predict its long-term behaviour in any way other than by setting the system in motion and letting it run ([Peitgen & Richter, 1986]).

For example, if we compare the orbit obtained from the function defined in (ii) above with $a = 4$ and $x_0 = 0.3$ with the orbit obtained by setting $x_0 = 0.301$ (see Figure 2.3.3), we see that after only a few iterations they bear little resemblance.

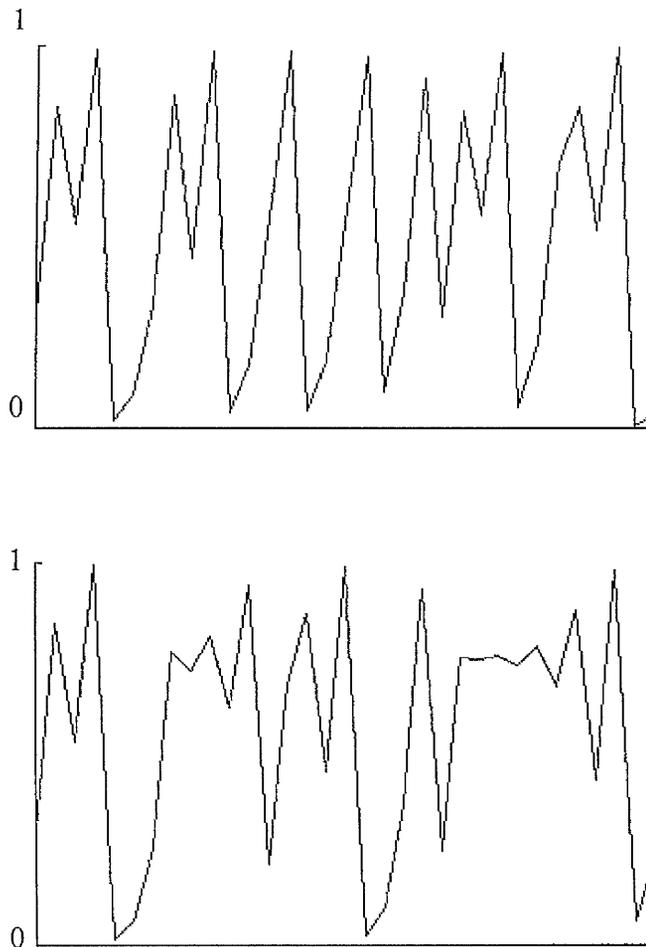


Figure 2.3.3 – Graphical representation of the orbits that arise from $x_{n+1} = 4x_n(1 - x_n)$ with $x_0 = 0.3$ (top) and $x_0 = 0.301$ (bottom). Note the divergence after just 7 iterations.

2.3.2 Implementing iterative processes musically

When implementing an iterative process to produce musical output, there are a number of things that should be taken into consideration, namely:

What iterative system should be used? Generally speaking, the human ear tends to enjoy music that sounds familiar enough to keep the listener comfortable, yet contains sufficient new material to generate interest and avoid boredom. Hence, an iterative process that produces orbits that quickly converge towards stable fixed points may not be the best choice for a music generating system, as the output values quickly become static.

Oscillatory behaviour offers more scope for generating interesting results, particularly if the period is large. However, in many cases, the period is often quite small, and the frequent repetition of the basic musical material can quickly become tedious. By far the most promising type of behaviour from a musical (and indeed, mathematical) point of view is chaotic behaviour.

As previously mentioned, a chaotic orbit will often wander among a fixed range of points, visiting similar, though not identical points each time. Thus the material that is generated has a degree of correlation with its past, whilst still producing new material. In fact, generating music by using chaotic orbits is sometimes viewed as a process of theme and variations, or exposition and development, because of the nature of this near-repetition.

How are the orbits of the process to be mapped to the musical output? This is, without a doubt, one of the most important considerations a composer faces when creating any algorithmic system that utilises the output of a non-musical process for creation of the final work. Choosing a mapping that is too simplistic may strip a potentially rich orbit of its detail, producing music that is dull and uninteresting.

Similarly, a mapping which is too complex may lead to difficulties when coding the system for a computer to calculate the compositions. Clearly, a balance must be struck.

The form the mapping takes will, to a certain extent, be dictated by the dimensionality of the iterative process itself. For example, a two dimensional process would allow

direct control over two musical parameters (e.g. pitch and duration). It is possible, however, to obtain similar levels of control by, for example, calculating several orbits of a lower dimensional system in parallel, by using several orbit points from a lower dimensional mapping simultaneously, or by using systems of more than one equation in a mapping.

It is also important to decide upon the musical parameters that should be controlled by the iterative process. The important parameters are likely to vary from composition to composition, and will depend to a large extent on the subjective preferences of the composer. Some examples include pitch, duration, velocity and timbre.

What starting value is to be chosen for the system? This is important, because, as we have seen, even for non-chaotic systems, orbits whose starting values lie close together can vary drastically after only a few iterations. This effect can also be used to the composer's advantage, however, and may be employed to produce cycles of works, or variations of a work, which begin in a similar manner, but which quickly diverge.

For further details of the use of iterative processes in music see [Little, 1993] and [Johnson, 1997b].

2.3.3 Fractal Algorithms

Fractals are a relatively recent development in mathematics. The term, derived from the Latin *fractus*, meaning *broken* ([Mandelbrot, 1982]), was coined in 1977 by the Polish mathematician, Benoit Mandelbrot. The essence of fractals is that they contain an infinite amount of detail at all scales. We illustrate with the coastline analogy:

We ask *how long is the coastline of Britain?* A study is commissioned, and a member of the research team duly sets off with his trundle-wheel and measures the length of the coastline by walking along the coastal roads. When he makes his report, it is pointed out to him that the coastline twists and turns away from the road. In fact, he has under-recorded the value.

He sets off again and this time moves away from the roads, measuring the coastline a metre from the edge all the way round the country. He records a higher value for the

distance than on his first expedition. Again, however, when he makes his report, it is noted that he has missed much of the finer detail at the coastline's edge. He sets off again, the time armed with a tape measure, and calculates a value higher still than the previous two forays, but here he notices that each rocky outcrop contains detail that is obscured by the tape measure. His recorded value is still too small! The finer the coastline was to be measured, the more detail was discovered.

This is fundamentally different to non-fractal structures, which at sufficient magnification look locally like straight lines. Figure 2.3.4 demonstrates the difference between successive magnifications of a fractal known as the Mandelbrot set, and a graph of the (non-fractal) sin function.

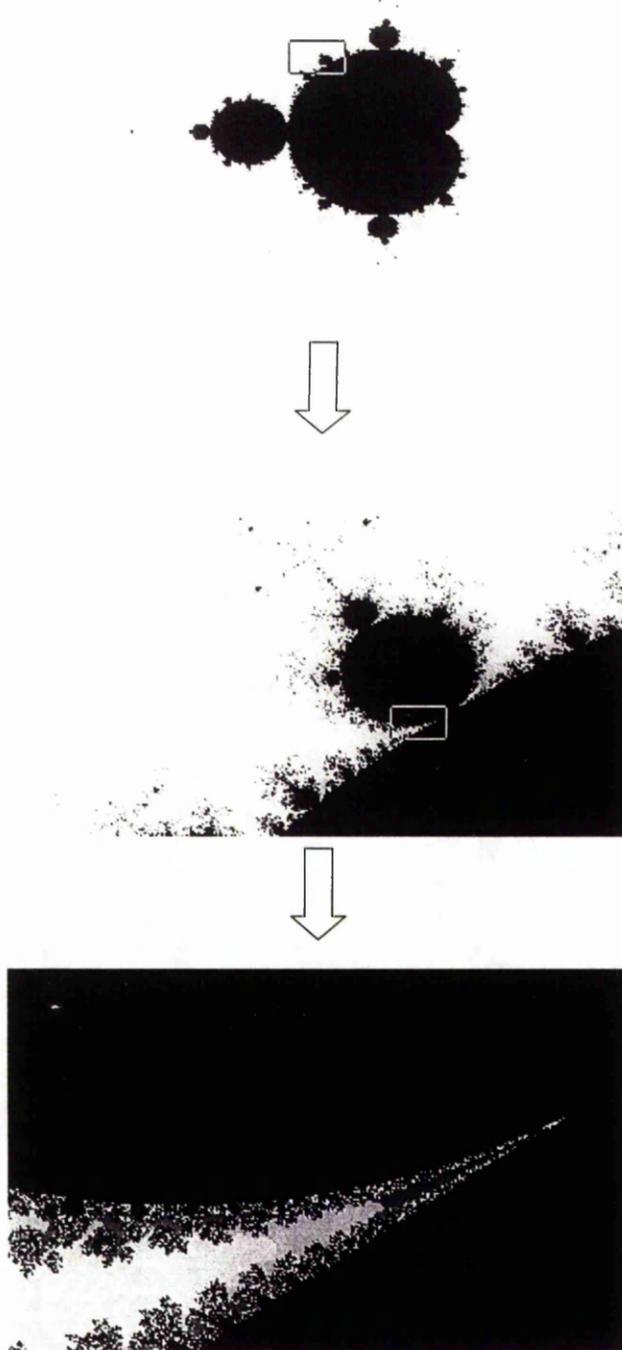


Figure 2.3.4 a – The complete Mandelbrot set and two successive magnifications showing detail at every scale.

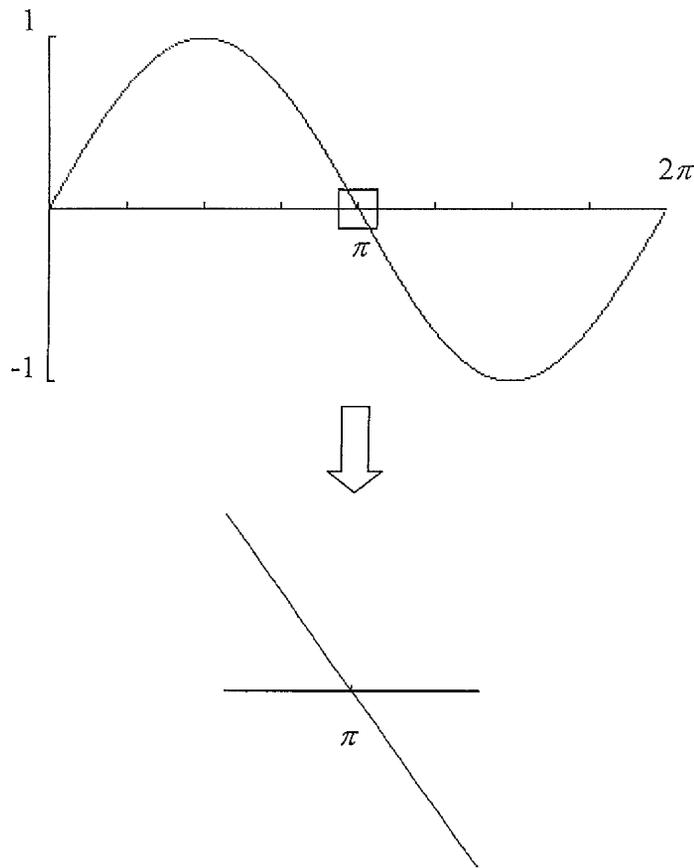


Figure 2.3.4 b – Graph of $\sin x$ and magnification showing that the graph looks locally like a straight line.

A characteristic feature of many fractal forms, which describes how they appear similar when viewed under different levels of magnification, is *self-similarity*.

There are three different kinds of self-similarity ([Peitgen & Saupe, 1988]):

- i.) *Exact self-similarity*, in which magnified small parts of the object in question are identical to the whole
- ii.) *Statistical self-similarity*, in which the magnified portion of the image has the same statistical properties as the whole object
- iii.) *Generalised self-similarity*, also known as *self-affinity*, in which scaled copies of the whole figure undergo some *affine transformation*⁹.

⁹ An affine transformation is one that is composed of a linear transformation and/or a translation. In two dimensions, an affine transformation, a , has the general form $a(\mathbf{x}) = A\mathbf{x} + \mathbf{b}$, where A is an invertible 2×2 matrix.

Figure 2.3.5 below shows a sample of $1/f$ noise (see Section 2.3.5), which demonstrates statistical self-similarity. A good example of an exactly self-similar figure is the *Sierpinski Gasket* (see Figure 2.3.6), which is made up of transformed copies of a triangle. Figure 2.3.7 shows a fractal spiral, and a skewed and rotated detail.

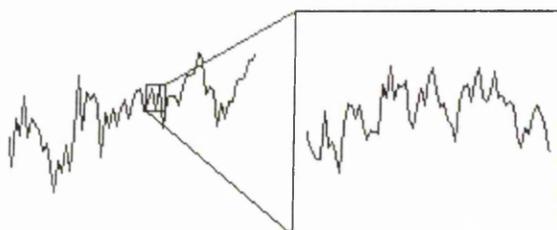


Figure 2.3.5 – $1/f$ Noise and magnification

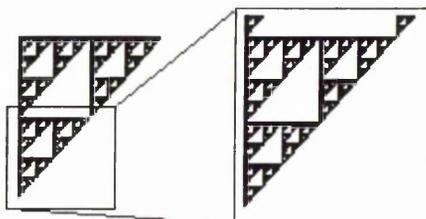


Figure 2.3.6 – Sierpinski gasket and magnification.

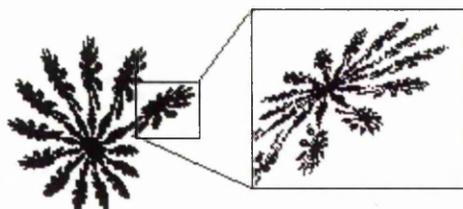


Figure 2.3.7 – Fractal spiral and detail.

It should be noted that a fractal figure need not exhibit self-similarity, the Mandelbrot set, for example, does not, though it does have detail on every scale.

2.3.4 Musical fractals

There are essentially two methods that can be employed when using fractals as a means of generating music:

(i) Using fractal equations¹⁰ to generate musical data.

(ii) Using fractal pictures to generate musical data.

Using fractal equations to generate music has a great deal in common with the iterative algorithms discussed earlier. Indeed, most fractal equations are iterative. One exception to this is $1/f$ noise.

2.3.5 $1/f$ Noise

The use of fractal techniques in music composition began when it was discovered that almost all musical melodies mimic $1/f$ noise ([Peitgen & Saupe, 1988]).

$1/f$ noise is a term which is used to describe any fluctuating quantity in which the 'amount' of frequency, f , varies as $1/f$.

This particular statistical blueprint shows itself in a great many physical situations, from semiconductors to ocean flows ([Peitgen & Saupe, 1988]).

There is no simple mathematical model that produces $1/f$ noise, although it is possible to generate close approximations. One such method for generating an approximate $1/f$ sequence is given in [Roads, 1996].

¹⁰ Note here that the term *fractal equations* is used to denote an equation (iterative or otherwise) that specifies a particular fractal figure. This is a generalisation of the term *fractal function*, which is used to denote any continuous function that is everywhere non-differentiable. The graphs of such functions are, in general, fractal sets.

2.3.6 The Mandelbrot Set

Perhaps the single most famous mathematical structure is the Mandelbrot set. The Mandelbrot set can be considered as a graphical representation of the behaviour of the complex plane according to the iterative equation

$$z_{i+1} = z_i^2 + c$$

where c is a complex constant. For each c in the plane, the system is initialised with $z_0 = 0$, and iterated. This results in exactly one of the following mutually exclusive outcomes:

- (i) $|z_n| \rightarrow \infty$ as $n \rightarrow \infty$.
- (ii) $|z_n|$ remains bounded as $n \rightarrow \infty$.

Each c is assigned a colour according to the number of iterations required to send the point to infinity, or a default colour (usually black) if the point does not increase without bound (in practice, we assign a threshold value, T , and a maximum number of iterations, MI , and stop the iterative process when either $|z_n| \geq T^{MI}$, or when $n > MI$). The Mandelbrot set is then the set of all values of c which are coloured black.

The area of interest occurs within the region bounded by the lines $x = -2.5$ and $x = 1.5$ on the real axis, and $y = -1.5i$ and $y = 1.5i$ on the imaginary axis.

The Mandelbrot set's distinctive squashed beetle-like profile is illustrated in the top image of Figure 2.3.6a above.

At, and near the boundary of the Mandelbrot set, there is described a structure of inconceivable complexity, with curves which spiral onwards for eternity, and detail at every scale.

A simplistic (and rather naïve) way of harnessing the structure contained within Mandelbrot's equation is to iterate the equation and associate the number of iterations required to send a point to infinity with a particular pitch. This is equivalent to mapping the colour bands of the set to pitches. An example is presented in Algorithm 2.3.1 below.

¹¹ We generally check to see whether the point $z = x + iy$ lies within the circle $x^2 + y^2 = T^2$. If not, we can be certain that it will tend to infinity ([Hoggar, 1992]).

For other applications of fractals in music composition and sound synthesis see [Bolognesi, 1983], [Dodge, 1988] and [Monro, 1995].

```

procedure Mandelbrot()
{
    REM Screen dimensions
    Width = 320
    Height = 240

    REM Region of complex plane to be 'played'
    MaxReal = 1.5
    MinReal = -2.5
    MaxIm = 1.5
    MinIm = -1.5

    REM Scaling factors
    HorizScale = (MaxReal - MinReal)/Width
    VertScale = (MaxIm - MinIm)/Height

    for i = 0 to Width - 1
        for j = 0 to Height - 1
            Iterations = 1
            RealC = MinReal + i * HorizScale
            ImaginaryC = MinIm + j * VertScale
            x = 0
            y = 0
            while (Iterations < 256 AND x*x + y*y < 4)
                NewX = x*x - y*y + RealC
                NewY = 2*x*y + ImaginaryC
                x = NewX
                y = NewY
                Iterations = Iterations + 1
            end while
            Play a note with pitch parameter Iterations
        next j
    next i
}

```

Algorithm 2.3.1 – Simplistic fractal music generator.

2.4 Evolutionary¹² Algorithms

2.4.1 Artificial neural networks

The human brain is, without a doubt, the most complex computing device known to mankind. Essentially a massively parallel array of simple information processing devices known as *neurons*, it is uniquely responsible for all of the works of art, all of the music and all of the scientific breakthroughs that surround us.

Neurons typically operate several orders of magnitude slower than the silicon logic gates used in today's computers ($\sim 10^{-3}$ seconds as opposed to $\sim 10^{-9}$ seconds) ([Haykin, 1994]). This, however, is compensated for by the sheer volume of and massive interconnectivity between them. It is estimated that the human cortex contains in the region of 10 billion neurons, connected together by some 60 trillion links, known as *synapses*. As a result, the human brain can perform certain tasks, such as pattern or voice recognition many times faster than even the speediest digital computer using current methods.

The brain also has the ability to program itself, forming rules and responses to situations that it has encountered and extrapolating from experience to deal with new phenomena. This process is commonly known as *learning*.

2.4.2 Modelling neural networks

Since the neuron is fundamental to the operation of a neural network, any attempt to model the workings of the brain must offer a satisfactory representation of it. In particular, the model must offer the following four attributes:

- (i) A set of connecting links (synapses) between neurons. Each of the synapses should have associated with it a weight which scales the signal arriving at the input. In particular, a signal, x , at the input to the j^{th} synapse of neuron i is multiplied by the synaptic weight, w_{ij} . The weight w_{ij} is positive if the associated

¹² Here, the term *evolutionary* refers to the computing techniques employed by these algorithms. The algorithms themselves do not evolve.

synapse is *excitatory* (signal boosting) and negative if it is *inhibitory* (signal attenuating).

- (ii) An addition unit for summing the weighted input signals of the neuron.
- (iii) An activation function, ϕ , which scales the output of the neuron to some prescribed range of values. Typical choices for the normalised output range of a neuron are $[0, 1]$ and $[-1, 1]$.
- (iv) An externally applied threshold, θ , which has the effect of lowering the net input to the activation function. A negative threshold value constitutes a *bias* term, which has the effect of boosting the input to the activation function.

Keeping these four points in mind, we arrive at a neural model as presented in Figure 2.4.1 below.

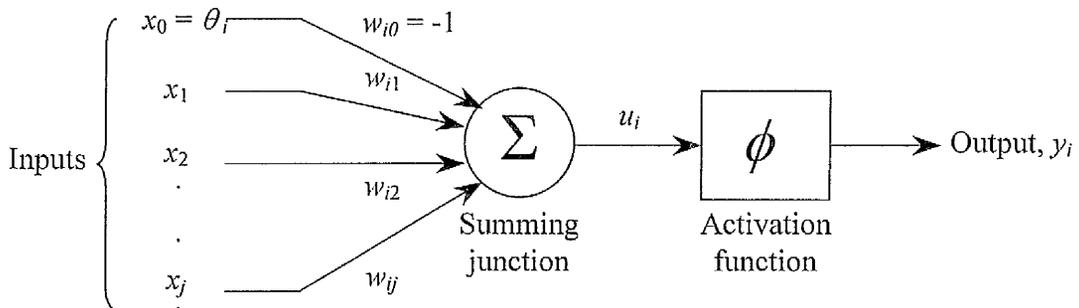


Figure 2.4.1 – Model of an artificial neuron.

We can specify the neuron of Figure 2.4.1 algebraically by the equations

$$u_i = \sum_{n=0}^j w_{in} x_n$$

which corresponds to the summing junction and

$$y_i = \phi(u_i)$$

which corresponds to the activation function, and by specifying the parameters corresponding to the synaptic weights and the threshold and the activation function associated with the neuron.

2.4.3 Types of activation function

The activation function, ϕ , defines the behaviour of a neuron in terms of its input. Three basic types of activation function are used:

(i) *Threshold functions* are generally of the following form

$$\phi(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{otherwise} \end{cases}$$

although functions of the form

$$\phi(x) = \begin{cases} -1, & \text{if } x \leq 0 \\ 1, & \text{otherwise} \end{cases}$$

are also used.

In those neural models that utilise threshold functions, there is no output from the neuron until a prescribed level of input activity is reached. Figure 2.4.2 shows a typical threshold function.

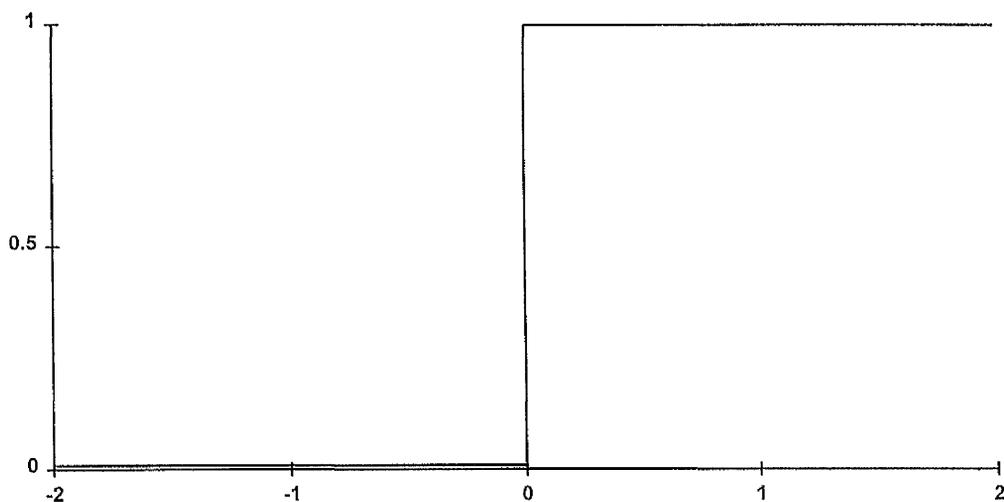


Figure 2.4.2 – Threshold function.

(ii) *Piecewise linear functions.* (For a definition of piecewise linearity see *Linear distributions* in the *Stochastic algorithms* section.) An example of a piecewise linear function is

$$\phi(x) = \begin{cases} 0, & x \leq 0 \\ x, & 0 \leq x \leq 1 \\ 1, & x \geq 1 \end{cases}$$

A piecewise linear function may be viewed as an approximation to a non-linear activation function.

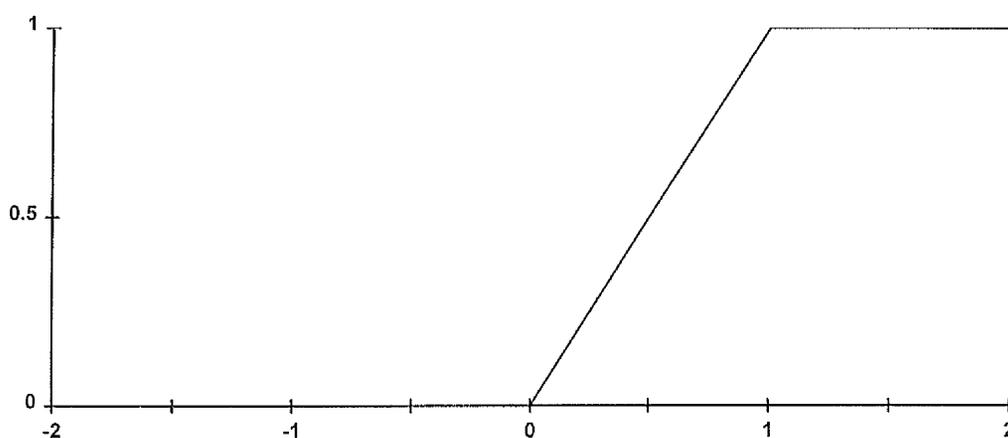


Figure 2.4.3 – Piecewise linear function

(iii) *Sigmoid functions* are the most widely used form of activation function. A sigmoid function is defined to be a smooth, strictly increasing function with horizontal asymptotes. Figure 2.4.4 shows the sigmoid function

$$\phi(x) = \tanh\left(\frac{x}{2}\right) = \frac{1 - e^{-x}}{1 + e^{-x}}$$

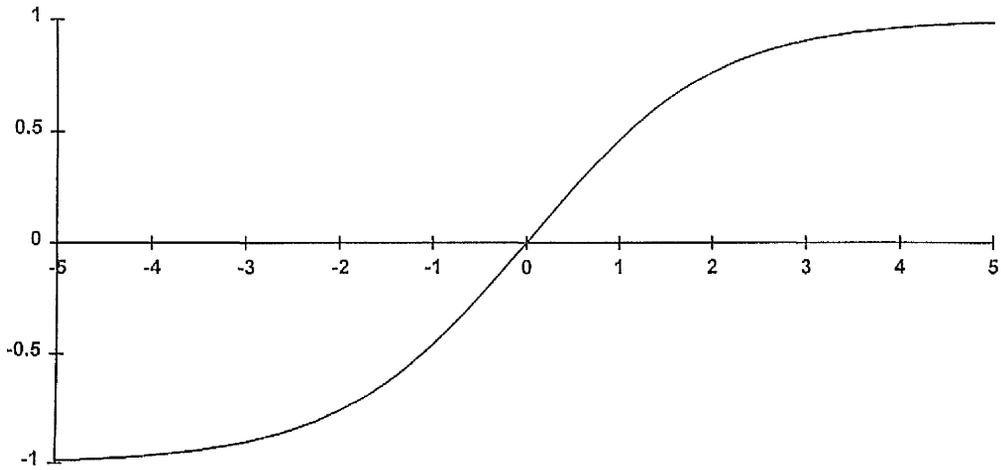
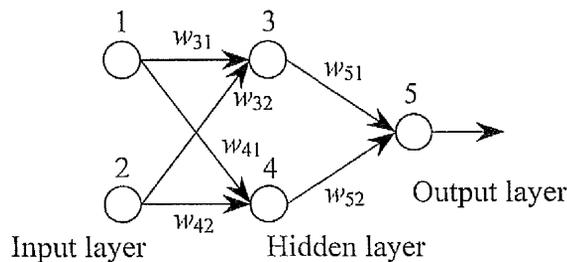


Figure 2.4.4 – Sigmoid function.

2.4.4 Graphical representation of a neural network

A neural network is essentially a type of automaton – a self-contained computing machine – and, as such, it is convenient to represent these networks as weighted directed graphs, whose nodes represent the individual neurons, and whose weighted edges represent the synaptic links. In addition, with each node, i , we associate an activation function, ϕ_i , which has the effect of scaling the output values to some prescribed range. Figure 2.4.5 below shows a simple neural network.



$$\phi_1(x) = \phi_2(x) = \phi_3(x) = \phi_4(x) = \phi_5(x) = \phi(x) = \tanh(x/2)$$

Figure 2.4.5 – A simple neural network.

The neural network of Figure 2.4.5 illustrates several important ideas. Firstly, notice that the neurons in the network may be grouped into three vertical layers as illustrated in Figure 2.4.5. These are the *Input*, *Hidden* and *Output* layers. The network is of the

feedforward type. That is, each of the neurons in any given layer is connected only to neurons in higher¹³ layers. The network is also said to be *fully connected*, in that each node in the i^{th} layer is connected to each node of the $i + 1^{\text{th}}$ layer. A network in which some of these synaptic links are not present is said to be *partially connected*. Since there are several layers of neurons in this network, we say that it is a *multilayer network*. In fact, we go further and label it a 2-2-1 multilayer feedforward network, where the numbers refer to the number of neurons in each of the three layers. The middle layer of neurons is referred to as a hidden layer, since these lie out of sight to an external observer. The hidden layer performs additional processing on the data flow from the input to the output neurons.

2.4.5 Learning in a neural network

Machine learning is, conceptually at any rate, no different to the learning processes experienced by, for example, human babies. We identify three key steps in the learning process:

- (i) The neural network receives stimulus from the environment.
- (ii) The neural network changes in some way as a result of the stimulus.
- (iii) The neural network now responds differently to environmental stimuli due to the changes that have taken place.

The problem can be approached in several ways. We now identify three learning paradigms that can be employed.

- (i) *Supervised learning*. This is analogous to the classroom notion of learning, in which an external ‘teacher’ supervises the whole of the learning process to ensure that the network behaves and progresses in the correct manner.
- (ii) *Reinforced learning*. Here, a critic that evolves through a trial and error process is employed to guide the learning process.

¹³ Note that the layers in the network are numbered from left to right. Thus, when we talk of *higher layers*, we mean those layers to the right of the one in question.

(iii) *Unsupervised learning*. As the name suggests, no external teacher or critic is employed in learning algorithms of this type. The process is entirely self-organised.

It is neither practical nor particularly useful to study each of the learning algorithms in detail at this point. The interested reader who wishes to read further on alternative learning algorithms is referred to [Haykin, 1994] and [Kartalopoulos, 1996].

2.4.6 Neural networks and music

The application of neural networks to musical problems should really come as no surprise. After all, creative processes like musical composition are often considered to be quintessentially human processes, and neural networks were themselves introduced as a means of modelling the human thought processes.

Neural networks have been applied in a number of musical situations. One usage has been for pitch detection ([Jenkins & Sano, 1989]).

Here, a sound is fed into the trained neural network. The different partials present in the sound excite neurons in the input layer. The weighted outputs from these neurons are processed by hidden layers before finally exciting the single output neuron, resulting in a single best guess at the input pitch.

Pitch detection networks of this sort have several advantages over more traditional pitch detection techniques which try to find periods in the input waveform to guess the input pitch. In particular, they are often easier and cheaper to implement once the initial training period is complete. Neural-based pitch-detection techniques are also not as prone to delays in guessing as traditional techniques, which must wait for the initial noisy attack and at least one period of the waveform to complete before attempting to determine the pitch. In fact, the neural approach to pitch detection has been so successful that the technique has been commercialised by Yamaha as the basis for their MIDI guitar system ([White, 1996]).

Neural networks have also been used to compose music (see, for example [Dolson, 1989]), to model the human perception of tonality ([Bharucha & Todd, 1989]) and to generate sounds ([Warthman, 1993])

2.4.7 Genetic Algorithms

We now introduce a parameter optimisation technique known as a *genetic algorithm* (GA). This technique was originally introduced in the field of engineering as a means of finding optimal solutions for problems that were not easy to solve using traditional optimisation techniques, and operates on a set of binary codewords.

There are four basic types of operation which can be performed on the codewords: selection; crossover; mutation and inversion. We shall consider only the first three, since inversion can be derived from these. The crossover operation exchanges information between a pair of codewords. Mutation alters the value of a single bit in a codeword. Typically, the genetic algorithm is used as an optimisation technique, and so the selection operation is used to find the 'best' possible codewords for some predetermined criterion. However, for compositional purposes, an optimisation of this sort may be unnecessary if the operations are used solely for the development of the compositional parameters. Alternatively, if the composer wishes to search the space of compositions for the best melody, some care must be taken over the choice of fitness function, since this determines the melodies that are selected and those that are discarded.

A typical parameter evolution step is shown in Table 2.4.1.

Explanations	Codewords
Consider a set of n codewords which represent the values of some musical parameters.	$C_1: 11010110$ $C_2: 10010111$... $C_n: 01001001$
Selection: Codewords are chosen to undergo evolution according to some stochastic mechanism (<i>c.f.</i> Darwin's theory of evolution – chance mutation leading to population development ([Darwin, 1981])).	$C_2: 10010111$ $C_7: 11000101$ $C_{11}: 01111001$
Crossover: Some portion of a pair of codewords (in this case, C_7 and C_{11}) is exchanged at the dotted position randomly specified, producing the two offspring C_7' and C_{11}' , whose values are then assigned to C_7 and C_{11} . This is applied to some predetermined section of the population, specified by the crossover rate.	$C_2: 10010111$ $C_7: 11000 101$ $C_{11}: 01111 001$ $C_7': 11000 001$ $C_{11}': 01111 101$
Mutation: The bit values of some codewords are inverted at a mutation rate of 1 - 5%. The value that has been changed as an example is highlighted by an underline.	$C_2: 10010111$ $C_7: 11100001$ $C_{11}: 01111101$

Table 2.4.1 – Typical parameter development step for a GA.

The genetic algorithm can be utilised as a parameter development technique as follows: firstly, each parameter is assigned a binary codeword according to its initial value. At each timestep, the genetic algorithm is performed on the set of codewords as illustrated in Table 2.4.1 above, leading to population development.

2.4.8 Musical applications of GAs

Genetic algorithms have been applied to the composition problem. One example, which uses the simple genetic algorithm to produce an 'optimal melody'¹⁴, is the program *GAMusic* by Software Visions ([Moore, 1994]).

Of course, the problem of how to define an optimal melody is by no means trivial. The subjective qualities of music are such that a truly objective definition of musical worth may remain forever elusive.

GAMusic side-steps the issue slightly by leaving the aesthetic judgement of the generated melodies' fitnesses to the user, who auditions each melody and awards it a fitness value. The possible fitness values are poor – 1, average – 2 and good – 3. Each melody in the initial population starts out with an average fitness. Following the fitness assignment, the GA is iterated once by the user.

The iterative evolution of the codewords, and the mutation and recombination frequencies are also controlled by the user. Each series of musical notes is represented in binary form by an array of length 128, allowing a maximum of 30 notes per melody and providing a solution space with approximately 3.4×10^{38} possible melodies. An unintelligent search for an ideal melody would require testing every possible solution, and even allowing an optimistic estimate, it would take years to search, play and evaluate each of these solutions. In theory, the GA should be able to find a near optimum melody after searching only a fraction of the solution space.

Genetic algorithms have also been used in sound synthesis (see, for example [Horner, Beauchamp & Haken, 1993]).

¹⁴ It is difficult to specify precisely what is meant by an 'optimal melody', since the judging of such is necessarily subjective and fraught with difficulty. We cannot hope to offer a concrete definition, but for the purposes of this discussion, we use the phrase to refer to a melody that is more pleasant and which provides a closer match to the composer's aesthetic ideals than any other.

2.5 Serial Algorithms

2.5.1 Serialism

One composition technique that lends itself very well to algorithmic treatment is that of *serialism*. This is the method of composition that derives from Arnold Schoenberg's twelve-note style, which arose from the composer's experiments in writing music that was free from tonal constraints and which did not follow the traditional ways of building chords [Schoenberg, 1984].

The essence of serialism is that all chords and melodies are constructed as arrangements of the twelve notes of the chromatic scale in a particular order. This is referred to as a *tone-row* or *series*.

Each tone-row has associated with it 48 different forms, which are obtained by subjecting the series to one of the following transformations.

- (i) *Transposition*. The tone-row can begin on any of the twelve semitones.
- (ii) *Retrograde motion*. The tone-row can be played backwards on each of the twelve semitones.
- (iii) *Inversion*. The tone-row can be played 'upside down'. More formally, each of the intervals through which the melody proceeds is replaced by its inversion.
- (iv) *Retrograde inversion*. The tone-row can be played backwards and upside down.

Clearly then, there are 12 forms for each of (i) – (iv) above, resulting in 48 distinct transformations of the original tone-row.

There are often further constraints placed on the development of the tone-row throughout a composition. For example, the composer may decide that the tone-row is not to be repeated after its initial exposition. In addition, the composer may decide to permute the tone-row after a number of steps to obtain a new series.

The tone-row can then be thought of as a blueprint for the entire composition. Everything that appears subsequently in the composition is derived in some way from the original tone-row.

2.5.2 Implementing a serial algorithm

In order to construct a serial algorithm, the composer must decide upon the following.

- (i) *How should the initial tone-row be chosen?* There are two options here. It can be specified manually by the composer, or automatically generated by computer.
- (ii) *How should the transformations be chosen?* For example, the composer may specify the transformation at each stage in the composition process, or may employ a decision making routine within the composition algorithm. The decision making routine could be deterministic, using properties of the tone-row to choose which transformation to apply next, or could be stochastic in nature.
- (iii) *How should the other composition parameters develop?* There are a number of different methods that could be employed here. One technique, in keeping with the serial nature of the notes, is to use formal mathematical series to control the parametric evolution. For example, the composer could map the values of a geometric series¹⁵ to tempo to obtain *accelerandi* and *ritardandi*.
- (iv) *At what stage should the tone-row be permuted?* Again, this is a decision that the composer may wish to make personally, or which could be handled by a decision making routine from within the algorithm.

We illustrate with the following algorithm:

¹⁵ A geometric series is a sequence of real (or complex) numbers, s_0, s_1, s_2, \dots in which s_{n+1} , the next number in the series is obtained by multiplying s_n by a factor, g . The n^{th} term of the series, s_n , is therefore s_0g^n .

```

procedure serial()
{
    generate the initial tone-row
    repeat
    {
        if (MIDI value for last note in series  $\equiv 0 \pmod{4}$ )
            transpose series to begin on last note
        else if (MIDI value for last note in series  $\equiv 1 \pmod{4}$ )
            apply retrograde motion to series
        else if (MIDI value for last note in series  $\equiv 2 \pmod{4}$ )
            apply inversion to series
        else
            apply retrograde inversion to series
    until user interrupts
    }
}

```

Algorithm 2.5.1 – Serial composition algorithm.

The algorithm employs a deterministic decision making routine for choosing the transformation that is to be applied. There is a danger here, however, in that the original tone-row may be repeated before the full cycle of 48 transformations has been completed. It is therefore prudent to have a further subroutine which would compare the newly-transformed tone-row with the transformations that have already been played, and transform it again if it had already been played. If this additional check were not applied, we would have formed a cycle of fewer than 48 transformations which would cycle indefinitely, since the decision making routine is deterministic.

The generation of the tone-row could be achieved by a routine similar to the following:

```

procedure tone_row()
{
    for i = 0 to 11
        N[i] = i
    next i
    for i = 0 to 100
        a, b = random integers between 0 and 11
        swap N[a] and N[b]
    next i
}

```

Algorithm 2.5.2 – A simple tone row generator.

Here, we have assumed that the notes of the tone-row are stored in an 11-element array, $N[]$, and are specified by an integer between 0 and 11.

Transposition is achieved by adding the transposition value to each of the notes in the tone-row:

```

procedure transpose(amount)
{
    for i = 0 to 11
        N[i] = N[i] + amount
    next i
}

```

Algorithm 2.5.3 – Algorithm for transposing a tone row.

Retrograde motion is achieved by re-ordering the elements in the array:

```

procedure retrograde()
{
    for i = 0 to 11
        temp[i] = N[11 - i]
    next i
    for i = 0 to 11
        N[i] = temp[i]
    next i
}

```

Algorithm 2.5.4 – Algorithm for calculating a retrograde tone row.

Finally, inversion is obtained by inverting the interval between the successive notes of the tone-row:

```

procedure inversion()
{
    temp[0] = N[0]
    for i = 1 to 11
        interval = N[i] - N[i - 1]
        temp[i] = temp[i - 1] - interval
    next i
    for i = 0 to 11
        N[i] = temp[i]
    next i
}

```

Algorithm 2.5.5 – Algorithm for inverting a tone row.

2.5.3 Quality of results

Serial music is not judged by traditional aesthetic criteria. The main objective is not to produce pleasing melodies, but to subject the original tone-row to complex transformations. Because of this highly mathematical structure, serial composition lends itself very well to the algorithmic treatments discussed here, and, with a little work, it is possible to create complex compositions that sound convincingly like the works of human proponents of this style. After all, Schoenberg, Stockhausen et al used exactly the same transformations as we have presented above. The use of a computer merely removes the necessity for menial calculation.

Few algorithmic composition packages have been dedicated entirely to serial algorithms. Most often such transformations form part of a larger composition environment. Such is the case with Datamusic's *Fractal Music* for the Atari ST ([Sansom, 1992], [Russ, 1992], [Johnson, 1997a]).

Fractal Music is a composition package that uses fractal and iterative techniques to drive the generative elements of the system. However, once the program has generated a melodic line, it is then subject to serial (and other) transformations. Amongst the alternatives to the basic transposition, inversion, retrograde motion and retrograde inversion options are:

- (i) *Quantise*, which allows the composer to quantise the note values in a track to a specific resolution (say to the nearest semiquaver).
- (ii) *Stretch/move* which allows the composer to alter a track's position in space (i.e. transposition) and time. It is also possible to compress and expand a track in both space and time (i.e. augmentation and diminution).
- (iii) *Reflect/rotate* allows the composer to apply precise mathematical transformations to the note data. This is simply a generalisation of the more traditional inversion (i.e. reflection in the horizontal pitch axis), retrograde (i.e. reflection in the vertical time axis) and retrograde inversion (i.e. rotation around 180 degrees) transformations.

Combining algorithmic techniques like this vastly increases the scope of a system, and can aid the composer both with the generation of new musical ideas and with the development of these initial ideas into complete musical works.

2.6 Rule-based algorithms

2.6.1 What is a rule-base?

The final class of algorithm that we examine is the class of *rule-based algorithms*. This class of algorithm uses a set of desired responses, known as a *rule-base* (or *knowledge-base*), which is stored in permanent memory and consulted when the system receives input ([Ullman, 1990]).

The rules essentially define the behaviour of the system, which should have at least one desired response for each possible input.

A system of this sort, which contains the coded knowledge of one or more experts in a particular field is often referred to as an *expert system*.

2.6.2 Knowledge representation in an expert system

The way that information is stored in a rule-base depends on a number of factors. For example:

i.) *For what purpose is the expert system to be used?*

Clearly, an expert system whose purpose is to advise medical staff of likely diagnoses for given symptoms is unlikely to have the same data storage requirements as an expert system for music composition. There may, of course, be certain common requirements – both systems may use ASCII text to store certain types of data, for example. However, on the whole, different requirements require different methods of storage.

ii.) *What resources are available for input?*

Will the composer be entering composition data manually from a keyboard? Will the data be entered as textual information? Perhaps some algorithmic process is at work in the background and will generate some other form of composition data. All of these will have a direct bearing on the way that the rule-base is stored in memory. Clearly, it is desirable to have a reasonably close match with the input data and the rule-base to avoid unnecessary translations. For example, if composition data are to be entered as text, whilst the rule data are stored as MIDI information, there must be at least one translation to ensure that a meaningful comparison can take place.

iii.) *What resources are available for output?*

Again, this will have a direct bearing on how the responses are to be stored in memory. For musical purposes, this is extremely likely to be in the form of short MIDI files, or score files for composition languages.

iv.) *What resources are available for the system?*

This is a further consideration that has some relevance on how the rule-base should be implemented. For example, suppose that the computer on which the system is to be implemented offers less storage space than is needed to fully specify the rule-base. This would mean either omitting some of the rules or the information associated with them, which is not desirable since it alters the behaviour of the system, or finding some alternative means of representation which requires less storage space.

2.6.3 Some benefits and drawbacks of rule-base systems

One of the greatest benefits of using rule-base systems of this sort is that the results are potentially exceptionally good, because the system draws on the knowledge of experts to create new compositions.

Another benefit of the rule-base approach to composition is that it allows the creation of 'hybrid' works, which would have otherwise been unlikely (or impossible). After all, there is no reason why an expert system should refer to the knowledge of only one expert.

It is easy to envisage a situation where several experts, each with their own area of expertise, have contributed to an expert composition system. On composition, then, the system has at hand the combined knowledge of each of the experts, making possible the creation of a funk-baroque, or classic-blues work.

There are, however, problems, albeit problems which are not insurmountable.

In order to create a truly expert system, one must consult true experts. This makes the creation of an expert system both time consuming and costly: One must first find experts on whatever subject the system is to cover. Having found these experts, their entire knowledge of the subject must somehow be discovered and formalised, then encoded and stored.

Of course, further problems arise if, for example, there are no living experts on the particular subject of interest. Thankfully, there is a wide range of musical analysis available which may be used to assist in the creation of a knowledge-base.

For example, if we wish to construct a Mozart algorithm, it would be possible to examine the numerous analyses of his work and obtain a clear idea of how the composer dealt with certain situations. This could then be coded and stored just as with the knowledge of contemporary experts.

Also, having built an expert system for composition, there is a danger that it will respond to similar input in an identical manner. The problem arises if the system has been designed to repeat exactly the experts' responses (i.e. quote verbatim). For example, if the system has been designed using an input checking routine of the sort shown in Figure 2.6.1 below, there is a real danger that the expert system will become little more than an expensive musical quotation machine.

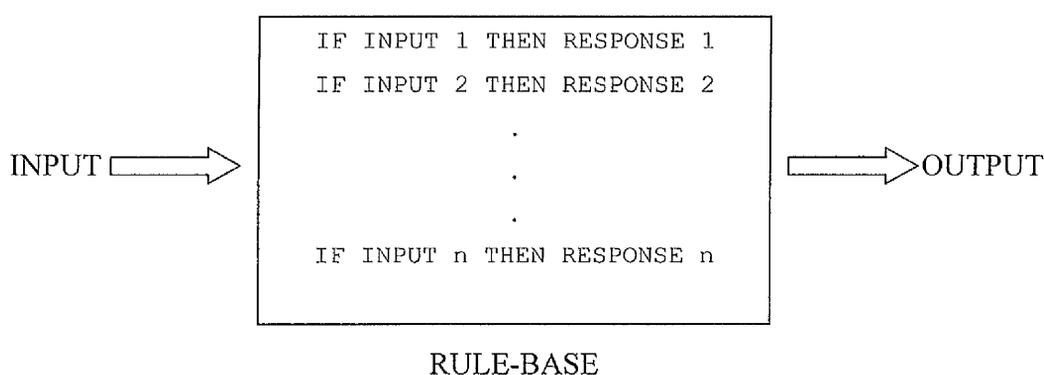


Figure 2.6.1 – Simplistic expert system that repeats the experts' responses exactly.

One way to avoid this is to employ a subroutine that takes as input an expert's response and returns a variation of that response.

A good example of a composition system which uses a knowledge base is [Cope, 1987].

2.7 Turing Systems: A new means of classification

We now propose a method of classifying certain types of musical algorithm that depends not on how the music is created within a composition system, but on how the composition system itself behaves.

Recently, with the explosion of interest in the world wide web, a number of music generation packages have arisen in order to satisfy the need for music over the

internet. By far, the most interesting of these packages have been those that allow interactive generation of music – literally jamming via the web. One such system is the Rocket Network, which allows composers and audio professionals anywhere in the world to work together on the same recording over the Internet, eliminating the need for all contributors to be in a common location (see [Wentk, 1995] and [Res Rocket, 1999]).

The packages in question allow several musicians to link up via the internet and create music together. Some music may be generated algorithmically; some may be played by human performers, and some may have been pre-recorded and replayed. The actual means by which the music is generated is not, in itself, important. What is important is that each individual musician is receiving musical input at their computer terminal and is providing musical output in response. The musician does not know (nor particularly care) whether the music has come from the fingers of a pianist halfway around the world or from the logic gates of a PC on the other side of town.

One can see straight away the similarities between this and the writings of the great theoretical computer scientist, Alan Turing, whose infamous test for machine intelligence ([Turing, 1950]) states that:

True machine intelligence can be said to have been achieved when a human being can converse via a computer terminal with an unseen operator and a machine and is unable to distinguish between the two.

For this reason, we have grouped interactive algorithmic music packages of this sort together under the title *Turing Systems*.

One might imagine that it would be simple to tell the difference between a real jamming musician and a computer running algorithmic composition software. In practice, though, if a listener is given no additional clues to the origin of a piece of music, it is often very difficult to tell – there may be little difference in the output from a poorly designed generative music algorithm and a musician who cannot improvise!

2.8 Summary

We conclude this section by presenting the following table, which highlights the main features of the algorithms discussed herein.

It should be noted from this table, that each algorithm has its advantages and its disadvantages. There is no 'right' or 'wrong' algorithm. The composer should examine the problems that face him when composing a work and should make an informed choice of algorithm accordingly.

Algorithm Type	Key Features	Advantages	Disadvantages
Stochastic	Driven by the laws of probability. Rely on random functions.	Easy to implement and understand.	Output is random – unsuitable for replicative composition. Impossible to predict in advance exactly how the composition will sound, making it difficult to plan compositions.
Formal Grammars and Automata	Uses language theory to specify macroscopic structure and fine detail.	Can generate extremely convincing music. Formulation rules easy to specify.	Underlying mathematics is quite complex and difficult for non-mathematicians to comprehend.
Iterative and Fractal	Uses fractal pictures and mathematical equations to generate note data.	High correlation between visual and sonic representations of data. Chaotic orbits often produce interesting results.	Very many uninteresting orbits. Complex mathematics may be difficult to understand. Poor choice of mapping often leads to dull or non-musical results. Often difficult to predict how the system will behave for chaotic orbits.
Evolutionary	Neural techniques use a model of the workings of the human brain to analyse and replicate compositions. Genetic algorithms evolve an initial melody to some optimum.	Exceptionally powerful technique, particularly when combined with other classes of algorithm. The composer only has to provide the building blocks of a work, the system can complete it.	Extremely complex. Neural-based systems often have to undergo extensive training. Difficult to define what is meant by an optimal melody.
Serial	Formalisation of a well-established composition technique. A series of notes is subject to various musical transformations.	Easy to implement and understand. Output is of a similar standard to that obtained manually.	Serial composition is not to everyone's taste!
Knowledge-base	System responds to input from the user by consulting an expert knowledge-base.	Convincing results. Puts a wide range of musical expertise at the composer's fingertips. Fusion of styles possible.	Difficult and costly to implement. Danger of straight repetition of stored knowledge.

Table 2.8.1 – Summary of the six main algorithm types.

3. CAMUS – A cellular automata based music algorithm

3.0 Introduction

In this chapter we introduce the fundamentals of our research work involving the application of cellular automata (see Section 2.2.8) to music composition and present CAMUS, an algorithmic composition system which uses cellular automata as the basis for its control system.

3.1 Cellular Automata MUSic

The CAMUS (Cellular Automata MUSic) system uses the *Game of Life* ([Gardner, 1971]) and *Demon Cyclic Space* ([Miranda, 1993]) automata to generate compositions.

3.1.1 The Game of Life

The Game of Life automaton consists of an $m \times n$ array of cells, which can exist in two states, alive, denoted by 1, or dead, denoted by 0. The rule which determines the development of the automaton is: *A cell will be alive at timestep $t + 1$ if and only if it has precisely 3 live neighbours at timestep t .* Figure 3.1.1 shows six successive timesteps of the Game of Life. The live cells are shaded in black.

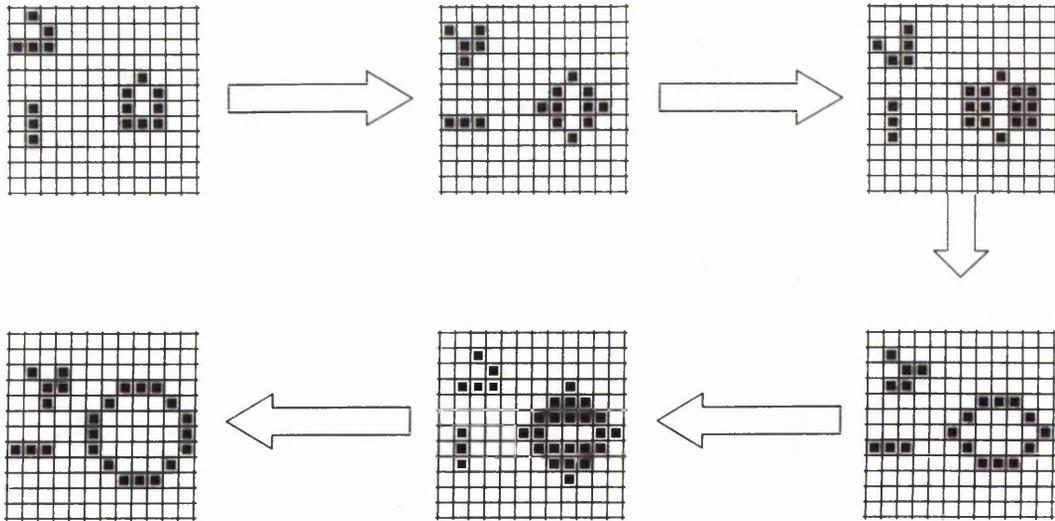


Figure 3.1.1 – Six successive timesteps of the Game of Life.

3.1.2 The Demon Cyclic Space

The Demon Cyclic Space automaton is an $m \times n$ array of cells which can exist in k states. The evolution of the automaton is determined by: *A cell which is in state j at timestep t will dominate any neighbouring cells which are in state $j - 1$, so that they increase their state to j at timestep $t + 1$.*

A very important feature of the Demon Cyclic Space is that it is *cyclic*, meaning that we treat k as 0. This has the effect that cells in state 0 dominate cells in state $k - 1$. Thus the influence that a cell exerts on its neighbours has far reaching consequences, often extending beyond the eight cells that immediately border the cell in question. This, taken in conjunction with the fact that the cells are checked in parallel, results in a self-organising structure as shown in Figure 3.1.2 below. The image on the left-hand side is the initial, randomised cell configuration. The image on the right-hand side is taken 20 timesteps later and shows how the cells arrange themselves into a 'patchwork' type of pattern.

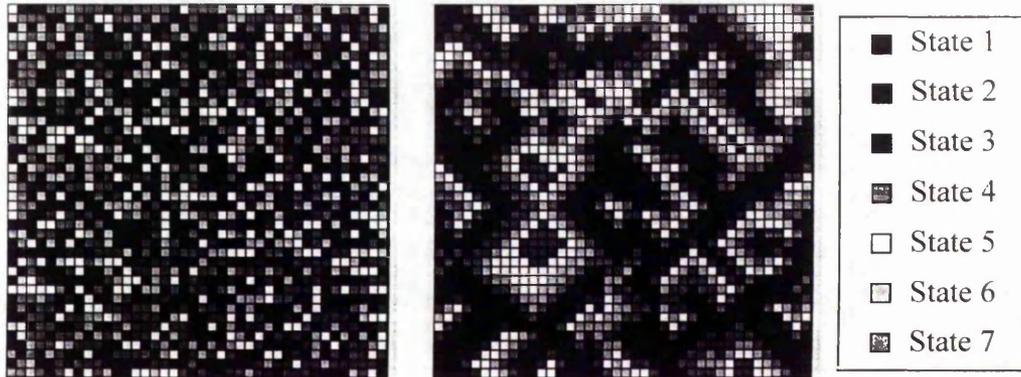


Figure 3.1.2 – Two timesteps of the Demon Cyclic Space. The image on the left is the initial, randomised state. The image on the right is from the same region taken 20 timesteps later.

3.1.3 The Automata Space

The Game of Life and Demon Cyclic Space automata described in Sections 3.1.1 and 3.1.2 above inhabit a space which is *toroidal* in nature. That is to say that the right edge of the automata space wraps around to meet the left edge and the top edge of the automata space wraps around to meet the bottom edge – the planar spaces of Figures 3.1.1 and 3.1.2 are mapped onto a torus (see Figure 3.1.3). For reasons of legibility, however, we will continue to present the automata as planar spaces, it being a simple enough matter to track the wrapping of cells in one’s own mind.

Mapping the automata spaces onto toruses in this way is not essential. It is possible to treat the spaces as being truly planar by imposing boundaries at the edges, just as it is possible to map the spaces onto other spaces, such as cylindrical spaces, where the space is bent so that only two opposing edges meet.

Whilst these alternative spaces may also produce interesting cell dynamics, the toroidal spaces win out for ease of use, since there is no need for extra code to check the boundary conditions, and predictability – both the top and bottom ‘edges’ behave in an identical manner. It is for these reasons that toroidal spaces have been employed here.

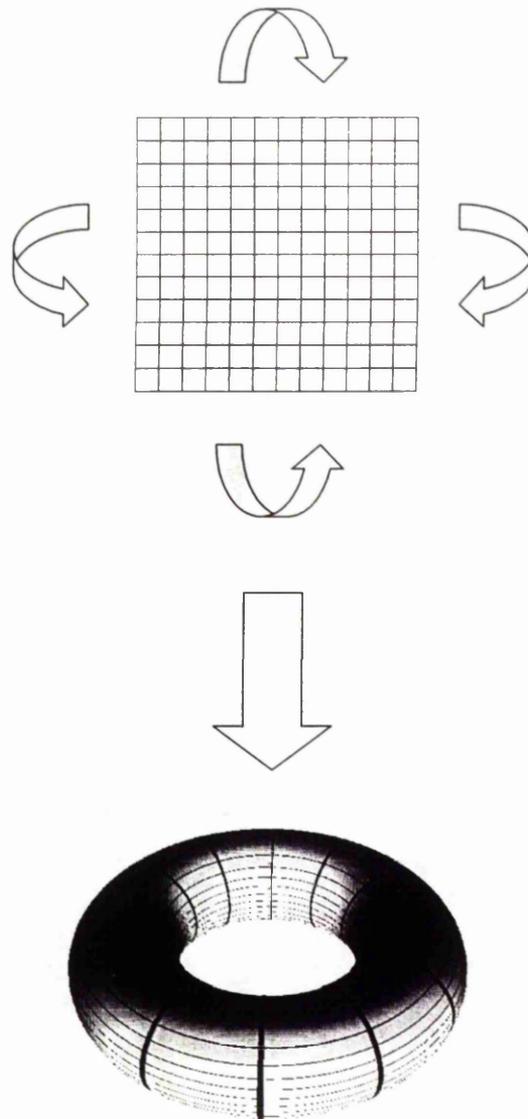


Figure 3.1.3 – Mapping a 2-dimensional automaton space onto a torus.

3.1.4 The CAMUS algorithm

To begin the composition process, an $m \times m^{16}$ Game of Life automaton is set up with an initial cell configuration, the Demon Cyclic Space automaton is initialised with random states, and both are set to run.

¹⁶ Observe that here we take $n = m$ in our definition.

At each time step, the cells of the Game of Life are analysed column by column¹⁷, starting with cell (0, 0), and continuing until cell (m, m) has been checked¹⁸.

3.1.5 Chords in CAMUS

We define an *n*-note chord to be a set of *n* notes which may or may not sound simultaneously. Thus we have that



is a 3-note chord, as is



and



and so on.

We can describe an *n*-note chord by specifying its *fundamental pitch*, that is the lowest note in the chord, and an (*n* - 1)-tuple that gives its *intervallic content*, that is the width of the intervals between the notes of the chord in semitones. In essence, the fundamental provides us with a reference point from which we can calculate the remaining notes from the chord's co-ordinates in an (*n* - 1)-dimensional Euclidean space known as the *von-Neumann music space* ([Miranda, 1993], [Miranda, 1994]).

We use the notation

$$X(x_{1,2}, x_{2,3}, \dots, x_{n-1,n})$$

to denote the *n*-note chord whose fundamental pitch is X, with the semitone interval between the lowest and next-lowest pitches being $x_{1,2}$, the semitone interval between the second lowest and next-lowest pitches being $x_{2,3}$ and so on.

¹⁷ The automaton is analysed in this way to remain backwardly compatible with Dr Miranda's original system.

¹⁸ We use the term *checked* here to denote the examination of a cell's state for the purpose of generating musical data and not for the updating of the automata. This terminology is retained throughout the thesis.

Thus we have that the three examples of a 3-note chord above may all be described as $A4(8, 2)$, since the fundamental pitch is $A4$ (i.e. the fourth A above $C1$, the lowest C on the piano keyboard), the interval from $A4$ to $F5$ is 8 semitones and the interval from $F5$ to $G5$ is 2 semitones.

In many instances we are concerned only with the chord type, such as major or minor. If this is so, we may omit the fundamental pitch from the notation, since this is simply a reference point and all the information about the chord type is stored in the $(n - 1)$ -tuple that follows.

Table 3.1.1 below shows some common (3-note) chord types and the corresponding integer pairs.

	Major	Minor	Augmented	Diminished
Root position	(4, 3)	(3, 4)	(4, 4)	(3, 3)
First inversion	(3, 5)	(4, 5)	(4, 4)	(3, 6)
Second inversion	(5, 4)	(5, 3)	(4, 4)	(6, 3)

Table 3.1.1 – Some common 3-note chord types in duple notation.

3.1.6 From automata to music

Converting the cells of the Game of Life to music is accomplished as follows:

When CAMUS arrives at a live cell, its co-ordinates are taken to be the duple representation of a 3-note chord.

The state of the corresponding cell of the Demon Cyclic Space automaton is used to determine the *instrumentation*, that is the instrument that 'plays' the cell, of the piece. This configuration is demonstrated in Figure 3.1.4. In this case, the cell in the Game of Life at (5, 5) is alive, and thus constitutes a sonic event. The corresponding cell in the Demon Cyclic Space is in state 4, which means that the sonic event would be played by instrument number four (e.g., using MIDI channel 4). The co-ordinates (5, 5) describe the intervals in a triad: the note five semitones above some fundamental pitch, and the note ten semitones above the fundamental.

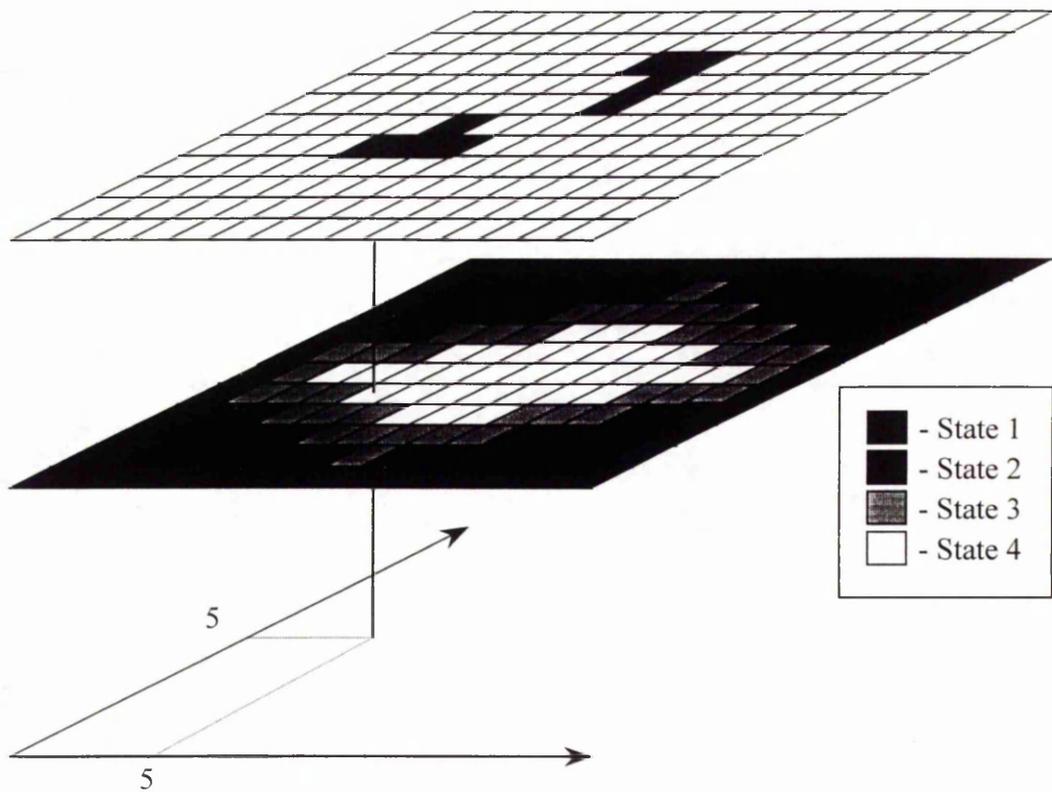


Figure 3.1.4 - Configuration for a typical timestep of the cellular automata music algorithm used in CAMUS.

Now, in order to specify fully the triad, we also require a fundamental pitch to use as a reference point. CAMUS employs a user-specified parameter list known as an *articulation* for this.

An articulation consists of a twelve-pitch sequence of notes that serve as the fundamental pitches for the triads, and a number of other composition-specific parameters. These are:

Speed. This allows the user to specify the tempo of the composition in beats per minute (BPM).

Speed Variation. This allows the user to specify a rubato setting which will vary the basic tempo by up to \pm this number of BPM.

Dynamics. This allows the user to specify the intensity of the generated music. The dynamic level is given as a MIDI note velocity in the range 0 – 127.

Dynamics Variation. This allows the user to specify a dynamic range. The dynamic level will then vary by up to \pm this value.

Random Number Generation. This allows the user to alter the random number generation settings used in the calculation of the speed and dynamic variation values in the composition. The user can specify the maximum and minimum numbers returned by the random number generation and the type of distribution used. The three distribution types are *uniform*, meaning that each real number in the range [0, 1] is equally likely; *linear*¹⁹, meaning that real numbers closer to 1 are much more likely than those close to 0, and *triangular*²⁰, which returns numbers close to 0.5 far more often than those close to 0 or 1. Note that the uniform and linear distributions described above are only one example of each of these distribution types. They are, however, hard-coded into the system and are a legacy from CAMUS v1.0 for Atari ST ([Miranda, 1993]) from which this version was developed.

¹⁹ Note that this is a particular example of a linear distribution. This distribution was chosen by Dr. Miranda and is hard-coded into the system.

²⁰ Similarly, this is a particular example of a triangular distribution which is coded into the system.

Number of Loops. This allows the user to specify the number of *loops* that CAMUS uses in the generation of a composition. A loop is a twelve-triad long section of the composition to which the user may assign any articulation. CAMUS allows for up to 9999 loops.

Use Articulation x for Loop y . This allows the user to associate any articulation with any loop.

CAMUS allows for the specification of up to 22 such articulations, which, combined with the ability to associate the articulations with any of the available loops gives a fairly wide scope for composition.

The articulation system described above, is a development of the system employed in CAMUS v1.0 for Atari ST. The number of parameters-per-articulation and their usage derives from this earlier version.

Once the triad for each cell has been determined CAMUS calculates the note velocities for each of the notes by generating a random number in the range specified by the user, scaling by the dynamics variation parameter and adding (or subtracting) to the dynamics parameter.

Now in order to avoid the composition being composed entirely of block chords, we need some method of staggering the starting (and possible ending) times of the notes of the triad. CAMUS uses the states of the neighbouring cells in the Game of Life to calculate the temporal position and duration of each note as follows:

Suppose we wish to determine the temporal positioning of the cell (x, y) . We can construct a set of values from the states of the neighbouring cells – the value being 1 if the cell is alive and 0 if it is dead:

$$\begin{aligned}
a &= \text{cell}(x, y - 1) & b &= \text{cell}(x, y + 1) \\
c &= \text{cell}(x + 1, y) & d &= \text{cell}(x - 1, y) \\
m &= \text{cell}(x - 1, y - 1) & n &= \text{cell}(x + 1, y + 1) \\
o &= \text{cell}(x + 1, y - 1) & p &= \text{cell}(x - 1, y + 1)
\end{aligned}$$

We then form the four 4-bit words, *abcd*, *dcba*, *mnop* and *ponm*. Next, we perform the bitwise inclusive OR operation, '|', to form two four-bit words, *Tgg* and *Dur*:

$$\begin{aligned}
Tgg &= abcd \mid dcba \\
Dur &= mnop \mid ponm
\end{aligned}$$

From *Tgg*, we derive the note trigger information, and from *Dur*, the note duration information. With each relevant four-bit word, we associate a code, known as an **AND** (cellulAr geNetic coDe). Note that the **AND** code is distinct from the bitwise AND operator. The author concedes that the terminology may cause some confusion, but has been retained here for consistency with Dr. Miranda's original program. Thus, for the sake of clarity, when referring to the **AND** code throughout this thesis, we use bold typeface. Logical operations will always be printed with plain typeface.

Each of the three letters in the **AND** codeword is used to represent a note in the triad, with **a** denoting the lower pitch, **n** the middle pitch, and **d** the upper. The square brackets are used to indicate that the note events contained within that bracket occur simultaneously. The codewords are assigned as follows:

0000 : a[dn]	0001: [dna]	0010 : adn	0011 : dna	0101 : and
0110 : dan	0111 : nad	1001 : d[na]	1011 : nda	1111 : n[da]

Table 3.1.2 – The AND codewords and their respective numerical values.

With each codeword we associate a particular temporal configuration, as shown in Figure 3.1.5.

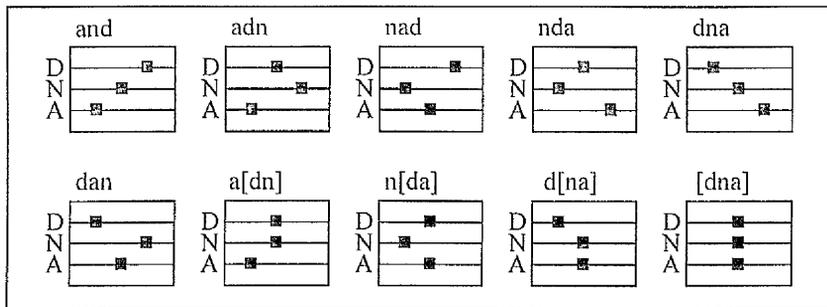


Figure 3.1.5 - Ten different temporal codes.

These codes determine the *temporal shape* of each triad, the actual values for the trigger and duration parameters are calculated using a random number generator and the distribution and upper and lower bounds specified in the relevant articulation.

Finally, the music is written to a storage file and sent to a MIDI-equipped instrument to be played.

Figure 3.1.6 below illustrates the principal steps of the CAMUS algorithm in the form of a flowchart.

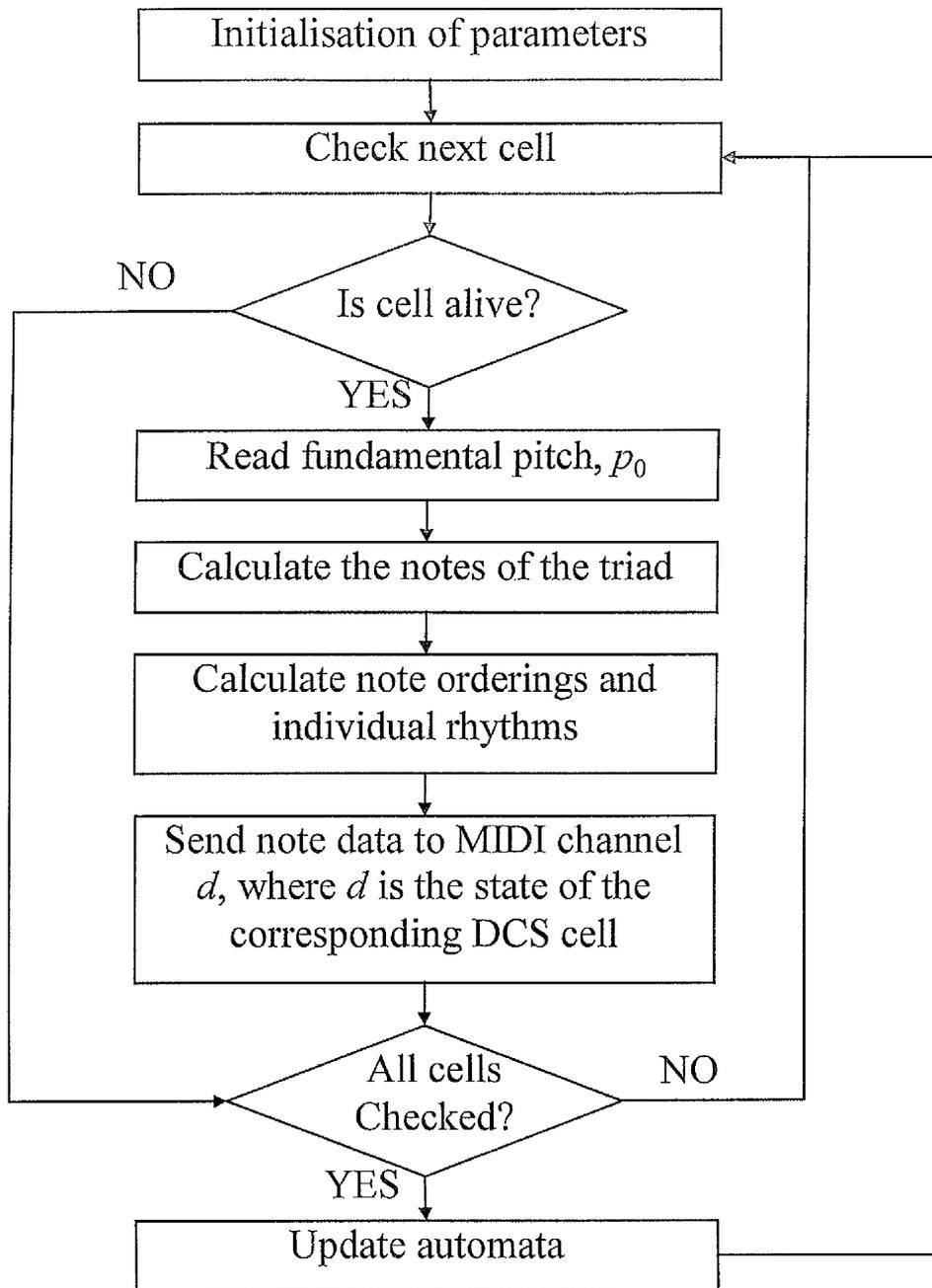


Figure 3.1.6 – The CAMUS algorithm.

Figure 3.1.7 expands further the step labelled *Calculate the notes of the triad*.

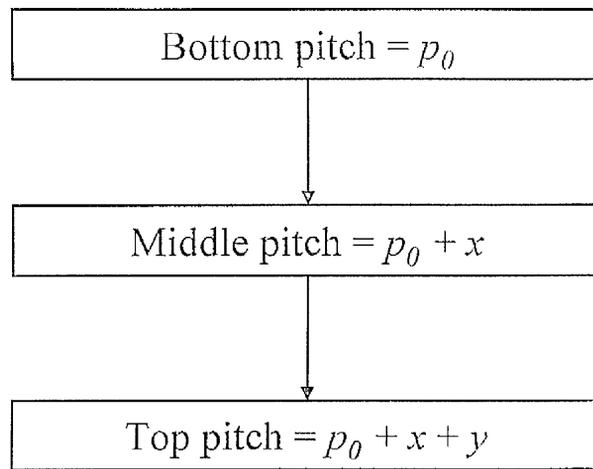


Figure 3.1.7 – Algorithm for calculating the notes of the triad defined by cell (x, y) and fundamental pitch p_0 .

In Figure 3.1.7, we have used a one-to-one correspondence to map the chromatic pitches from C-2 through to G8 onto the integers from 0 to 127. An increase in pitch of a semitone corresponds to an increase by 1 of the relevant note number. This mapping forms the basis of the MIDI representation and is discussed in more detail in Appendix C.

In Figure 3.1.8a below, we illustrate how the parameters a , b , c , d , m , n , o and p are assigned values depending on the states of the cells neighbouring the one under examination. The value assigned to the parameter is taken to be 1 if the corresponding cell is alive and 0 otherwise.

It is also important to note that the neighbouring cell co-ordinates are calculated modulo $Size$, where $Size$ is assigned the integral value m for an $m \times m$ automaton. This ensures that the automaton space is *toroidal*, that is, each edge of the automaton space wraps around to meet the opposite edge.

From these parameters, we form the four four-bit binary words $abcd$, $dcba$, $mnop$ and $ponm$ and calculate the trigger and duration parameters, Tgg and Dur by performing the bitwise inclusive OR operation.

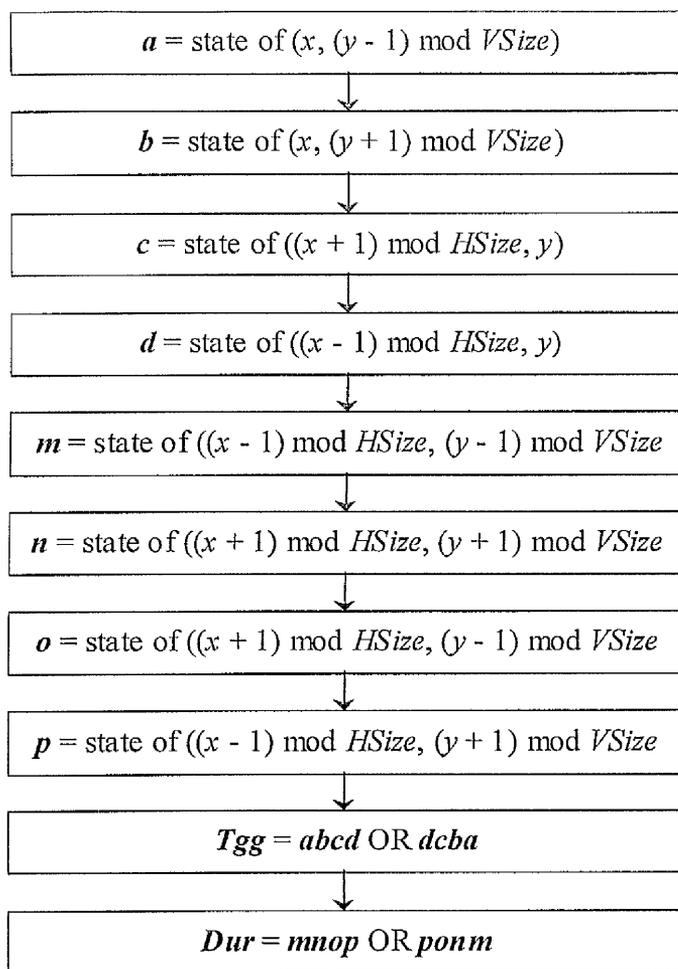


Figure 3.1.8a – Initialisation of parameters for calculating the note orderings of the triad defined by (x, y) in a toroidal Game of Life automaton of size $(Hsize, Vsize)$.

In Figures 3.1.8b, c, d and e, we show how the trigger and duration parameters are used to select the note orderings and durations according to their respective values. The values used in the decision-making routines were chosen subjectively and then hard-coded into the system.

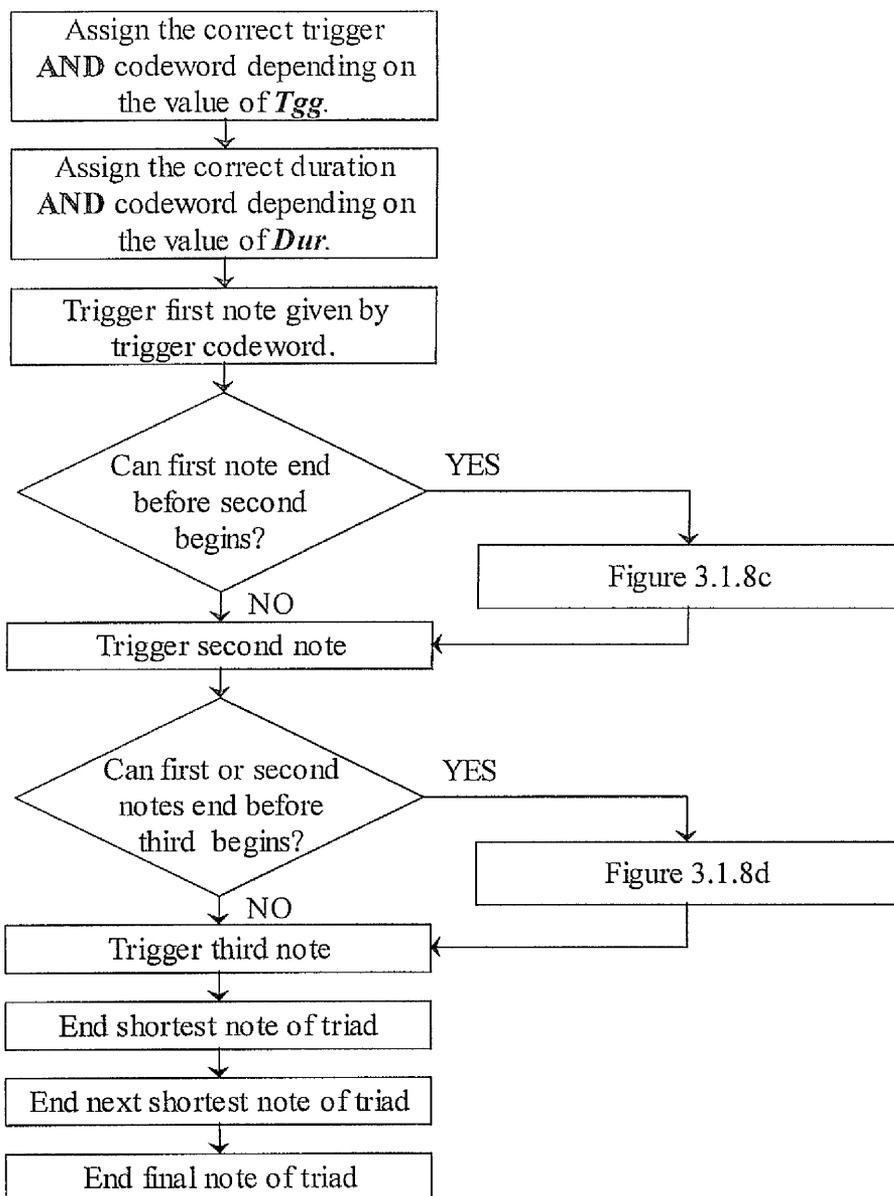


Figure 3.1.8b – Decision routine based on the *Tgg* and *Dur* parameters.

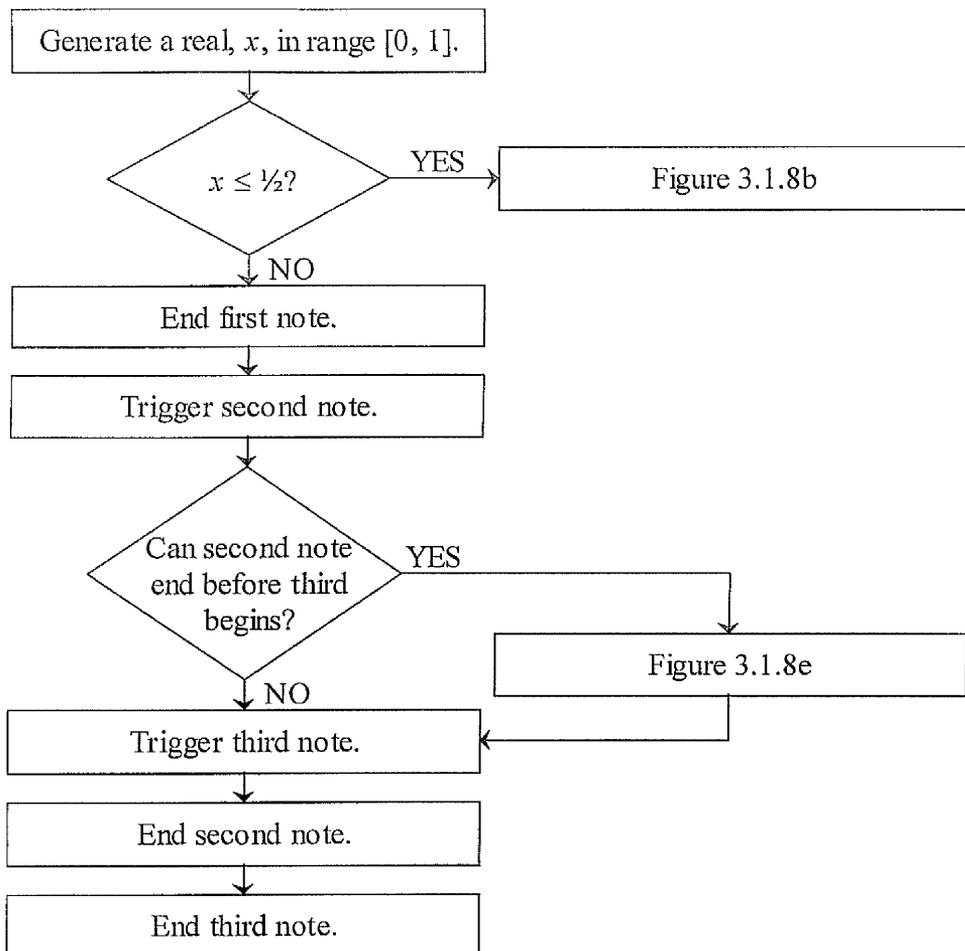


Figure 3.1.8c – Decision routine based on the *Tgg* and *Dur* parameters.

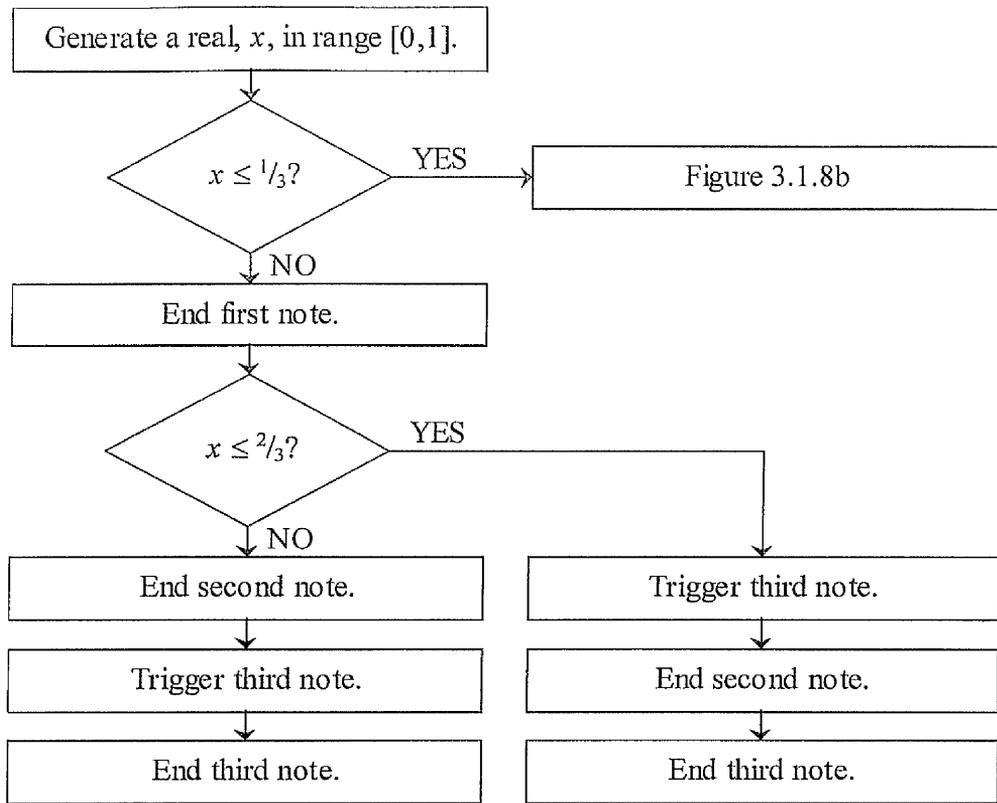


Figure 3.1.8d – Decision routine based on the *Tgg* and *Dur* parameters.

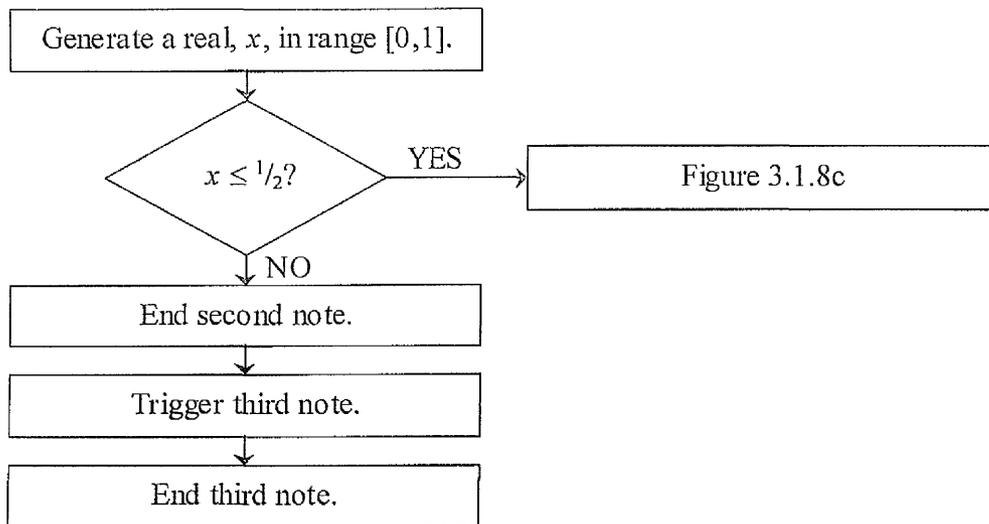


Figure 3.1.8e – Decision routine based on the *Tgg* and *Dur* parameters.

It should be noted that this decision making routine also works correctly even if the notes are not strictly ordered (i.e. two or more sound simultaneously). Here, we still order the notes, but allow starting times and durations of value 0. This works because MIDI is serial in nature and commands must be sent one at a time (see Appendix C).

3.2 Composition example

We now illustrate the workings of the CAMUS algorithm by tracing the development of a simple composition using CAMUS version 2.0 for Windows95 [McAlpine, Miranda & Hoggar, 1997a], [McAlpine, Miranda & Hoggar, 1997b], which was developed from CAMUS version 1.0 for Atari ST [Miranda, 1993].

3.2.1 The interface

CAMUS has three main windows from which all of the functionality of the program can be accessed. The first of these is the *CAMUS toolbar*:

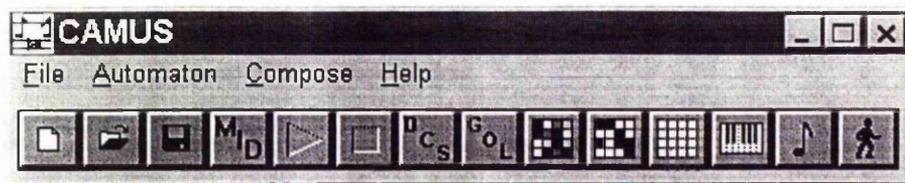


Figure 3.2.1 – The CAMUS toolbar.

This is the main ‘control centre’ for CAMUS, allowing the user to load and save files, stop and start the composition process, and change all of the available parameters. Reading from left to right, the buttons are:

New. Creates a new CAMUS project.

Open. Opens a previously saved CAMUS project.

Save. Saves the current CAMUS project to disk.

Export MIDI file. Saves the current CAMUS composition as type 0 standard MIDI file (see Appendix C).

Go. Begins the composition process and plays the generated music in real-time.

Stop. Terminates the composition process and silences any music that is currently playing.

Alter Demon Cyclic Space Rules. Allows the user to customise the rules that determine the cellular evolution of the Demon Cyclic Space automaton.

Alter Game of Life Rules. Allows the user to customise the rules that determine the cellular evolution of the Game of Life automaton.

Load Demon Cyclic Space with Random Values. Assigns each cell of the Demon Cyclic Space automaton a random value between 0 and $n - 1$, where n is the total number of available states.

Load Game of Life with Random Values. Assigns each cell of the Game of Life automaton with a random binary value.

Clear Game of Life Cells. Kills each cell of the Game of Life automaton.

Change Instrumentation. Allows the user to associate General MIDI (see Appendix C) instruments with each of the cells of the Demon Cyclic Space automaton.

Change Composition Settings. Allows the user to alter the articulation parameters (see Section 3.1.6) that control the main musical aspects of the composition.

Step Through. The system updates the automata once and stops, generating and playing music as it does so.

The second of the three main windows is the *Game of Life window*, which displays the states of the cells in the Game of Life automaton:

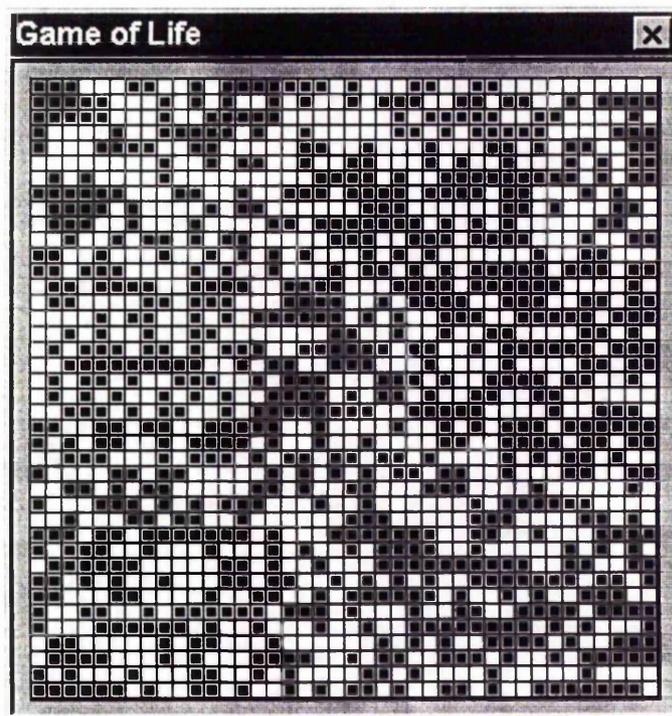


Figure 3.2.2 – The Game of Life window.

A good way to think of this window is as CAMUS's musical score. This analogy is fairly natural in that both a musical score and the Game of Life window display information about musical events in a graphical form. Here, though, the live (shaded) cells that are responsible for the musical events should be thought of as chords rather than individual notes as in a traditional score.

The state of any cell in the Game of Life window can be altered by clicking on it. In addition, the Clear and Randomise buttons described above can be used to alter the states of all of the cells in the automata at once.

The last of the three main windows is the *Demon Cyclic Space window*, which displays the states of the cells in the Demon Cyclic Space automaton:

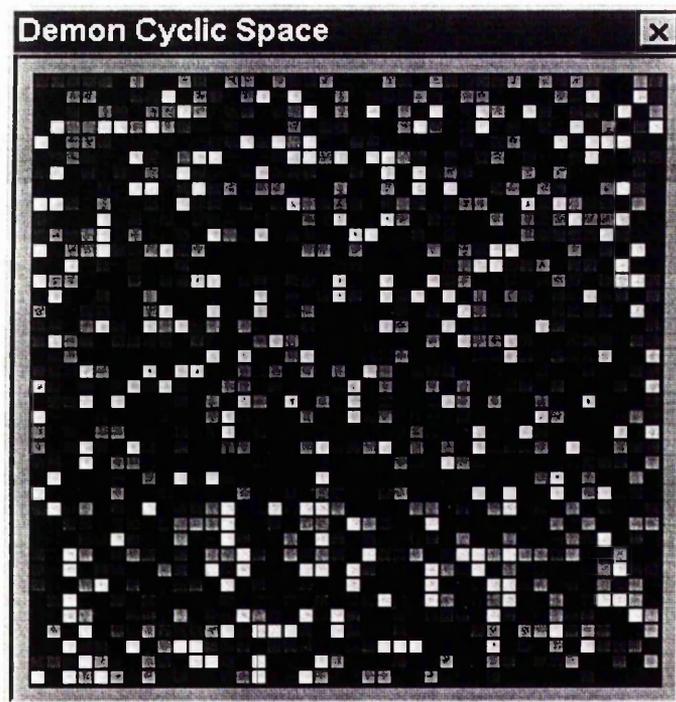


Figure 3.2.3 – The Demon Cyclic Space window.

The state of each cell in the Demon Cyclic Space is indicated by its colour²¹. The user cannot directly alter the states of the cells in the Demon Cyclic Space, but can randomise them using the Randomise button described above.

Since the cells in the Demon Cyclic Space determine which instrument plays the notes generated by the corresponding cell in the Game of Life, we have that the number of states in which a Demon Cyclic Space cell can exist defines the number of instruments that will play any music that is generated.

²¹ Due to the monochrome printing capabilities available to the author, it has been impossible to display many figures in colour. Instead we use greyscale to display shades. The computer software, however, does display these screens in full colour.

3.2.2 Changing the Demon Cyclic Space rules

We now begin work on the composition. Let's suppose that we wish to write a short piece for four instruments. First of all, we must alter the Demon Cyclic Space rules so that each cell can exist in one of four possible states.

The Demon Cyclic Space Rules window (see Figure 3.2.4) is displayed by clicking on the *Alter Demon Cyclic Space Rules* button on the CAMUS toolbar or by selecting the corresponding menu option.

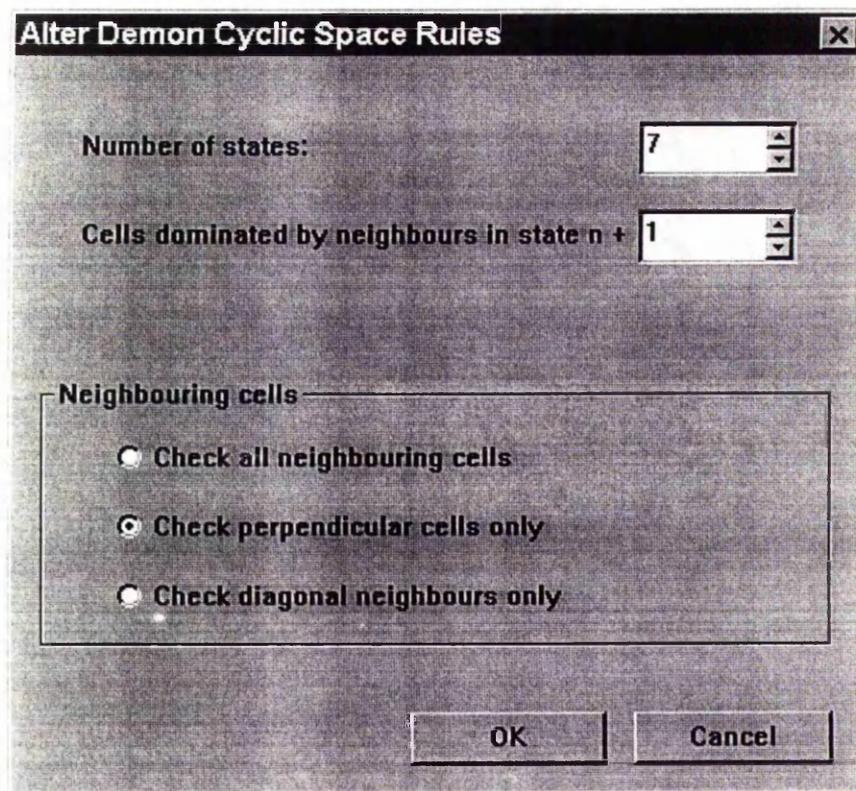


Figure 3.2.4 – The Alter Demon Cyclic Space Rules window.

In order to ensure that the music that is generated is played on four instruments, the *Number of states* parameter is set to 4, as shown in Figure 3.2.5 below.

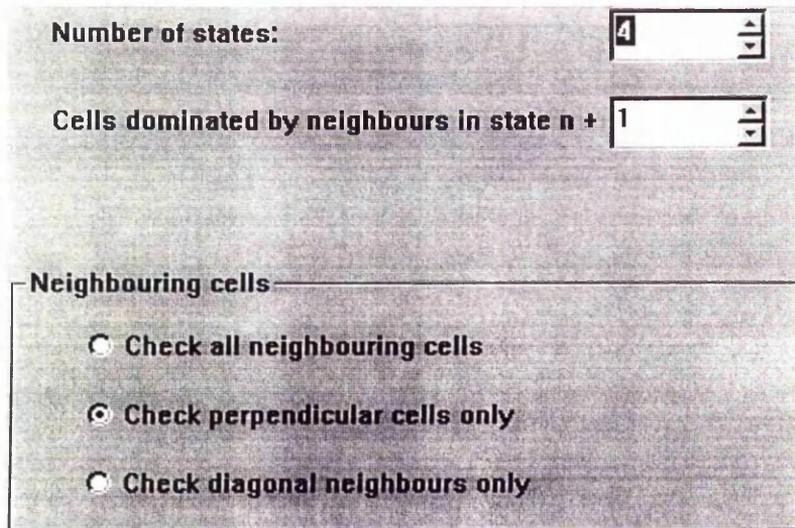


Figure 3.2.5 – Altering the number of instruments by limiting the number of possible states in the Demon Cyclic Space.

The other options allow the user to change the way that the Demon Cyclic Space evolves. The parameter *Cells dominated by neighbours in state $n +$* , displayed in Figure 3.2.5 above, can be set to any value between 1 and the total number of states, and specifies which cells will dominate their neighbours at each timestep. The *Neighbouring cells* option allows the user to select one of three neighbourhoods to be examined at each timestep. The options are:

Check all neighbouring cells examines the states of all the cells surrounding the cell currently being examined.

Check perpendicular cells only examines only the cells directly above, below, to the left and to the right of the cell currently being examined.

Check diagonal cells only examines only the cells above-left, above-right, below-left and below-right of the cell currently being examined.

These options alter the patterns (and thus the music) produced by the automata.

3.2.3 Changing the Game of Life rules

It also is possible that the user might wish to make changes to the Game of Life rules.

The window, *Game of Life Rules* (see Figure 3.2.6 below) is displayed by clicking on the *Alter Game of Life Rules* button on the CAMUS toolbar or by selecting the corresponding menu option.

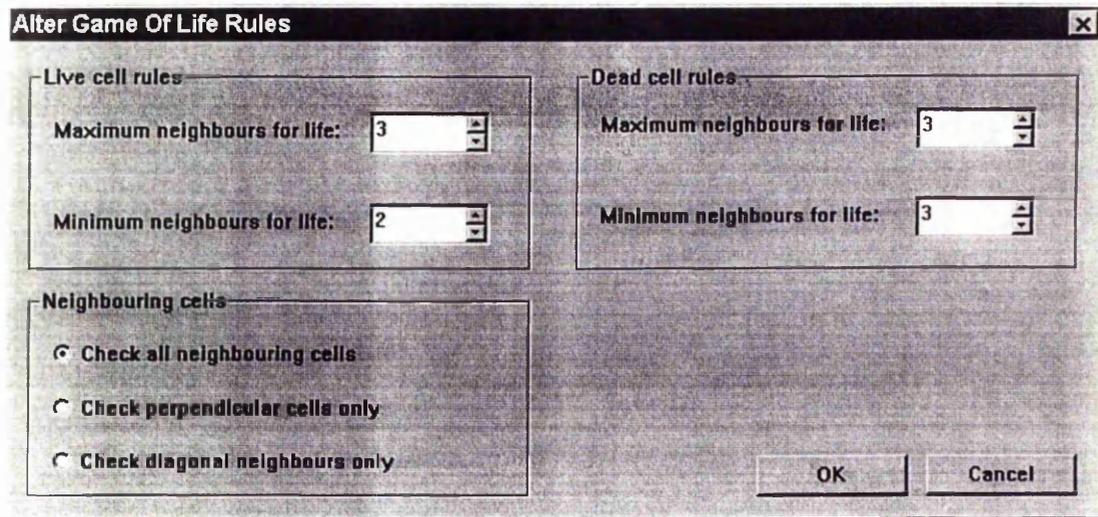


Figure 3.2.6 – The Alter Game of Life Rules window.

The *Maximum neighbours for life* parameters in the *Live cell rules* and *Dead cell rules* boxes determine the maximum number of live neighbours that can surround a live or dead cell and yet preserve or create life on the next timestep.

Similarly, the *Minimum neighbours for life* parameters determine the minimum number of live neighbours that can surround a live or dead cell and yet preserve or create life on the next timestep.

The settings in Figure 3.2.6 above signify that any live cells will continue to live if they have either two or three live neighbours (not including the cell being checked), and any dead cells become live if they have exactly three live neighbours. All other cells will die or remain dead.

As with the Alter Demon Cyclic Space Rules window, the Neighbouring cells box allows the user to specify precisely which neighbours are examined at each timestep.

3.2.4 Setting the instrumentation

Next, the user should decide on the instruments that will perform the composition. The *MIDI Instrumentation Setup* window (see Figure 3.2.7) can be opened by clicking on the *Change Instrumentation* button on the CAMUS toolbar.

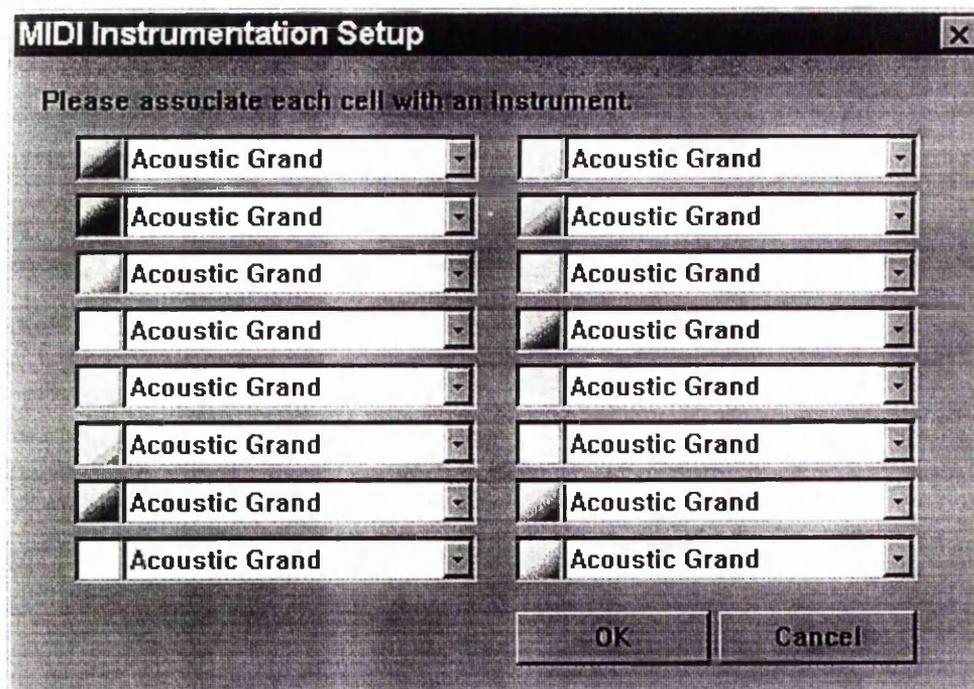


Figure 3.2.7 – The MIDI Instrumentation Setup window.

Suppose that the composer wishes the piece to be performed by piano, string section, lead synth tone and synth pad tone. The first four list boxes, corresponding to MIDI channels 1 – 4 would then be set to something similar to those of Figure 3.2.8:



Figure 3.2.8 – Changing the MIDI instruments of the first four MIDI channels.

Notice, here, that the user is still free to alter the settings for any of the other instruments, and indeed, patch change data will be sent for each, but this will have no effect on the composition, because only the first four instruments are configured for playback.

Note also, that although we are, in effect, assigning instruments to each of the 16 available MIDI channels, no mention is made of channel assignments in the dialog box. The interface was deliberately designed like this so as to make those musicians who are uncomfortable with the MIDI protocol feel more at ease with the software. It is a simple matter to associate the instrument names with the colours of the corresponding Demon Cyclic Space cells. CAMUS then takes care of the channel assignments in the background, leaving the user to get on with composing the music.

3.2.5 Setting the articulations

The next step in the composition process is to configure the parameters that determine the actual notes that CAMUS performs. To do this, the user must open the *Change Composition Settings* window (see Figure 3.2.9). This can be achieved by clicking on the *Change Composition Settings* button on the CAMUS toolbar or by selecting the corresponding menu option.

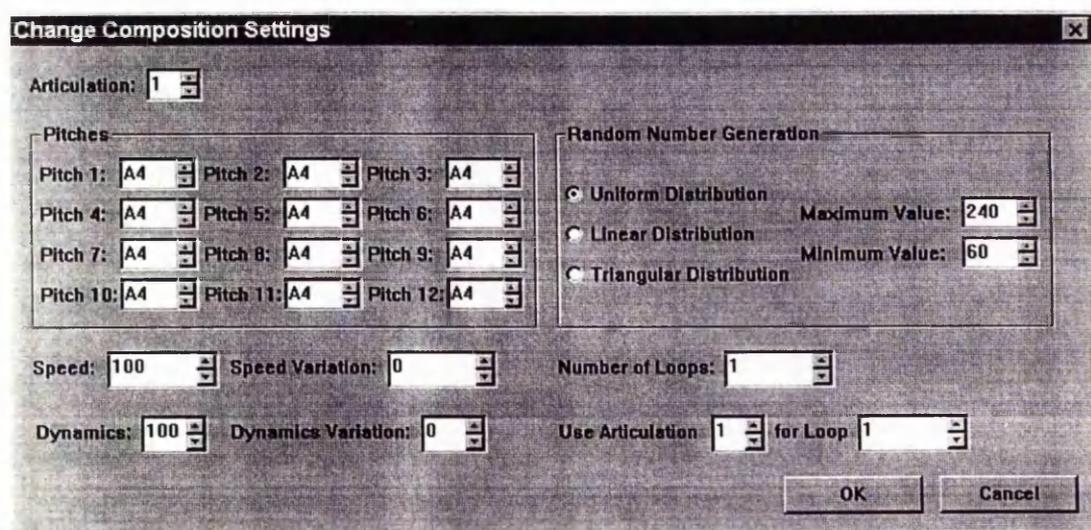


Figure 3.2.9 – The Change Composition Settings window.

The Change Composition Settings window has a number of parameters that can be altered. Let us consider them one at a time.

The first parameter we see is the one marked *Articulation* in the top left corner of the window. An articulation is simply a convenient way of referring to all of the parameters that you can set on this page. A single articulation consists of a 12-pitch sequence, parameters for controlling the speed and the speed variation, the dynamics (i.e. the loudness or softness of the notes) and dynamic variation, and parameters for controlling the random number generation employed by the composition algorithm. CAMUS allows for up to 22 such articulations which can be played back repeatedly and in any order.

The next step for the user is to define the 12-pitch sequence for the first articulation. The pitch values are displayed textually in the form Xn , where X denotes the pitch name (e.g. C#) and n is an integer denoting MIDI octave number (see Appendix C). The values may be changed by clicking on the text and typing in a new value, or by clicking on the increment and decrement buttons at the side of each text box to raise or lower the pitch by a semitone. The result of such an action may look like Figure 3.2.10 below.

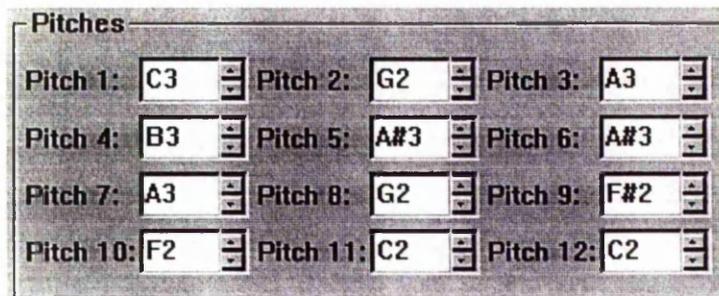


Figure 3.2.10 – Altering the twelve-pitch sequence of an articulation.

The next parameters we meet are called *Speed* and *Speed Variation*. Speed allows the user to set the tempo of the generated music in beats per minute (BPM). The Speed Variation parameter is a *rubato* control which allows the user to vary the speed of the composition by up to \pm Speed Variation BPM. The values in Figure 3.2.11 below indicate that the underlying tempo of the composition is 100 BPM, with a rubato parameter of 30 BPM, giving tempos in the range [70, 130] BPM.



Figure 3.2.11 – Setting the speed and rubato parameters.

Again, the values may be altered by either clicking on the text and typing a new value, or by clicking on the increment and decrement buttons at the side of the text box.

Next, we meet two parameters named *Dynamics* and *Dynamics Variation*, which allow the user to alter the dynamic level of the instruments and the amount by which this level can vary. The parameters are stored in MIDI volume format, and so range from 0 to 127. Thus, setting the values as in Figure 3.2.12 below, we would have a base note volume of 90, with a variation of ± 40 .



Figure 3.2.12 – Setting the dynamic level and range.

However, since, as mentioned above, the maximum value MIDI allows for note volume commands is 127, it means that the dynamic range for the articulation is actually [50, 127].

Because of this possible source of integer overflow, boundary checking must be employed not only for this particular case, but for all of the composition parameters.

In the top right of the articulation window, we see a box labelled *Random Number Generation* (see Figure 3.2.13). This is where the user can alter all of the settings concerned with random number generation.

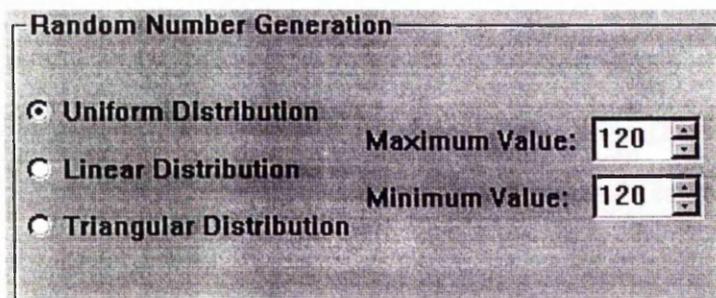


Figure 3.2.13 – Random number control.

Random numbers feature prominently in CAMUS. They form the basis of all of the decision-making routines, note start-time and duration routines and dynamic and tempo variation routines. The random numbers generated by these variables form the starting times and durations of the note events that are generated elsewhere. CAMUS works at a note resolution of 120 ppn (pulses per quarter note). That is, each unit of time corresponds to 1/120th of a crotchet.

Here, we are presented with three radio buttons marked *Uniform Distribution*, *Linear Distribution*, and *Triangular Distribution*, which uses the appropriate distribution to generate the random numbers. To the right of these buttons are two edit boxes marked respectively *Maximum Value* and *Minimum Value* which allow the user to specify the maximum and minimum values returned by the random number generation routine.

Thus the settings of Figure 3.2.13 above would result in the random number routine always returning a value of 120 (i.e. a crotchet), since the upper and lower bounds for the routine are identical.

In the bottom right corner of the screen, there are a set of parameters labelled *Number of loops*, and *Use articulation ... for loop ...* (see Figure 3.2.14). These parameters allow the user to control the order in which the articulations are performed and are independent of the settings of each of the 22 available articulations.

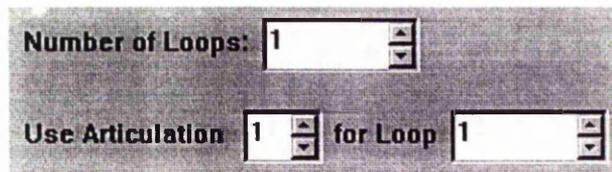


Figure 3.2.14 – Ordering the articulations.

A *loop* is simply a measure of time which lasts for the duration of one articulation. CAMUS allows up to 9999 of these. Let us suppose that the user wishes to have four loops in the composition. The *Number of loops* parameter would then be set to 4, as shown here:

Number of Loops: 4

Figure 3.2.15 – Changing the number of loops.

At present, each of the four available loops will have the articulation 1 associated with them by default. In order to introduce some movement, the user must associate different articulations with each loop. For example, in Figure 3.2.16, we have linked articulation 2 with the second loop.

Use Articulation 2 for Loop 2

Figure 3.2.16 – Altering the articulation order.

Let us suppose that the user now associates articulation 2 with the third loop, and articulation 1 with the fourth in exactly the same way. This means that the order of play will be articulation 1 for the first loop, articulation 2 for the second loop, articulation 2 for the third loop, and articulation 1 for the fourth. Once the fourth loop has been performed, CAMUS will return to the beginning and start again with loop 1.

Once the user has specified the second articulation, as shown in Figure 3.2.17 below, the parametric specification of the system is complete.

Pitches

Pitch 1: B2 Pitch 2: A#2 Pitch 3: A2
 Pitch 4: A2 Pitch 5: A2 Pitch 6: C3
 Pitch 7: B3 Pitch 8: G2 Pitch 9: F3
 Pitch 10: F#3 Pitch 11: G3 Pitch 12: A4

Speed: 160 Speed Variation: 35

Dynamics: 60 Dynamics Variation: 20

Random Number Generation

Uniform Distribution
 Linear Distribution
 Triangular Distribution

Maximum Value: 240
 Minimum Value: 120

Figure 3.2.17 – Settings for the second articulation.

3.2.6 – Initialising the Game of Life

Finally, in order for CAMUS to actually produce music, the user must draw an initial configuration of cells in the Game of Life window. As mentioned in Section 3.2.1, this is achieved by left-clicking and dragging the mouse over the Game of Life window.

Let us suppose that the user draws the following configuration:

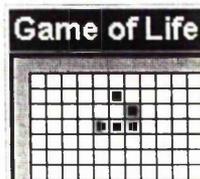


Figure 3.2.18 – Initial cell configuration for the Game of Life.

3.2.7 – Generating the composition

Once all of the composition parameters have been initialised and the initial cell configuration is in place, the composition process can be set in motion.

Once composition is underway, CAMUS begins scanning the Game of Life column by column for live cells. Note that the point of origin in the Game of Life is the top-leftmost cell. The x -axis runs left-to-right, whilst the y -axis runs top-to-bottom. This gives the following cell ordering:

Order	Cell
1	(4, 3)
2	(5, 1)
3	(5, 3)
4	(6, 2)
5	(6, 3)

Table 3.2.1 – Cell ordering for the configuration of Figure 3.2.18.

The first cell that CAMUS encounters is the cell at position (4, 3). By examining the first note of the articulation associated with the first loop, we see that the fundamental pitch of this chord is C3, resulting in the chord C3(4, 3):



Figure 3.2.19 – C3(4, 3) chord generated by the first live cell of the configuration of Figure 3.2.18.

Now, CAMUS calculates the note ordering using the **AND** code.

This results in the parameter values of Table 3.2.2 below.

Parameter	Cell	State
<i>a</i>	(4, 2)	0
<i>b</i>	(4, 4)	0
<i>c</i>	(5, 3)	1
<i>d</i>	(3, 3)	0
<i>m</i>	(3, 2)	0
<i>n</i>	(5, 4)	0
<i>o</i>	(5, 2)	0
<i>p</i>	(3, 4)	0

Table 3.2.2 – *AND* code parameter values.

We now form the trigger and duration codewords, *Tgg* and *Dur*:

$$Tgg = abcd | dcba = 0010 | 0100 = 0110$$

$$Dur = mnop | ponm = 0000 | 0000 = 0000.$$

Comparing these calculated values with the table of codewords presented in Table 3.1.2, we see that the trigger codeword is **dan**, and the duration codeword is **a[dn]**. In other words, the notes are triggered with the order top note, bottom note, middle note, and released with the order bottom note, and the middle and top notes simultaneously.

Now CAMUS calculates the start times and durations for each of the notes of the chord using the random number parameters defined in the current articulation (number 1). These are set to return the value 120 each time the random number procedure is called. Also, the dynamic level is set to give a base velocity of 90 with a range of 40. This results in start times, durations and note velocities for the first chord as given in Table 3.2.3 below.

Pitch	Start Time	Duration	Velocity
C	120	480	119
E	240	120	56
G	480	120	105

Table 3.2.3 – Note Parameters for each of the notes of the chord generated by live cell (4, 3). Values are given as absolute times.

The resulting chord is presented in Figure 3.2.20 below.



Figure 3.2.20 – The resulting chord obtained from live cell (4, 3) in the Game of Life.

The corresponding cell of the Demon Cyclic Space is in state 2, meaning that the note information is sent to MIDI channel 2 for playback.

Continuing in this manner results in the following:

Figure 3.2.21 – Music generated by performing two steps of the CAMUS algorithm with composition parameters set as described in the text.

3.3 Towards an efficient working procedure in CAMUS

As we have seen, CAMUS relies on the evolution of two deterministic cellular automata to generate musical data. Thus, in order for us to discuss the musical output of the CAMUS system, it is first necessary to consider the effects that different initial configurations and evolution rules have on the system's development and output.

In this section, we consider several different starting configurations, and trace the resulting pattern propagation. We then discuss the implications for the musical output.

3.3.1 Two important considerations

There are essentially two things to be considered when examining CAMUS' output.

Firstly, one must consider the triads that are generated. It is desirable to generate a range of triads, both consonant and dissonant in a work. There are many complex inter-relations between triads that should be present if the output is to be musically pleasing. At present, CAMUS offers no facilities for manually imposing restrictions on the output, and so if we are desirous of producing music that exhibits some of these triadic relations, it must be through careful choice of our starting configuration.

Secondly, one must consider the effect that applying different articulations to the triadic sequence obtained from a particular configuration has on the output. The music generated by CAMUS is essentially atonal. However, by analysing the triadic progressions that result from a given configuration, one can choose pitch sequences for the articulations such that a tonal quality is imposed on the music. This will be discussed at length in Section 3.3.7.

3.3.2 Some terminology

When discussing the musical output that arises as a result of the evolution of the automata, it will prove useful to introduce terminology relevant to the control mechanism. When composing using cellular automata, we are essentially treating music as a means of pattern propagation (see Section 1.3.1). Thus, in using geometric terms to describe the states and transformations of the automata, we are also describing the transformations that are applied to the musical output. We will speak of reflections, rotations and clusters of cells, and associate each with a traditional musical device.

3.3.3 Initial states of the game of Life

Here, we introduce a number of different starting configurations for the Game of Life, and trace the development of each through to its conclusion. As we saw in Section 2.2.8, there are essentially three different classes of starting configuration in the Game of Life – configurations that die out completely after a number of timesteps,

configurations that translate across space, and configurations that reach a steady cyclic state. We refer to these categories as *doomed*, *gliders* and *cyclic* respectively.

3.3.4 Doomed cells and their musical legacy

The single cell

Single live cells (see Figure 3.3.1) in CAMUS allow the user complete control over the tonality of the resulting composition. By positioning a live cell at position (x, y) , the user can generate precisely one triad, with exactly the intervallic content that he or she desires. With the default evolution rules, this solitary cell will die on the next timestep, and so no further note data will be generated until the next live cell is selected. Using this single-cell-per-timestep technique, the user can generate lengthy chord progressions, which evolve exactly as desired.



Figure 3.3.1 – Single live cell

This high level of control over tonality is, however, not without cost. Recall that the temporal aspects of the composition are determined by the states of the neighbours of the live cell. Thus, since a single live cell has no live neighbours, this means that both the trigger parameter and the duration parameter will have a value of

$$0000 \mid 0000 = 0000,$$

meaning that both the start shape and duration shape will be constant with codeword **a[dn]** for each triad generated.

While it is true that in practice there will be timing variations in the performance of the triads due to the random selection routines utilised in the algorithm, the effects are negligible when compared to the repetitive nature of the note orderings.

It is, of course, entirely feasible to alter manually the note data generated by CAMUS, and impose one's own note orderings on the composition. However, since the purpose of this section is to consider the effect that different cell configurations have on the

output from the composition system, this technique is outside the scope of our discussion.

The use of single live cells as the basis for a CAMUS composition is then a technique that offers excellent user controllability over the tonal aspects of the piece, but which generates note data that quickly becomes monotonous, due to the constantly repeating nature of the note orderings.

Cell pairs

The second type of doomed configuration we consider is a pair of adjacent cells (see Figure 3.3.2).



Figure 3.3.2 – Adjacent pair of live cells

This configuration shares many of the properties of the single cell. For example, in both cases, no cells remain living after just one timestep. As mentioned above, this allows the user a great deal of controllability over the tonality of the music, since the user is free to choose cell positions that fit the general mood of the music being composed. This freedom is inhibited slightly, in that here, the user must choose pairs of cells, but in so doing, we arrive at one major advantage over dealing with single cells, namely that we now have some temporal variation in the triads generated. With the vertical arrangement of cells, this would lead to the following temporal configuration: for the topmost cell, we would arrive at a trigger parameter of

$$T_{gg} = 0100 \mid 0010 = 0110,$$

and a duration parameter of

$$D_{ur} = 0000 \mid 0000 = 0000,$$

corresponding to trigger codeword **dan** and duration codeword **a[dn]**. Similarly, for the bottom cell, we obtain the trigger codeword **d[na]** and duration codeword **a[dn]**.

By changing the alignment of the cell pairs, and carefully choosing the position of the cells in the automaton, different temporal effects and musical triad progressions can be achieved. Table 3.3.1 below shows the four possible arrangements of cell pairs and their corresponding trigger and duration codewords. Cell 1 is always taken to be the top-leftmost of the two.

Orientation	Cell 1 Tgg	Cell 1 Dur	Cell 2 Tgg	Cell 2 Dur
 	a[dn]	dan	a[dn]	d[na]
 	dan	a[dn]	d[na]	a[dn]

Table 3.3.1 – Trigger and Duration codewords for each of the four possible cell pair configurations.

It can be clearly seen from the above table that rotating a cell pair through 45 degrees has the effect of swapping the temporal ordering of that cell’s trigger and duration parameters. The reason for this is that as we rotate through 45 degrees, the neighbouring cell in which the live cell falls varies as:

$a \rightarrow o$ $b \rightarrow p$ $c \rightarrow n$ $d \rightarrow m$
 $m \rightarrow a$ $n \rightarrow b$ $o \rightarrow c$ $p \rightarrow d$

Now, since we are dealing with pairs of cells, only one neighbouring cell will be live. This means that if we begin with

$$abcd = 0000$$

and

$$mnop = 0010$$

as in the first row of Table 3.3.1, the above cell mapping results in the parameters

$$abcd = 0010$$

and

$$mnop = 0000$$

for the next row, and

$$abcd = 0000$$

and

$$mnop = 0100$$

for the next, and so on. It can be seen easily that when the OR operation is performed on these parameters, the swapping motion of Table 3.3.1. is obtained.

3.3.5 Gliders

This configuration of cells is so named because under default conditions, the live cells appear to 'glide' across the automaton. The cell evolution is shown in Figure 3.3.3.

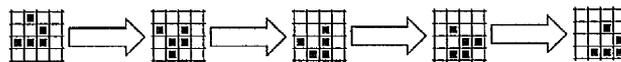


Figure 3.3.3 – Cellular evolution for the Glider configuration

Taking the top left cell of this configuration to have co-ordinate (x, y) , we get the following sequence of chords:

Frame 1	$(x, y + 2)$	Frame 2	$(x, y + 1)$
	$(x + 1, y)$		$(x + 1, y + 2)$
	$(x + 1, y + 2)$		$(x + 1, y + 3)$
	$(x + 2, y + 1)$		$(x + 2, y + 1)$
	$(x + 2, y + 2)$		$(x + 2, y + 2)$
Frame 3	$(x, y + 2)$	Frame 4	$(x + 1, y + 1)$
	$(x + 1, y + 3)$		$(x + 1, y + 3)$
	$(x + 2, y + 1)$		$(x + 2, y + 2)$
	$(x + 2, y + 2)$		$(x + 2, y + 3)$
	$(x + 2, y + 3)$		$(x + 3, y + 2)$

Table 3.3.2 – Sequence of live cells generated by a glider.

When the system reaches frame 5, we return to the same configuration as frame 1, with x and y replaced by $x + 1$ and $y + 1$ respectively.

This glider configuration gives us a cluster of five triads at each timestep. It should be noted here that although the triads are generated in clusters, the ultimate note data that are performed may be very widely dispersed across the musical range. This is due to the mapping technique employed in CAMUS: recall that the system generates only chord *structures* automatically – the fundamental pitches of the triads are user-specified. Therefore, even though two triads are clustered closely together, for example the major triad, (4, 3) and the minor triad (3, 4), if the user specifies widely separated fundamental pitches, such as C1 and G7, the chords will seem very widely spaced.

Every fifth frame, the initial pattern repeats, with the initial co-ordinates incremented by one. This corresponds musically to augmentation of each of the intervals within the triad. Reflecting and/or rotating the initial configuration of cells leads to a repeating pattern of five-triad clusters in which one or both of the internal triadic intervals are diminished.

The actual usefulness of the triad clusters generated here, as for all of the initial conditions presented in this discussion, depends on precisely where the cluster is positioned to begin with. That is, the desirability of a cell configuration is not only pattern-dependent, but also position-dependent.

This is not to say, however, that a given configuration should only be used when positioned within a limited region of the screen. On the contrary, the cell configurations will generate useable chord sequences regardless of where they are positioned on screen, but only a limited range of positions will be useful within a given musical context. Therefore, it is important that the composer who has clear musical ideas that he or she wishes to realise with this system considers not only the triad ordering which is generated, but also the effect that different positionings will have on the resulting musical output.

One possible starting position for the Glider and the corresponding triad type and fundamental pitch are presented in Figure 3.3.3. The initial position was chosen so that $x = 4$ and $y = 1$.

Cell co-ordinates	Chord type	Fundamental pitch
(4, 3)	major, root	60 (C4)
(5, 1)	p4 and passing note	55 (G3)
(5, 3)	minor, 2 nd inversion	57 (A4)
(6, 2)	major 7 th , 1 st inversion	59 (B4)
(6, 3)	diminished, 2 nd inversion	58 (Bb4)
(4, 2)	major, root with diminished 5 th	58(Bb4)
(5, 3)	minor, 2 nd inversion	57 (A4)
(5, 4)	major, 2 nd inversion	55 (G3)
(6, 2)	major 7 th , 1 st inversion	54 (Gb3)
(6, 3)	diminished, 2 nd inversion	53 (F3)
(4, 3)	major, root	48 (C3)
(5, 4)	major, 2 nd inversion	48 (C3)
(6, 2)	major 7 th , 1 st inversion	47 (B3)
(6, 3)	diminished 7 th	46 (Bb3)
(6, 4)	major, 2 nd inversion with diminished 5 th	45 (A3)
(5, 2)	augmented, root	45 (A3)
(5, 4)	major, 2 nd inversion	45 (A3)
(6, 3)	diminished, 2 nd inversion	60 (C4)
(6, 4)	major, 2 nd inversion with diminished 5 th	59 (B4)
(7, 3)	indeterminate 7 th (major or minor)	55 (G3)

Table 3.3.3 – One possible chording of the glider starting configuration.

3.3.6 Cyclic configurations

The broken cross

The broken cross, shown in Figure 3.3.4 below, is a two-state cyclic structure made up of four *blinkers*.

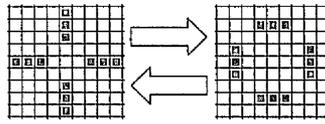


Figure 3.3.4 – Cellular evolution for the broken cross

The broken cross is a structure that arises frequently in the Game of Life, as many cell configurations ultimately arrive at this state before cyclic behaviour begins. The following triadic sequence is obtained from this configuration:

Frame 1	$(x, y + 4)$	Frame 2	$(x + 1, y + 3)$
	$(x + 1, y + 4)$		$(x + 1, y + 4)$
	$(x + 2, y + 4)$		$(x + 1, y + 5)$
	$(x + 4, y)$		$(x + 3, y + 1)$
	$(x + 4, y + 1)$		$(x + 3, y + 7)$
	$(x + 4, y + 2)$		$(x + 4, y + 1)$
	$(x + 4, y + 6)$		$(x + 4, y + 7)$
	$(x + 4, y + 7)$		$(x + 5, y + 1)$
	$(x + 4, y + 8)$		$(x + 5, y + 7)$
	$(x + 6, y + 4)$		$(x + 7, y + 3)$
	$(x + 7, y + 4)$		$(x + 7, y + 4)$
	$(x + 8, y + 4)$		$(x + 7, y + 5)$

Table 3.3.4 – Sequence of live cells generated by the broken cross.

This pattern turns out to have some very musical properties. For instance, in each cell, there are two groups of three cells arranged in a horizontal line, and two groups of three cells laid out in a vertical line. In each case, the horizontal line corresponds to the augmentation of the lower interval in the triad, so that if we began with the minor triad, (3, 4), we would progress through the augmented triad, (4, 4), and finish at the major triad (5, 4). Similarly, the vertical lines correspond to the augmentation of the upper interval in the triad.

It should also be apparent from Figure 3.3.4 that as this configuration evolves, the horizontal groups of cells become vertical groups of cells, and the vertical groups become horizontal. This is essentially a mixture of the familiar musical devices

known as sequence and inversion, although in this case, the devices are applied to the triadic development, rather than to the triads themselves.

To see this, first notice that the act of augmentation of the triads described by the horizontal lines and that described by the vertical lines follow a pattern: first we have a triad, whose lower interval is augmented twice, then we are presented with a triad whose upper interval is augmented twice. This may be thought of as a sequence of two augmentations. Now we are given another triad whose upper interval is augmented twice and then another whose lower interval is augmented twice. Once more, we have a sequence of augmentations, but this time, the sequence is the inverse of the first, in the sense that the interval ordering is altered. We now consider this as a sequence of augmentations.

In the next frame, we are presented with a further sequence: a triad whose upper interval is augmented twice, two triads whose lower intervals are augmented twice, and finally, a further triad whose upper interval is augmented twice. It is easy to see that this, in turn is simply the inverse of the sequence of four augmentations from the previous frame.

One possible drawback to using the broken cross configuration is that we are dealing with an arrangement of cells that repeats every two frames. If caution is not exercised, it is possible that the system will settle into a routine in which the music generated by this configuration repeats note for note. It is, however, possible to utilise this repetitive nature in a musical way. For example, the triadic sequence repeats exactly every 24 triads. In other words, the period of the triadic sequence is 24. If we choose the fundamental pitches in such a way that they repeat with period p , where p is *relatively prime*²² to 24, the resulting music will evolve in a predictable way, with both the triadic movement and the fundamental pitches following a fixed evolutionary pattern, but will not repeat exactly until the $24p^{\text{th}}$ triad.

²² Two integers, not both of which are zero, are said to be relatively prime if and only if their greatest common divisor is 1.

Three-quarter cross

The three-quarter cross configuration forms a lengthy sequence of cell clusters which eventually settle into the steady cyclic state of the broken cross (see Figure 3.3.5).

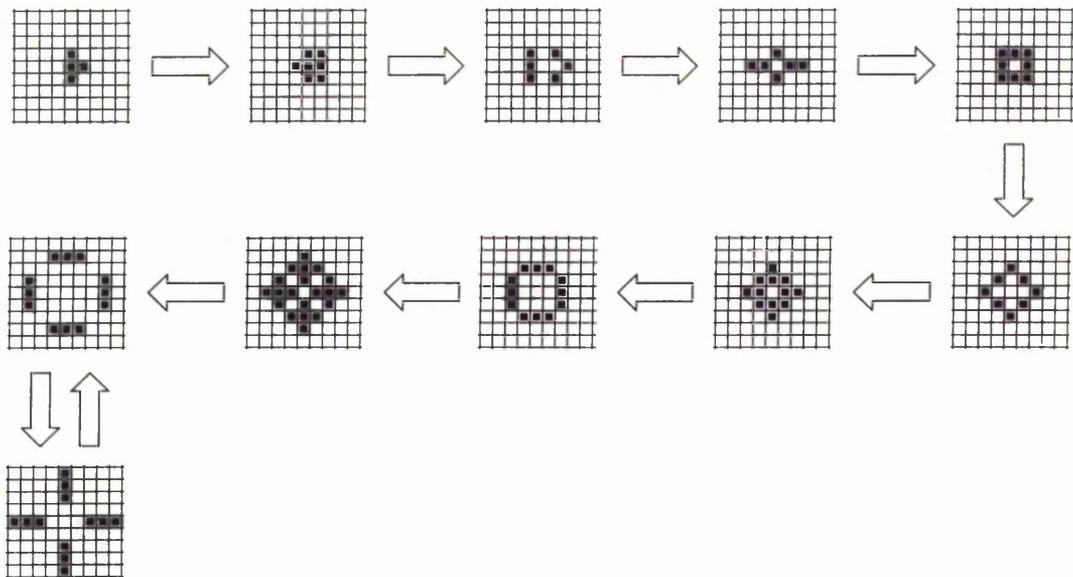


Figure 3.3.5 – Cellular evolution for the three-quarter cross configuration

The triad sequence generated by the three-quarter cross differs from that generated by the glider in two ways.

Firstly, the whole glider sequence repeats itself every five frames, whereas the three-quarter cross gradually evolves to a cyclic state between two frames. Secondly, as the walker evolves, it gradually works its way across the screen. The three-quarter cross, on the other hand, evolves centred on the cell $(x + 4, y + 4)$.

There are also several similarities between the two cell configurations. For example, both the walker and the three-quarter cross generate clusters of cells; both are sensitive to their position within the automaton, and both configurations, unlike the single cell, will generate musical data with a variety of temporal configurations.

Again, taking (x, y) to denote the top-left cell, we get the following sequence of live cells:

Frame 1	$(x + 4, y + 3)$		$(x + 4, y + 5)$		$(x + 3, y + 2)$		$(x + 6, y + 4)$
	$(x + 4, y + 4)$		$(x + 5, y + 3)$		$(x + 3, y + 6)$		$(x + 6, y + 5)$
	$(x + 4, y + 5)$		$(x + 5, y + 4)$		$(x + 4, y + 2)$		$(x + 7, y + 4)$
	$(x + 5, y + 4)$		$(x + 5, y + 5)$		$(x + 4, y + 6)$	Frame 10	$(x + 1, y + 3)$
Frame 2	$(x + 3, y + 4)$	Frame 6	$(x + 2, y + 4)$		$(x + 5, y + 2)$		$(x + 1, y + 4)$
	$(x + 4, y + 3)$		$(x + 3, y + 3)$		$(x + 5, y + 6)$		$(x + 1, y + 5)$
	$(x + 4, y + 4)$		$(x + 3, y + 5)$		$(x + 6, y + 3)$		$(x + 3, y + 1)$
	$(x + 4, y + 5)$		$(x + 4, y + 2)$		$(x + 6, y + 4)$		$(x + 3, y + 7)$
	$(x + 5, y + 3)$		$(x + 4, y + 6)$		$(x + 6, y + 5)$		$(x + 4, y + 1)$
	$(x + 5, y + 4)$		$(x + 5, y + 3)$	Frame 9	$(x + 1, y + 4)$		$(x + 4, y + 7)$
	$(x + 5, y + 5)$		$(x + 5, y + 5)$		$(x + 2, y + 3)$		$(x + 5, y + 1)$
Frame 3	$(x + 4, y + 3)$		$(x + 6, y + 4)$		$(x + 2, y + 4)$		$(x + 5, y + 7)$
	$(x + 4, y + 4)$	Frame 7	$(x + 2, y + 4)$		$(x + 2, y + 5)$		$(x + 7, y + 3)$
	$(x + 4, y + 5)$		$(x + 3, y + 3)$		$(x + 3, y + 2)$		$(x + 7, y + 4)$
	$(x + 6, y + 3)$		$(x + 3, y + 4)$		$(x + 3, y + 4)$		$(x + 7, y + 5)$
	$(x + 6, y + 5)$		$(x + 3, y + 5)$		$(x + 3, y + 6)$	Frame 11	$(x, y + 4)$
	$(x + 7, y + 4)$		$(x + 4, y + 2)$		$(x + 4, y + 1)$		$(x + 1, y + 4)$
Frame 4	$(x + 2, y + 4)$		$(x + 4, y + 3)$		$(x + 4, y + 2)$		$(x + 2, y + 4)$
	$(x + 3, y + 4)$		$(x + 4, y + 5)$		$(x + 4, y + 3)$		$(x + 4, y)$
	$(x + 4, y + 3)$		$(x + 4, y + 6)$		$(x + 4, y + 5)$		$(x + 4, y + 1)$
	$(x + 4, y + 5)$		$(x + 5, y + 3)$		$(x + 4, y + 6)$		$(x + 4, y + 2)$
	$(x + 5, y + 4)$		$(x + 5, y + 4)$		$(x + 4, y + 7)$		$(x + 4, y + 6)$
	$(x + 6, y + 4)$		$(x + 5, y + 5)$		$(x + 5, y + 2)$		$(x + 4, y + 7)$
Frame 5	$(x + 3, y + 3)$		$(x + 6, y + 4)$		$(x + 5, y + 2)$		$(x + 4, y + 8)$
	$(x + 3, y + 4)$	Frame 8	$(x + 2, y + 3)$		$(x + 5, y + 4)$		$(x + 6, y + 4)$
	$(x + 3, y + 5)$		$(x + 2, y + 4)$		$(x + 5, y + 6)$		$(x + 6, y + 4)$
	$(x + 4, y + 3)$		$(x + 2, y + 5)$		$(x + 6, y + 3)$		$(x + 6, y + 4)$

Table 3.3.5 – Sequence of live cells generated by the three-quarter cross.

There are some interesting features to note in the three-quarter cross configuration. For example, there is a strong visual correlation between frames 1 and 2. Taking $x = y = 0$ for illustrative purposes, we have that the initial pattern is reflected in the line $x = 4$, and three extra cells are added: (5, 3), (5, 4) and (5, 5). To interpret this musically, first note that reflection in the line $x = 4$ leaves the cells (4, 3), (4, 4) and (4, 5) unchanged. Only cell (5, 4) is altered, becoming (3, 4) after transformation. This is equivalent to moving from a second inversion major triad to a minor triad in root position, that is, the cell in question undergoes a modulation.

Of course, since the user is free to choose the fundamental pitches for the triads, the modulation can begin and end in whatever key the user desires. Therefore, it is entirely feasible to choose fundamental pitches so that the triads in the first frame relate to the key of C major, and those in the second frame relate to A minor, giving a modulation from major to relative minor.

It is important to note that the cell modulation here is a result of the initial configuration *and* the positioning of the initial configuration within the automaton space. If, for example, we had chosen the cells so that $x = y = 1$, the cell subject to the transformation would be (6, 5) – a combination of the tritone interval and a major seventh – which would have become (2, 3) – a dominant seventh chord with the 1st, 3rd and 7th present.

It should also be noted that the three-quarter cross ends up in the broken cross configuration when it reaches its cyclic state. This configuration, as seen above, generates triads whose internal intervals are successively augmented. The discussion concerning the broken cross will, of course, hold here.

One possible chord sequence for the first few frames of the three-quarter cross are shown in Table 3.3.4 below.

Cell co-ordinates	Chord type	Fundamental pitch
(16, 15)	major, root	50 (D3)
(16, 16)	augmented, root	50 (D3)
(16, 17)	minor, 1 st inversion	50 (D3)
(17, 16)	major, 2 nd inversion	50 (D3)
(15, 16)	minor, root	47 (B2)
(16, 15)	major, root	50(D3)
(16, 16)	augmented, root	50 (D3)
(16, 17)	minor, 1 st inversion	50 (D3)
(17, 15)	minor, 2 nd inversion	47 (B2)
(17, 16)	major, 2 nd inversion	50 (D3)
(17, 17)	major, 2 nd inversion with augmented 3 rd	49 (C#3)
(16, 15)	major, root	47 (B2)
(16, 16)	augmented, root	47 (B2)
(16, 17)	minor, 1 st inversion	40 (E2)
(18, 15)	diminished 7 th , no third	51 (D#3)
(18, 17)	minor, 1 st inversion with diminished 3 rd	50 (D3)
(19, 16)	major 7 th , added 5 th	49 (C#3)

Table 3.3.6 – One possible chording of the three-quarter cross starting configuration.

The bow tie

The final cell configuration that we consider is named the bow tie for obvious reasons (see Figure 3.3.6).

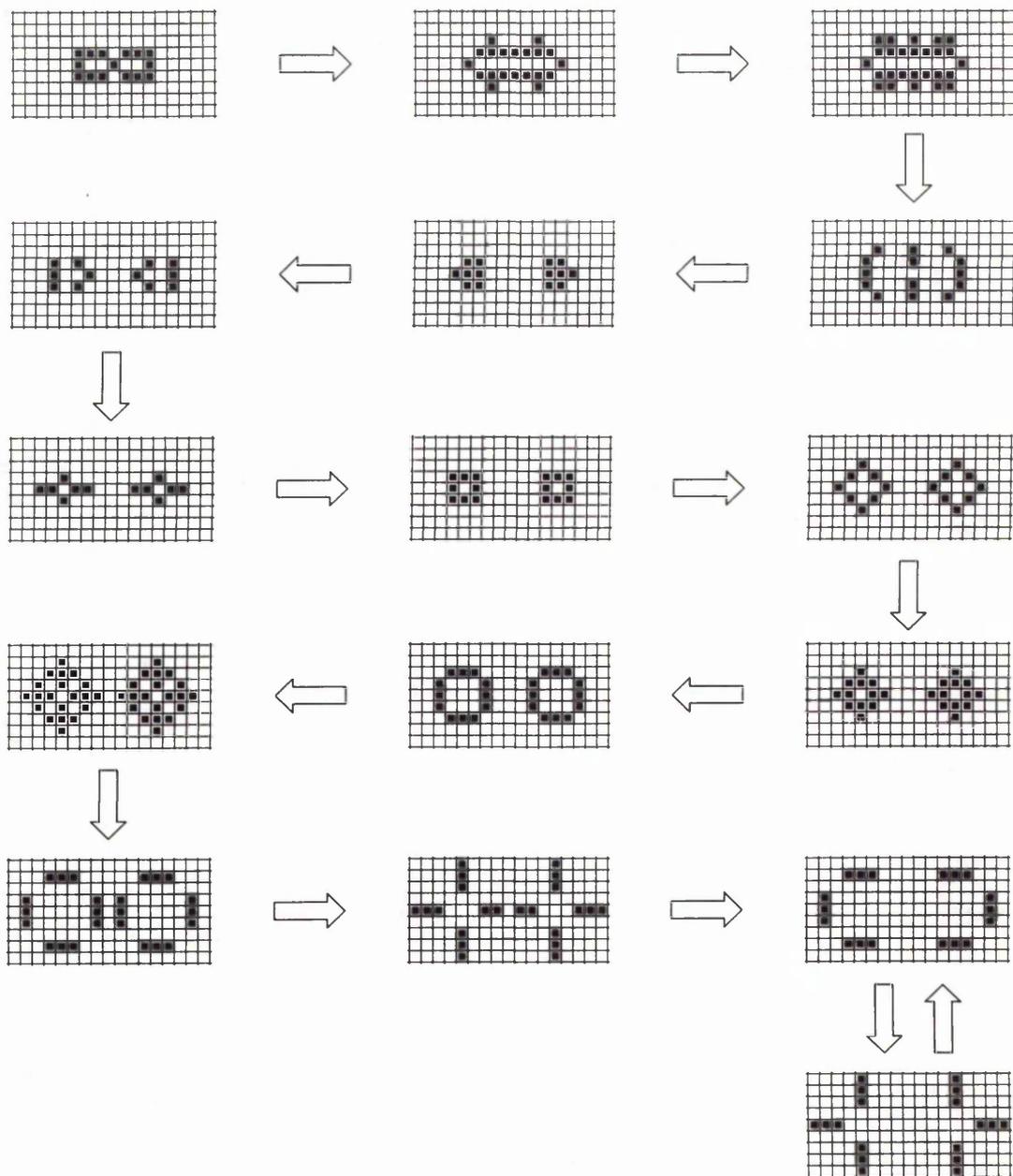


Figure 3.3.6 – Cellular evolution for the bow tie configuration.

The bow tie configuration evolves for 15 timesteps before reaching its cyclic state, which is a slight variation on the broken cross. The slight difference between the cyclic states of this configuration and, for example the three-quarter cross arises as a direct result of the pattern duplication seen in frame 4 onwards. The duplication arises because of two things. Firstly, the reflective symmetry in the initial frame ensures that

all subsequent frames are symmetric, and secondly, as the cells evolve, we obtain a dense clustering of live cells around $(x + 8, y + 4)$, forming a hostile environment for this neighbourhood. This in turn causes the central cells to die, and the remaining cells evolve symmetrically, but in isolation.

The pattern duplication of the bow tie configuration can be considered musically in several ways. One interpretation is to consider the patterns in frames 7 onwards. Here, each of the leftmost patterns exhibit mirror symmetry, and so they are duplicated exactly 8 cells to the right. If we consider the leftmost pattern in each frame as a theme, we can view this as an exposition step as the theme (i.e. the leftmost pattern) is played, and a development step as the developed theme (i.e. the rightmost pattern) is played. Here, the development of the thematic material in each frame consists of an augmentation of each of the lower triadic intervals by eight semitones, while the underlying triadic relationships – the cell pattern – remains unchanged.

This view of the configuration is, however, simply a special case of the true behaviour of the system. The system is symmetric about the line $x = x' + 8$ (taking (x', y') as the top-left cell co-ordinate for each frame), and so remains symmetric about this line for all subsequent steps. After the initial pattern has split into two parts, it is only because we arrive at a frame in which further symmetry is introduced – in this case about the line $x = x' + 4$ – that we obtain exact copies of the component patterns. The system itself makes no distinction between the patterns in frames 6 and 7. It is only due to the added symmetry that the developmental behaviour appears to change.

The triadic sequence generated by the bow tie configuration is shown below.

Frame 1	$(x + 5, y + 3)$		$(x + 5, y + 3)$		$(x + 11, y + 3)$		$(x + 3, y + 7)$
	$(x + 5, y + 4)$		$(x + 5, y + 4)$		$(x + 11, y + 4)$		$(x + 4, y + 1)$
	$(x + 5, y + 5)$		$(x + 5, y + 5)$		$(x + 11, y + 5)$		$(x + 4, y + 7)$
	$(x + 6, y + 3)$		$(x + 11, y + 3)$		$(x + 12, y + 2)$		$(x + 5, y + 1)$
	$(x + 6, y + 5)$		$(x + 11, y + 4)$		$(x + 12, y + 3)$		$(x + 5, y + 7)$
	$(x + 7, y + 3)$		$(x + 11, y + 5)$		$(x + 12, y + 5)$		$(x + 7, y + 3)$
	$(x + 7, y + 5)$		$(x + 12, y + 3)$		$(x + 12, y + 6)$		$(x + 7, y + 4)$
	$(x + 8, y + 4)$		$(x + 12, y + 4)$		$(x + 13, y + 3)$		$(x + 7, y + 5)$

	$(x+9, y+3)$		$(x+12, y+5)$		$(x+13, y+4)$		$(x+9, y+3)$
	$(x+9, y+5)$		$(x+13, y+4)$		$(x+13, y+5)$		$(x+9, y+4)$
	$(x+10, y+3)$	Frame 6	$(x+3, y+3)$		$(x+14, y+4)$		$(x+9, y+5)$
	$(x+10, y+5)$		$(x+3, y+4)$	Frame 11	$(x+2, y+3)$		$(x+11, y+1)$
	$(x+11, y+3)$		$(x+3, y+5)$		$(x+2, y+4)$		$(x+11, y+7)$
	$(x+11, y+4)$		$(x+5, y+3)$		$(x+2, y+5)$		$(x+12, y+1)$
	$(x+11, y+5)$		$(x+5, y+5)$		$(x+3, y+2)$		$(x+12, y+7)$
Frame 2	$(x+4, y+4)$		$(x+6, y+4)$		$(x+3, y+6)$		$(x+13, y+1)$
	$(x+5, y+3)$		$(x+10, y+4)$		$(x+4, y+2)$		$(x+13, y+7)$
	$(x+5, y+5)$		$(x+11, y+3)$		$(x+4, y+6)$		$(x+15, y+3)$
	$(x+6, y+2)$		$(x+11, y+5)$		$(x+5, y+2)$		$(x+15, y+4)$
	$(x+6, y+3)$		$(x+13, y+3)$		$(x+5, y+6)$		$(x+15, y+5)$
	$(x+6, y+5)$		$(x+13, y+4)$		$(x+6, y+3)$	Frame 14	$(x, y+4)$
	$(x+6, y+6)$		$(x+13, y+5)$		$(x+6, y+4)$		$(x+1, y+4)$
	$(x+7, y+3)$	Frame 7	$(x+2, y+4)$		$(x+6, y+5)$		$(x+2, y+4)$
	$(x+7, y+5)$		$(x+3, y+4)$		$(x+10, y+3)$		$(x+4, y)$
	$(x+8, y+3)$		$(x+4, y+3)$		$(x+10, y+4)$		$(x+4, y+1)$
	$(x+8, y+5)$		$(x+4, y+5)$		$(x+10, y+5)$		$(x+4, y+2)$
	$(x+9, y+3)$		$(x+5, y+4)$		$(x+11, y+2)$		$(x+4, y+6)$
	$(x+9, y+5)$		$(x+6, y+4)$		$(x+11, y+6)$		$(x+4, y+7)$
	$(x+10, y+2)$		$(x+10, y+4)$		$(x+12, y+2)$		$(x+4, y+8)$
	$(x+10, y+3)$		$(x+11, y+4)$		$(x+12, y+6)$		$(x+6, y+4)$
	$(x+10, y+5)$		$(x+12, y+3)$		$(x+13, y+2)$		$(x+7, y+4)$
	$(x+10, y+6)$		$(x+12, y+5)$		$(x+13, y+6)$		$(x+9, y+4)$
	$(x+11, y+3)$		$(x+13, y+4)$		$(x+14, y+3)$		$(x+10, y+4)$
	$(x+11, y+5)$		$(x+14, y+4)$		$(x+14, y+4)$		$(x+12, y)$
	$(x+12, y+4)$	Frame 8	$(x+3, y+3)$		$(x+14, y+5)$		$(x+12, y+1)$
Frame 3	$(x+4, y+4)$		$(x+3, y+4)$	Frame 12	$(x+1, y+4)$		$(x+12, y+2)$
	$(x+5, y+2)$		$(x+3, y+5)$		$(x+2, y+3)$		$(x+12, y+6)$
	$(x+5, y+3)$		$(x+4, y+3)$		$(x+2, y+4)$		$(x+12, y+7)$
	$(x+5, y+5)$		$(x+4, y+5)$		$(x+2, y+5)$		$(x+12, y+8)$

	$(x+5, y+6)$		$(x+5, y+3)$		$(x+3, y+2)$		$(x+13, y+4)$
	$(x+6, y+2)$		$(x+5, y+4)$		$(x+3, y+4)$		$(x+14, y+4)$
	$(x+6, y+3)$		$(x+5, y+5)$		$(x+3, y+6)$		$(x+15, y+4)$
	$(x+6, y+5)$		$(x+11, y+3)$		$(x+4, y+1)$	Frame 15	$(x+1, y+3)$
	$(x+6, y+6)$		$(x+11, y+4)$		$(x+4, y+2)$		$(x+1, y+4)$
	$(x+7, y+3)$		$(x+11, y+5)$		$(x+4, y+3)$		$(x+1, y+5)$
	$(x+7, y+5)$		$(x+12, y+3)$		$(x+4, y+5)$		$(x+3, y+1)$
	$(x+8, y+2)$		$(x+12, y+5)$		$(x+4, y+6)$		$(x+3, y+7)$
	$(x+8, y+3)$		$(x+13, y+3)$		$(x+4, y+7)$		$(x+4, y+1)$
	$(x+8, y+5)$		$(x+13, y+4)$		$(x+5, y+2)$		$(x+4, y+7)$
	$(x+8, y+6)$		$(x+13, y+5)$		$(x+5, y+4)$		$(x+5, y+1)$
	$(x+9, y+3)$	Frame 9	$(x+2, y+4)$		$(x+5, y+6)$		$(x+5, y+7)$
	$(x+9, y+5)$		$(x+3, y+3)$		$(x+6, y+3)$		$(x+11, y+1)$
	$(x+10, y+2)$		$(x+3, y+5)$		$(x+6, y+4)$		$(x+11, y+7)$
	$(x+10, y+3)$		$(x+4, y+2)$		$(x+6, y+5)$		$(x+12, y+1)$
	$(x+10, y+5)$		$(x+4, y+6)$		$(x+7, y+4)$		$(x+12, y+7)$
	$(x+10, y+6)$		$(x+5, y+3)$		$(x+9, y+4)$		$(x+13, y+1)$
	$(x+11, y+2)$		$(x+5, y+5)$		$(x+10, y+3)$		$(x+13, y+7)$
	$(x+11, y+3)$		$(x+6, y+4)$		$(x+10, y+4)$		$(x+15, y+3)$
	$(x+11, y+5)$		$(x+10, y+4)$		$(x+10, y+5)$		$(x+15, y+4)$
	$(x+11, y+6)$		$(x+11, y+3)$		$(x+11, y+2)$		$(x+15, y+5)$
	$(x+12, y+4)$		$(x+11, y+5)$		$(x+11, y+4)$	Frame 16	$(x, y+4)$
Frame 4	$(x+4, y+3)$		$(x+12, y+2)$		$(x+11, y+6)$		$(x+1, y+4)$
	$(x+4, y+4)$		$(x+12, y+6)$		$(x+12, y+1)$		$(x+2, y+4)$
	$(x+4, y+5)$		$(x+13, y+3)$		$(x+12, y+2)$		$(x+4, y)$
	$(x+5, y+2)$		$(x+13, y+5)$		$(x+12, y+3)$		$(x+4, y+1)$
	$(x+5, y+6)$		$(x+14, y+4)$		$(x+12, y+5)$		$(x+4, y+2)$
	$(x+8, y+2)$	Frame 10	$(x+2, y+4)$		$(x+12, y+6)$		$(x+4, y+6)$
	$(x+8, y+3)$		$(x+3, y+3)$		$(x+12, y+7)$		$(x+4, y+7)$
	$(x+8, y+5)$		$(x+3, y+4)$		$(x+13, y+2)$		$(x+4, y+8)$
	$(x+8, y+6)$		$(x+3, y+5)$		$(x+13, y+4)$		$(x+12, y)$

	$(x + 11, y + 2)$		$(x + 4, y + 2)$		$(x + 13, y + 6)$		$(x + 12, y + 1)$
	$(x + 11, y + 6)$		$(x + 4, y + 3)$		$(x + 14, y + 3)$		$(x + 12, y + 2)$
	$(x + 12, y + 3)$		$(x + 4, y + 5)$		$(x + 14, y + 4)$		$(x + 12, y + 6)$
	$(x + 12, y + 4)$		$(x + 4, y + 6)$		$(x + 14, y + 5)$		$(x + 12, y + 7)$
	$(x + 12, y + 5)$		$(x + 5, y + 3)$		$(x + 15, y + 4)$		$(x + 12, y + 8)$
Frame 5	$(x + 3, y + 4)$		$(x + 5, y + 4)$	Frame 13	$(x + 1, y + 3)$		$(x + 13, y + 4)$
	$(x + 4, y + 3)$		$(x + 5, y + 5)$		$(x + 1, y + 4)$		$(x + 14, y + 4)$
	$(x + 4, y + 4)$		$(x + 6, y + 4)$		$(x + 1, y + 5)$		$(x + 15, y + 4)$
	$(x + 4, y + 5)$		$(x + 10, y + 4)$		$(x + 3, y + 1)$		

Table 3.3.7 – Sequence of live cells generated by the three-quarter cross.

3.3.7 The process of ‘reverse engineering’

During the course of the author’s experimentation with CAMUS, a number of different working styles that users tend to adopt when composing with the system were observed. However, despite the apparent diversity of working procedures, users can essentially all be classed under two distinct headings – *dabblers* and *visionaries*.

Dabblers

Dabblers tend to play around with the software. They tend to be more interested in what the system does rather than what it can do for them – they are concerned more with the *hows* and the *what ifs* than in using the system as a compositional aid. Dabblers tend to make random changes to the parameters ‘just to see what happens’ but without having any musical or mathematical purpose for so doing.

There is absolutely nothing wrong with this type of working technique. Indeed, it is perhaps the best way to get to grips with the functionality of the software and the underlying compositional procedures. However, the music that is produced when dabbling is seldom noteworthy, and often consists of nothing but unrelated pitches and difficult rhythms. It is therefore the latter class of visionaries on which we wish to focus for the remainder of this discussion.

Visionaries

Visionaries are so called because they approach the algorithmic composition process with a definite goal in mind. This goal may be the core of a musical work that the user wishes to realise with the aid of the system, or it may be that the user simply wishes to 'hear' certain cell configurations. Whatever the ultimate aim of the user, the very fact that there is some structure behind the choice of parameters and patterns usually leads to music that is of a much higher standard than that which is achieved by dabbling.

Let us suppose, by means of illustration, that the user wishes to produce a composition with a particular tonal quality. How can this be achieved when the composition system itself has no regard for key? As was mentioned earlier, this can be done, but the tonality must be imposed manually through careful choice of the control parameters. The most efficient way of doing this is to employ a 'reverse engineering' technique as follows.

Firstly, one must choose an initial cell configuration for the Game of Life. When choosing such a configuration, the user must ask his or herself what design objectives are desired. For example, should the composition consist of densely clustered triads or should there be a large triadic spread? How should the triads evolve? Should the composition continue indefinitely, or should it come to an end after a specified time? Only when the user is confident that the initial cell configuration is consistent with the overall composition plan should he or she progress to the next stage.

Once an initial pattern has been chosen, the user should then preview the cellular evolution of the automaton, noting the live cells of the Game of Life. In so doing, a detailed map of the triadic progression is arrived at, although at this stage, the map is still in the form of cell co-ordinates, rather than the more musically useful chord notation. From here, it is relatively straightforward, if a little time-consuming, to analyse the co-ordinate list and convert it to chord notation. Clearly, some co-ordinates will give rise to more than one possible chord, so that the final chording will depend on the tonality that the user wishes to impose upon the work.

Finally, the pitch sequences for each articulation should be chosen so that the chord progressions that arise are consistent with the overall plan. The remaining parameters can then be chosen according to the user's own aesthetic criteria.

One can easily see how this technique can be considered as a form of reverse engineering by considering the technique of harmonising for four voices. In this case, one begins with a sequence of notes – either a melody or bass line. A chord sequence is then created, and finally the remaining parts are written, adhering to the rules of four-part harmony. Here, we approach from the opposite end – we begin with three distinct parts, which have been arrived at by following the composition rules, and use these to construct a chord sequence. Then, we analyse the chord sequence, and, depending on the tonal qualities that we desire, use this sequence to choose what is essentially a bass line.

Using this method, compositions can be created that share tonal aspects with other musical forms, thus bringing the comforting familiarity that the human ear finds so pleasing, whilst still retaining a high degree of individuality from the stylistic features typified by the triadic structure of CAMUS-inspired compositions.

Two examples of this method of working, *Sonatina* and *Minuet and Trio*, both for woodwind ensemble, are provided on tracks 3 and 4 of the accompanying compact disc.

4. Developing the system

4.0 Introduction

In this chapter we discuss the limitations of the system as it currently stands and suggest ways in which they may be overcome. The actual implementation and workings of these solutions are presented in later sections.

We also suggest ways in which the system can be extended to take into account, for example, different algorithm types and different mappings from the underlying control systems to musical output.

4.1 Limitations with the present system

We now identify several features which, although not critical, limit the usefulness of the system.

4.1.1 Chord complexity

The first thing that should be noted when discussing the limitations of the system is that the use of a two-dimensional von Neumann Music Space to generate triads effectively limits the complexity of the resulting music. This is further compounded by the serial method that CAMUS uses to check cells (see Section 4.1.5).

It is fair to say that no matter how clever or elegant a composition system is, it is unlikely to be genuinely useful as a musical tool unless it generates music or musical ideas that end listeners will find pleasant and enjoyable, or that a composer can use from which to build complete works.

Although the system can and does generate useful musical ideas which can be built upon, the monochordal three-note chords that form the basis of all CAMUS compositions simply do not hold the interest of a listener for more than a short time. The system cannot, therefore, be said to produce music that end listeners will find pleasant and enjoyable.

With this in mind, the author proposes two methods for increasing the complexity of the chordal events that form the basis of CAMUS compositions. These are:

- i.) Extending the von Neumann Music Space to a higher dimensionality, which we discuss below.
- ii.) Introducing parallel cell checking, which we introduce in Section 4.1.5 below, and discuss further in Section 6.1.7.

4.1.2 New dimensions of complexity

Extending the von Neumann Music Space to a higher dimensionality is a fairly logical progression of the system, and allows for the development of much more complex chordal and temporal structures within each composition.

The work required to implement the extensions to the automata is feasible, though not trivial. It involves a corresponding extension of the arrays of cells in the Game of Life and Demon Cyclic Space automata, an increase in the number of neighbours whose states must be checked at each timestep, and an alteration to the rules that determine the evolution of the automata.

The mechanics of the increase in dimensionality are not difficult, rather it is the *determination of suitable transition rules* for automata of higher dimensionality that proves to be most problematic. We discuss this in Section 4.1.3 below.

In addition, two further sources of difficulty present themselves.

The first is due to the increase in the number of calculations brought about by the higher dimensionality of the control automata. In general, the n -dimensional automata used by CAMUS require $3^n - 1$ cell calculations per timestep. One can see that as the number of dimensions is increased, the number of calculations required simply to update the automata can become quite large. However, today's processors are more than capable of dealing with this number of calculations for arrays of cells which would be used in practical cases, and if significant timing problems did arise, it would be perfectly feasible to compose off-line, saving the resulting file for editing at a later time.

The second problem, however, does not have such a neat solution. The problem is one of human-computer interface – an issue that has a very real effect on the usefulness of any system.

It is desirable to display the data from the control automata in graphical form. The reasons for this are twofold. Firstly, visual display of data is often an aid to the understanding of a system, by clicking on cells and making general parameter changes, and seeing the results implemented immediately onscreen, one can quickly develop an intuitive feel for how the system behaves. Secondly, seeing the evolution of the control automata helps to correlate what is happening in the system to the musical output that is generated. This leads to a better aesthetic feel for the system as the user associates different cell configurations with the musical output that they generate.

Graphical displays are easily implemented for 2-dimensional automata, but a difficulty arises when one considers how to obtain and display the data from an automaton of higher dimensionality. The three-dimensional case has a partial solution in that it is possible to display an isometric view of a three-dimensional automaton on a two-dimensional computer screen, but this has its limitations in that some cells may be hidden from view by neighbouring cells. In any case, this is still not a viable method for the input of cell data using a two-dimensional input device, such as a mouse.

If the dimensionality is increased still further then we hit an intractable problem - the human brain can only perceive three spatial dimensions. Some sort of compromise must be decided upon, and to this end, we propose the following.

Rather than try to display a two dimensional (or isometric) representation of an automaton with dimensionality 3, the user is presented with a series of two-dimensional *embeddings* of the automaton.

For the purpose of this discussion, we take an embedding to be a projection of the three-dimensional Game of Life space onto a general plane.

Let the plane be denoted P , and the 3-dimensional Game of Life be denoted GOL. Then a cell, c , on P will be alive if and only if the line tangential to P from c meets at least one live cell in GOL. This is illustrated in Figure 4.1.1 below. Note that for reasons of legibility this illustrates just one plane of the 3-dimensional automaton space. The cells extend into the screen, resulting in a plane embedding rather than the line illustrated in the figure.

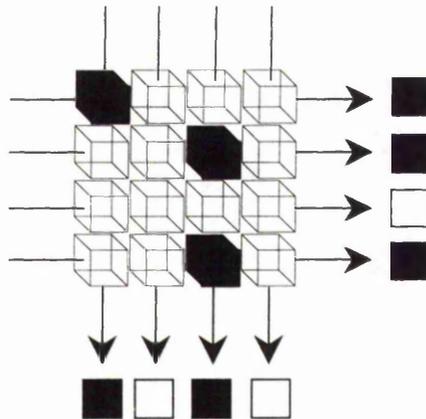


Figure 4.1.1 – Constructing an embedding by projecting a plane from the 3-dimensional Game of Life to a row (or column) of cells in a 2-dimensional array.

The embeddings offer the user feedback from the control system as to the current states of the automata.

Input from the user can be accomplished in two ways.

The first method is by the direct entry of co-ordinates for cells. This, however, is not a particularly intuitive method, and does not aid the user's visualisation of the current state of the automaton in question, which must be done in the user's mind's eye.

The second method involves clicking cells in the various embeddings of the automaton, which is a more visual, if cumbersome method of input. Here, the user clicks on a cell in one embedding, fixing the two co-ordinates described by that embedding. Clicking on a cell in the next embedding fixes a further point and so on.

As an example of this input method, consider again the three-dimensional case, in which each cell is specified by a co-ordinate of the form (x, y, z) (see Figure 4.1.2).

The three embeddings with which we will work are the x - y plane ($z = 0$); the x - z plane ($y = 0$), and the y - z plane ($x = 0$).

Suppose the user clicks on the point $(3,5)$ on the x - y plane. This would then specify a line on both the x - z plane ($x = 3$) and the y - z plane ($y = 5$). The user would then click on a point on one of these lines (say $(3,4)$ on the x - z plane) to fix the z -co-ordinate as $z = 4$, resulting in the unique point $(3,5,4)$.

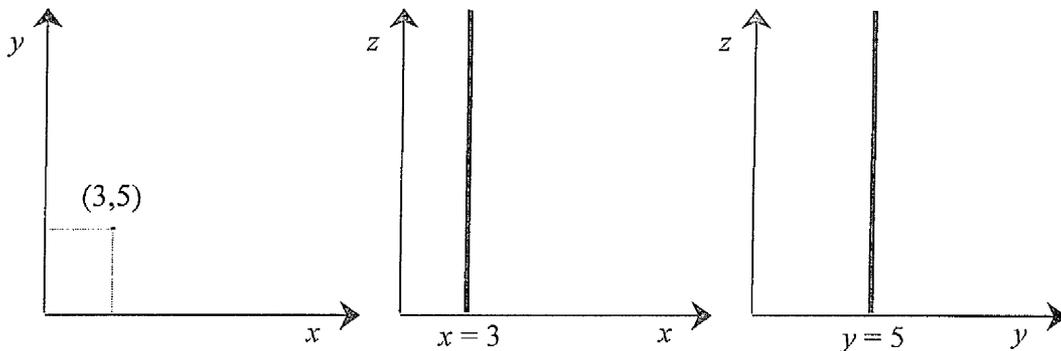


Figure 4.1.2a – The user clicks on the point $(3,5)$ in the x - y plane, fixing the lines $x = 3$ and $y = 4$ in the other two embeddings.

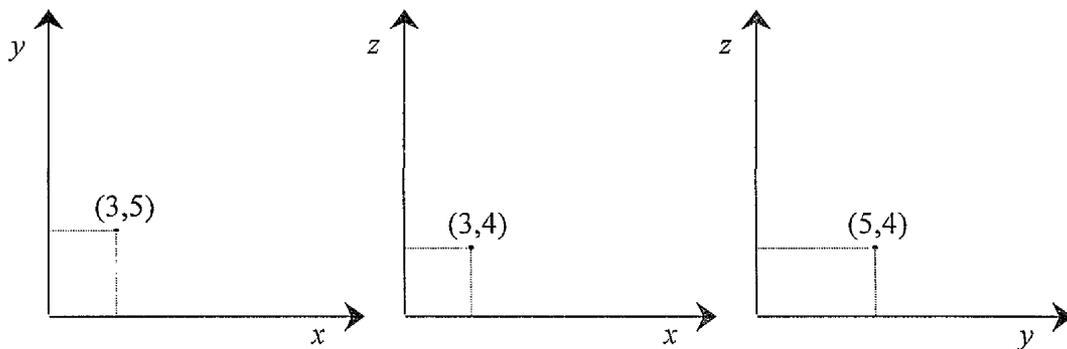


Figure 4.1.2b – The user clicks on the point $(3,4)$ in the x - z plane, fixing the unique point $(3, 5, 4)$ in the three-dimensional space.

This system is very similar to that used by 3-D modelling packages, and serves us well as a prototype. Practical experience with the test system indicated desirable adaptations, which are discussed in Section 5.2.2.

4.1.3 – Rules for life in a 3-dimensional universe

Just as with Conway's 2-dimensional Game of Life, not every rule set for 3-dimensional life produces cellular evolution worthy of the name life (in the sense described in Section 2.2.4).

Recall from Section 2.2.4 that the *environment*, E , is defined as the number of living neighbours that surround a particular cell. The *fertility*, F , is defined as the number of living neighbours required to give birth to a currently dead cell. Then, we define the *transition rule*, R , as the quadruple $(E_{min}, E_{max}, F_{min}, F_{max})$, where

$$E_{min} \leq E \leq E_{max}$$

and

$$F_{min} \leq F \leq F_{max}.$$

Then the rule R defines a Game of Life if and only if the following statements hold:

- i.) A glider must exist and occur "naturally" if we apply R repeatedly to primordial soup configurations (as defined in Table 2.2.1).
- ii.) All primordial soup configurations when subjected to R must exhibit bounded growth.

The only way to be completely certain that these statements are satisfied is to perform an exhaustive search on the rule space. Unfortunately, this is a very time-consuming business. We must test a large number of primordial soup configurations for each rule to say that the above criteria have been fulfilled with any degree of conviction.

Now, for a three dimensional automaton, each cell has 26 neighbours other than itself. This means that for live cells alone we have the following 351 possible rules:

$$(E_{min}, E_{max}) = (1, 1), (1, 2), \dots, (1, 26), (2, 2), (2, 3), \dots, (2, 26), \dots, (26, 26).$$

A similar number of rules for dead cells results in a total of $351 \times 351 = 123\,301$ possible rule sets for the three-dimensional Game of Life.

We cannot hope to test exhaustively each of these rule sets by hand. Fortunately, we can reduce the possibilities to a more manageable number.

Theorem 4.1.1 (Bays, 1987)

Any rule $(E_{min}, E_{max}, F_{min}, F_{max})$ with $F_{min} \geq 10$ cannot support a glider.

To see this, note that a dead cell which lies adjacent to a plane of living neighbouring cells can have at most nine living neighbours (see Figure 4.1.3). Thus, any formation under a rule R of the form $(X, Y, 10, Z)$ will ultimately either shrink and disappear or will form a convex blob whose outer surface may continually evolve, but which will never translate across the universe.

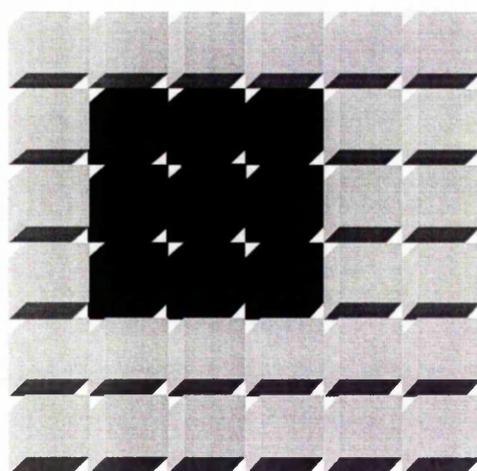


Figure 4.1.3 – A dead cell, illustrated here as a wireframe box, which lies next to a plane of live cells has at most nine live neighbours, shown here as dark grey.

Theorem 4.1.2 (Bays, 1987)

Any rule $(E_{min}, E_{max}, F_{min}, F_{max})$ with $F_{min} \leq 4$ leads to unlimited growth.

This can be seen quite easily by considering a starting configuration of four live cells arranged in a square (see Figure 4.1.4). Such configurations always grow without bound.

In his paper, *Candidates for the Game of Life in Three Dimensions* ([Bays, 1987]), Carter Bays suggests two candidates for a three-dimensional Game of Life. These are

$R = (4, 5, 5, 5)$ and $R = (5, 7, 6, 6)$. We discuss these rule systems at length in Section 5.1.

An alternative to working with life in a true 3-dimensional space is to treat the space as a number of stacked 2-dimensional spaces. This is similar in notion to the time-space barriers proposed by Bays ([Bays, 1987]). Here, however, we do not depend on stable cell constructions to restrict the development of cells in three dimensions. Instead, we insist that only the neighbours bordering the cell in the plane of interest (which is decided upon by the user) are checked.

This has the significant advantage that the 3-dimensional space now behaves as several parallel Conway Games, so that the tried and tested rules and starting configurations can be used for the 3-dimensional game. Thus we have the increased complexity offered by a higher dimensional control system, whilst still retaining the ease of use of the 2-dimensional system.

In addition, the number of neighbours that are required to be checked for each cell is reduced from 26 (in the 3-dimensional case) to 8.

This method can also be extended to higher dimensions. For example, suppose we are employing cell-checking on the x - y plane in the 4-dimensional case. Then for the cell (x, y, z, w) , we check the cells $(x \pm 1, y, z, w)$, $(x, y \pm 1, z, w)$, $(x \pm 1, y \pm 1, z, w)$ and $(x \pm 1, y \mp 1, z, w)$.

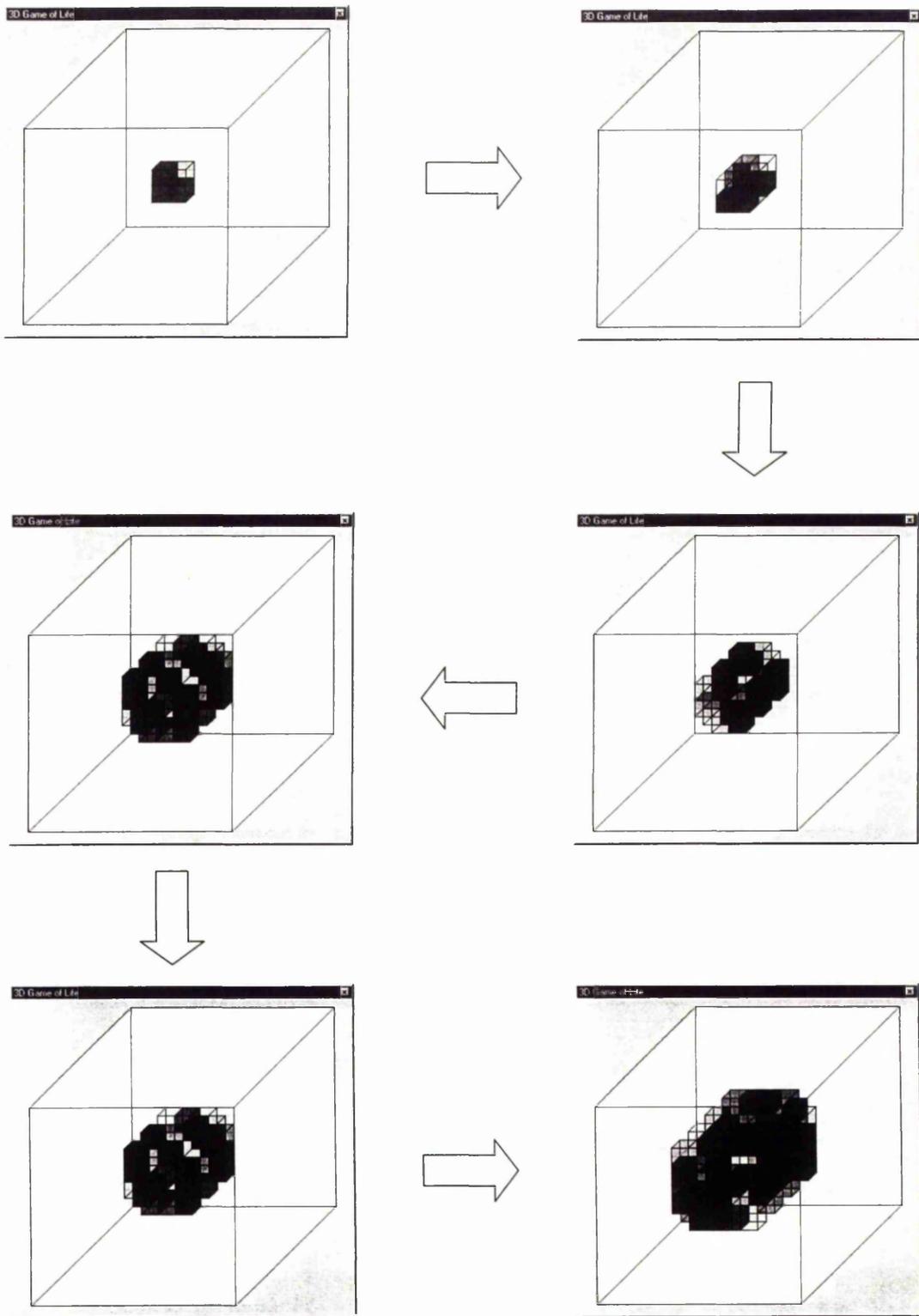


Figure 4.1.4 – The rule of the form (3, 3, 4, 5) leads to unlimited growth.

4.1.4 Needles in haystacks

A further limitation arises when CAMUS is used as generator of musical ideas.

Here, the system is configured and set in motion. The resulting output is used as a rough musical sketch by a human composer. This sketch is then rearranged and expanded to give a complete musical composition.

As mentioned in Section 4.1.1 above, rarely, if ever, does CAMUS produce complete compositions of any real worth. Nevertheless, the system does produce some interesting ideas that can be used to build complete compositions. Several compositions have been penned by the author and Dr. Eduardo Miranda in this way.

Of particular note is Dr. Miranda's, *Entre l'Absurde et le Mystère* for chamber orchestra, which was premiered in Edinburgh in March 1995 by The Chamber Group of Scotland, conducted by Martyn Brabbins. This is illustrated, along with other CAMUS-inspired²³ works on tracks 2 – 9 and 18 – 20 of the accompanying CD.

Genuinely useful musical ideas are relatively few and far between, however, and a considerable amount of useless material is also generated. This greatly increases both the time taken to build a finished composition and the likelihood that useful snippets are missed because the composer has to trawl through the entire output from the system.

The problem is then how best to increase the yield of the system. There is no simple solution, but a partial answer may lie in getting the system to behave more musically. For example, at present, the method of rhythm generation is very unmusical: A random number generator returns an integer value between two limits. This is then used as a value for the number of 'ticks' of the note event in question. A note event may be a note duration, or the length of time before a new note is triggered. The tick

²³ We use the term 'inspired' here as opposed to 'composed' or 'generated' because the musical output generated by the system was used as a sketch from which the finished compositions were developed. It is a matter of philosophical debate as to the true authors of such works. Does authorship lie with the computer that produced the work, with the designer of the algorithm that was responsible, or with the human operator who initialised the parameters and arranged the output to create the final output. These issues are addressed in a later section. The reader is also directed to ([Miranda, 1997]).

system has an effect on the temporal resolution of the music and is discussed more fully in Appendix C.

This system is very unsatisfactory. It produces rhythmic events that are irregular and difficult to listen to (see Figure 4.1.5). The human ear tends to welcome pseudo-regularity which, despite the underlying probability distributions at work, are not apparent in the musical output.

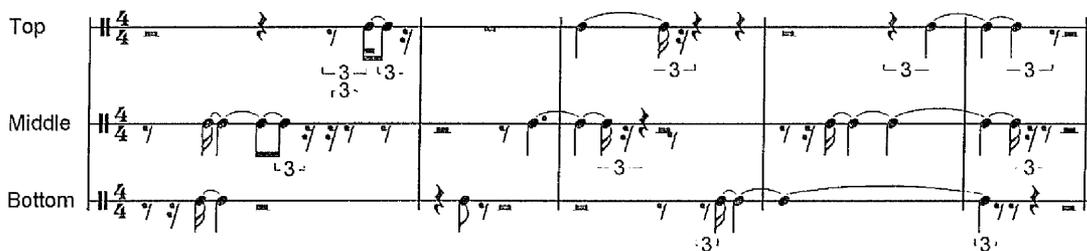


Figure 4.1.5 – A typical rhythmic figure produced by CAMUS's rhythm engine.

The musical phrase in Figure 4.1.5 above was generated using the default random number selection routines in CAMUS. The phrase was saved as a MIDI file and imported into Steinberg's *CUBASE* sequencer for scoring. The triads were split into three separate rhythm staves in order to render readable the overlapping rhythms. If this had not been done, the rhythms would have been unintelligible.

We compare this with Figure 4.1.6, a four-bar phrase from J. S. Bach's three-voice Fugue no. 2 in C minor from the *Well-Tempered Clavier* ([Bach, 1923]).



Figure 4.1.6 – A four-bar phrase from Bach's Fugue no. 2 from the *Well-Tempered Clavier*.

The differences in the rhythmic make-up of these two compositions is immediately apparent. Whilst each voice in Figure 4.1.6 has a great deal of movement, the 'voices'

of Figure 4.1.5, in contrast, are sparse. This is not necessarily problematic in itself, but leaves little scope for complex interaction between voices.

We also see that the rhythmic figures in Figure 4.1.6 are relatively simple (mainly straight quavers and semiquavers), and that similar figures occur frequently. This cannot be said for the rhythms in Figure 4.1.5, which begin on and last for very complex fractions of beats.

Even in the four-bar phrase of Figure 4.1.6, there is considerable evidence of structural detail that simply does not exist in Figure 4.1.5. Rhythmic figures are repeated (for example bars 22 and 23 in all three voices) or altered slightly (for example bars 22 of the second voice and 23 of the third).

We discuss these issues and a possible solution in Section 5.2.5.

The rhythmic structure is, unfortunately, not the only unmusical detail that arises as a result of the CAMUS algorithm.

Looking back at Figure 4.1.5, we see that the notes of the triad often overlap in a way that is more akin to polyphonic writing, whilst the music itself is generated homophonically, albeit with spread chords. This results in conflicting signals being received by the listener – on the one hand, the listener may interpret the note overlap as several independent voices, but hears music that essentially consists of smeared block chords.

Remember also that each triad is played on one MIDI channel, i.e. by a single instrument. This raises the possibility that polyphonic note data may be generated for an instrument that is only capable of playing monophonically, such as the majority of wind instruments, or some synthesisers. This means that music may be generated that is unable to be played on the instruments for which it was written. Therefore, it is important that this particular issue is addressed. We suggest a practical solution to this problem in Section 5.2.7.

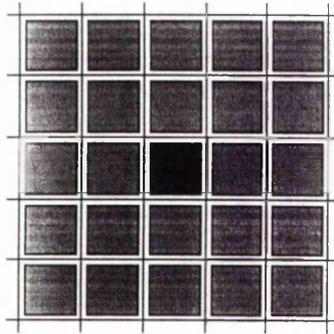


Figure 4.1.8 – Cell neighbourhood of size 2.

At present, the cell checking algorithm is as shown in Algorithm 4.1.1 below.

```

for each cell in array
    if cell is alive
        play cell
    end if
next cell

```

Algorithm 4.1.1 – Standard cell checking routine for the CAMUS algorithm.

The algorithm used to check cell neighbourhoods is very similar. Here, we illustrate for a neighbourhood of size i (see Algorithm 4.1.2).

```

for each  $(2i + 1)$ th cell in array
    for each cell in neighbourhood
        if cell is alive
            calculate note data
        end if
    next cell in neighbourhood
    play calculated note data
next cell in array

```

Algorithm 4.1.2 – Parallel cell checking for a neighbourhood of size i .

Parallel cell-checking of this sort has a distinct advantage over the present one-cell-per-sonic-event technique. Namely that the number of simultaneous notes that the system can produce is limited only by the size of the cell neighbourhood being used. For example, a neighbourhood of size 2 would allow a maximum of $5 \times 5 \times 3 = 75$ simultaneous notes.

Unfortunately, we are still limited by the fixed cell sequence, although the effect of this is lessened because many cells are played at once. However, it is possible to rid ourselves of this restriction: If we extend the size of the cell neighbourhood to that of the control automaton, then each automaton timestep can be considered as a single musical event. Thus a fixed interval series no longer applies because the cells are checked in parallel, so that no one cell will be played before any another. The maximum number of simultaneous notes is also now limited only by the size of the control automata.

However, there are two further sources of difficulty which now present themselves.

The first is concerned with real time performance: We are now generating music at a higher level, and thus we need considerably more processing power than before. If we consider a timestep of x seconds, this means that for real-time output, we must calculate the next timestep of the automata, determine the live cells and their corresponding instrumentations, sort the cells in temporal order at triad level, sort the cells in temporal order at note level, update the score files and output the musical data all within the x -second window. Some study must be given to the feasibility of producing real-time output using this technique.

The second source of difficulty arises as a result of another limitation of the CAMUS system. At present, CAMUS does not, in any way, distinguish between triads. Thus each live cell contributes sonically to the composition, whether or not the user desires its presence. Therefore, there is an overwhelming likelihood that a sizeable portion of the music will sound extremely unpleasant due to the absence of harmonic structure.

We comment further on this in Section 6.

4.1.6 Dynamical systems for a dynamic system

There is one more restriction, though not necessarily a limitation, of the system to discuss at this time.

At present, CAMUS allows only for one music algorithm to drive the composition. System versatility could be improved by offering the user a choice of several algorithms, which could be selected at will to best suit the music being composed.

Possible developments include the provision of alternative mappings from the control automata to music and the introduction of new dynamical systems, such as fractal zooms and neural networks to generate raw musical data.

One such algorithm, which draws on certain ideas behind the CAMUS system, utilises two properties of the fractal classes known as the Mandelbrot set, and Julia sets.

We saw in Chapter 2 that the Mandelbrot set can be considered as a graphical representation of the behaviour of the complex plane according to the iterative equation

$$z_{i+1} = z_i^2 + c$$

where c is a complex constant.

For each c in the plane, the system is initialised with $z_0 = 0$, and iterated. This results in the following possible outcomes:

- i.) $|z_n| \rightarrow \infty$ as $n \rightarrow \infty$.
- ii.) $|z_n|$ remains bounded as $n \rightarrow \infty$.

Each c is assigned a colour according to the number of iterations required to send the point to infinity, or a default colour (usually black) if the point does not increase without bound²⁴. The Mandelbrot set is then the set of all values of c which are coloured black.

The area of interest occurs within the region bounded by the lines $x = -2.5$ and $x = 1.5$ on the real axis, and $y = -1.5i$ and $y = 1.5i$ on the imaginary axis. At, and near the boundary of the Mandelbrot set, there is described a structure of inconceivable complexity, with curves which spiral onwards for eternity, and detail at every scale.

Julia sets are determined in much the same way as the Mandelbrot set, using the same iterative formula, but in this case, the complex parameter, c , remains fixed. After each

²⁴ In practice, we assign a threshold value, T , and a maximum number of iterations, MI , and stop the iterative process when either $|z_n| \geq T$, or when $n > MI$.

iteration, the initial value, z_0 , is altered. This gives rise to the Julia set, J_c , which is defined to be the boundary formed between the set of all initial values, z_0 , for which $|z_n| \rightarrow \infty$ as $n \rightarrow \infty$, and the set of all initial values for which $|z_n|$ remains bounded.

When c is controlled from within the confines of the Mandelbrot set, say by the user clicking on a point, which is then passed to the Julia algorithm, one can see a striking visual correlation between the region which was clicked, and the type of Julia set which is obtained (see Figure 4.1.9). Parameters taken from the exterior of the Mandelbrot set result in non-connected “dust” sets, whilst parameters taken from within the set itself give rise to connected Julia sets.

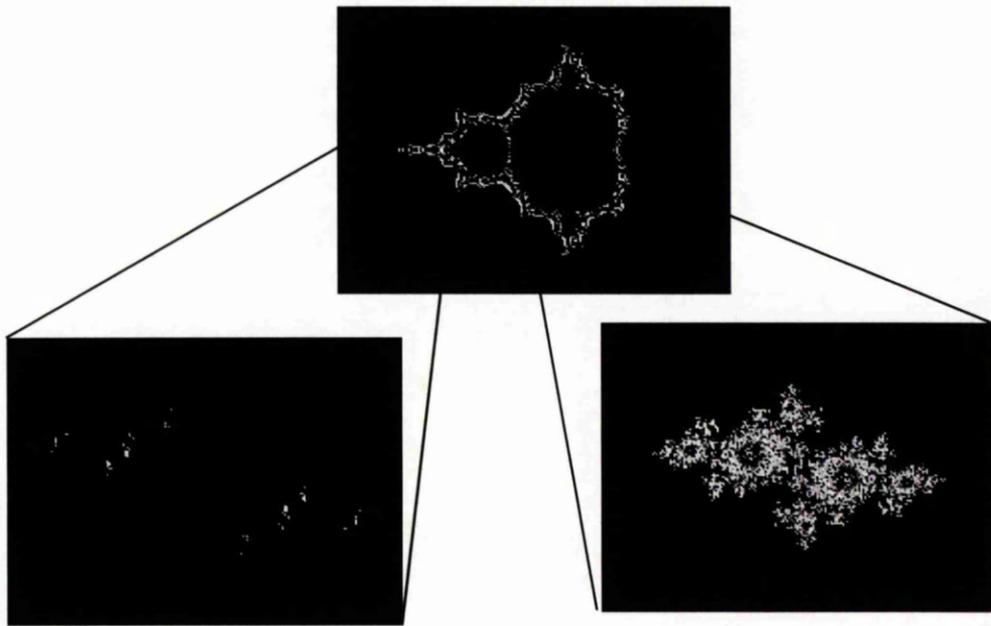


Figure 4.1.9 – The Mandelbrot set and two Julia sets with respective regions.

Like the Mandelbrot set, Julia sets exhibit detail at every scale, and it is this feature along with the relationship between the two on which the composition system proposed by the author is based.

To begin with, the user is presented with a window showing the Mandelbrot set. The user then draws a path on the screen, specifying a series of Julia sets, which will be calculated in sequence.

Next, a further path, which may involve zooming in or out of the Mandelbrot set, is drawn. This defines a sequence of regions of the Mandelbrot set, to be calculated in parallel with the Julia sets. The user specifies the number of timesteps required to complete the composition, along with other composition specific parameters, such as the number of instruments and their timbres.

The sets are then mapped onto the von Neumann Music Space, with each Julia colour band corresponding to a different instrument, and points on the boundary²⁵ of the

²⁵ Here, the 'boundary' of the Mandelbrot set is taken to mean the quantised graphical boundary.

Mandelbrot set contributing to the notes of the composition. In order to control the notes and dynamics, a set of articulations, similar to those of the CAMUS system could be used. Alternatively, a stochastic routine, with user-defined probability tables controlling the evolution of the composition, or some other automated parameter control routine, such as that described below could be used.

One alternative method for controlling the development of the composition parameters is derived from a computation technique known as a genetic algorithm. This is an evolutionary algorithm, that is one which is based on the various biological phenomena associated with evolution, and operates on a set of binary codewords.

There are four basic types of operation that can be performed on the codewords. These are selection, crossover, mutation and inversion. We shall consider only the first three, since inversion can be derived from these.

The crossover operation exchanges information between a pair of codewords. Mutation alters the value of a single bit in a codeword. Typically, the genetic algorithm is used as an optimisation technique, and so the selection operation is used to find the 'best' possible codewords for some predetermined criterion. However, for compositional purposes, no such optimisation is necessary, since the operations are used solely for the development of the compositional parameters. A typical parameter evolution step is shown in Table 4.1.1.

Explanations	Codewords
Consider a set of n codewords which represent the values of some musical parameters.	$C_1 : 11010110$ $C_2 : 10010111$... $C_n : 01001001$
Selection: Codewords are chosen to undergo evolution according to some stochastic mechanism (<i>c.f.</i> Darwin's theory of evolution - chance mutation leading to population development ([Darwin, 1981])).	$C_2 : 10010111$ $C_7 : 11000101$ $C_{11} : 01111001$
Crossover: Some portion of a pair of codewords (in this case, C_7 and C_{11}) is exchanged at the dotted position randomly specified, producing the two offspring C_7' and C_{11}' , whose values are then assigned to C_7 and C_{11} . This is applied to some predetermined section of the population, specified by the crossover rate.	$C_2 : 10010111$ $C_7 : 11000$ 101 $C_{11} : 01111$ 001 $C_7' : 11000$ 001 $C_{11}' : 01111$ 101
Mutation: The bit values of some codewords are inverted at a mutation rate of 1 - 5%. The value that has been changed as an example is highlighted by an underline.	$C_2 : 10010111$ $C_7 : 11100001$ $C_{11} : 01111101$

Table 4.1.1 – A typical parameter development step.

The author proposes the following as a genetic algorithm for parametric development: firstly, each parameter is assigned a binary codeword according to its initial value. At each timestep, the genetic algorithm is performed on the set of codewords as illustrated in Table 4.1.1 above, leading to population development.

4.1.7 Alternative mappings

Finally, we consider the effect of many different mappings from both cellular automata and fractals onto music.

The technique of mapping co-ordinates to musical triads through the von Neumann Music Space has already been described, but several alternatives exist.

For example, we could map cells onto a discrete two-dimensional space defined by pitch vs. a musical parameter (see Figure 4.1.10). Similarly, we could combine automata with stochastic techniques by using the states of cells to determine probability tables from a set held in memory. Once a probability table has been established for an individual cell, a random selection routine can then be used to select the actual note values and musical parameters with the cell (see Figure 4.1.11).

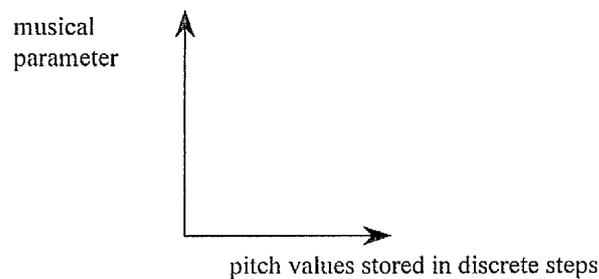
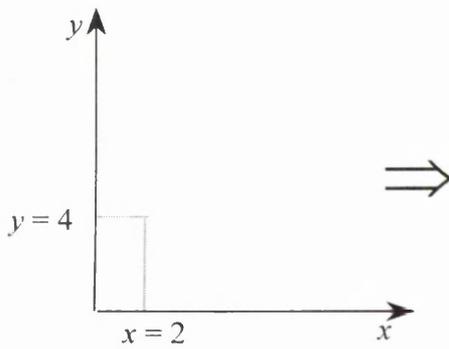


Figure 4.1.10 - An alternative two-dimensional space for use in mapping dynamical systems to music.

Probability table stored at array position
(2, 4)



<i>Note</i>	<i>Dynamic level</i>	<i>Probability</i>
C#	<i>p</i>	0.2
E	<i>mf</i>	0.35
Ab	<i>pp</i>	0.4
Rest	-	0.05

Figure 4.1.11 - A cell at position (2, 4) determines a probability table associated with that note. The actual note values and parameters can then be calculated using a random selection technique.

5. CAMUS 3D

5.0 Introduction

In this chapter we present CAMUS 3D (Cellular Automata MUSIC in 3 Dimensions). The system is clearly derived from the original 2-dimensional system, but it is the author's belief that there are sufficient differences to allow us essentially to consider the 3-dimensional system as a new composition algorithm in its own right.

We continue by illustrating the workings of the new system with a composition example, and conclude by highlighting the similarities and the differences between the 2-dimensional and the 3-dimensional systems.

5.1 The development of the system

CAMUS 3D has undergone continual revision since the prototype system was developed in 1997. In this section we trace the development of the system from its initial implementation through to the introduction of the latest composition tools.

5.1.1 Into the third dimension

The CAMUS 3D system is a development of the earlier 2-dimensional system ([Miranda 1993], [Miranda, 1994], [McAlpine, Miranda, and Hoggar 1997a] [McAlpine, Miranda & Hoggar 1997b]), and uses three-dimensional extensions of the Game of Life and Demon Cyclic Space automata to generate compositions.

5.1.2 The 3-dimensional Game of Life

The 3-dimensional Game of Life automaton consists of an $l \times m \times n$ array of cells, each of which can exist in two states, alive, denoted by 1, or dead, denoted by 0. As in the 2-dimensional case, live cells are shaded black, whilst the dead cells are left unshaded.

5.1.3 Candidates for rules in the 3-dimensional Game of Life

As mentioned in Section 4.1.3, only a small subset of the possible rule sets result in games that satisfy the criteria for life. By using Theorems 4.1.1 and 4.1.2, we showed how most of the 123 301 possible rule sets can be discarded, whilst Carter Bays, who is surely the leading researcher in higher dimensional Life, further reduced the number of candidates to just two ([Bays 1987]) – the rules $R = (4, 5, 5, 5)$ and $R = (5, 7, 6, 6)$. We now discuss these rules in more detail.

5.1.4 The rule $R = (4, 5, 5, 5)$

The first point to note about $R = (4, 5, 5, 5)$ is that it is obtained simply by adding 2 to the standard Conway rule, $R = (2, 3, 3, 3)$.

Before examining this more closely, let us first introduce some terminology that will help us to compare configurations of cells in the 2 and 3-dimensional automata.

An *expansion* of a 2-dimensional Conway cell configuration²⁶ is formed in three dimensions by making copies of all living cells, $(x_i, y_i, 0)$ into the adjacent z -plane, $(x_i, y_i, 1)$. Thus, the expansion of a 2-dimensional cell configuration will have exactly twice as many cells as the original object. The behaviour of the expansion will depend on the rule set employed by the automaton.

The expansion of one such 2-dimensional cell cluster is shown in Figure 5.1.1.



Figure 5.1.1 – Expanding 2-dimensional objects to 3-dimensions. The image on the left shows a 2-dimensional Conway Life object. The image on the right shows the expansion of this object into the third dimension.

²⁶ We use the term *Conway object* or *Conway cell configuration* to denote any cell configuration in the 2-dimensional Game of Life described by $R = (2, 3, 3, 3)$.

A similar operation can be defined in order to obtain 2-dimensional objects from the 3-dimensional automaton: we say that a *projection* of a 3-dimensional cell configuration into two dimensions exists if and only if both of the following are true.

- i.) All of the living cells, (x_i, y_i, z_i) lie in two adjacent planes. For the sake of argument let these planes be $z = 0$ and $z = 1$.
- ii.) The pair of cells $(x_i, y_i, 0)$ and $(x_i, y_i, 1)$ are either both alive or both dead.

We may now define an *analogue* of a 2-dimensional Life object in three dimensions as an expansion which, when subjected to appropriate 3-dimensional rules, yields after each and every generation a projection identical to the original 2-dimensional cell configuration for the same generation under the 2-dimensional rule $R = (2, 3, 3, 3)$.

Theorem 5.1.1 (Bays, 1987)

A stable 2-dimensional Conway object has an analogue under $R = (4, 5, 5, 5)$ if and only if each living cell in the Conway object has exactly two neighbours.

The proof of this is obtained from Tables 5.1.1 and 5.1.2 below.

Conway Object		(4, 5, 5, 5) Expansion			
Number of neighbours, N, when cell at $(x_i, y_i, 0)$ is alive		Number of neighbours, N, of cells at $(x_i, y_i, 0)$ and $(x_i, y_i, 1)$		Number of neighbours, N, of cells at $(x_i, y_i, -1)$ and $(x_i, y_i, 2)$	
N	Next State	N	Next State	N	Next State
0	dead	1	dead	1	dead
1	dead	3	dead	2	dead
2	alive	5	alive	3	dead
3	alive	7	dead	4	dead
4	dead	9	dead	5	alive
5	dead	11	dead	6	dead
6	dead	13	dead	7	dead
7	dead	15	dead	8	dead
8	dead	17	dead	9	dead

Table 5.1.1 – Comparison of the number of neighbours and the status for live Conway cells and 3-dimensional Life (4, 5, 5, 5) cells. For example, suppose that a 2-dimensional cell is alive and has 2 live neighbours (the third row in the table). Then on the next timestep, the cell will live, as will the pair of cells in the (4, 5, 5, 5) expansion.

Conway Object		(4, 5, 5, 5) Expansion			
Number of neighbours, N, when cell at $(x_i, y_i, 0)$ is alive		Number of neighbours, N, of cells at $(x_i, y_i, 0)$ and $(x_i, y_i, 1)$		Number of neighbours, N, of cells at $(x_i, y_i, -1)$ and $(x_i, y_i, 2)$	
N	Next State	N	Next State	N	Next State
0	dead	0	dead	0	dead
1	dead	2	dead	1	dead
2	dead	4	dead	2	dead
3	alive	6	dead	3	dead
4	dead	8	dead	4	dead
5	dead	10	dead	5	alive
6	dead	12	dead	6	dead
7	dead	14	dead	7	dead
8	dead	16	dead	8	dead

Table 5.1.2 – Comparison of the number of neighbours and the status for dead Conway cells and 3-dimensional Life (4, 5, 5, 5) cells.

Now, in order for a Conway Life object to be stable, it is necessary that it does not die off after a number of timesteps. As we saw in Section 3.3.4, those live cells that have either one or no living neighbours fall into this doomed category, and so can be excluded from consideration. Similarly, objects that contain live cells with four or more neighbours tend to lose cells to form a stable object. Of the remaining possibilities only the case $N = 2$ provides us with consistent information – live cells that have three neighbours in the 2-dimensional Game of Life will live on the

subsequent timestep whereas similar cells will die in the 3-dimensional case. Similar inconsistencies arise in the table for dead cells.

The following corollary is immediate.

Corollary 5.1.2 (Bays, 1987)

A Conway object that changes from one generation to the next has no analogue under the 3-dimensional Life rule $R = (4, 5, 5, 5)$.

The consequence of this is that although $R = (4, 5, 5, 5)$ has an occasional analogous form (see, for example, Figure 5.1.2), such objects are rare and the game's behaviour is ultimately very different from the more familiar 2-dimensional game.



Figure 5.1.2 – 3-Dimensional analogue of a stable 4-cell object under $R = (4, 5, 5, 5)$

5.1.5 The rule $R = (5, 7, 6, 6)$

We saw in the previous section how the three-dimensional Game of Life characterised by the rule $(4, 5, 5, 5)$ supports only a very limited number of analogous forms of Conway objects. Here we prove a similar, though broader result.

Theorem 5.1.3 (Bays, 1987)

A Conway object has an analogue under the three dimensional Game of Life rule $(5, 7, 6, 6)$ if and only if the two-dimensional object satisfies the following conditions at every timestep:

- i.) A dead cell in the neighbourhood of the object cannot have six living neighbours.*
- ii.) A live cell cannot have five neighbours.*

We prove the result, as before, by examining the behaviour of live and dead cells under the two and the three-dimensional rule systems. The results are presented in Tables 5.1.3 and 5.1.4 below.

Conway Object		(5, 7, 6, 6) Expansion			
Number of neighbours, N, when cell at $(x_i, y_i, 0)$ is alive		Number of neighbours, N, of cells at $(x_i, y_i, 0)$ and $(x_i, y_i, 1)$		Number of neighbours, N, of cells at $(x_i, y_i, -1)$ and $(x_i, y_i, 2)$	
N	Next State	N	Next State	N	Next State
0	dead	1	dead	1	dead
1	dead	3	dead	2	dead
2	alive	5	alive	3	dead
3	alive	7	alive	4	dead
4	dead	9	dead	5	alive
5	dead	11	dead	6	alive
6	dead	13	dead	7	dead
7	dead	15	dead	8	dead
8	dead	17	dead	9	dead

Table 5.1.3 – Comparison of the number of neighbours and the status for live Conway cells and 3-dimensional Life (5, 7, 6, 6) cells.

Conway Object		(5, 7, 6, 6) Expansion			
Number of neighbours, N, when cell at $(x_i, y_i, 0)$ is alive		Number of neighbours, N, of cells at $(x_i, y_i, 0)$ and $(x_i, y_i, 1)$		Number of neighbours, N, of cells at $(x_i, y_i, -1)$ and $(x_i, y_i, 2)$	
N	Next State	N	Next State	N	Next State
0	dead	0	dead	0	dead
1	dead	2	dead	1	dead
2	dead	4	dead	2	dead
3	alive	6	alive	3	dead
4	dead	8	dead	4	dead
5	dead	10	dead	5	dead
6	dead	12	dead	6	alive
7	dead	14	dead	7	dead
8	dead	16	dead	8	dead

Table 5.1.4 – Comparison of the number of neighbours and the status for dead Conway cells and 3-dimensional Life (5, 7, 6, 6) cells.

Note here that the behaviour of the expansion in the planes $z = 0$ and $z = 1$ is identical to the two-dimensional game. Only the planes $z = -1$ and $z = 2$ affect the analogue, and it is these deviations that are addressed by the restrictions imposed by Theorem 5.1.3.

Upon examining a number of stable and oscillating two-dimensional Conway objects it becomes clear that many of them satisfy the conditions imposed by Theorem 5.1.3. Some examples are presented in Figures 5.1.3 – 5.1.5 below.

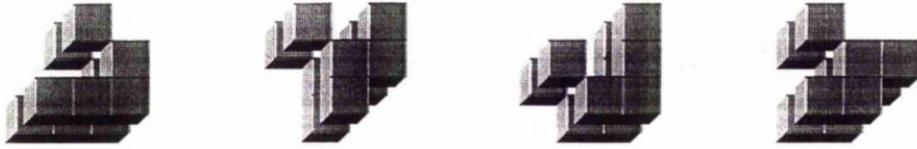


Figure 5.1.3 – 3-Dimensional analogue of the Conway glider under $R = (5, 7, 6, 6)$.



Figure 5.1.4 – 3-Dimensional analogue of a simple three-cell oscillator under $R = (5, 7, 6, 6)$.



Figure 5.1.5 – 3-Dimensional analogue of a four-cell stable object under $R = (5, 7, 6, 6)$.

It seems, therefore, that the 3-dimensional Game of Life characterised by the rule set $R = (5, 7, 6, 6)$ is by far the best candidate for our control system. Not only does this rule set produce interesting cell development that is worthy of the name life, it is also the 3-dimensional system that behaves most like the familiar 2-dimensional game that was utilised by the previous version of CAMUS. It is for this reason that the rule set $R = (5, 7, 6, 6)$ was chosen as the default rule set for CAMUS 3D, although the user is still free to chose whichever legal rule set best suits their compositional ideal.

However, as we shall see in the next section, it is possible to engineer Conway's life in three dimensions by imposing restrictions on the cell checking mechanism.

5.1.6 Conway's Life in 3 dimensions

It is possible to configure a 3-dimensional Game of Life to behave in the same way as Conway's 2-dimensional game. This is a highly desirable state of affairs, since it gives us the benefits of a 3-dimensional control system – namely the increased chord complexity – together with the ease of use and well-understood behaviour of the 2-dimensional Game of Life.

There are two methods that may be used to achieve this.

The first of these is to treat the 3-dimensional space as a series of parallel stacked 2-dimensional spaces. This can be done quite simply:

Suppose we wish to treat a 3-dimensional Game of Life as a series of 2-dimensional games stacked parallel to the plane $x = 0$. Then each of the stacked planes have the form $x = a$, where $a \in \mathbb{Z}$ is a constant. Thus, when we come to examine the neighbouring cells of an arbitrary cell (a, b, c) , where $a, b, c \in \mathbb{Z}$, we need restrict our attention only to the cells

$$(a, b + 1, c), (a, b - 1, c), (a, b, c + 1), (a, b, c - 1), \\ (a, b + 1, c + 1), (a, b + 1, c - 1), (a, b - 1, c + 1) \text{ and } (a, b - 1, c - 1),$$

since we are concerned only with the neighbouring cells in the plane $x = a$. This means that each of the stacked 2-dimensional games evolves in total isolation – none of the cells from the neighbouring games can exert any influence on the cells in any of the other games.

This configuration is illustrated in Figure 5.1.6 below.

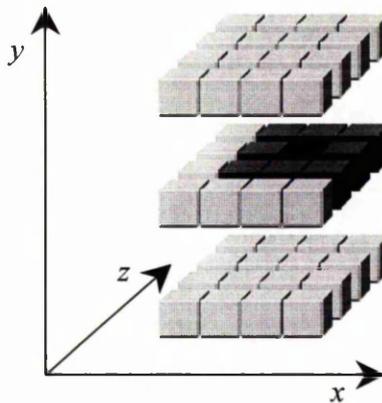


Figure 5.1.6 – A 3-dimensional Game of Life as a series of parallel stacked 2-dimensional games. The 2-dimensional games in this case are stacked parallel to the plane $y = 0$. The dark grey cell in the middle layer is currently under examination. Since we treat the y co-ordinate as a constant, only the shaded neighbouring cells in the same plane are also examined. Thus, each 2-dimensional game evolves in isolation.

The second option, proposed by Carter Bays ([Bays, 1987]) involves the construction of *time-space barriers* in the three dimensional space.

Time-space barriers are essentially stable planar structures in which each living cell has precisely seven living neighbours (see Figure 5.1.7). This has the effect of preventing any births in the two planes adjacent to the barrier – these planes can be thought of as being *barren*.

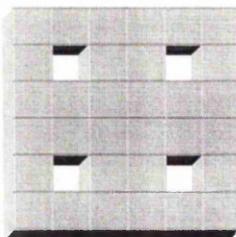


Figure 5.1.7 – A section of a time-space barrier. Note that the barrier extends infinitely in all directions and that each live cell has precisely 7 live neighbours.

Suppose we construct such a time-space barrier in the plane $z = 0$. Then this means that no glider approaching from a direction perpendicular to that plane can ever

penetrate the planes $z = \pm 1$. In effect the time-space barrier acts as a shield for its neighbouring planes.

To use the time-space barrier technique to force a three-dimensional Conway life, one proceeds in the following manner:

Firstly, two infinitely large²⁷ parallel time-space barriers are constructed and placed in the universe so that they are separated by four parallel planes (see Figure 5.1.8). Then, since the planes adjacent to the barriers are barren, all life must be confined to the two middle planes. However, since the three-dimensional analogue to Conway's life under the rule set $R = (5, 7, 6, 6)$ breaks down only because of growth in the z -direction (see Tables 5.1.3 and 5.1.4), and we have prevented such growth.

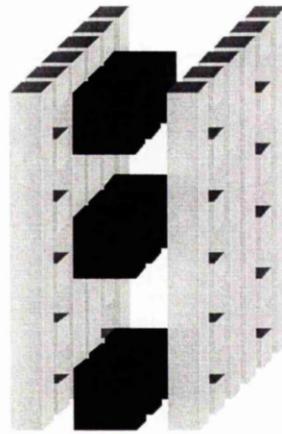


Figure 5.1.8 - Two infinitely large parallel time-space barriers – shown here using light-coloured cells – are constructed and placed in the universe so that they are separated by four planes. The planes immediately adjacent to the barriers are barren, so that all life is confined to the two middle planes, which here contain dark-coloured cells.

We conclude that:

²⁷ It is, of course, impossible to use truly infinite time-space barriers. However, since we are dealing with an automaton space that is toroidal in nature, the same effect may be achieved by extending the barriers to fill the plane in which they exist.

Theorem 5.1.4 (Bays, 1987)

An analogue to the entire 2-dimensional Game of Life exists between two infinite parallel time-space barriers separated by four planes under the rule set $R = (5, 7, 6, 6)$.

5.1.7 The Demon Cyclic Space in three dimensions

The construction of a 3-dimensional Demon Cyclic Space is very similar to the construction of a 3-dimensional Game of Life, albeit with a much simpler rule set.

Recall that the Demon Cyclic Space is a k -state $l \times m$ array of cells whose evolution is determined by the following rule:

A cell which is in state j at timestep t will dominate any neighbouring cells which are in state $j - 1$, so that they increase their state to j at timestep $t + 1$.

We then define the 3-dimensional extension of the Demon Cyclic Space to be a k -state $l \times m \times n$ array of cells whose evolution is determined by the same rule.

However, as with the two-dimensional Demon Cyclic Space, only a subset of each cell's neighbours is actually examined at each timestep.

Recall that in two dimensions a cell (x, y) requires only the cells $(x \pm 1, y)$ and $(x, y \pm 1)$ to be examined.

By extension, a cell (x, y, z) in the 3-dimensional case necessitates the checking only of cells $(x \pm 1, y, z)$, $(x, y \pm 1, z)$ and $(x, y, z \pm 1)$.

As with the 2-dimensional CAMUS system, CAMUS 3D offers scope to alter the evolution rules of the control automata in order to fine-tune the system. This is discussed more fully in Section 5.2.4.

5.1.8 The emergent behaviour of the 3-dimensional Demon Cyclic Space

Due to the higher dimensionality of the automaton, spotting long-term behaviour in the cells of the 3-dimensional Demon Cyclic Space is not an easy task. However, by examining two-dimensional cross-sections of the system we may hope to discover the

emergence of self-organising behaviour similar to that seen in the 2-dimensional case. An example is shown in Figure 5.1.9 below.

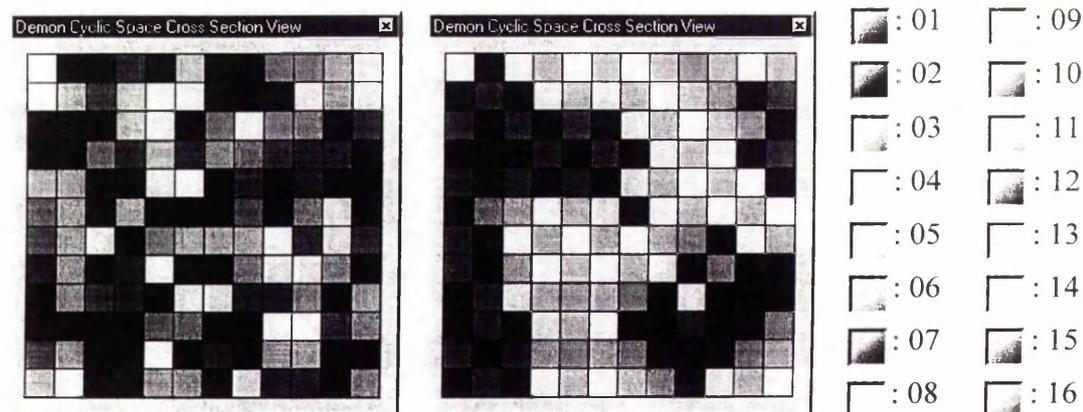


Figure 5.1.9 – Self-organising behaviour in the x-y plane of the 3-dimensional Demon Cyclic Space. The image on the left is the initially randomised automaton. The image on the right shows the automaton a number of timesteps later.

Note that the pattern is not as clearly defined as in the 2-dimensional case. Nevertheless, such self-organising behaviour clearly exists and should be noted. The existence of such behaviour, however, is not critical, and by no means as important as the behaviour of the 3-dimensional Game of Life, since the states of the Demon Cyclic Space cells determine only the instruments that perform sections of music and not the music itself.

5.2 Mapping the control system to music

Now, having seen how the underlying control system has been extended to three dimensions, we describe how they are used to generate CAMUS 3D's musical output.

5.2.1 The mapping

To begin the composition process, the Game of Life automaton is initialised with a starting cell configuration, the Demon Cyclic Space automaton is initialised with random states, and both are set to run.

At each timestep, the co-ordinates of each live cell are analysed and used to determine a 4-note chord²⁸ that will be played during the corresponding temporal window in the composition. The state of the corresponding cell of the Demon Cyclic Space automaton is used to determine the instrumentation of the piece.

This configuration is demonstrated in Figure 5.2.1. In this case, the cell in the Game of Life at (5, 5, 2) is alive, and thus constitutes a musical event.

The co-ordinates (5, 5, 2) describe the intervals in a 4-note chord: The x cell-position (starting at 0 in the bottom left corner) defines a semitone interval from a fundamental pitch to the second-lowest pitch of the chord. The y -cell position defines a semitone interval from the second-lowest to the second-highest pitch in the chord. The z -cell position defines a semitone interval from the second-highest pitch to the top note of the chord. Note that if the cell position is 0 (corresponding to the first cell in each direction) the 'higher' pitch defined by the associated interval will be identical to the 'lower' pitch.

The corresponding cell in the Demon Cyclic Space is in state 4, which means that the sonic event would be played by instrument number 4 (for example, by using MIDI channel 4). Note that for the sake of clarity, the first two layers of the Demon Cyclic Space have been omitted in the figure below.

²⁸ A four-note chord is a set of four (not necessarily distinct) notes that may or may not sound simultaneously.

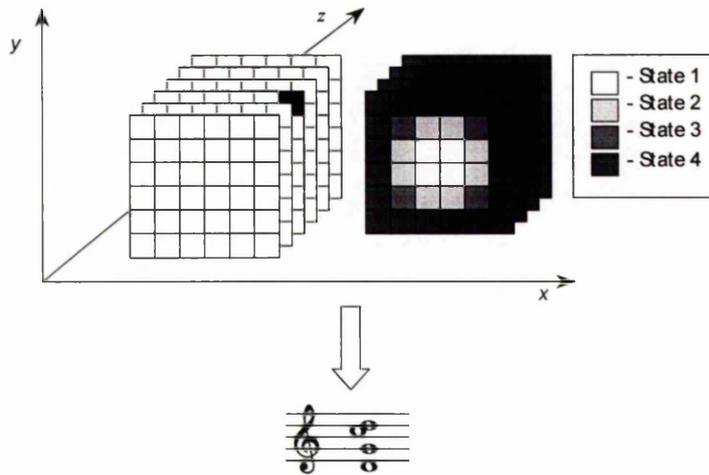


Figure 5.2.1 – Configuration of a typical timestep in CAMUS 3D.

The use of a discrete 3-dimensional Euclidean space to represent musical intervals is an extension of the 2-dimensional von Neumann Music Space used in an earlier version of the system ([Miranda 1993], [Miranda, 1994]).

Having established the intervallic content of the chord associated with a live cell, we must establish the fundamental note in order to specify fully each of the pitches in the chord. This is done automatically using stochastic selection routines, which allow the composer to specify the relative weightings of the pitch values for the fundamental pitches (see Section 5.2.6).

Now, in order to avoid a piece being composed entirely of block chords, we must implement a routine that staggers the starting (and possibly ending) times of each of the notes of the chord.

The temporal structure of each of the chords is represented by a 4×4 grid. The bottom row of boxes represents the lowest pitch of the chord, the second bottom row represents the second lowest note, the second top row represents the second highest pitch and the top row represents the highest pitch. The order of the shaded cells from left-to-right determines the order in which each of the notes is played.

It is a simple matter to calculate that there are 24 different ways of arranging the starting order of these 4 (non-simultaneous) notes. CAMUS 3D uses a stochastic selection routine that consults a user-specified table for the associated probabilities of

each of the 24 possible starting arrangements (see Figure 5.2.2). The routine used to select the note order is a variation of the non-uniform cumulative probability selection routine detailed in Section 2.1.3.

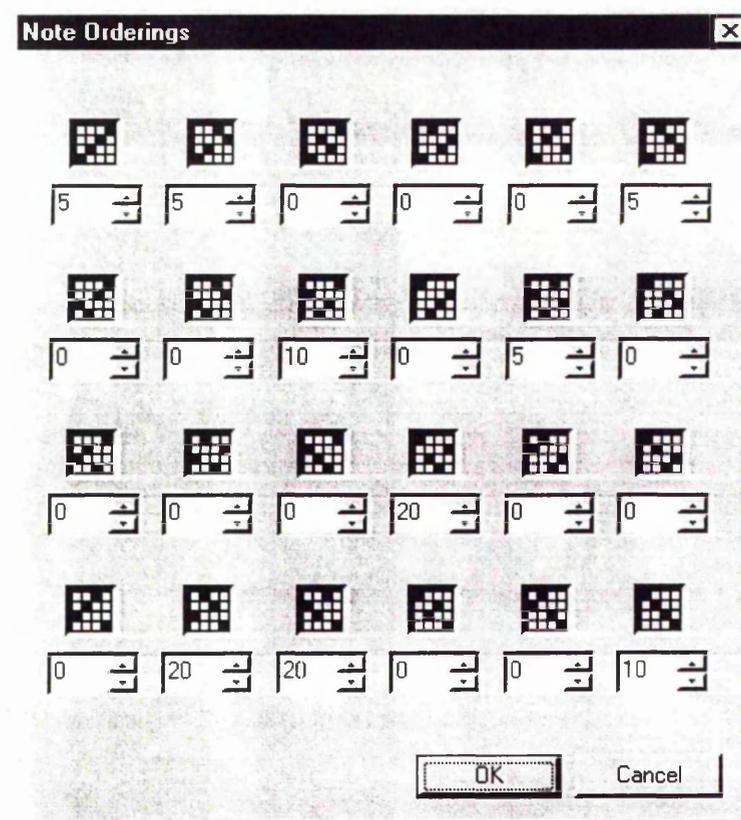


Figure 5.2.2 – The Note Orderings dialog box.

When a starting arrangement has been chosen, CAMUS 3D calculates the precise note durations. This is achieved by means of a first order Markov chain (see Section 5.2.5).

The probabilities in the transition matrix of the Markov chain are, again, user-specified. Note lengths are quantised to semiquaver resolution, and simultaneous note events are catered for by allowing starting times of duration 0.

CAMUS 3D offers two methods of specifying the probabilities in the Markov state-transition matrix. The standard option offers the composer a graphical means of viewing and altering the probability values. The probabilities are represented on screen by a graduated colour scheme which ranges from pure red (probability value 0) to pure green (probability value 1). This scheme is very natural to use, tying in as it

does with the natural colour schemes of the physical world: red is often signifies a warning or danger (impossible), whilst green indicates safety (certain).

When the composition process is started, the music is performed in real time, and can be saved as a type 0 standard MIDI file, fixing the composition for all time.

CAMUS 3D also allows the user to save the composition as a set of parameters that correspond to the states of the automata and the probability tables for the selection routines. Whilst this allows the composer to re-create the 'same' composition, the resulting music may sound quite different, since although the automata are wholly deterministic, and so produce identical chord sequences and instrumentations each time they run with identical initial configurations, the stochastic selection routines may lead to quite different fundamental pitches, note orderings and note durations²⁹.

This is analogous to the performance of a given piece of music by a musician – no two performances of the same piece will be identical. Indeed, it is likely that over time, the performance will change dramatically as the musician reinterprets the work. Of course, if a digital recording is made on DAT tape, for example, then the performance is fixed for all time. This is exactly the case when saving a CAMUS composition as a parameter set and as a MIDI file – the parameter set records the musical score which the program will interpret in a different manner each time it performs the composition, whilst the MIDI file records the musical performance and will not change.

5.2.2 Data entry in the third dimension

As was mentioned in Section 4.1.2, two methods of data entry were initially proposed for CAMUS 3D.

²⁹ In fact, the random number generators used by computers are also deterministic and only simulate randomness. Thus, a random number generator will produce identical sequences for any given seed. This restriction is overcome in CAMUS 3D by using the state of the internal clock on the host PC as the seed value for the initialisation of the generator. Since this value changes as time progresses, the probability that the random number generator will produce an identical sequence of numbers on two successive iterations is quite negligible.

The first of these involved the direct entry of co-ordinates using a tabular system, whilst the second allowed the user to click on a series of embeddings using the mouse.

Upon experimentation with a preliminary system, however, it was discovered that the method of clicking on two-dimensional embeddings was cumbersome and often confusing for the user. This was then dropped from the final software and the co-ordinate system was developed.

In order to achieve a user-friendly input method from the co-ordinate system it was decided that the 3D modelling analogy would be developed.

3D modelling packages such as Autodesk's *3D Studio Max* ([Autodesk, 1999]) often offer preset shapes and objects that may be placed in 3-dimensional space and tailored to the user's needs. These shapes are often referred to as *primitives*. Some common examples of primitives are cuboids, ellipsoids, toruses and pyramids.

CAMUS 3D uses a similar system of primitives, where the cellular automata primitives are preset clusters of cells which can be positioned at any point in the 3-dimensional automaton space. The primitives available to the user are *Invert Cell*, *Line* and *Cube*.

The *Invert Cell* function displays the *Invert Cell* dialog box (see Figure 5.2.3) which allows the user to invert the state of the cell at position (x, y, z) .

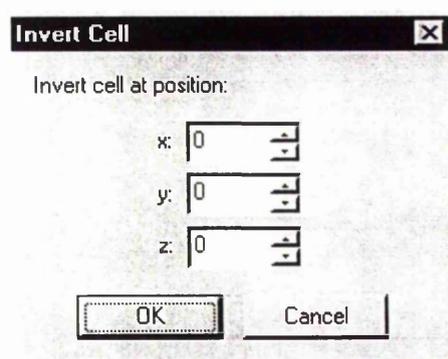


Figure 5.2.3 – The *Invert Cell* dialog box.

The routine used for inverting the cell (x, y, z) is presented below. Note that although for reasons of consistency we term the procedure an algorithm, we concede that this is perhaps too ostentatious a term for a single instruction!

```

procedure InvertCell(x, y, z)
{
    cell(x, y, z) = NOT cell(x, y, z);
}

```

Algorithm 5.2.1 – Invert cell.

The *Line* function displays the *Line* dialog box (see Figure 5.2.4), which allows the user to invert the states of those cells that lie on the unique line between the cells (a, b, c) and (x, y, z) .

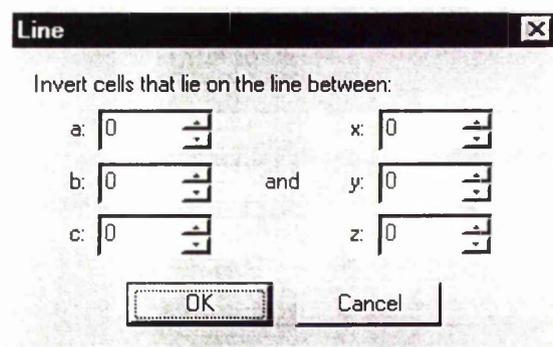


Figure 5.2.4 – The Line dialog box.

It should be noted, however, that since the Game of Life space is discrete and quite small ($12 \times 12 \times 12$), the quantisation of the line may cause it to look blocky and, in extreme cases, almost like a curve.

The algorithm used to ‘draw’ the line is based on Bresenham’s Algorithm ([Burger & Gillies, 1989]).

Suppose we wish to draw a line between the points (x_1, y_1) and (x_2, y_2) . We may assume without any loss of generality that $x_1 \leq x_2$. We may also assume that the error at point (x_1, y_1) is 0, that is, the line passes through the initial point.

The essence of the algorithm is that we step through the x co-ordinates from x_1 to x_2 , calculating the error made by not drawing the line at its true position. At each step a

decision is made as to which of the two points that bound the real position of the line at that x ordinate is the closest and that pixel is set to ON.

For our 3-dimensional case, however, we need a slightly different approach.

Here we wish to determine which cells in our 3-dimensional Game of Life lie on, or rather closest to the line between (a, b, c) and (x, y, z) . Again, we may assume that the line passes through the point (a, b, c) .

Firstly, we calculate the direction numbers of the line, l . These are

$$x_l = x - a$$

$$y_l = y - b$$

$$z_l = z - c$$

Starting at live cell (a, b, c) , we move in a direction (x_l, y_l, z_l) until we leave the cell. We now calculate the error made by not drawing the line at its true position as with Bresenham's Algorithm, and invert the state of the cell which minimises the error.

We continue in this manner until we reach the point (x, y, z) .

The algorithm is presented below.

```

procedure Line(a,b,c,x,y,z)
{
    xl = x - a;
    yl = y - b;
    zl = z - c;
    repeat until cell (x,y,z) is reached
    {
        while (still within previous cell)
            move in direction (xl,yl,zl);
        for (all cells immediately neighbouring the true line
            position)
            e(i) = distance from neighbour i to true
            position;
        next neighbour
        (x',y',z') = coordinates of cell with min(e(i));
        InvertCell(x',y',z');
    }
}

```

Algorithm 5.2.2 – The 3-dimensional line algorithm.

The *Cube* function displays the *Cube* dialog box (see Figure 5.2.5), which allows the user to invert the states of those cells that lie within a cube with sides of length l cells, whose top-left-nearmost corner is given by the cell (x, y, z) .

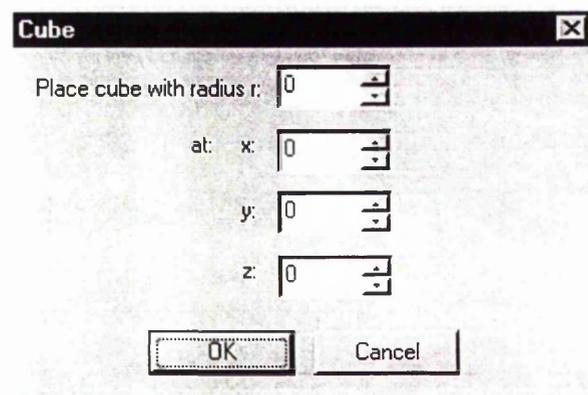


Figure 5.2.5 – The Cube dialog box.

The algorithm used to create the cube is shown in Algorithm 5.2.3 below.

```

procedure Cube(r, x, y, z)
{
    // We assume that the automaton is of dimension
    // (Size × Size × Size)
    for i = x to (x + r)
    {
        x' = i mod Size;
        for j = y to (y + r)
        {
            y' = j mod Size;
            for k = z to (z + r)
            {
                z' = k mod Size;
                InvertCell(x', y', z');
            }
            next k;
        }
        next j;
    }
    next i;
}

```

Algorithm 5.2.3 – The Cube Algorithm.

From these three basic tools, a wide variety of objects can be produced.

For example, a hollow cube, that is a cube whose boundary cells are live and whose interior cells are dead, may be quickly constructed in the 3-dimensional Game of Life by splitting the cube into a live boundary and a dead interior. The cells that lie in the interior of the cube are brought to life first by an application of the cube primitive. Then, the larger boundary is placed on top, bringing those cells that lie on the boundary to life, and killing those that lie within the interior.

Other tools offered to the user are *Clear*, *Randomise* and *Extend*.

The *Clear* function kills all cells in the Game of Life, essentially providing a blank score on which the user can work. The routine for accomplishing this is very simple and is identical to Algorithm 5.2.1 with the line

```
cell(x, y, z) = NOT cell(x, y, z);
```

replaced by

```
cell(x, y, z) = FALSE;
```

Two *Randomise* functions are available. The first randomly assigns each cell in the 3-dimensional Game of Life a state, either alive or dead. The second randomly assigns each cell in the 3-dimensional Demon Cyclic Space a state $0, 1, \dots, j - 1$, where j is the total number of states of the automaton.

Again, these algorithms are very simple. For the former we have:

```
procedure RandomiseGOLCell(x, y, z)
{
    i = rand();
    if (i mod 2) then
        cell(x, y, z) = FALSE;
    else
        cell(x, y, z) = TRUE;
}
```

Algorithm 5.2.4 – The Randomise Game of Life Cell Algorithm.

Algorithm 5.2.5 below shows how the latter may be achieved.

```
procedure RandomiseDCSCell(x, y, z)
{
    i = rand();
    cell(x, y, z) = i mod j;
}
```

Algorithm 5.2.5 – The Randomise Demon Cyclic Space Cell Algorithm.

The *Extend* function displays the *Extend* dialog box (see Figure 5.2.6), which copies all cells in a user-specified plane to the adjacent plane in the positive direction, thus creating an expansion of all objects in that plane.

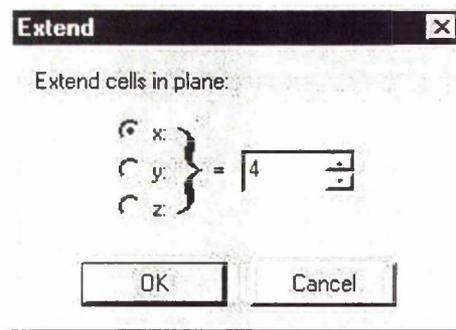


Figure 5.2.6 – The Extend dialog box.

We also intend to support a number of common Conway objects, either as a library of shapes or as further primitives.

5.2.3 The display

As was mentioned in Section 4.1.2, displaying data from a 3-dimensional control system on a 2-dimensional computer screen presents us with some problems.

An isometric display gives the user the most complete picture of the current state of the automaton, although it is accepted that it can be difficult to obtain a sense of depth in this way, and that occasionally some live cells may be obscured by those in front.

The problem of depth perception can be overcome by displaying the 3-dimensional automaton as a series of embeddings. When used in conjunction with the isometric view these provide the user with sufficient visual cues to complete the three dimensional picture in his or her own mind.

Also, by using certain graphics modes we may overcome the problem of obscured cells. In order to do this, a graphics transparency feature offered by the Windows operating system was utilised. This allows live cells to be shaded in such a way that their colours are translucent, like coloured glass, meaning that cells that would otherwise be hidden may be viewed through the obscuring cells.

Unfortunately, this approach is not suitable for the Demon Cyclic Space in which all cells must be visible simultaneously. The reason for this is that even with translucent colouring, cells which have one or two layers in front of them quickly become

obscured, resulting in a large dark mass in the centre of the automaton (see Figure 5.2.7).

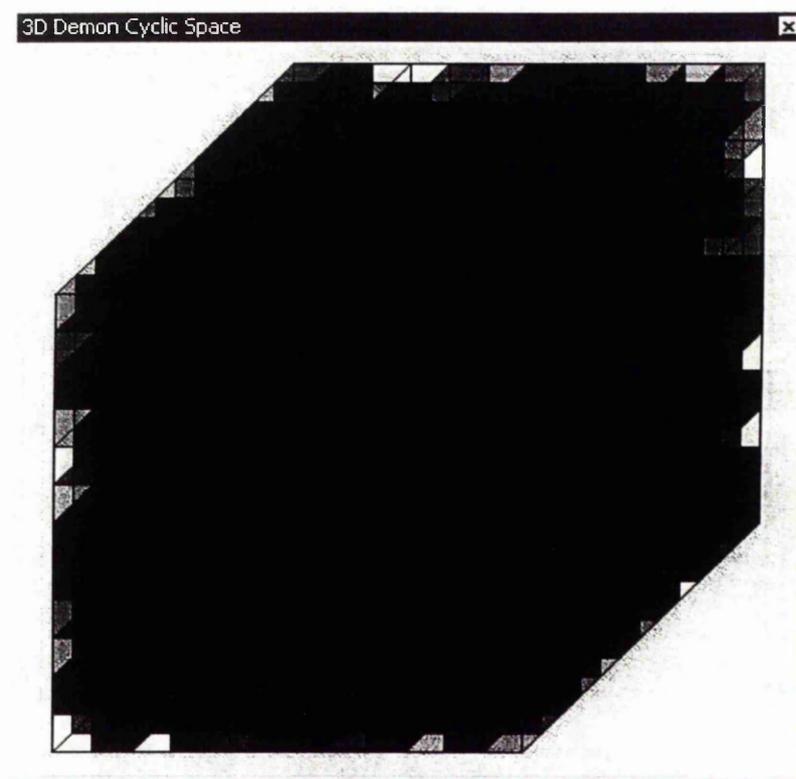


Figure 5.2.7 – Cells in the 3-dimensional Demon Cyclic Space very quickly become obscured by those in front.

However, it is fair to say that a great deal of the information stored within the Demon Cyclic Space is redundant, since the states of cells are used by the composition system only when there is a corresponding live cell in the 3-dimensional Game of Life. Therefore, it was decided that the 3-dimensional Demon Cyclic Space data should be incorporated into the 3-dimensional Game of Life display. The states of the Demon Cyclic Space cells are then displayed only when there is a corresponding live cell in the Game of Life, and this is achieved by colouring the live Game of Life cell with the colour associated with the relevant state of the Demon Cyclic Space.

Thus, the user is presented with an isometric display that incorporates the Demon Cyclic Space as the main visual feedback (see Figure 5.2.8), but may also open any of the three available embeddings (see Figures 5.2.9 – 5.2.11) for a plan, elevation or end-elevation view.

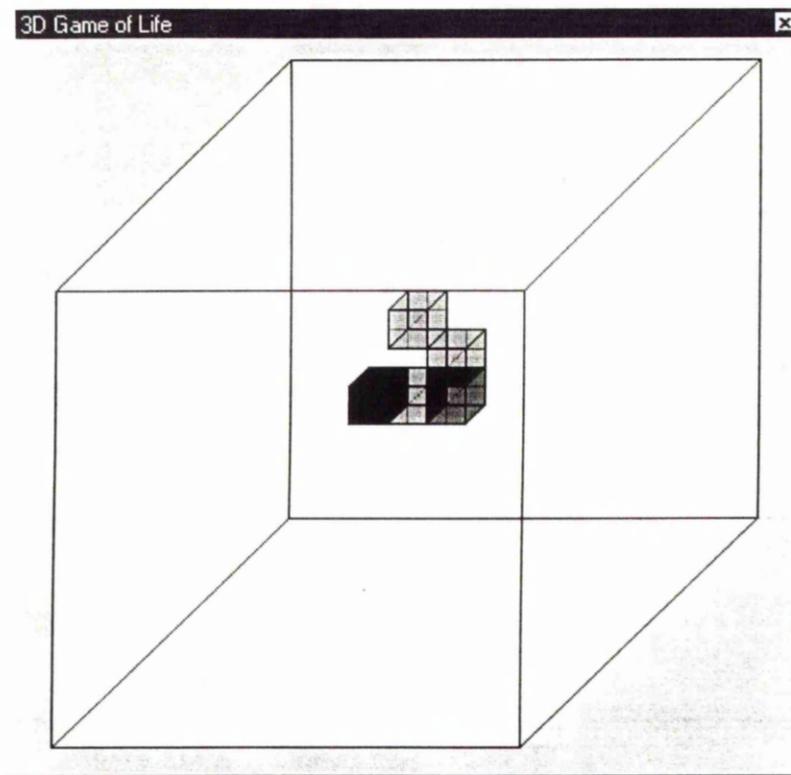


Figure 5.2.8 – The isometric view of the 3-dimensional Game of Life.

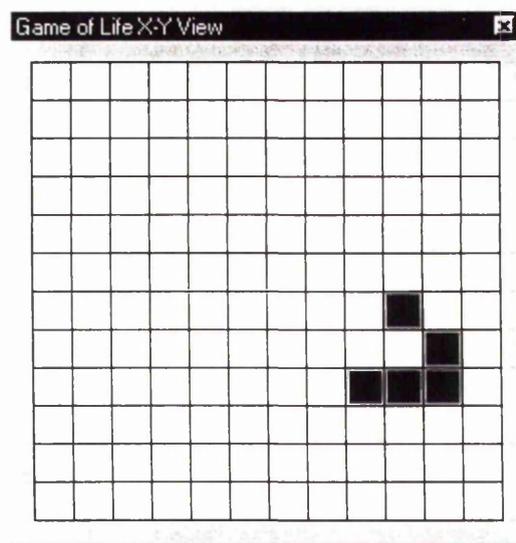


Figure 5.2.9 – The x-y plane view of the 3-dimensional Game of Life.

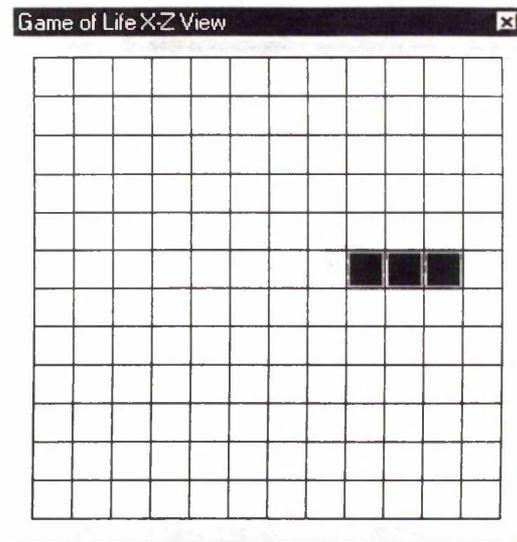


Figure 5.2.10 – The x-z plane view of the 3-dimensional Game of Life.

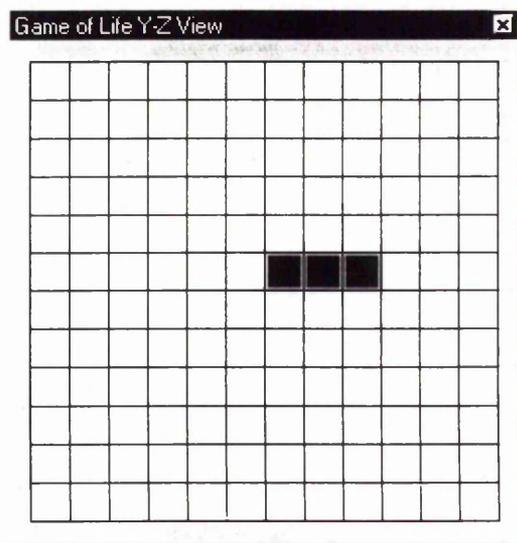


Figure 5.2.11 – The y-z plane view of the 3-dimensional Game of Life.

5.2.4 – Altering the evolution rules

The development of each composition created using CAMUS depends to a great extent on the evolution of the two automata that control the system.

In order to provide flexibility within the system, CAMUS allows the user to alter the evolution rules for both the 3-dimensional Game of Life and the 3-dimensional Demon Cyclic Space.

If the user wishes to alter these rules, he or she is presented with the relevant dialog box that presents all user-changeable parameters.

The *Alter Game of Life Rules* dialog box (see Figure 5.2.12) allows the user to customise the way in which the 3-dimensional Game of Life evolves.

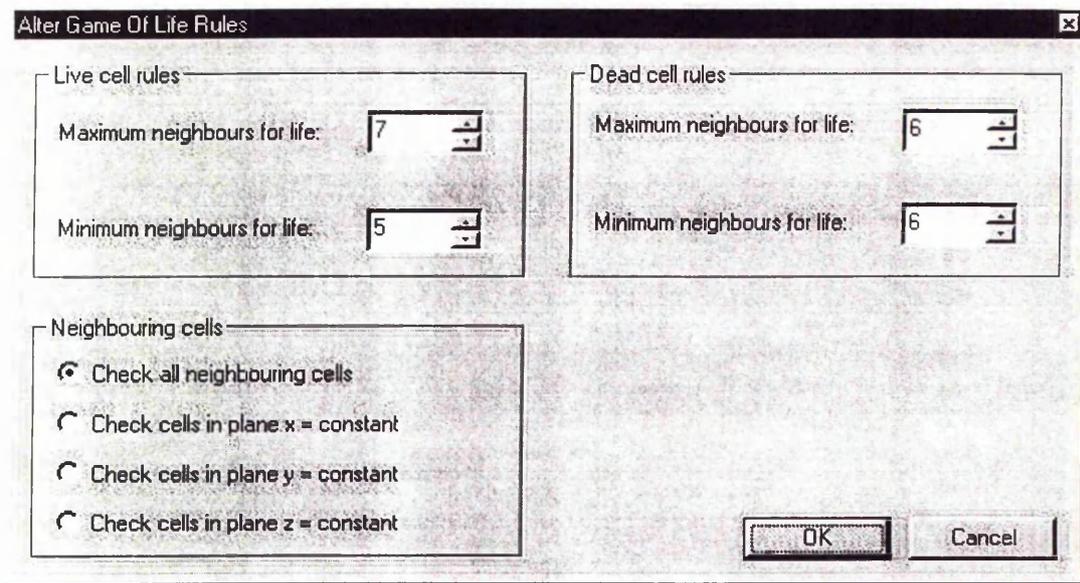


Figure 5.2.12 – The *Alter Game of Life Rules* dialog box.

The parameters affect the system as follows:

The *Live Cell Rules* group consists of all parameters that have a direct influence on the evolution of live cells.

Maximum neighbours for life is the largest number of live neighbours that a live cell may have in order to live on the next timestep.

Minimum neighbours for life is the smallest number of live neighbours that a live cell may have in order to live on the next timestep.

The *Dead Cell Rules* group consists of all parameters that have a direct influence on the evolution of dead cells.

Maximum neighbours for life is the largest number of live neighbours that a dead cell may have in order to live on the next timestep.

Minimum neighbours for life is the smallest number of live neighbours that a dead cell may have in order to live on the next timestep.

The *Neighbouring cells* group allows the user to specify how the 3-dimensional automaton is treated for cell-checking purposes.

If *Check all neighbouring cells* is selected, the automaton will behave like a true 3-dimensional space and all neighbouring cells will be counted at each timestep.

If *Check cells in plane $x = \text{constant}$* is selected, the automaton will behave like a series of stacked 2-dimensional spaces and only neighbours that lie in the plane parallel to the yz -plane will be counted at each timestep.

The remaining two options cause the automaton to behave in a similar manner.

The *Alter Demon Cyclic Space Rules* dialog box (see Figure 5.2.13) allows the user to customise the way in which the 3-dimensional Game of Life evolves.

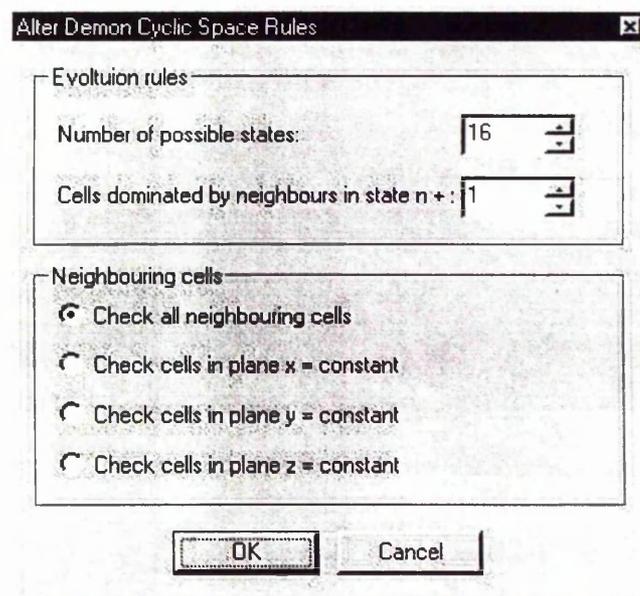


Figure 5.2.13 – The *Alter Demon Cyclic Space Rules* dialog box.

The parameters affect the system as follows:

The *Evolution rules* group consists of all parameters that directly alter the evolution of the automaton.

Number of possible states is the total number of states in which each cell can exist.

Cells dominated by neighbours in state $n +$ allows the user to specify the domination rule, so that cells may be influenced by neighbours whose state is greater than the one under examination by this value, working modulo the total number of states.

The *Neighbouring Cells* group behaves exactly as for the 3-dimensional Game of Life.

5.2.5 – Choosing rhythms in CAMUS 3D

As mentioned earlier, the use of random generators for rhythm in CAMUS produced irregular rhythms that were unnatural and often difficult to listen to. The main reason for this is, of course, that the human ear welcomes regularity. Random note durations merely serve to confuse the brain as it looks for rhythmic patterns and fails to find them.

However, there is always a danger that a rhythm generator will go too far in the opposite direction and impose a stifling regularity on the rhythms it generates. Such music tends to be very dull and uninteresting and should be avoided if we are to succeed in providing composers with a useful compositional tool.

In order to solve the problem, then, we require an easy-to-use system of rhythm generation that provides us with semi-regular rhythms. Thankfully, such a system exists and is known as a *Markov Chain*.

We discussed Markov Chains in Section 2.1.4. Recall that a Markov Chain may be partitioned into equivalence classes known as a *recurrent classes* and *transient classes*. It can be shown that every finite Markov chain consists of at least one recurrent class, which, once reached can never be left, and some number (possibly none) of transient classes.

There are a number of musical applications to which Markov Chains may be applied (see, for example [Ames, 1989] and [Jones, 1980]). However, it is the author's belief that Markov processes are particularly well suited for rhythm selection. Rhythmic lines often exhibit semi-cyclic behaviour in that short phrases are often repeated

exactly or slightly altered as the line progresses – the human ear tends to like this sort of regularity. Similar behaviour can be engineered by careful manipulation of the recurrent and transient classes within the Markov chain.

CAMUS 3D utilises a first order Markov Chain in the generation of rhythms. The probabilities are entered using the *Note Lengths* dialog box (see Figure 5.2.14).

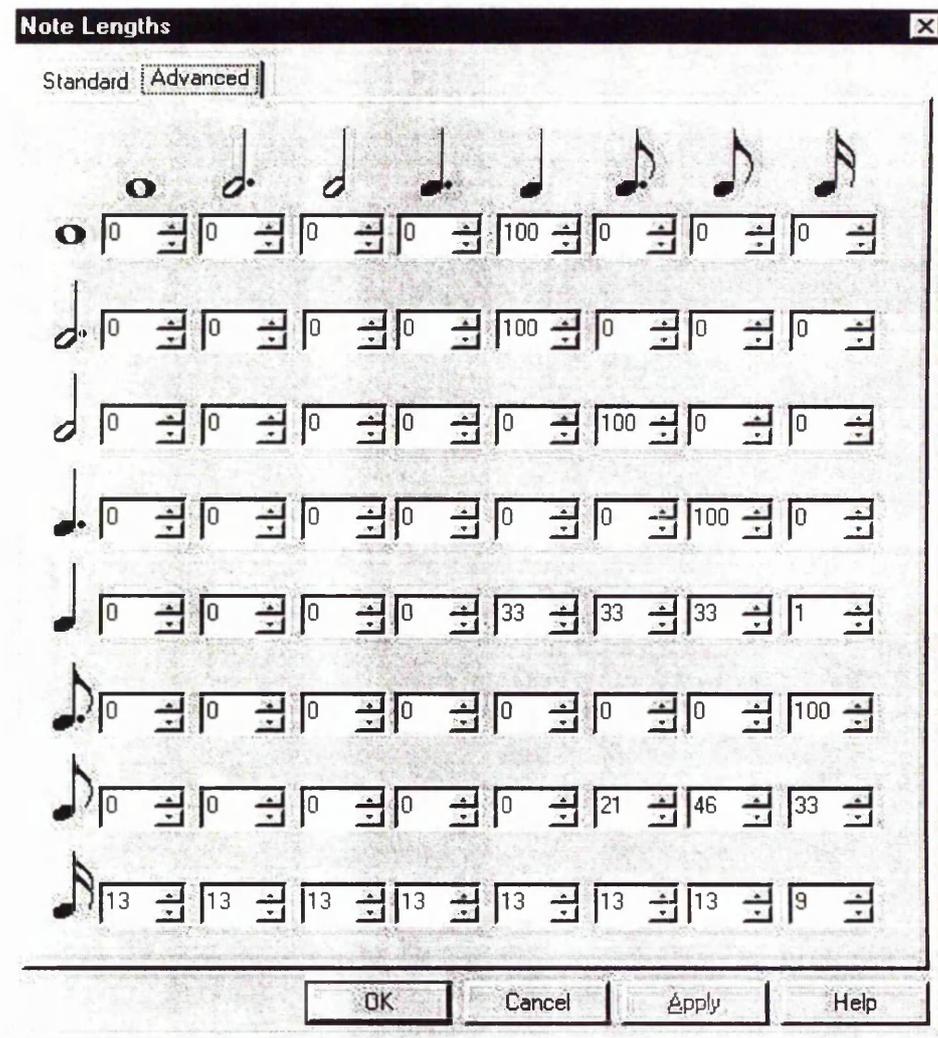


Figure 5.2.14 – The Note Lengths dialog box.

The column of notes down the left-hand side of the screen indicates the current note value. The row of notes along the top indicates the next note in the rhythm with the probability of transition given by the cell indexed by the relevant note values.

For example, in Figure 5.2.14 above, the probability of a dotted crotchet being followed by a quaver is found by looking down the column of notes until the dotted crotchet is reached, and then by moving along this row to the quaver column. It can be clearly seen that the probability of this transition is 100 – i.e. it occurs with absolute certainty.

The algorithm used by CAMUS 3D in the generation of rhythms is illustrated in Algorithm 5.2.6 below.

```
procedure GenerateRhythm()
{
    check value of current note
    go to relevant row of table
    x = rand(0, 100)
    //Here, P(i) is the probability that
    //the current note will be followed
    //the note indexed by i.
    interval = P(0)
    i = 0
    while (x < interval)
    {
        interval = interval + P(i)
        i = i + 1
    }
    current note = note value indexed by i
    return current note
}
```

Algorithm 5.2.6 – The note generation algorithm employed by CAMUS 3D.

In Figure 5.2.14 the reader will notice that there are two tabs marked *Standard* and *Advanced* along the top of the Note Lengths dialog box. These allow the user to alter the way in which data are represented in the dialog box.

The standard option (see Figure 5.2.15) offers the composer a graphical means of viewing and altering the probability values. The probabilities are represented on screen by a gradated colour scheme which ranges from pure red (probability value 0) to pure green (probability value 1). This scheme is very natural to use, tying in as it

does with the natural colour schemes of the physical world: red often signifies a warning or danger (impossible), whilst green indicates safety (certain).

The advanced option displays all data as numerical values and allows for the direct entry of probabilities via the numeric keypad.

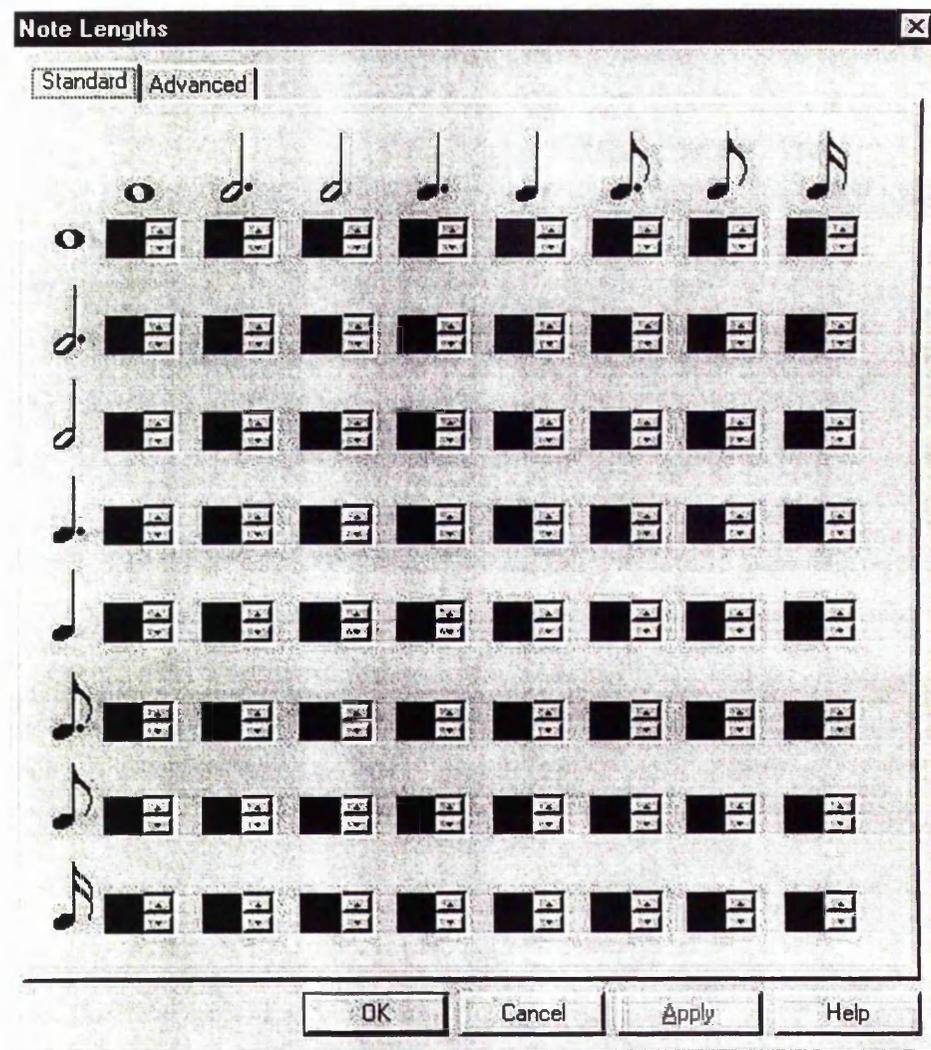


Figure 5.2.15 – The Standard Note Lengths dialog box.

5.2.6 – Pitch generation

Recall from Section 3.1.6 that the original CAMUS relied on user-specified lists of notes to determine fully the triads that were performed at each step of the composition.

This system worked reasonably well, particularly since the user had the ability to control to a very great extent the tonality (or atonality) of the resulting composition. This was demonstrated in the ‘reverse engineering’ method of composition described in Section 3.3.7

However, this system is not ideal. For complex or very long lines it is extremely tedious to configure the large-number of articulations required to give the desired effect, and practical experience with the system has indicated that users are far more likely to become bored and give up rather than to persevere to the end. Indeed, it seems that this hindrance of interface negates many of the positive aspects of the system and contributes greatly to its limited yield.

One possible solution is the use of probability tables to select starting notes for the chords.

As with Markov Chains, probability tables allow the user to impose long-term trends on the music without having to spend a great deal of time specifying individual notes. Probability tables are particularly well-suited for this task because it is simple for the user to emphasise the important notes in the composition by assigning them higher probability values, whilst limiting or excluding the other notes by assigning them small probability values or 0. In addition, the nature of probability tables is such that improbable results are often generated in places that would not have been expected by the user, thus taking the composition in a new direction.

The probabilities for the note generation table (a 0th order Markov Chain) are entered via the *Pitches* dialog box, shown in Figure 5.2.16 below.

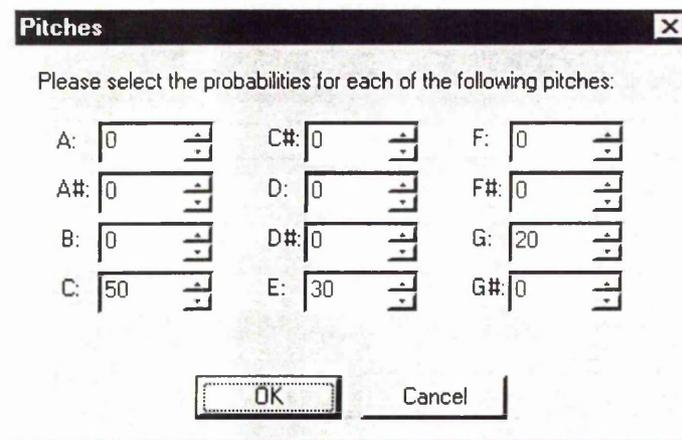


Figure 5.2.16 – The Pitches dialog box.

The probability values are entered as integers and are automatically normalised to the range [0, 100].

The algorithm used by CAMUS 3D in the generation of pitches is illustrated in Algorithm 5.2.7 below.

```

procedure GenerateRhythm()
{
    x = rand(0, 100)
    //Here, P(i) is the probability that
    //the pitch indexed by i will be played.
    interval = P(0)
    i = 0
    while (x < interval)
    {
        interval = interval + P(i)
        i = i + 1
    }
    pitch = pitch value indexed by i
    return pitch
}

```

Algorithm 5.2.7 – The note generation algorithm employed by CAMUS 3D.

It is important to note that this system only returns information on the *pitch class* of the note that will be played. In other words, the algorithm only determines that an A,

G or Bb will be played, not *which* A, G or Bb. For this we require a further algorithm that will specify an octave value for the pitch class, thus tying the root of the chord to a precise note.

At present, the octave generator for CAMUS 3D is very rudimentary and does not produce musically satisfactory results. It works by adding a randomly selected octave value to a predefined base octave, below which no notes can be generated. The problem with this routine is that although it indeed generates music that covers a range of octaves, the resulting music ‘jumps around’ far too much for it to be valid.

Possible solutions may include the use of a pink noise generator to control the leaping motion, since this would limit the frequency of large jumps, or the use of higher order Markov Chains, which would take into account the number of large jumps in the recent past when calculating the next octave value.

Further development of this routine, presented in Algorithm 5.2.8 below, is planned for future work.

```
procedure GenerateOctave(pitch, range)
{
    octave = rand() mod range;
    base_line = 45;
    note = 45 + 12*octave + pitch;

    return Note;
}
```

Algorithm 5.2.8 – The octave generation algorithm employed by CAMUS 3D.

5.2.7 – Instrumentation

The final stage in the composition process is the initialisation of the instrumentation.

This is achieved by means of the *Instrumentation* dialog box, as shown in Figure 5.2.17 below.

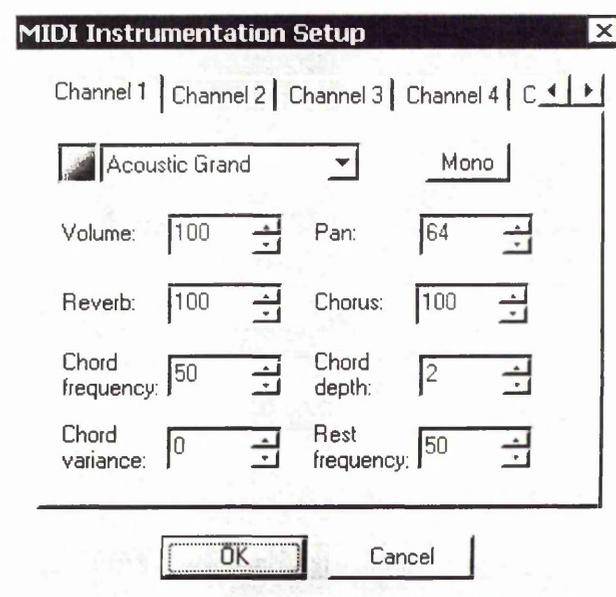


Figure 5.2.17 – The instrumentation dialog box.

Along the top of the dialog box are a row of tabs labelled *Channel 1*, *Channel 2*, ..., *Channel 16*. Clicking on tab *i* displays the parameter set corresponding to state *i* of the Demon Cyclic Space. The composition data generated by the system are then transmitted on MIDI channel *i*.

The coloured box in the top-left corner of the dialog box shows the colour of the corresponding state of the Demon Cyclic Space.

The text box to the immediate right of the coloured box allows the user to select the General MIDI instrument that will perform the music transmitted on that channel. For a complete list of the General MIDI instrument set see Appendix C.

The *Mono* button in the top-right corner of the dialog box allows the user to specify whether the music will be generated monophonically – that is, one note at a time – or homophonically – that is, using block chord movement. The default setting, Mono off, is indicated by the button being in its unchecked state as shown in Figure 5.2.17 above.

The *Volume* parameter allows the user to specify the overall level of the music that will be transmitted on the corresponding MIDI channel. Acceptable values range from 0 to 127 in standard MIDI volume units. This feature may be used as a simple MIDI

mixer in order to balance the levels of the various MIDI channels used in a CAMUS composition.

Pan allows the user to position the MIDI channel anywhere in the sonic panorama, from 0, denoting hard left positioning, through to 127, denoting hard right positioning. The default value of 64 denotes central positioning.

Reverb allows the user to alter the amount of reverb effect (if available) applied to the corresponding MIDI channel. A full description of reverberation effects is outside the scope of this thesis, but a very good article appears in [Roads, 1996]. Reverb is now offered as a standard effect type on most synthesisers. Acceptable values for this parameter range from 0 to 127.

Similarly, the *Chorus* parameter allows the user to alter the amount of chorusing (see [Roads, 1996]) applied to the corresponding MIDI channel.

Chord frequency is a parameter that determines how often chord events are generated. It is given as a percentage. Thus, the default setting of 50 indicates that a chord event may be expected half of the time. It is important to note, however, that this parameter will only have an effect provided that the *Mono* button is unchecked.

Chord depth, an integer between 1 and 4, is the number of notes that combine to form a typical chord event.

Chord variance, an integer between 0 and 3, is the number of notes by which the chord depth can vary.

Thus, a chord depth of 2 with a chord variance of 1 would mean that 1, 2 or 3 notes are all possible values for the number of simultaneous notes in a chord, whilst the generation of a 4-note chord is impossibility.

The *Rest frequency* parameter is very similar in operation to the *Chord frequency* parameter. It is an integer percentage that determines how often rests are generated. The remainder of the time, notes begin immediately following the termination of those that immediately precede them.

The general algorithm for controlling the instrumentation of the composition is not particularly difficult, as it involves only a check of the relevant Demon Cyclic Space cell. However, the algorithm for generating note events is quite complex and is best viewed in the form of a flowchart. This is presented in the next section.

5.3 The CAMUS 3D algorithm as a flowchart

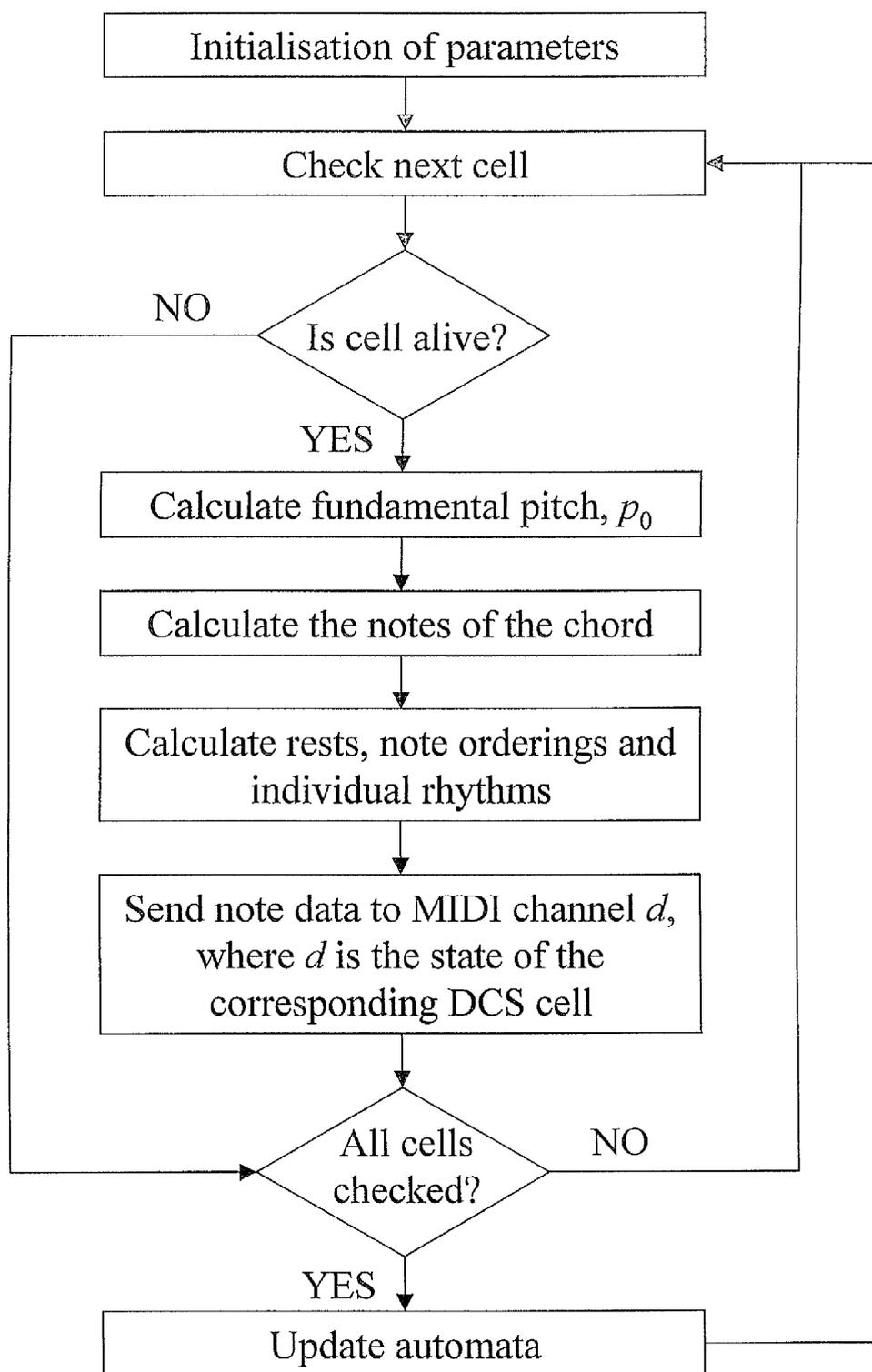


Figure 5.3.1 – The CAMUS 3D algorithm.

As can be seen by comparing Figures 5.3.1 and 3.1.6, the underlying algorithms for CAMUS and CAMUS 3D are outwardly very similar. As we shall see, however, the algorithms are quite different in detail.

Figure 5.3.2 expands further the steps labelled *Calculate fundamental pitch, p_0* and *Calculate the notes of the chord*. The algorithm makes use of a user-defined probability table, p .

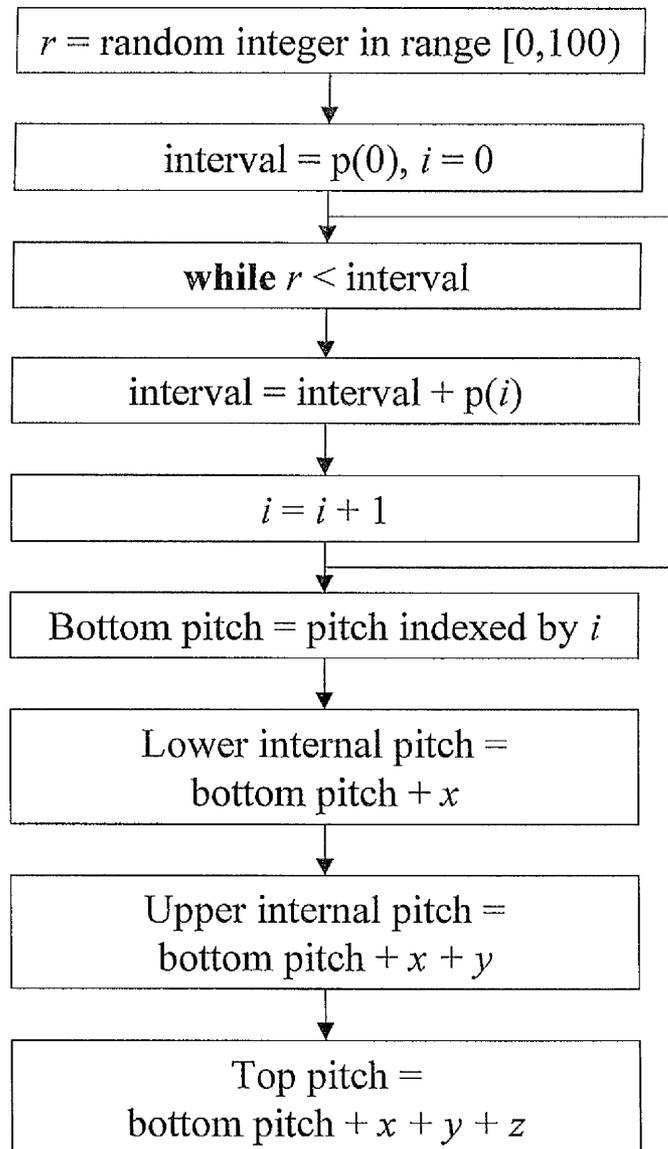


Figure 5.3.2 – Algorithm for calculating the notes of the four-note chord defined by cell (x, y, z) .

As before CAMUS 3D uses the one-to-one correspondence described in Appendix C to map the chromatic pitches from C-2 through to G8 onto the integers from 0 to 127.

The next step in the algorithm is the calculation of rests, note orderings and individual rhythms. Figure 5.3.3 illustrates the procedure for determining the insertion of a rest.

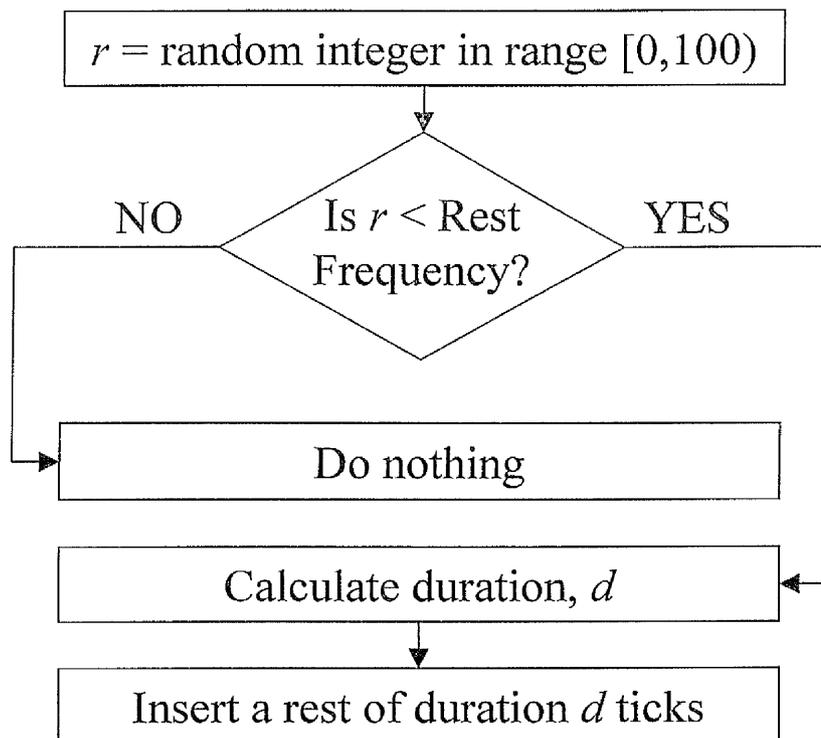


Figure 5.3.3 – Algorithm for determining the insertion of rests.

Figure 5.3.4 shows the algorithm used for determining the chord shape given the probability table p .

It should be apparent that the algorithm is almost identical to the *Calculate fundamental pitch, p_0* section of the algorithm described in Figure 5.3.2, the reason being, of course, that both algorithms make use of the cumulative distribution method.

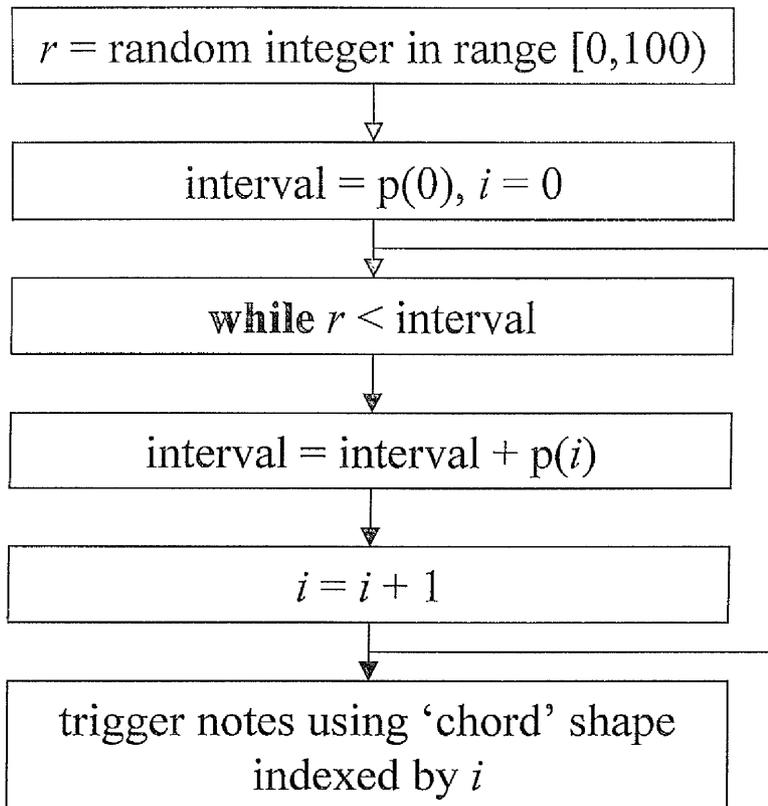


Figure 5.3.4 – Algorithm for determining chord shape.

The next stage of this section of the algorithm is concerned with determining whether or not a chord (i.e. a true simultaneous multi-note event) is to be played. As was mentioned earlier, this depends on the state of a user-specified binary control device and a number of probabilistic parameters. The step used to calculate the number of notes in each chord is illustrated in Figure 5.3.5a below.

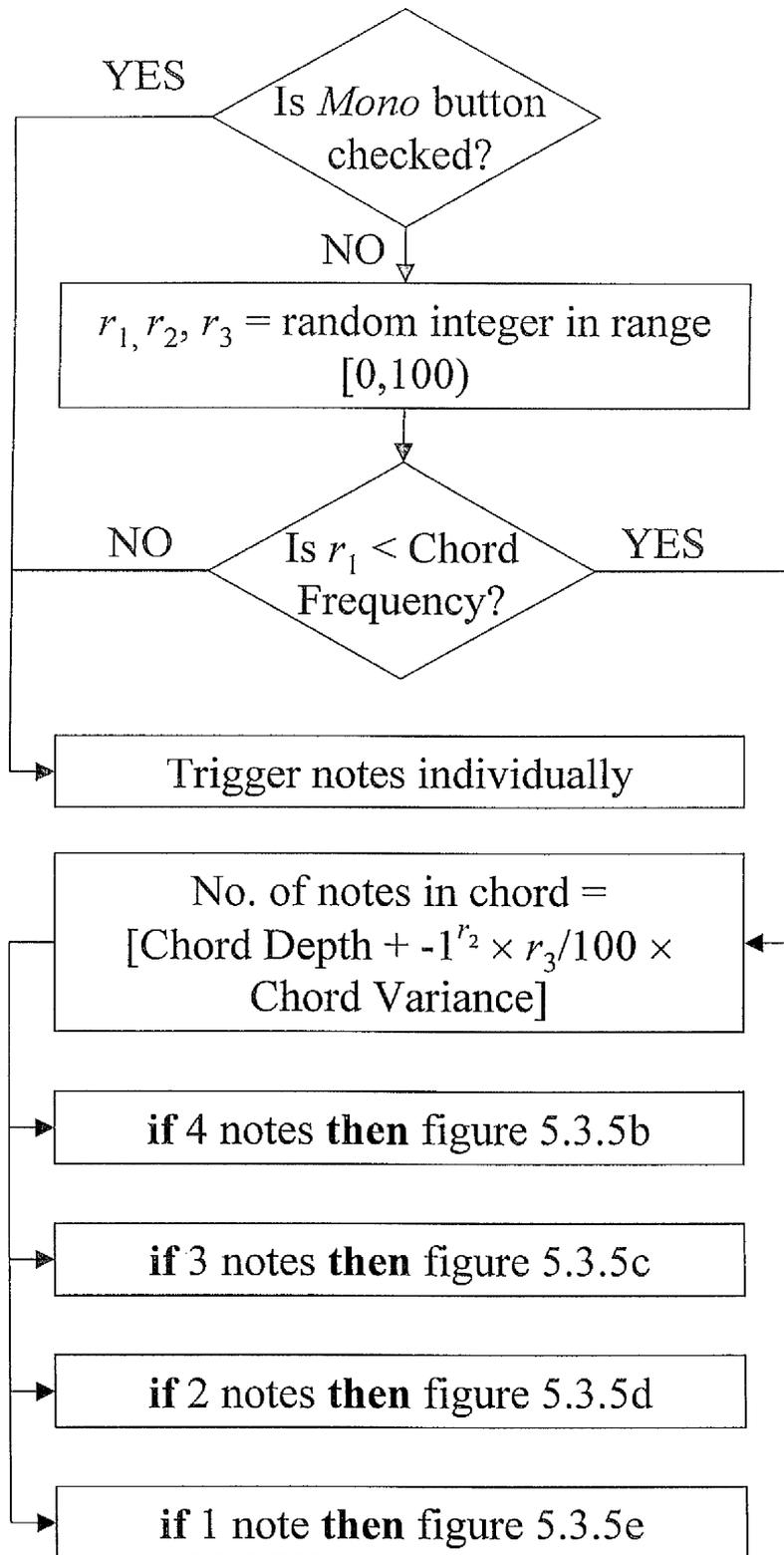


Figure 5.3.5a – Decision algorithm for generating chords.

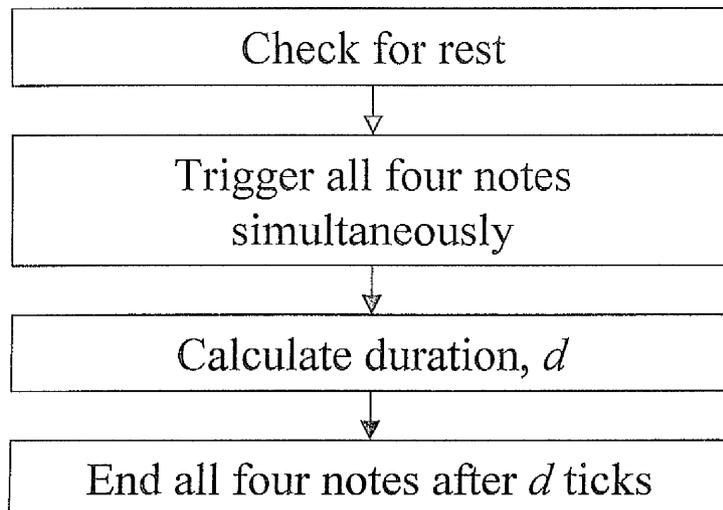


Figure 5.3.5b – Algorithm for performing four-note chords.

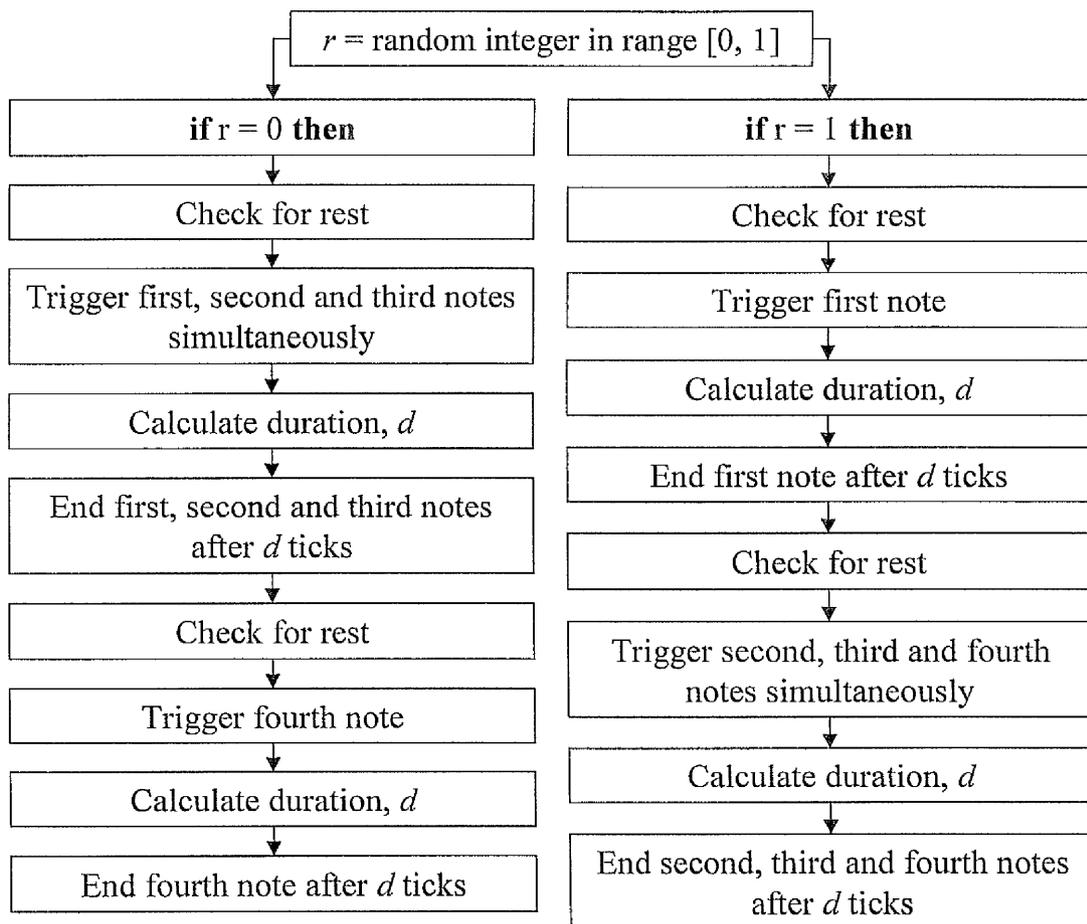


Figure 5.3.5c – Algorithm for performing three-note chords.

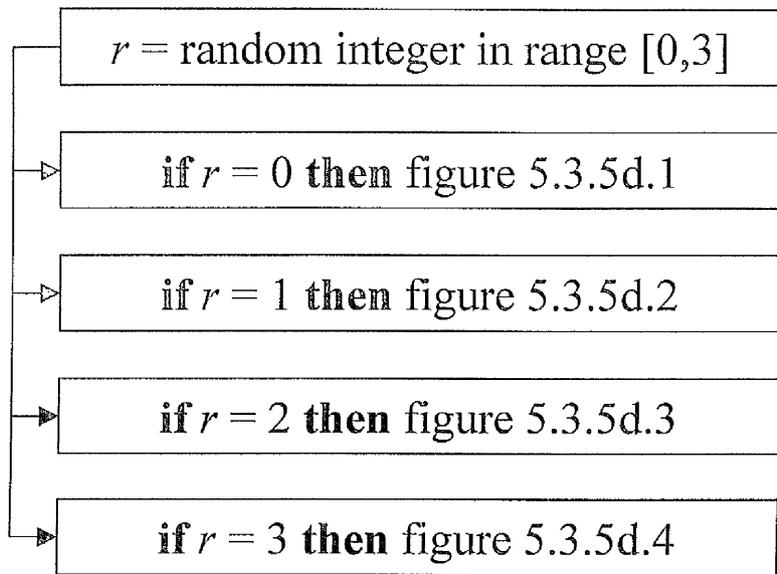


Figure 5.3.5d – Algorithm for performing two-note chords.

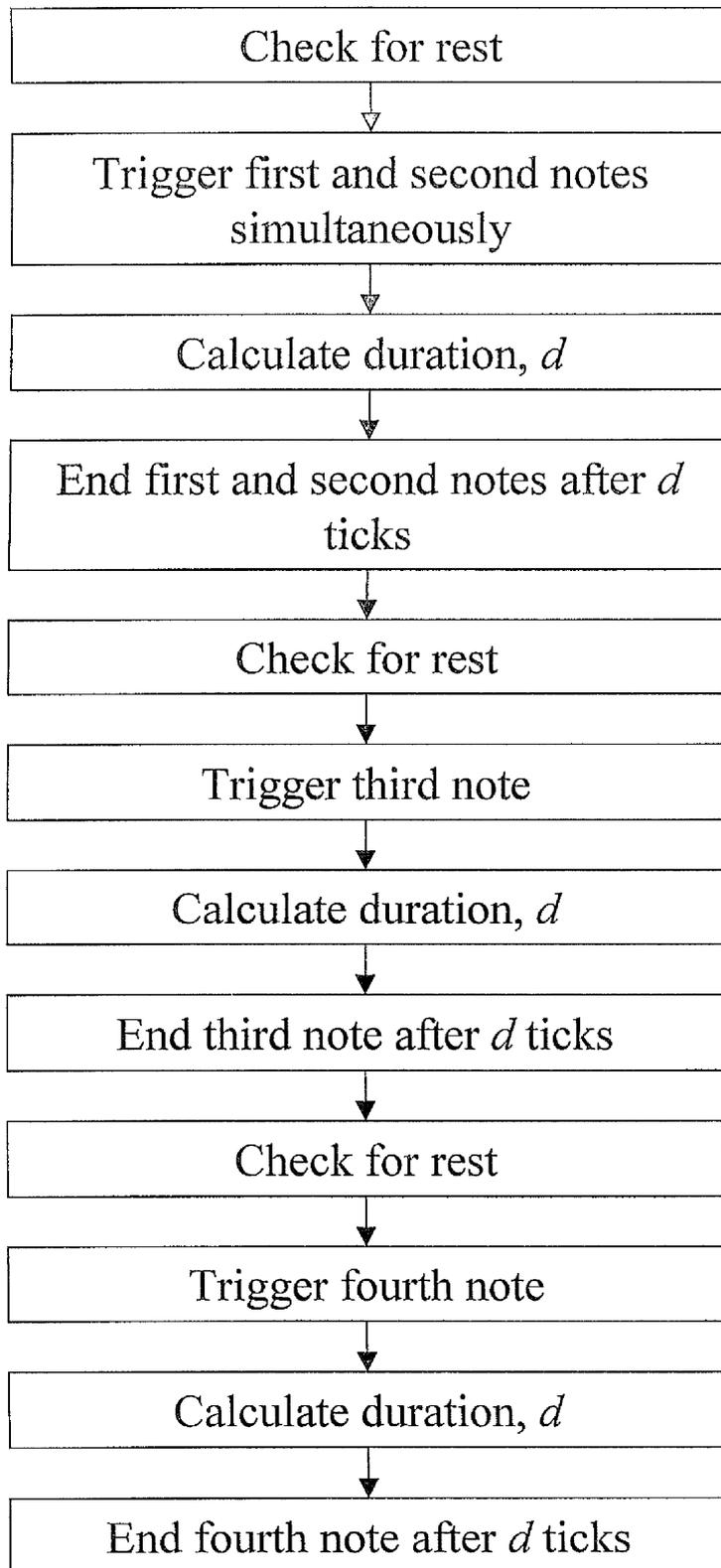


Figure 5.3.5d.1 – Algorithm for performing two-note chords.

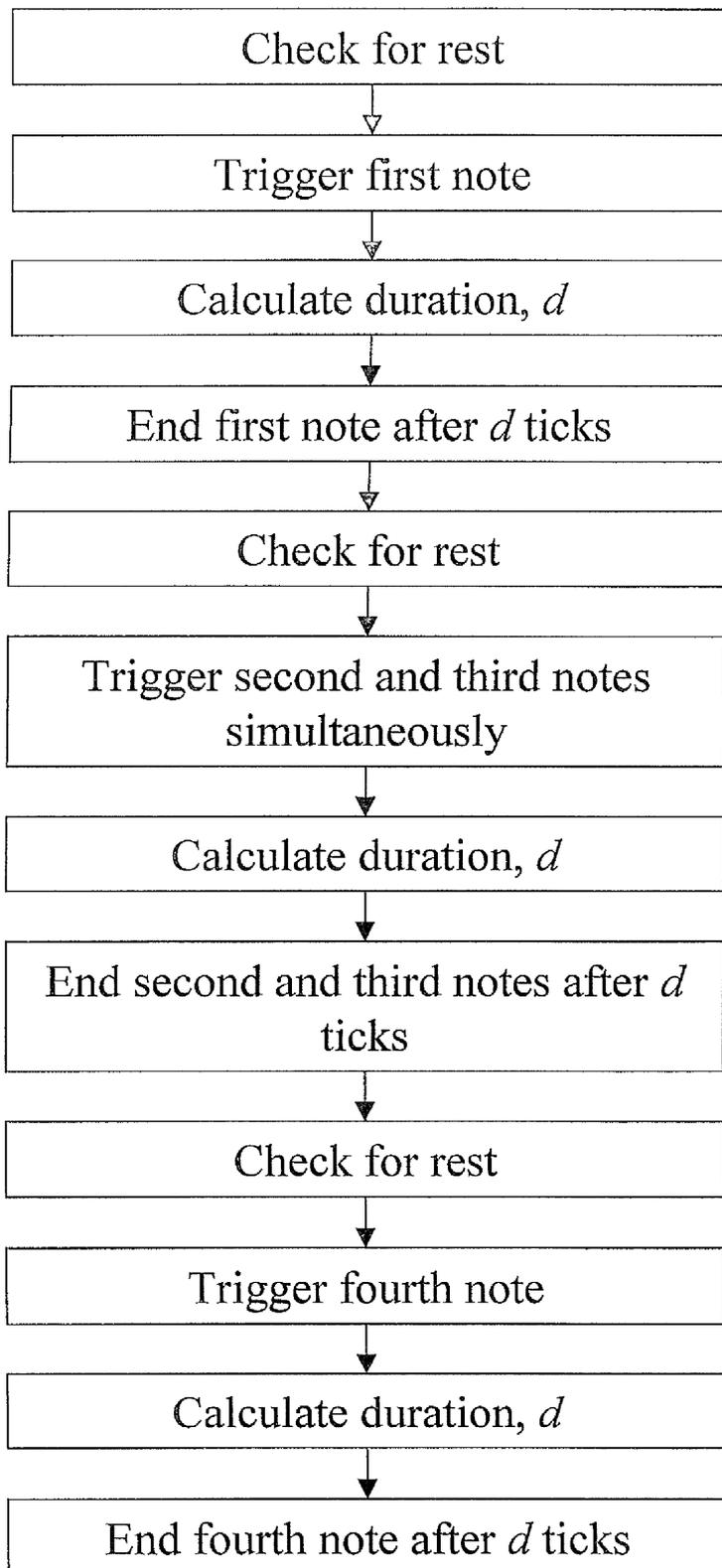


Figure 5.3.5d.2 – Algorithm for performing two-note chords.

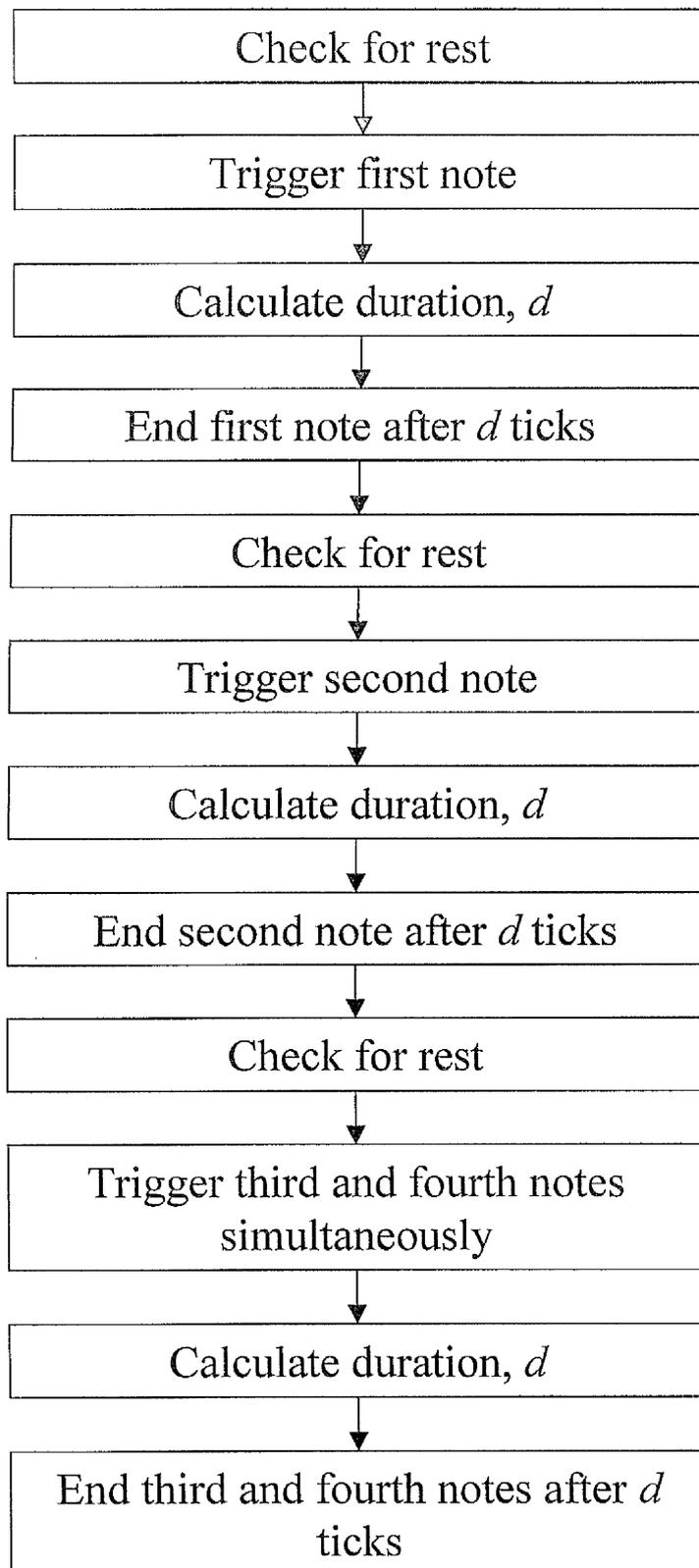


Figure 5.3.5d.3 – Algorithm for performing two-note chords.

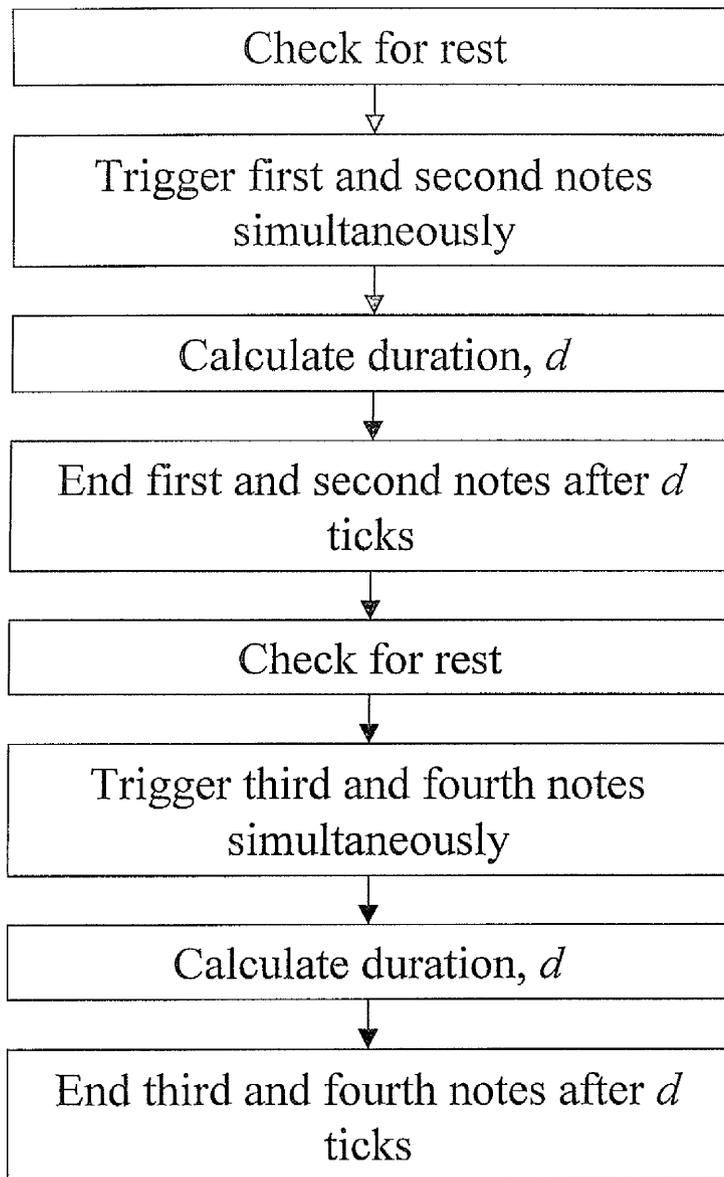


Figure 5.3.5d.4 – Algorithm for performing two-note chords.

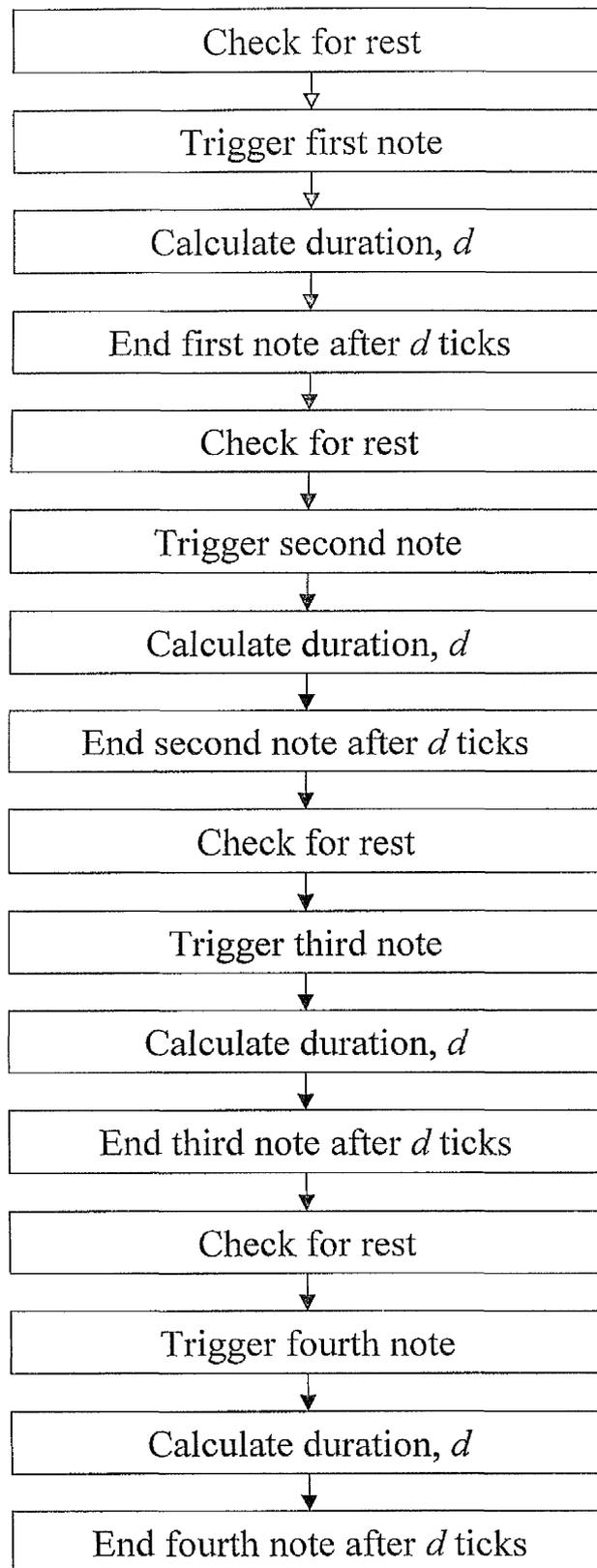


Figure 5.3.5e – Algorithm for performing one-note chords.

Finally, the application of the 1st order Markov Chain is used to calculate each note value.

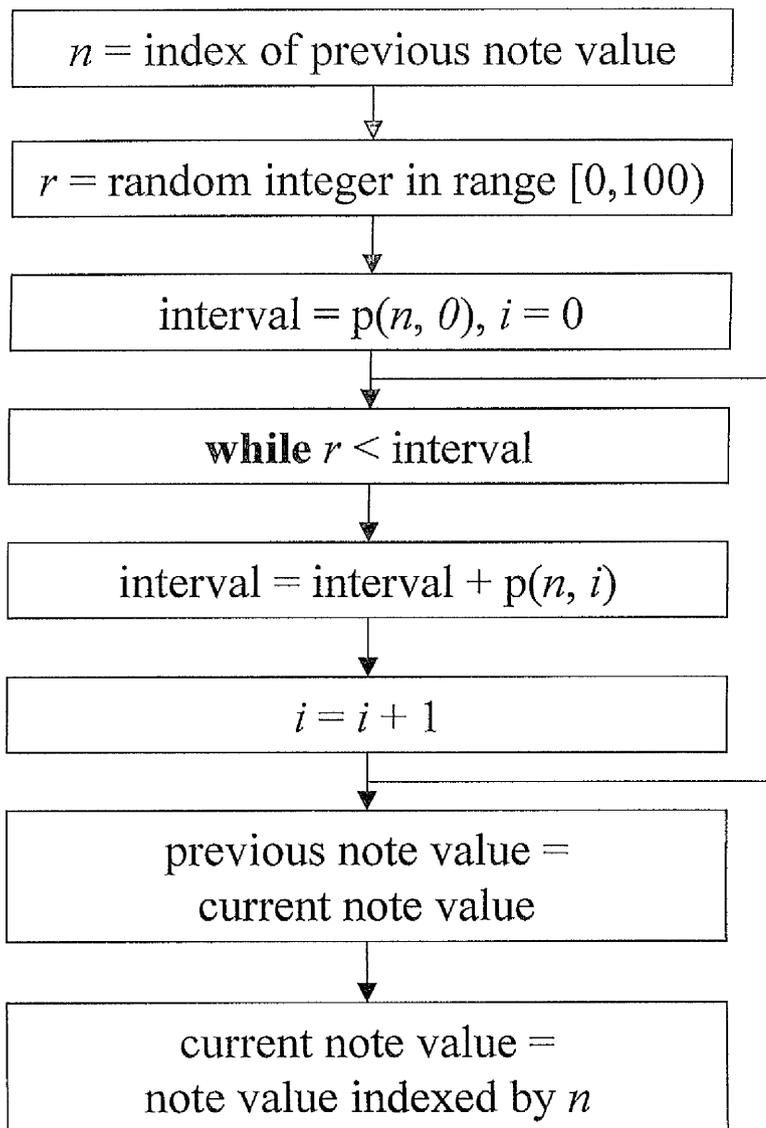


Figure 5.3.6 – Algorithm for determining note values using a first order Markov Chain with transition matrix $p(x, y)$.

5.4 Composition Example

As with the 2-dimensional system, it is useful to illustrate the workings of the system by providing a worked composition example to show the effects that parametric changes have on the resulting musical output.

As mentioned earlier, CAMUS 3D essentially operates from within two main windows. The first of these is the *CAMUS 3D toolbar*, shown in Figure 5.4.1 below.

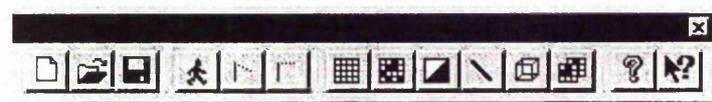


Figure 5.4.1 – The CAMUS 3D Toolbar.

This is the main ‘control centre’ for CAMUS 3D. It allows the user to load and save files, stop and start the composition process, and change the composition parameters. All of the functions are accessed from either the menu options or by clicking on the buttons. Reading from left to right, the icons are described in Table 5.4.1 below.

Icon	Effect
<i>New</i>	Creates a new document.
<i>Open</i>	Opens an existing document.
<i>Save</i>	Saves an opened document using the same file name.
<i>Step Through</i>	Updates the automata once and stops. Musical data is generated and performed.
<i>Go</i>	Updates the automata continuously until cancelled by the user. Musical data are generated and performed.
<i>Halt</i>	Halts the composition process and silences any music that is currently playing.
<i>Clear</i>	Clears the Game of Life of all live cells.
<i>Randomise</i>	Randomises the cells in the Game of Life.
<i>Invert Cell</i>	Allows the user to invert the state of any cell in the Game of Life.
<i>Line</i>	Inverts all cells in the Game of Life that lie on a line between two user-specified cells.
<i>Cube</i>	Inverts all cells in the Game of Life that lie within a user-specified cube.
<i>Extend</i>	Creates an extension of all Game of Life objects in a user-specified plane into the next adjacent plane.
<i>Help Topics</i>	Offers the user an index to topics on which help is available.
<i>About</i>	Displays the version number of this application.

Table 5.4.1 – Description and functionality of the CAMUS 3D Toolbar icons.

The second main window is the 3D Game of Life window, which was illustrated in Figure 5.2.6. As was mentioned earlier, this may be thought of as CAMUS 3D's

musical score. Recall that the 3D Game of Life window displays information about the states of cells in the Game of Life and Demon Cyclic Space. The live cells are those that give rise to musical events.

In order to begin a new composition, it is first necessary to create a blank score with which to work. This may be done by clicking on the New icon on the toolbar or by selecting *New* from CAMUS 3D's File Menu. Alternatively, a previously saved composition may be opened for further work by clicking on the Open icon on the toolbar or by selecting *Open* from CAMUS 3D's File Menu.

Once a file has been created or opened, the user should specify the settings of the two control automata, which will determine how the composition unfolds.

The Game of Life rules may be edited by selecting *Alter Game of Life Rules* from the Automaton/3D Game of Life menu. This action displays the *Alter Game of Life Rules* dialog box, which is illustrated in Figure 5.2.10, and whose contents are described in Section 5.2.4.

The default settings of the *Alter Game of Life Rules* dialog box are given as $R = (5, 7, 6, 6)$. That is, in the *Live Cell* group, *Minimum neighbours for life* is set to 5, *Maximum neighbours for life* is set to 7, and in the *Dead Cell* group, both *Minimum neighbours for life* and *Maximum neighbours for life* are set to 6. This is precisely the rule set discussed in Section 5.1.5, which was shown to be the 3-dimensional rule set that most closely matched the behaviour of the more familiar 2-dimensional game. In addition, note that the *Check all neighbouring cells* is selected, so that the automaton behaves like a true 3-dimensional space.

The user may now choose to alter these rules, either by clicking on the required edit box and typing a new value, or by clicking on the value increment and decrement buttons at the right hand side of each edit box. Entries that are outside the permitted range of values and those that are invalid generate non-critical error messages and cause the system to prompt the user for a new value.

We now alter the automaton settings to the following:

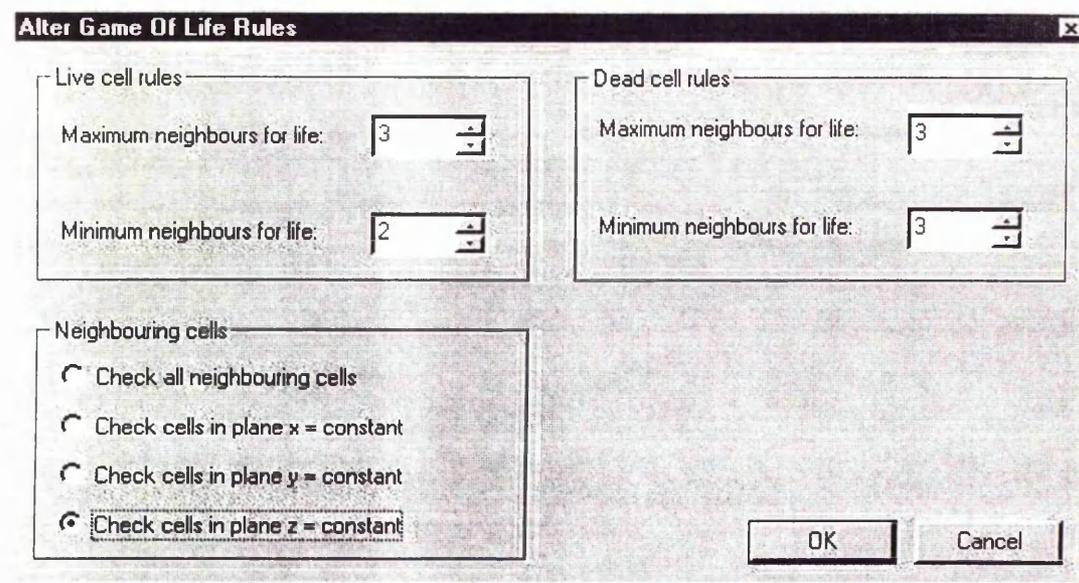


Figure 5.4.2 – Updated settings for the 3-dimensional Game of Life automaton.

By limiting the checking of cells to those neighbours that lie in planes parallel to the xy -plane, we are forcing the 3-dimensional automaton to behave like a number of 3-dimensional automata running in parallel. Further, changing the rule set to $R = (2, 3, 3, 3)$, has the effect of forcing the 3-dimensional automaton to behave as a number of Conway games evolving in parallel.

The evolution rules of the Demon Cyclic Space may be edited by selecting *Alter Demon Cyclic Space Rules* from the Automaton/3D Demon Cyclic Space menu. This action displays the *Alter Demon Cyclic Space Rules* dialog box, which is illustrated in Figure 5.2.11, and whose contents are described in Section 5.2.4.

The default settings of the Demon Cyclic Space allow each cell to exist in one of 16 states, each of which is represented by a coloured cell, and which is dominated by those neighbouring cells whose state is 1 higher.

Recall that the number of cells of the Demon Cyclic Space corresponds to the number of instruments that will perform the composition. Thus, if we set the parameters as in Figure 5.4.3 below, the resulting composition will be performed on just one instrument.

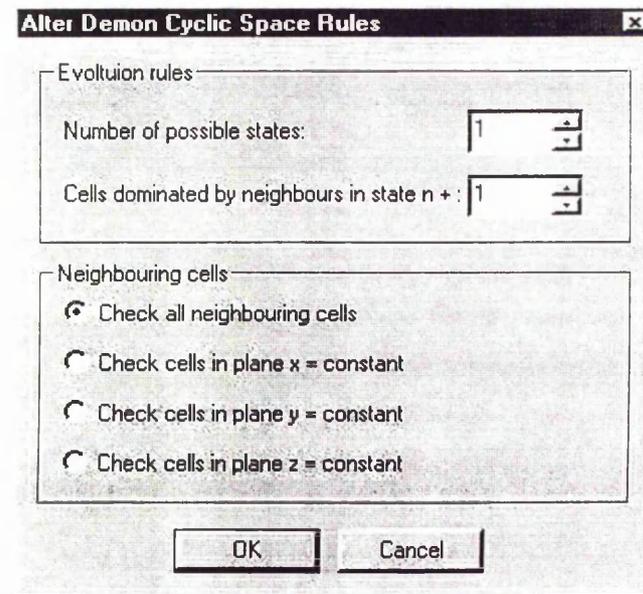


Figure 5.4.3 – Updated settings for the 3-dimensional Demon Cyclic Space automaton.

Note now that since each cell in the automaton can exist in only one state, the other parameters will have no effect on the cellular evolution.

The next stage in the process is the initialisation of live cells in the Game of Life. The functions involved in this process are described in some detail in Section 5.2.2. A typical initialised configuration is:

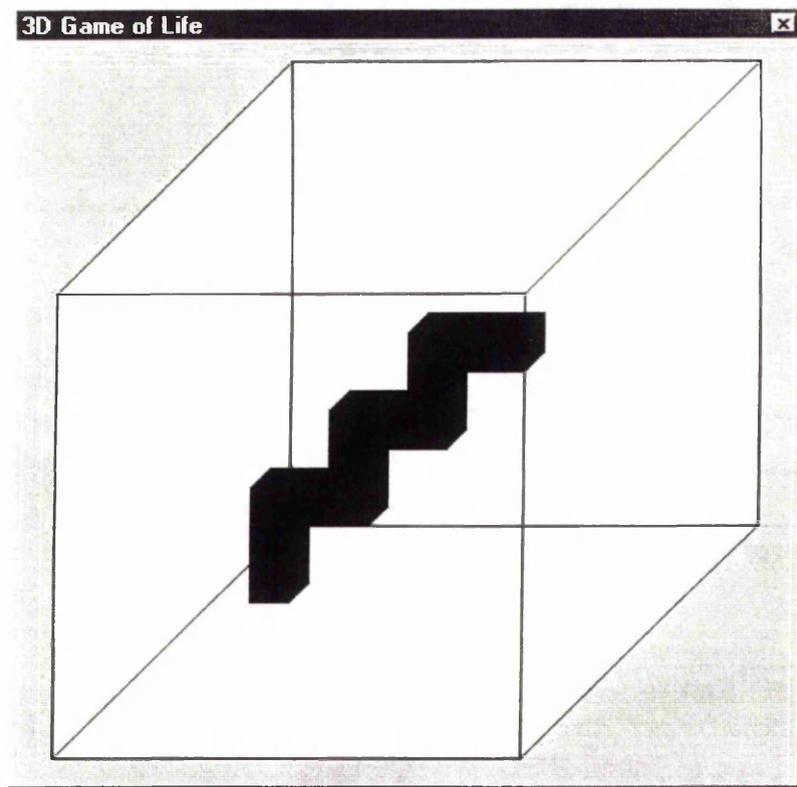


Figure 5.4.4a – A typical configuration of the 3-dimensional Game of Life.

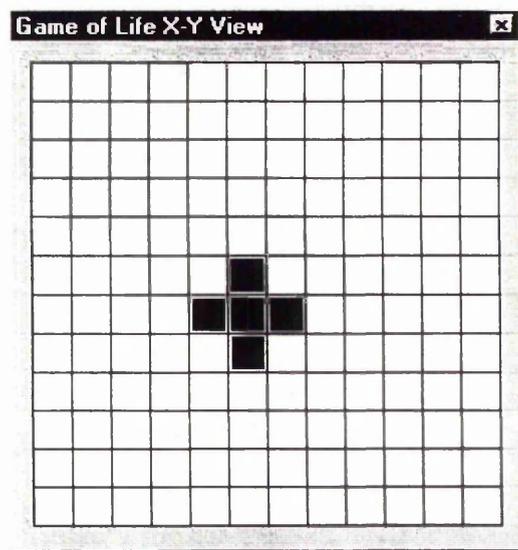


Figure 5.4.4b – A typical configuration of the 3-dimensional Game of Life viewed as an xy-plane projection.

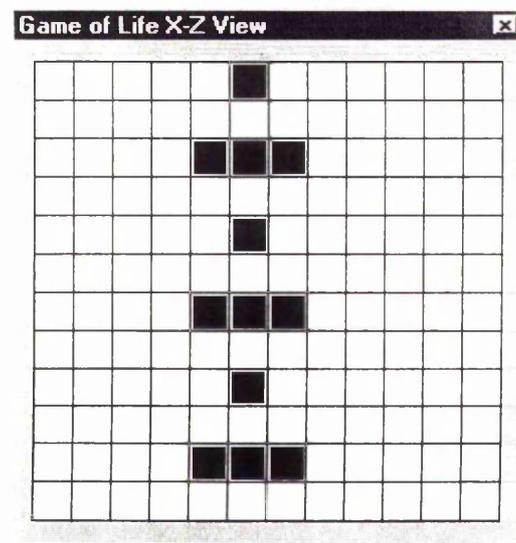


Figure 5.4.4c – A typical configuration of the 3-dimensional Game of Life viewed as an xz-plane projection.

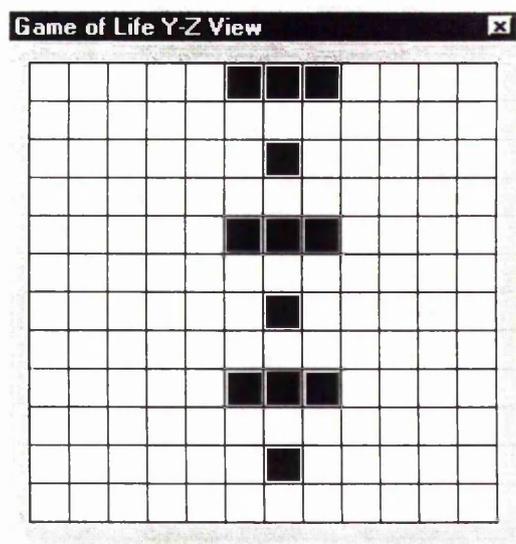


Figure 5.4.4d – A typical configuration of the 3-dimensional Game of Life viewed as a yz-plane projection.

Next, the instrumentation of the composition should be decided upon. This may be achieved using the *MIDI Instrumentation Setup* dialog box (see Figure 5.2.15), which is opened by selecting *Instrumentation* from the Control menu.

The parameters in this dialog box were described in Section 5.2.7 and are not re-examined here.

Since the number of states in the Demon Cyclic Space, that is the number of instruments that will perform the composition, was set to 1, we need only alter the parameter settings for Channel 1. Suppose we alter the values to those of Figure 5.4.5 below.

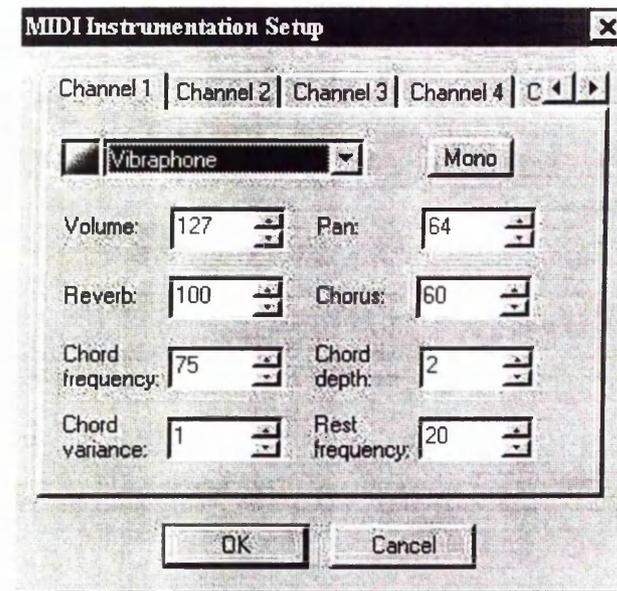


Figure 5.4.5 – Updated settings for the MIDI Instrumentation Setup dialog.

The Vibraphone is the instrument that has been selected to perform the piece as it is generated. However, since CAMUS 3D is configured to the General MIDI system (see Appendix C for further details) of instrument layout, the desired effect will only be achieved if the MIDI data are sent to an instrument that is also configured to this system. Most computer soundcards fulfil this criterion, meaning that the problem really only arises if the host PC is connected to an external sound synthesiser.

As with the original CAMUS system, the user is still free to alter the settings for any of the other instruments, and indeed, patch change data will be sent for each, but this will have no effect on the resulting composition, since only the first MIDI channel is configured for playback.

The Mono button in Figure 5.4.5 is unchecked, allowing for the possibility of both homophonic and monophonic note data. The frequency and depth of the chord events are determined by the settings of certain other parameters.

Chord Frequency is set at 75, meaning that on the average, chord events will be generated in three out of every four cases. *Chord Depth* is set at 2 and *Chord Variance* is set at 1, meaning that chords of 1, 2 or 3 notes are possible. *Rest Frequency* is set to 20, which will result in a rest between notes approximately once in every five cases.

Finally, the four parameters that determine MIDI effects, *Volume*, *Pan*, *Reverb* and *Chorus* are set to 127, 64, 100 and 60 respectively. This will result in the instrument on MIDI Channel 1 being played back at full volume in the centre of the sonic panorama with a reverberation setting of 100 and a chorus setting of 60.

The next step in the composition process is the initialisation of the parameters that determine the actual notes that CAMUS 3D performs. To do this, the user must open the Pitches dialog box, as illustrated in Figure 5.2.14. This may be achieved by selecting *Pitches* from the Control menu.

CAMUS 3D selects pitches for the chords using the probabilities stored in the form of a probability table. Acceptable values range from 0, signifying that the associated note will never occur, to 100, meaning that the note will occur with absolute certainty.

Suppose the user had set the values to the following:

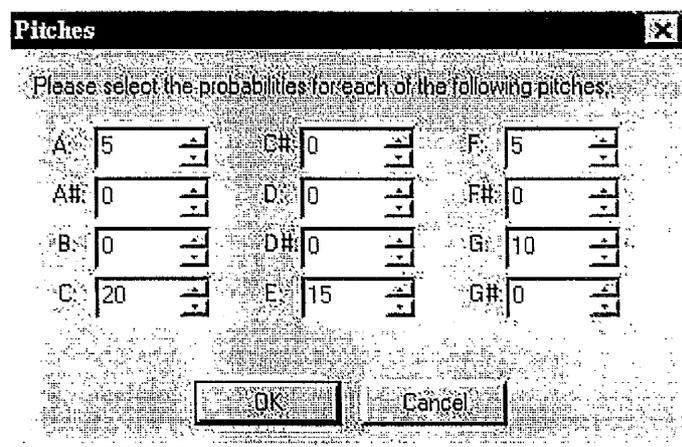


Figure 5.4.6a – Updated settings for Pitches dialog.

Upon re-opening the Pitches dialog, the probabilities read:

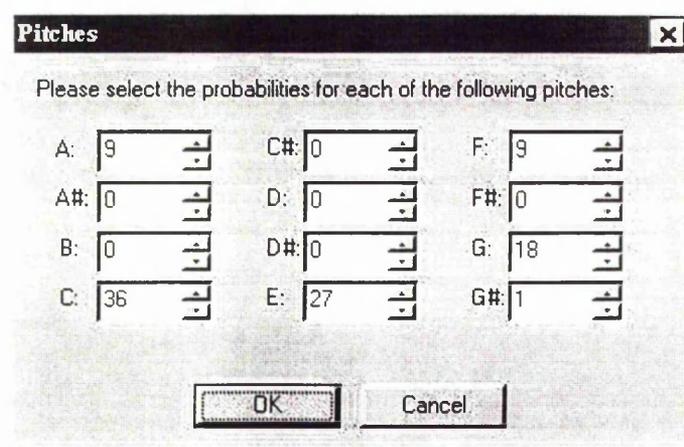


Figure 5.4.6b – Normalised settings for Pitches dialog.

The reason for this is that CAMUS 3D uses normalised probability tables in all of its procedures. Each of the dialogs that deal with probabilities are automatically normalised whenever they are changed.

The reader will notice, however, a slight anomaly. In the box labelled G# in Figure 5.4.6a the entry is 0, whereas in Figure 5.4.6b the entry is 1. This is a side-effect of the normalisation procedure used and the quantisation errors accrued during the normalisation process. In practice, it makes very little difference to the musical output and may be easily overcome with a little careful planning during the data entry stage.

A similar process is involved when altering the probability data stored within the Note Orderings dialog box, illustrated in Figure 5.2.2. The dialog box may be opened by selecting *Note Orderings* from the Control menu.

Recall that the temporal structure of each of the chords is represented by a 4×4 grid. The bottom row of boxes represents the lowest pitch of the chord, the second bottom row represents the second lowest note, the second top row represents the second highest pitch and the top row represents the highest pitch. The order of the shaded cells from left-to-right determines the order in which each of the notes is played.

Typically the values in this dialog box might resemble following:

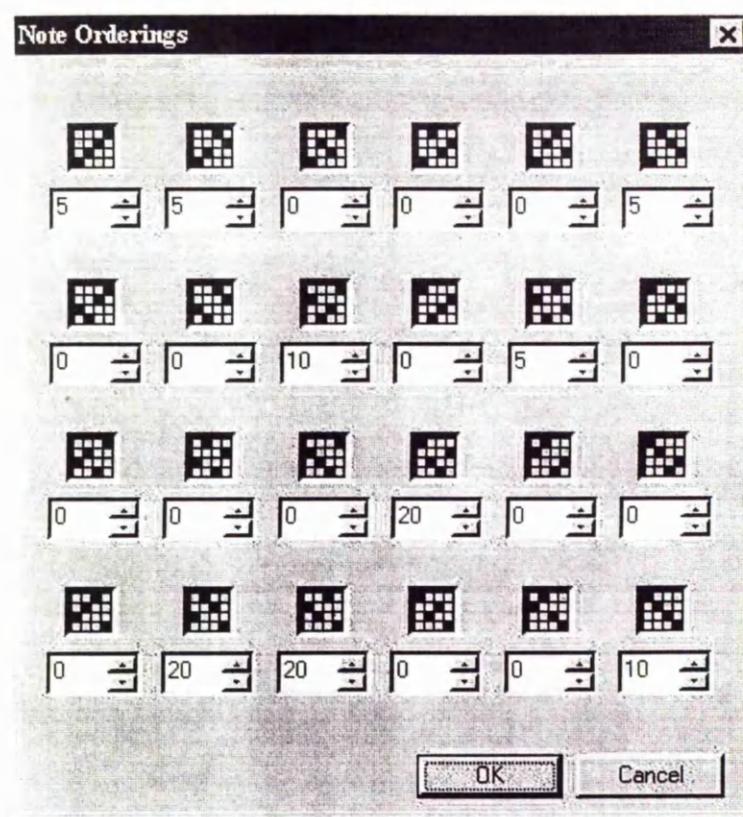


Figure 5.4.7 – Typical settings for the Note Orderings dialog.

The final stage in the composition setup process is the initialisation of the probability values that determine the rhythm of the composition. This is done via the Note Lengths dialog box (see Figure 5.2.12), which is opened by selecting *Note Lengths* from the Control menu.

The Note Lengths dialog box stores the probability data as a first-order Markov Chain, and offers two methods of viewing the probability values. The Standard window displays the data as a number of coloured boxes that smoothly change in colour from pure red (probability value 0) to pure green (probability value 100). The colour of the boxes can be changed by clicking on the arrow controls on the right-hand side of each box.

This window is probably best used to view the probability data to quickly see which note lengths are likely to arise. To actually set the probability values, the Advanced window allows direct access to the numerical values.

The user can switch between these windows by clicking on the tabs at the top of the screen.

Typically, the probability values would be set to something resembling the following:

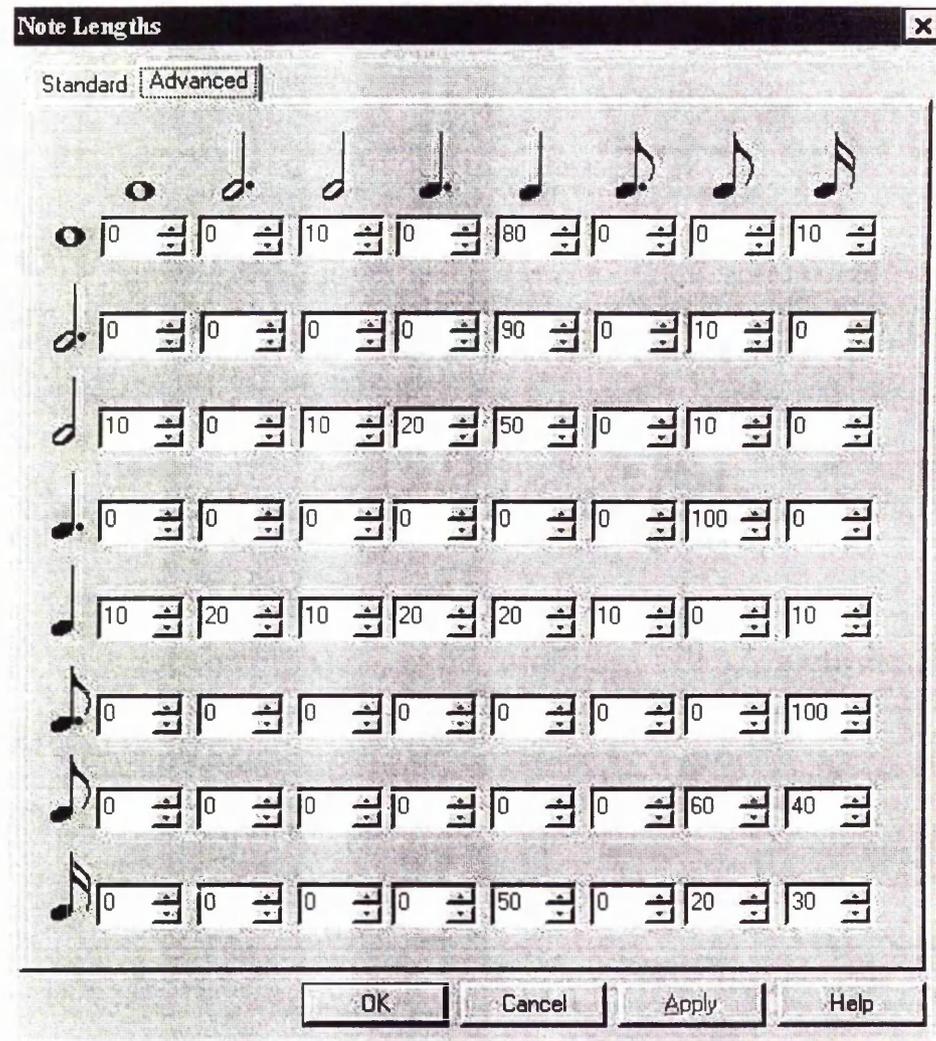


Figure 5.4.8a – Typical response for the Advanced Note Lengths dialog.

with the corresponding Standard view resembling:

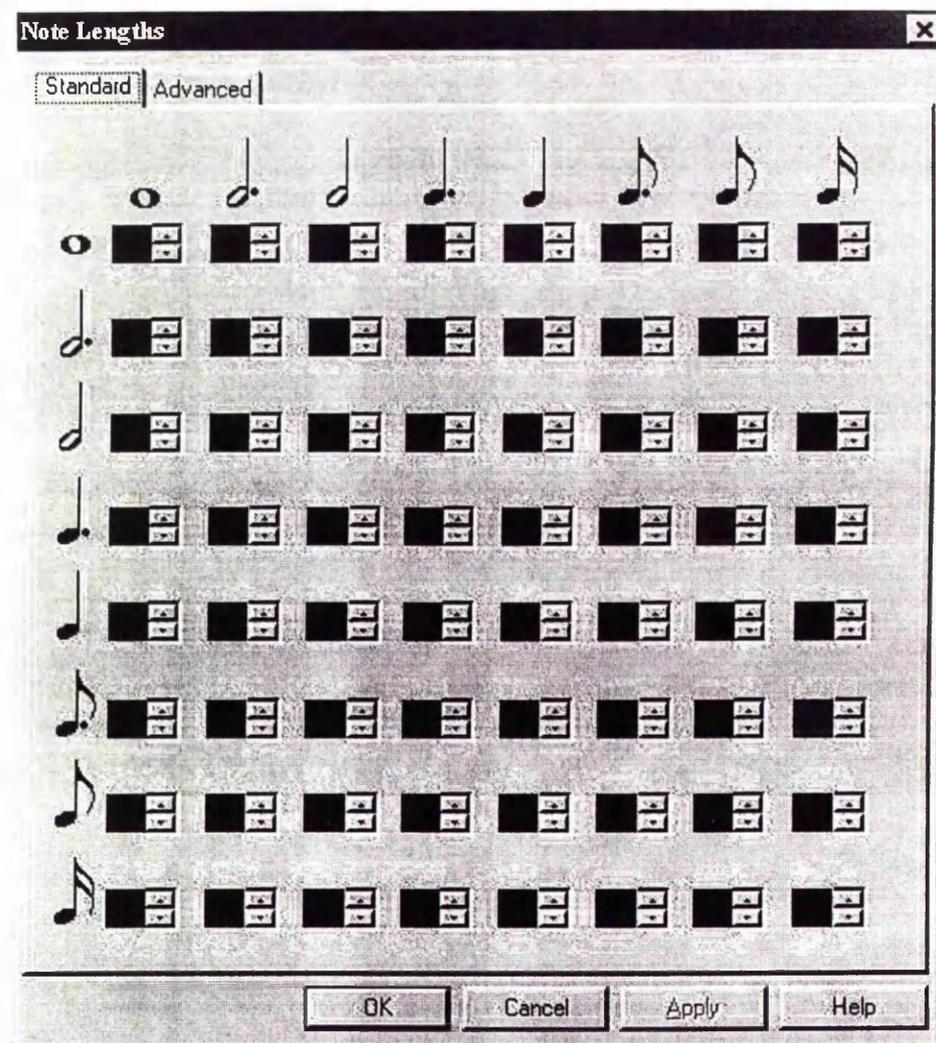


Figure 5.4.8b – Typical response for the Standard Note Lengths dialog.

The above settings correspond to the following labelled directed graph.

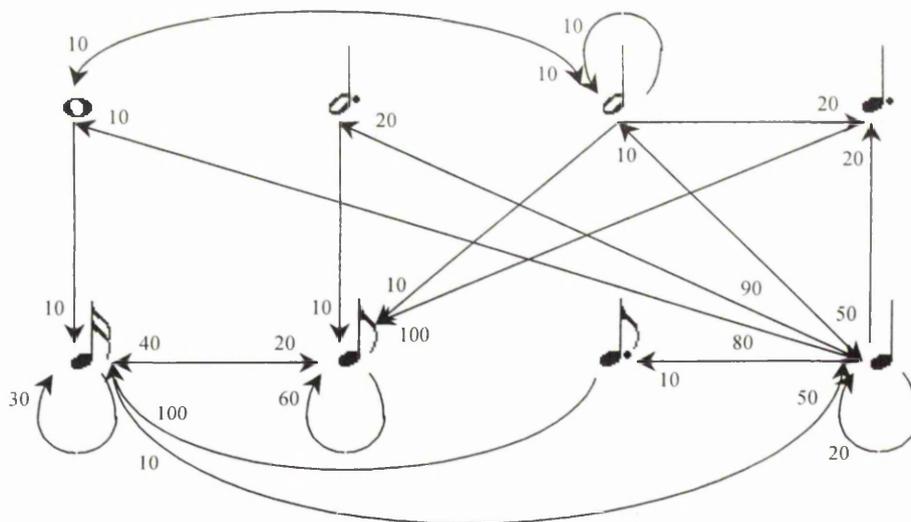


Figure 5.4.9 – Labelled directed graph corresponding to the probability settings of Figure 5.4.8.

At this point, it is beneficial for the user to save the CAMUS 3D project, so that it can be re-loaded at a later date. This is done by selecting *Save As* from the File menu, or by clicking the *Save* button on the CAMUS 3D toolbar. This action saves all of the composition-specific parameters and the states of the control automata. When this file is re-loaded and played back at a later date, the music that is generated is likely to differ slightly each time due to the random functions which are a part of the CAMUS 3D compositional process. If the user wishes to preserve a specific performance, the *Export MIDI File* option should be selected from the File menu. It is important to note, however, that exporting a MIDI file at this stage in the composition process will be of little use, since no musical data have yet been generated. Exporting a MIDI file will only have an effect once the musical generation process is underway.

A useful feature that is offered by CAMUS 3D is the *Preview* option, which may be selected from the Compose menu. This function simply runs through the evolution of the automata without generating any musical data, thus allowing the user to evaluate the automata rule configuration before generating the final composition. If at this stage the user decides that the automata rules do not produce the cellular evolution desired then he or she is free to revert to the saved project and alter the rules.

The music generation may now be started by selecting *Go* from the Compose menu, or by clicking the *Go* button on the CAMUS 3D toolbar. The music will become

audible after a few seconds. The generation of music continues until the user halts the composition process.

As an alternative method of music generation, the user may select *Step Through* from the Compose menu. This will step through one timestep of the composition process and stop. It is of most use to the composer who wishes to alter either the composition parameters or the position of live cells in the Game of Life as the composition progresses.

The Step Through function has the additional benefit of allowing the music to keep pace with the evolution of the automata. CAMUS 3D generates music faster than real time. In particular, if the Game of Life is particularly densely populated, several hours' worth of music may be generated at each timestep. When using the Step Through function, the user can wait – although possibly for quite some time – until the music has stopped before updating the system. However, when obeying the Go command, the system will continue to generate music regardless of the current state of the music, meaning that after a few seconds of generation, the system buffer may contain more music than it would be physically possible to evaluate.

The above settings result in the following:

CAMUS

The image displays a musical score for a piece titled "CAMUS". The score is written in 4/4 time and consists of five systems of two staves each (treble and bass clef). The first system starts at measure 1 with a piano (*p*) dynamic. The second system starts at measure 5, featuring a forte (*f*) dynamic in the treble and piano (*p*) in the bass. The third system starts at measure 9, with pianissimo (*pp*) in the treble and fortissimo (*ff*) in the bass. The fourth system starts at measure 13, with mezzo-forte (*mf*) in the treble. The fifth system concludes the piece. The score includes various musical notations such as notes, rests, beams, and dynamic markings.

Figure 5.4.10 – Music generated using the CAMUS 3D algorithm.

The score of Figure 5.4.10 was created in Steinberg's *Cubase VST/24* directly from the MIDI-file output of CAMUS 3D using the settings described above. Dynamic instructions were added manually.

5.5 Comparisons between the 2 and 3-dimensional systems

We conclude this chapter by briefly highlighting the differences and similarities between the original CAMUS and the 3-dimensional system.

5.5.1 Peas in a pod?

At first glance there are several similarities between CAMUS and CAMUS 3D. For example, both are algorithmic composition systems that utilise cellular automata as the main driving force behind the composition process.

This impression is further enhanced by the ability of CAMUS 3D to behave in an almost identical manner to the original system by limiting the evolution of the 3-dimensional automata to two dimensions.

However, on further inspection, we see that there are sufficiently many significant differences between the two to consider CAMUS 3D as a new algorithm, albeit one which is clearly derived from the original.

Indeed, a number of differences between the two algorithms – such as moving from 2 to 3-dimensional control – are clearly developments of the earlier system, whilst others are improvements in the human-computer interface and thus are excluded from consideration. Also, as can be seen from Figures 3.1.6 and 5.3.1, both systems use the same underlying algorithms to generate music.

We highlight the main differences in the following sections and summarise in the Comparison Table, 5.5.1

5.5.2 Key differences between the systems

There are three major differences between the two composition systems, all of which are concerned with note detail, and all of which were implemented after practical experience with the original system.

After a number of users had experimented with the original system, almost all commented on how cumbersome and incomprehensible the system of loops and articulations, used to calculate the main note detail, were in use.

Clearly, the removal of this system necessitates the introduction of alternative control measures, and it is here that the main differences lie.

The first of these is the introduction of the stochastic note generation routine described in Section 5.2.1. This was introduced primarily because user reports suggested that the original system of setting each subsequent root note for the composition was extremely laborious and time-consuming. The new method of using probabilities to determine root notes is far quicker and easier, and allows for the creation of long-term musical trends without the need to specify individual notes. It also overcomes the original system's limit to the number of root notes which could be defined. Previously this was limited to 264 notes since each of the 22 articulations on offer contained a single 12-note sequence.

Secondly, there is the stochastic selection routine to determine chord shape. Again, this was introduced because practical experience suggested that users wanted more control over chord shape than was offered by the **AND** codes described in Section 3.1.6. In addition, it was discovered that the commonly-used Conway objects gave rise to the same **AND** codewords time and time again, which can lead to listener boredom. The stochastic selection routine allows the user to include or reject chord structures as well as expressing a weighted preference for certain shapes.

Finally, and perhaps most importantly, is the Markov routine for pitch calculation. This was introduced because the previous method of rhythm generation was based on a random number generator, which gave rise to rhythmic structures that were irregular and difficult to appreciate. Whilst initialising the Markov chain with a uniform probability distribution does allow for irregular rhythms, there is at least some structure imposed because of the note duration quantisation that is applied. In addition, as was mentioned in Section 5.2.6, it is possible to harness the properties of Markov chains to produce structured rhythms in which the ear can detect regularity.

There are also a number of smaller algorithmic differences in CAMUS 3D. For example, fewer control parameters are 'hard-wired' into the system, which means that the user has more control over the musical output.

Comparison Table

	CAMUS	CAMUS 3D
Driving mechanism	2D cellular automaton	3D cellular automaton
Mapping type	2D von-Neumann	3D von-Neumann
Decision probabilities	Fixed	User-selectable
Max. chord depth	3-note	4-note, with control over chord depth
Rests	Random	Automated, user-controlled
Rhythm generation	Random	1 st -order Markov
Pitch generation	Manual note lists	Automated probability tables
Chord shape	AND codewords	Automated probability tables
Musical parts	Default polyphonic	Selectable monophonic / homophonic ³⁰
Timbrality	Mono	Mono
Output	Standard MIDI files	Standard MIDI files
Controllable MIDI parameters	MIDI instrumentation	MIDI instrumentation; channel level; spatial positioning; MIDI chorus and reverberation levels

Table 5.5.1 – Table offering comparisons between the CAMUS and CAMUS 3D algorithms.

³⁰ In monophonic mode, music is generated to play one note at a time. In homophonic mode, music is generated to play block chords.

5.5.3 Differences in interface

From the beginning CAMUS 3D was designed with the end user in mind, and great care has been taken to ensure that the system is as intuitive as possible. For example, in the original system, many of the devices, such as the rhythm generator, bore little resemblance to the musical properties they controlled. In CAMUS 3D this has been overcome to a great extent by using, for example, traditional score notation in the graphical interface. The graphics for chord shape also bear a close resemblance to the chord charts used by guitarists.

The extension from 2 to 3-dimensional control systems also required considerable changes in interface. These are discussed fully elsewhere and are not expanded upon here.

5.5.4 Differences in working techniques

In Section 3.3 we examined the 'reverse engineering' method of composition and showed how it could be used to improve the yield of the original system.

Unfortunately, such a working technique is not currently practical with the 3-dimensional system. The main problem is that it is much more difficult to trace the evolution of cells using the isometric view of the 3-dimensional Game of Life. It is occasionally possible to create a list of cells using a combination of the isometric view and the three available embeddings, but such cases only occur for fairly simple cell configurations.

This could be overcome by introducing a separate display window to trace cells as they are played, however, as we shall see in the next chapter the current development of CAMUS 3D takes us towards parallel cell checking. Thus, cell order becomes less important, since we consider each timestep as a musical event, rather than each individual cell.

6. Work in progress

6.0 Introduction

In this, our penultimate chapter, we present features and developments of CAMUS 3D, which for reasons of time have not yet been integrated fully with the system.

As has been noted in previous chapters, there is a gradual progression of the system towards parallel cell checking. We begin this chapter by introducing the principal development of the system – an automaton that allows the system to make informed choices between groups of triads depending on the user's own harmonic preference.

We then discuss how this can be put to use as part of a parallel cell-checking routine and conclude by speculating on how further developments might be integrated into the system.

6.1 An automatic chord classifier

We now present the design of an automaton which takes as input two chords and returns as output an index that corresponds to the dissonance of the resulting sound.

6.1.1 The dissonance function

Before we begin to design the automaton, which for our purposes may be regarded as a black box function approximator, we must first form a clear notion of the function we wish to calculate. Since the ultimate aim of the automaton is to return an index of dissonance between two or more chords played together, we call this the *dissonance function*.

The simplest case occurs when we consider chords of 1 note depth, that is, single notes. When two such chords are played together we obtain a single 2-note chord, or simple interval.

The dissonance of such an interval will depend principally on two things.

Firstly, and perhaps most importantly, the type of interval formed affects the dissonance of the interval. It is a simple matter to consult a text on music theory in order to obtain a discussion on the relative consonance and dissonance of the musical intervals (see, for example, [Lloyd & Boyle, 1963]). However, such discussions are generally based on the harmonic series, and do not take into account individual tastes or preferences.

For example, it is quite possible that a musician will have a quite different harmonic preference to a non-musician. Similarly, a lover of freeform jazz is likely to have a different harmonic preference to a proponent of classical music. We discuss our approach to this problem in Section 6.1.2.

Secondly, the amount by which the interval is 'stretched' affects the resulting sound. Intervals that are stretched over a number of octaves generally sound thinner and less pleasant than those whose component notes lie within the same octave. To illustrate, consider, for example, the shrill sound formed by a low bass voice and a high soprano and compare it with the much warmer sound of an alto and a tenor singing in close harmony.

The aim of the dissonance function is to take into account these two factors and return an index of dissonance, d , which reflects how pleasant or unpleasant the two single-note chords sound when played simultaneously.

The index of dissonance is given as a non-negative real number, with lower values of d representing consonant sounds and higher values representing dissonant sounds. In effect, it may be thought of as a measure of the harmonic distance between two single pitches.

The general form of the dissonance function is then:

$$d(c_1, c_2) = M_1 O(|c_2 - c_1| \pmod{12}) + M_2 \left[\frac{|c_2 - c_1|}{12} \right]$$

where c_1, c_2 are single-note chords; M_1 and M_2 are real constants; O is the *order-of-preference function* and $[\]$ denotes the 'integer part of'. The order-of-preference

function corresponds to the user's own harmonic preference and is specified separately.

The index of dissonance may be thought of as a measure of harmonic distance inasmuch as it provides us with a number that corresponds to the amount by which two single notes are separated harmonically. It is important to realise, however, that the dissonance function is *not* a metric. A metric is any function, m , which satisfies the following three conditions:

- i.) $m(x, y) \geq 0 \forall x, y$
- ii.) $m(x, y) = m(y, x) \forall x, y$
- iii.) $m(x, z) \leq m(x, y) + m(y, z) \forall x, y, z$

Clearly, the dissonance function satisfies the first two conditions. However, as will become apparent later, it is possible to specify the order-of-preference function in such a way that the third condition, usually referred to as the *triangle inequality*, does not hold.

For example, consider the tritone interval. It is generally accepted that for many this is a particularly pleasing interval, whilst for others it is extremely unpleasant. Now the tritone is composed of six semitones – two full minor third intervals, which are generally considered to lie in the consonant end of the interval spectrum. It is therefore possible that a human listener who finds the minor third particularly pleasing and the tritone particularly displeasing will return an order-of-preference in which the tritone has an index of dissonance of more than twice that of the minor third. Indeed, the second row of Table 6.1.2 displays just such an order-of-preference.

The dissonance function may be split into three distinct parts.

The first of these, $|c_2 - c_1| \pmod{12}$ essentially determines the interval type formed between the single-note chords c_1 and c_2 . The intervals are classified according to the difference between the two note numbers, working modulo 12. This is a formalisation of the processes at work when a musician calculates the interval between two notes and may be explained further as follows:

We may define a relation, ρ , on the set of pairs of note numbers as follows:

$$(c_1, c_2) \rho (c_3, c_4) \Leftrightarrow |c_2 - c_1| \pmod{12} = |c_4 - c_3| \pmod{12}.$$

It is a simple matter to conclude that ρ is an equivalence, and so partitions the 2-dimensional chord space. It is also easy to see that the equivalence classes here are of the form:

$$C_a = \{(c_1, c_2) \in C : |c_2 - c_1| \pmod{12} = a\}.$$

Thus, we associate as equivalent those pairs of single-note chords which, when combined form intervals of equal semitone width modulo twelve. In other words, we place pairs of single notes into twelve chord classes depending on their interval size. This is illustrated in Table 6.1.1 below.

Interval width in semitones	Interval class
0, 12, 24, ...	Unison
1, 13, 25, ...	Minor second
2, 14, 26, ...	Major second
3, 15, 27, ...	Minor third
4, 16, 28, ...	Major third
5, 17, 29, ...	Perfect fourth
6, 18, 30, ...	Tritone
7, 19, 31, ...	Perfect fifth
8, 20, 32, ...	Minor sixth
9, 21, 33, ...	Major sixth
10, 22, 34, ...	Minor seventh
11, 23, 35, ...	Major seventh

Table 6.1.1 – The twelve basic interval classes.

This information may be represented graphically:

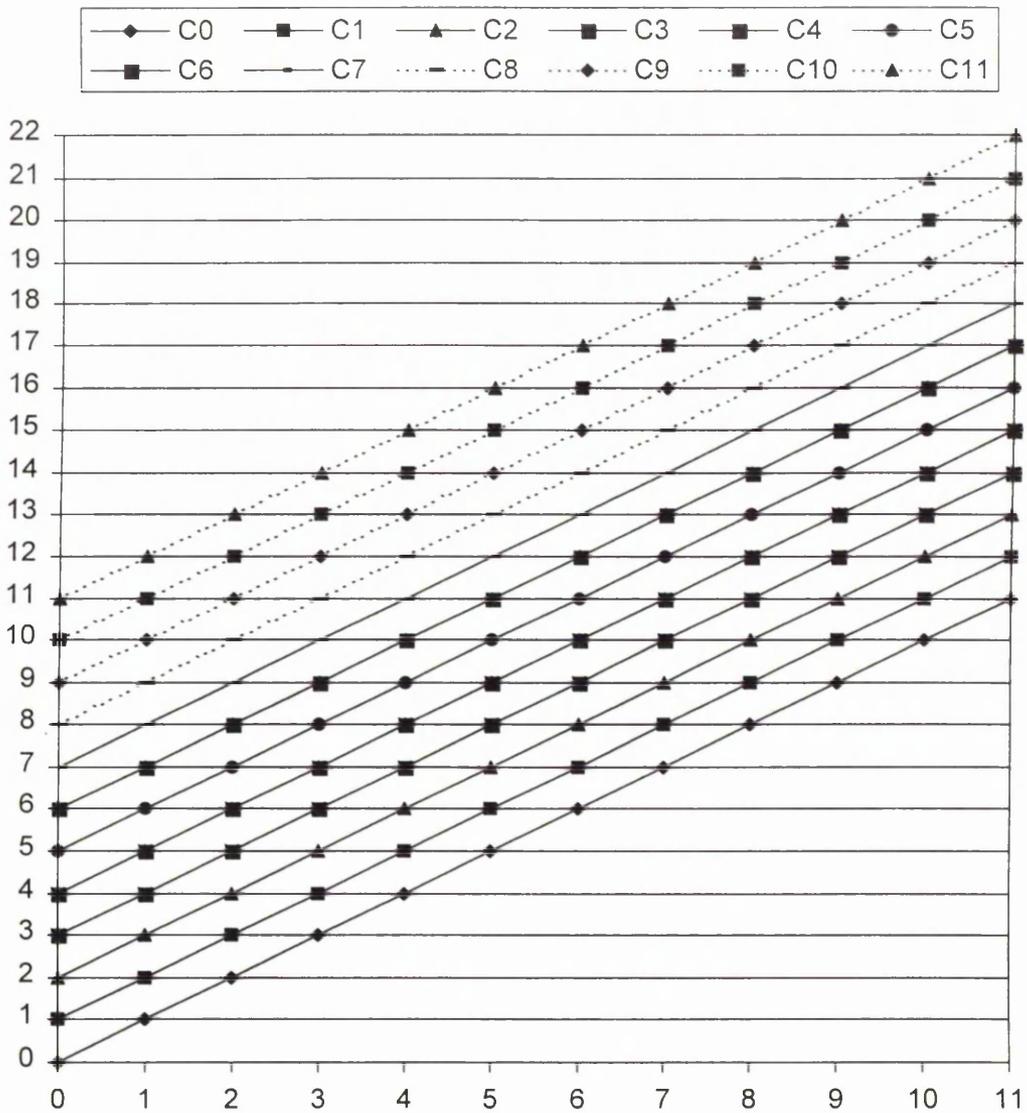


Figure 6.1.1 – Part of the twelve equivalence classes formed by the relation ρ as defined above.

The second part of the dissonance function is the application of the order-of-preference function, which is discussed more fully in Section 6.1.2.

Finally, the term $\left\lceil \frac{|c_2 - c_1|}{12} \right\rceil$ equals the total number of octaves over which the interval is stretched.

Thus, in order to determine fully the dissonance function with which we will work, we must specify three things – the two real constants, M_1 and M_2 , and the order-of-preference function, O .

We are free to select any real constants, M_1 and M_2 , which return satisfactory results for a human listener. In particular, if we assign the value $M_1 = 1$, then M_2 should be set to the reciprocal of the number of octaves over which an interval may be stretched before its harmonic character changes. This may be determined experimentally. The final term in the dissonance function then reflects a scaling factor that will only alter the input chord's overall position in the order of preference if it is stretched more than the critical value of $1/M_2$ octaves.

6.1.2 The order-of-preference function

As was mentioned in the previous section, it is a simple matter to consult a text on music theory in order to obtain a discussion on the relative consonance and dissonance of the musical intervals. However, these arguments are more often than not based on considerations of the harmonic series and do not take into account individual preference.

Therefore, in keeping with the practical nature of this research, it was decided that a statistical survey would be undertaken in order to determine an 'average' harmonic preference, taking into account the tastes of both trained musicians and non-musicians.

The object of this experiment was to try and establish the distribution that underlies harmonic preference. In order to achieve this the twelve simple intervals were played on piano, and recorded onto CD. The CD was then distributed along with a response form (see Figures 6.1.2a and 6.1.2b) to the sample audience, who auditioned the CD in private and completed and returned the response forms.

The sample audience was required to assign to each interval an integral value between 0 and 11 depending on how pleasant (or unpleasant) they thought they sounded. On this scale 0 represents the most consonant (pleasant) chords, whilst 11 represents the most dissonant. If it was felt that two or more chords were equally consonant, the

same integer could be assigned to each. This ranking corresponds to the order-of-preference function, O .

Thus, we conclude that the order of preference function is any many-to-one function of the form,

$$O: \mathbb{Z}_{12} \rightarrow \mathbb{Z}_{12}.$$

Note, however, that the function may not be one-to-many, since each chord must have a unique rank.

We may also, if necessary, extend the above definition slightly to allow functions of the form:

$$O: \mathbb{Z}_{12} \rightarrow [0, 12),$$

thus allowing for non-integral rankings.

However, for the purposes of this experiment, and in order to keep the values as simple as possible we will round the mean rankings for each chord to the nearest integer.

Sound Experiment No. 1

Harmonic Preference

Object

The object of this experiment is to try and establish the distribution that underlies harmonic preference.

Equipment

In order to participate in this experiment, you will need a copy of the Harmonic Preference CD, a CD player and this response form

Procedure

Retire to a quiet environment where you will not be disturbed. Place the CD in a CD player and listen to tracks 1 to 12. You will hear a number of two-note chords played consecutively. Each chord is played twice, with the individual notes sounding between playings to help gauge the overall width of the interval.

Once you have listened to the chords, assign to each an integer between 0 and 11 depending on how pleasant (or unpleasant) you thought they sounded. On this scale 0 represents the most consonant (pleasant) chords, whilst 11 represents the most dissonant. If you feel that two or more chords are equally consonant, then you may assign the same integer to each.

Try to evaluate each chord on its own merit. Do not try to imagine it in a musical context – all we are interested in is the pleasantness of the sound that two notes make when combined.

For example, suppose we have 5 chords, numbered 1, 2, 3, 4 and 5, which are to be labelled on a scale running from 0 (most consonant) to 4 (most dissonant). A typical response might look something like:

Chord Number	Rank
1	0
2	3
3	3
4	1
5	4

This would mean that the order of preference is 1, followed by 4, then 2 and 3, and finally 5. Note that we have omitted position 2. This signifies that whilst chords 2 and 3 are equally pleasant, they are harmonically closer to chord 5 than to chord 4 in the overall order of preference.

When ordering the chords, you may find that it helps to switch between them out of order to correctly judge their position in the order of preference. You can do this by using the track navigation controls on your CD player.

Once you have determined your order of preference, complete the response form on page 2 and return it to me with the CD.

Remember: There are no right or wrong answers to this experiment. Music appreciation is subjective and what some people find pleasant, others find excruciating. For this reason, I would appreciate it if you did not ask for the opinions of anyone else, whether or not they are taking part in this experiment – the opinions of a third party may skew the data.

Thank you for your time.

Figure 6.1.2a – First page of the harmonic preference response form.

Chord Number (Track)	Rank (0 – 11)
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

Figure 6.1.2b – Second page of the harmonic preference response form.

Intuitively one would expect the traditionally consonant intervals, such as the octave and the major third to be assigned the lowest rank, and the traditionally dissonant intervals such as the minor second and major seventh to be assigned the highest rank. The overall trend of the statistical survey does seem to back up this intuitive feeling.

The results are illustrated in Tables 6.1.2 and 6.1.3 below:

Chord	Octave	M7	m7	M6	m6	p5	TT	p4	M3	m3	M2	m2
	2	8	9	0	3	2	8	5	3	6	8	11
	6	11	9	3	3	2	8	6	0	2	5	11
	0	7	6	2	2	2	2	2	0	2	4	11
	0	10	5	1	3	2	9	4	8	8	6	11
	0	9	8	2	6	6	7	6	3	6	7	11
	6	10	5	4	9	3	7	2	1	0	8	11
	0	4	0	1	6	2	7	7	7	9	10	11
	11	3	3	8	10	11	7	11	9	10	2	0
	10	4	2	11	5	6	4	5	6	5	2	0
	6	9	7	5	11	2	0	4	0	4	9	1
	9	11	8	1	8	7	10	10	0	10	10	11
	6	6	6	6	6	6	6	6	6	6	6	6
	0	9	7	4	6	1	10	1	3	5	8	11
	0	9	4	1	2	4	1	0	0	0	6	11
	4	7	6	4	7	6	7	4	2	4	0	11
	8	6	8	8	11	6	6	6	3	3	3	0
Total	68	123	93	61	98	68	99	79	51	80	94	128
Mean	4.25	7.6875	5.8125	3.8125	6.125	4.25	6.1875	4.9375	3.1875	5	5.875	8
St. Dev.	4.0083	2.5224	2.5617	3.1245	3.0523	2.7203	2.9937	2.9545	3.1031	3.1833	3.0083	4.7889
Rank	4	8	6	4	6	4	6	5	3	5	6	8

Table 6.1.2 – Rank for each of the twelve simple intervals.

Chord	0	1	2	3	4	5	6	7	8	9	10	11	Total
Octave	6	0	1	0	1	0	4	0	1	1	1	1	16
M7	0	0	0	1	2	0	2	2	1	4	2	2	16
m7	1	0	1	1	1	2	3	2	3	2	0	0	16
M6	1	4	2	1	3	1	1	0	2	0	0	1	16
m6	0	0	2	3	0	1	4	1	1	1	1	2	16
p5	0	1	6	1	1	0	5	1	0	0	0	1	16
TT	1	1	1	0	1	0	2	5	2	1	2	0	16
p4	1	1	2	0	3	2	4	1	0	0	1	1	16
M3	5	1	1	4	0	0	2	1	1	1	0	0	16
m3	2	0	2	1	2	2	3	0	1	1	2	0	16
M2	1	0	2	1	1	1	3	1	3	1	2	0	16
m2	3	1	0	0	0	0	1	0	0	0	0	11	16

Table 6.1.3 – Number of votes cast in each category for the twelve simple chords.

Figures 6.1.3 and 6.1.4 below display the data graphically:

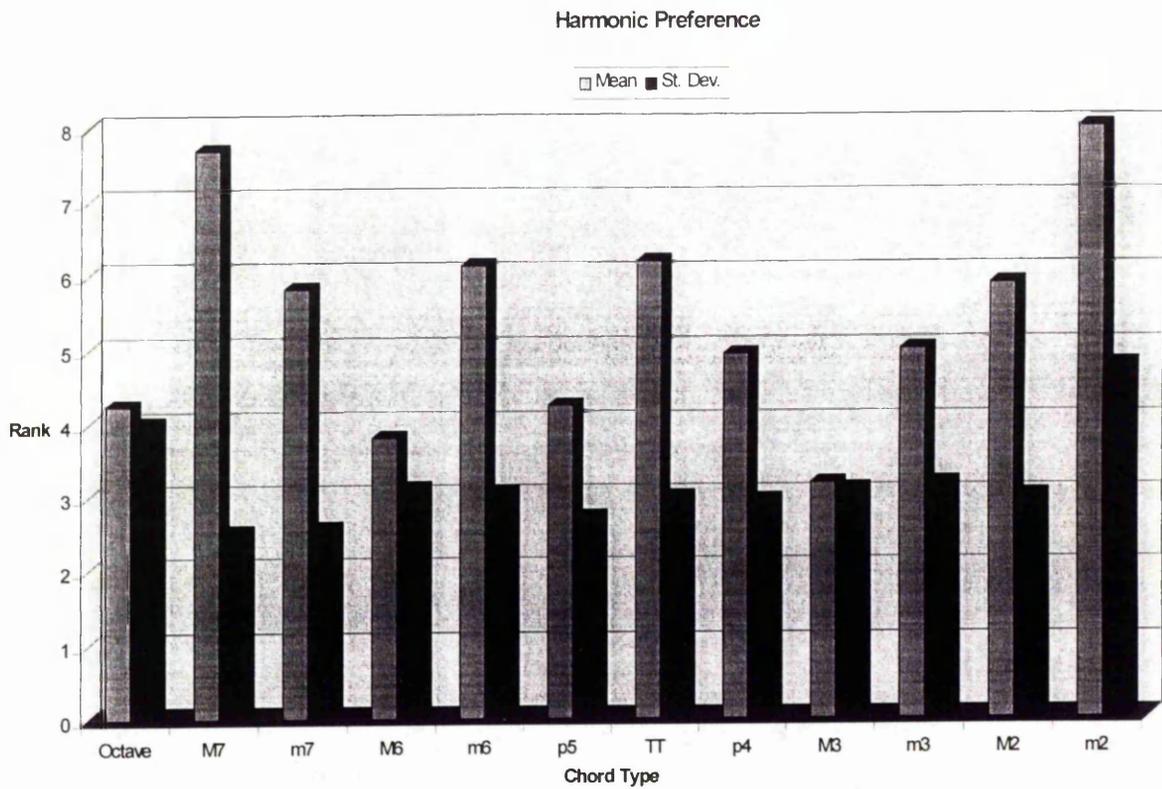


Figure 6.1.3 – Rank for each of the twelve simple intervals.

Harmonic Preference Histograms

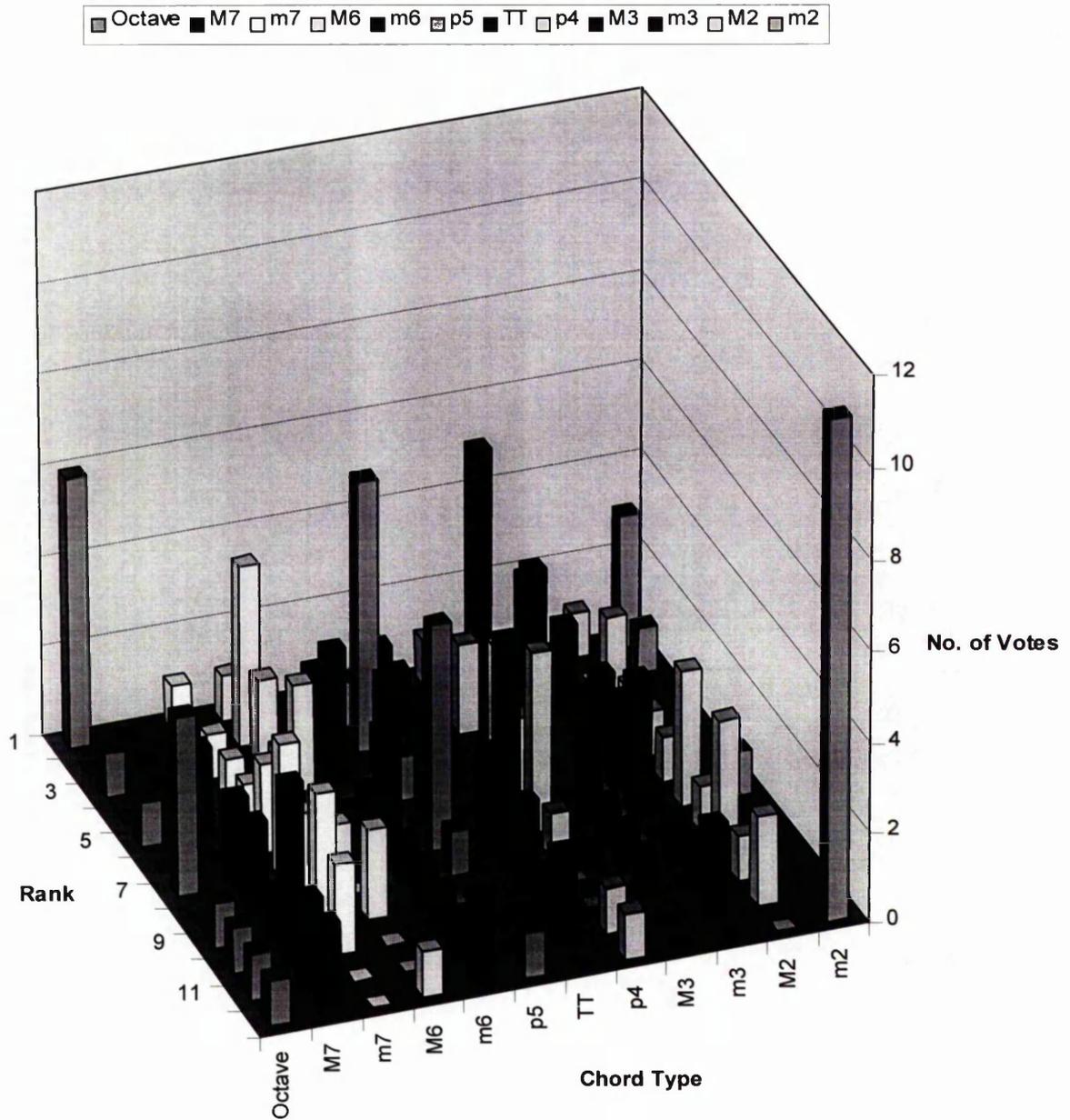


Figure 6.1.4 – Number of votes cast in each category for the twelve simple chords.

It can be seen from both the histogram and the graph of mean values that the underlying trend does seem to favour the traditionally consonant intervals over the

dissonant intervals, whilst votes for the intermediate intervals, such as the tritone and the minor seventh, are spread fairly evenly across the range of values.

It is interesting to note that while the octave interval received more 0 rankings than any other interval, as might be expected, it also received four votes for a ranking of 6 and a single vote for each of the four highest rankings. It is the author's belief that these listeners considered the harmony of the octave interval too 'pure'. As music has developed over time, harmonies have become more and more adventurous – in today's popular music, grating harmonies and noise are commonplace. An ear that has been cultured to such music may find itself unexcited by the natural harmony offered by the octave interval.

Note also that the standard deviation for each chord shape is, at present, fairly high. This is most likely due to the small statistical sample who agreed to participate in the experiment, and would in all probability be reduced with a larger sample set.

Using these data, we construct the following order-of-preference function to use as a typical response in the dissonance function:

Interval width in semitones	Rank
0	4
1	8
2	6
3	4
4	6
5	4
6	6
7	5
8	4
9	5
10	6
11	8

Table 6.1.4 – A typical order-of-preference function.

6.1.3 The dissonance network

In producing a workable computing implementation of the dissonance function defined in Sections 6.1.1 and 6.1.2 above, we must consider several possible solutions. In deciding with which to proceed we must consider which of the available solutions will

- consistently produce reliable results

- be the simplest to implement in computer code
- provide the user with the most transparent integrated interface

It is the author's belief that the pattern classification network described below is the best solution, since this not only produces provably correct³¹ results, but is also simple to code and provides the end user with an easy-to-use graphical interface.

We will call the completed automaton the *dissonance network*.

6.1.4 Constructing the network

We break down the construction of the dissonance network into three parts, corresponding to the three sections of the dissonance function, d .

The first stage in the construction is the classification of the input pair, c_1, c_2 into one of the twelve interval classes, C_0, C_1, \dots, C_{11} .

As we saw in Section 6.1.1, this may be reduced to a problem of simple geometry – the interval class C_i consists of all note pairs, (c_1, c_2) such that working modulo 12, (c_1, c_2) lies on the line $y = x + i$.

Classification is achieved by means of the following network:

³¹ That is, the network can be shown to return precisely the values predicted by the dissonance function, d .

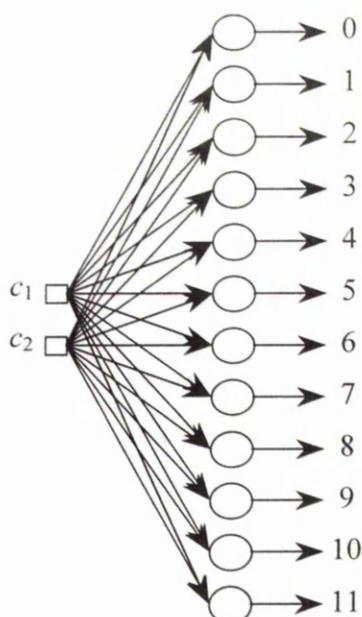


Figure 6.1.5 – The chord classification stage of the dissonance network.

Here, the input pair is fed to the two input nodes on the left, with c_1 being fed to the upper input node and c_2 being fed to the lower input node. The output nodes are labelled 0, 1, 2, ..., 11 from top to bottom.

We desire node i to be active, that is return the value 1, if and only if the input pair belongs to equivalence class C_i . All other nodes should be inactive, returning the value 0.

Then

$$\text{state of node } i = \begin{cases} 1, & \text{if } |c_2 - c_1| \pmod{12} = i \\ 0, & \text{otherwise} \end{cases}$$

The next part of the dissonance function is concerned with ordering the interval classes according to their rank. This is achieved by referring to the pre-determined order of preference function, O .

In order to apply such a function using the dissonance network, we simply scale the output of node i by a factor of $O(i)$.

The final part of the function adds a stretching coefficient that corresponds to the number of octaves over which an interval can be stretched before its character changes and alters its position in the order of preference. This may be achieved by summing the scaled outputs from the first layer and applying a bias term, which is the weighted width of the interval in octaves.

The complete network then looks like:

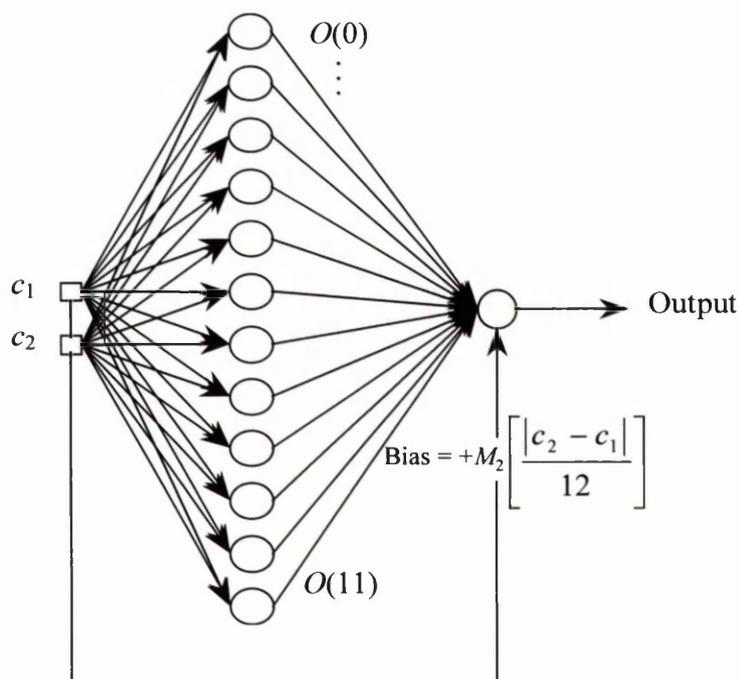


Figure 6.1.6 – Dissonance network for two single-note chords.

The dissonance network now offers an automated computational device for calculating the dissonance index of any two-note interval. Further, since it was constructed from the function using exact geometric arguments, the network is guaranteed to return the correct response.

The dissonance network is actually a simple neural network. It may seem a little grandiose to term it thus, given that no learning process – surely the major feature of neural networks – takes place, but this is because the general form of the dissonance function could be specified in this, the simplest of cases. We saw in Section 6.1.1 that the problem can essentially be reduced to that of simple pattern classification, which is easily solved with the network described above.

However, it is our ultimate aim to provide a system that will allow for chords of up to 8-note depth as are currently generated in CAMUS 3D. Given the variable chord depth within the system and the large number of component intervals that form such chords, it is extremely unlikely that we will be able to express the extended dissonance function in such a succinct manner. It is here that the neural approach will show its merit, since we may use an order-of-preference response, combined with certain adaptations described in Section 6.1.5 below, to provide the network with a training set from which it will create an approximation to the dissonance function.

The 2-note dissonance network has been coded and tested and is presented in the data section of the CD-ROM that accompanies this thesis.

Figures 6.1.7 and 6.1.8 illustrate the interface.

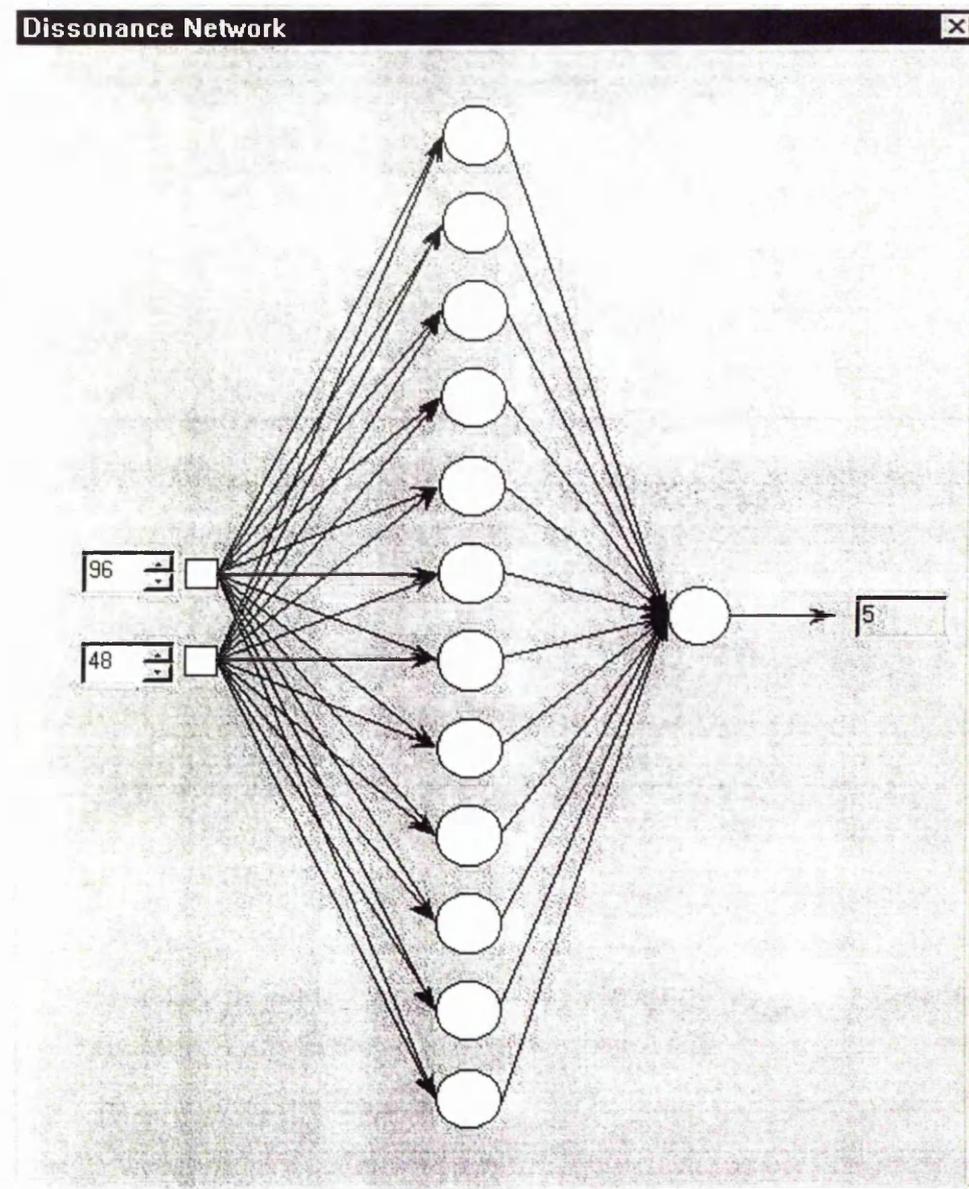


Figure 6.1.7 – Main operating window of the Dissonance Network.

Network Parameters

Order of Preference

Unison:	4	Tritone:	6
Semitone:	8	Perfect 5th:	5
Tone:	6	Minor 6th:	4
Minor 3rd:	4	Major 6th:	5
Major 3rd:	6	Minor 7th:	6
Perfect 4th:	4	Major 7th:	8

Interval's tonal quality changed after 4 octaves.

Figure 6.1.8 – Network parameter settings dialog box.

6.1.5 Validity of the index of dissonance

It is important to realise that the index of dissonance is based partly on experimental evidence and partly on hypothesis.

The two hypotheses that were made when the general form of the dissonance function was proposed in Section 6.1.1 are as follows:

- i.) That a human listener will favour certain interval types over others and that the harmonic preference is independent of the actual pitches involved.
- ii.) That the number of octaves by which a chord is stretched will influence the listener's preference. Eventually there will be a point at which this stretching will affect the preference to such an extent that the listener will favour other chords that are ranked lower in the order of preference, but which do not encompass such a large interval. This point is assumed to be constant for all interval types and is independent of the actual pitches involved.

Let us examine these two statements.

In constructing the dissonance function, d we have assumed that a human listener will favour certain interval types over others. This is a reasonable assumption to make,

since it is generally accepted that certain intervals are more pleasing to the ear than others.

However, we have also assumed that the harmonic preference is independent of the actual pitches involved. In other words we assume that the major third, say, sounds equally pleasant throughout the human hearing range. This is unlikely to be the case, although it is almost certainly sufficiently close to the truth to be used as a first approximation.

There is a considerable body of psycho-acoustic research which shows that the human hearing system is more attuned to sounds in the mid-range of frequencies, roughly equal to the human speech frequencies (see, for example [Jenkins & Sano, 1989]). Indeed, one does not need to venture too far from the mid-range to experience this change of harmonic character – by playing close harmonies at the extremes of the piano keyboard one can hear the ‘muddyness’ or shrillness of chords played in the extreme bass or treble. This suggests that the dissonance function could be improved upon by the introduction of a factor which accounts for the distance of the chord from the mid-range of frequencies. This factor, is likely to vary from person to person, and thus would have to be determined independently for each user. One solution may be to introduce a neural network that learns this parameter by subjecting each user to a training set and examining his or her responses. Similar systems are currently in use in commercially available speech recognition software, such as *FreeSpeech 98* by Philips ([Philips, 1999]).

In addition, in making the above assumption, we presume that similar intervals sound equally pleasant in different keys. Logically this is reasonable, particularly when we consider that the frequency ratios involved are identical in an equal-tempered pitch system. However, some human listeners insist that they can detect subtle changes in character between key signatures. However, research suggests that there is no psychoacoustical basis for these claims ([Corso, 1957]), although it is possible that certain physical circumstances (such as the effect of the fixed frequency resonances in piano soundboard) can create such changes.

Whether or not these are sufficient to necessitate the introduction of a correction parameter requires further study.

Secondly, we have assumed that the amount by which a chord is stretched will influence the listener's preference. Once more, this is a reasonable assumption and is illustrated well in Section 6.1.1.

However, we have also assumed that the stretching factor, M_2 , is constant for all interval types and is independent of the actual pitches involved. Again, this is unlikely to be valid. For example, there is evidence that certain pitches actually become *more* consonant as they are stretched -- one need look, for example, only as far as the music of George Gershwin for evidence of this ([Schwartz, 1973]).

Similarly, to assume that the stretching factor is independent of position in the frequency spectrum is to understate the case. For example, an interval stretched over several octaves whose lower note is in the bass range will generally sound richer than a similar interval whose lower note is in the mid to treble range. Clearly, these issues must be addressed if the present approximation to dissonance is to be improved upon.

6.1.6 Extending the dissonance network

The dissonance network as described above represents the simplest case. Clearly, in order to be integrated fully into the CAMUS 3D system, the network must be extended in order to cater for the increase in the number of notes produced by the CAMUS 3D mapping.

Recall that each live cell in the 3-dimensional Game of Life gives rise to a 4-note chord event. Introducing parallel cell checking will thus require the network to be able to cope with a minimum of eight notes, this being the total number of (not necessarily distinct) notes formed when two 4-note chords are combined.

At present, we can only speculate on what form such a network will take. Time constraints preclude us from carrying out the work required to produce detailed analysis of the network.

We now present one possible extension of the network. It is hoped that more detailed study of the network will be undertaken after the current research tenure is completed.

Suppose we wish to extend the capabilities of the dissonance network from the current limitations of two single-note chords up to two 2-note chords. The first thing to notice is that the resulting combined chord will be of 4-note depth.

Let us consider, for a moment, some common chord types available to the composer. These are

- i.) major triad with octave
- ii.) minor triad with octave
- iii.) dominant seventh
- iv.) diminished seventh

In each of these four chords, there are precisely $\binom{4}{2} = 6$ component intervals.

For example, consider the major triad with octave given by (CEGC'). The component intervals are

Notes of chord	Interval formed	Index of Dissonance
CE	major third	3
CG	perfect fifth	4
CC'	octave	4
EG	minor third	5
EC'	minor sixth	6
GC'	perfect fourth	5

Table 6.1.5 – Component intervals of the major triad with octave.

Similarly, the component intervals for the other four chords can be worked out and tabulated as in Table 6.1.6 below:

Chord	Component intervals
I	M3, p5, 8ve, m3, m6, p4
ii	m3, p5, 8ve, M3, M6, p4
iii	M3, p5, 8ve, m3, TT, m3
iv	m3, TT, M6, m3, TT, m3

Table 6.1.6 – Component intervals of the four chord types listed above.

In the author's opinion³², the chords listed above are amongst the most consonant. It is easy to see that on the whole, they are composed of intervals that lie towards the consonant end of the harmonic spectrum – the one exception being the tritone interval, which is indeterminate and can sound both consonant and dissonant depending on the context.

Similarly, if we consider a chord which, to the author's ears at any rate, is dissonant, such as that formed by the notes (CC#AB), we find that the component intervals – m2, M6, M7, m6, m7, M2 – mainly lie in the dissonant range.

We thus form the hypothesis that those chords that are formed from intervals that are consonant will, on the whole, sound more harmonious than those formed from mainly dissonant intervals. Chords that are formed from intervals which are indeterminate or which have a roughly equal mix of consonant and dissonant intervals will tend lie between the two extremes of consonance and dissonance and their character will depend on the musical context.

It may possible, therefore, to get a measure of the dissonance of a 4-note chord by summing the indices of dissonance of the component intervals, thus obtaining a combined index for the chord. As before, lower values for the combined index (representing chords composed of mainly consonant intervals) represent those chords which are considered consonant, and higher values (representing chords composed of mainly dissonant intervals) represent those chords which are considered dissonant.

Note, however, that as before, the results depend wholly on the user's particular harmonic preference. A small value of the index of dissonance does not, necessarily indicate that the chord will sound harmonious to all listeners, rather that it is composed of many of a particular listener's favourite intervals, and so closely matches his or her own aesthetic preferences.

³² These results must remain speculative until such times as statistical data can be accumulated to verify the claims.

The above argument can be extended to the situation of CAMUS 3D, which combines two 4-note chords to give a single 8-note chord. An automated network for calculating the combined index of dissonance may be constructed by running several networks of the type described in Section 6.1.4 in parallel. We illustrate in Figure 6.1.9 below.

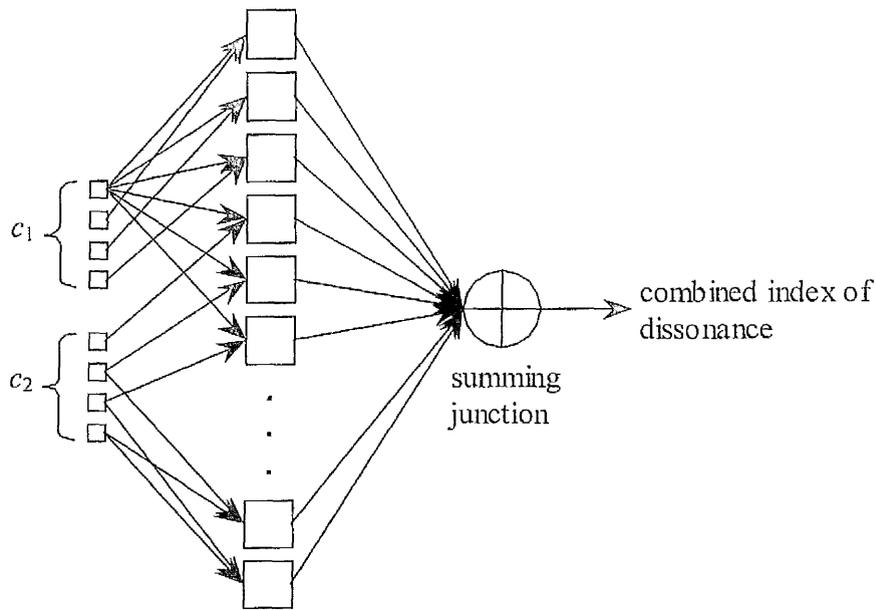


Figure 6.1.9 – Automated network for calculating the combined index of dissonance given two 4-note chords, c_1 and c_2 . The two chords, c_1 and c_2 , are fed to the input nodes on the left-hand side of the network. The note numbers for each of the component notes of the chords are sent in pairs to one of the 28 processing units in the first hidden layer. Each of these nodes represents a complete dissonance network of the type described in Section 6.1.4, the output from which is the index of dissonance for the input pair. These values are then combined at a summing junction to give the combined index of dissonance for the 8-note chord formed from c_1 and c_2 .

6.1.7 Integrating the dissonance network with CAMUS 3D

As has been mentioned earlier, the trend in the development of the CAMUS 3D system is towards parallel cell checking and performance so that more than one chord can be played at a time. We illustrate the workings of the dissonance network by presenting two examples of how the system might be utilised to aid in the production of music that performs two chords simultaneously.

Firstly, we suppose that the system scans through the cells row-by-row and column-by-column as in the standard CAMUS 3D algorithm until a live cell, c_{abc} , which lies at cell position (a, b, c) is reached.

Now a second scan is performed. The system scans through the 3-dimensional Game of Life space, excluding the current cell. In so doing, the system forms a table of data, which are the indices of dissonance for each of the cell pairs, (c_{abc}, c_{ijk}) , as i, j and k run from 0 to 11, with $i \neq a, j \neq c, k \neq b$.

Once the second scan is complete, it is a simple matter to examine the table and choose the cell pair that forms the best³³ harmonic mix according to the user's own harmonic preference.

This process is illustrated in Figure 6.1.10.

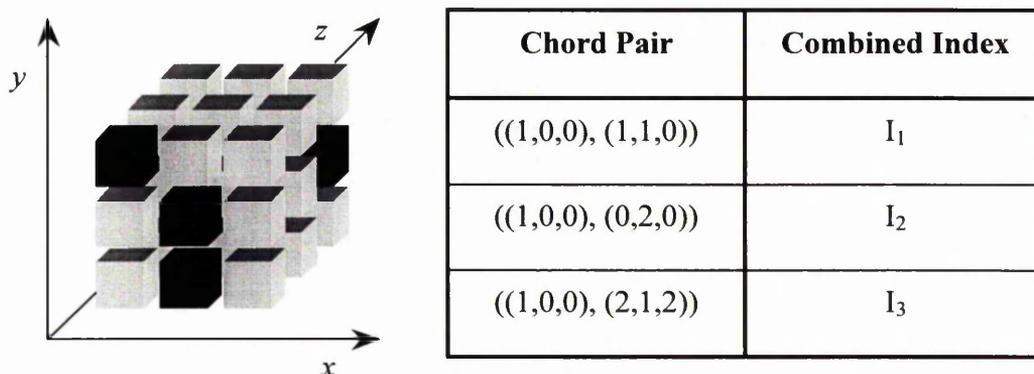


Figure 6.1.10 – An example usage of the dissonance network. The algorithm scans through the automaton until a live cell is reached. Here, the first cell would be the one at position $(1, 0, 0)$. Now a second scan, omitting the current cell, is performed. Live cells are also discovered at $(1, 1, 0)$, $(0, 2, 0)$ and $(2, 1, 2)$. The index of dissonance for each of the three chord pairs formed from $(1, 0, 0)$ and one of the other three live cells is calculated and tabulated. The chord pair that results in the lowest index of dissonance (i.e. that which provides the closest match to the composer's aesthetic) is selected by the algorithm for performance.

³³ In this case, the best harmonic mix is that which most closely matches the user's harmonic preference, that is, the cell pair that has the smallest index of dissonance.

Alternatively, we can envisage a system that examines only cells in the immediate neighbourhoods of the cell in question.

Suppose, as before, that the system scans through the cells row-by-row and column-by-column as in the standard CAMUS 3D algorithm until a live cell, c_{abc} , which lies at cell position (a, b, c) is reached.

This time, however, the system only examines the 26 cells that neighbour c_{abc} . Again, the system tabulates the indices of dissonance for each of the cell pairs.

Finally, the table is examined and the best cell pair chosen to contribute to the composition.

This process is illustrated in Figure 6.1.11.

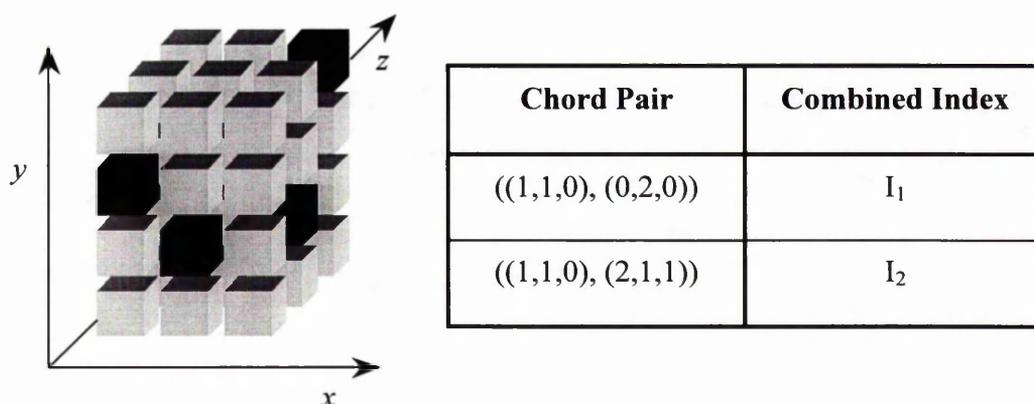


Figure 6.1.11 – An alternative example usage of the dissonance network. The algorithm scans through the automaton until a live cell is reached. Here, the first cell would be the one at position $(1, 1, 0)$. Now the algorithm examines the 26 cells in the immediate neighbourhood. Live cells are also discovered at $(0, 2, 0)$ and $(2, 1, 1)$. Note however, that although the cell at position $(2, 3, 2)$ is live, it does not lie in the immediate neighbourhood of $(1, 1, 0)$ and so is not considered. The index of dissonance for each of the two chord pairs formed from $(1, 1, 0)$ and one of the other two live cells in the neighbourhood is calculated and tabulated. Again, the chord pair that results in the lowest index of dissonance (i.e. that which provides the closest match to the composer's aesthetic) is selected by the algorithm for performance.

This system of chord selection mimics certain traditional composition practices. At each stage, there may be several choices of cell pairs (i.e. $2(n + 1)$ -note chords, where n is the dimensionality of the system). The system considers each of the possibilities in turn and finally decides on the one that most closely fits the user's aesthetic criteria. There are clear parallels here with the process of harmonising a bass or melody line – at each stage there may be several choices of chord available to the composer, who must consider each in turn and decide which one best fits a given situation.

However, it is important to note that whilst the composer is free to choose chords, the system assigns what it considers to be the best chord based on the index of dissonance. This has some significance when one considers that composers often introduce dissonances as a compositional device, and so at each stage may introduce unexpected chords, or chords which are not necessarily the most consonant available. This helps to keep the composition fresh and interesting. It may be worthwhile then, to introduce a further mechanism in the above decision routine which selects alternative chords to the one with the lowest index of dissonance.

6.2 Further development of the system

We conclude this section by discussing further developments of the system and alternative mapping algorithms.

6.2.1 Developing long term trends

It is a consequence of the design of the composition algorithm and the automata employed that the music that is generated tends to sound repetitive after a time. The author believes that this is one of the strongest arguments for retaining a human composer who has overall control of the composition.

As we discussed in Chapter 3, the music that is generated by the CAMUS system is best used by the composer as a source of ideas to be developed into a more complete and well-rounded composition. Thus, we are essentially using the arranging skills of the human operator to control the long-term development of the system.

However, although this setup undoubtedly produces the best overall results, it requires a tremendous amount of work on the part of the end user. Frequently the amount of

work required to produce a finished composition is equal to or exceeds the amount of work required to create a composition from scratch, which surely defeats the purpose of using the algorithmic composition system as an aid to composition in the first place!

The amount of work required to produce the end composition is such that the user may feel daunted by the prospect of composing such a piece. Clearly there is a case for introducing at least a partial system of automating the long-term development of the composition.

Following a talk given by the author to the Department of Computing's algorithmics research group at the University of Glasgow in June of 1998, Dr. Bill Findlay, suggested that it might be possible to develop the system to create interesting long-term development automatically.

He proposed the notion that a third automaton could be made to run alongside the Game of Life and the Demon Cyclic Space to control the structure of the music. This would work by, say, employing a formal grammar in the background to further manipulate the music generated by the system to adhere to a musical style defined by the grammar. As the third automaton developed, it would bring new grammatical rules into play, which would alter the structure of the music.

However, although this sort of configuration is feasible and has been utilised in other systems (see, for example, [Ames, 1983] and [Beyls, 1997]), it is the author's belief that it would remove too many of the human aspects of the composition system. Thus, we require some sort of intermediate measure.

To this end, we propose that the use of the system as a musical scratch pad be further developed. At each timestep, the music that is generated is fed into a transformer module, which allows the composer to apply musical transformations to it.

The transformations will include, but not be restricted to inversion, retrograde motion, transposition, and reflection about a note. The composer then decides whether to keep the transformed phrase and append it to the permanent music file held in memory or discard it and work with a new phrase. This allows for the long-term behaviour to be

controlled from within the system, whilst still allowing the composer to have overall control. Figure 6.2.1 shows how such a system might work.

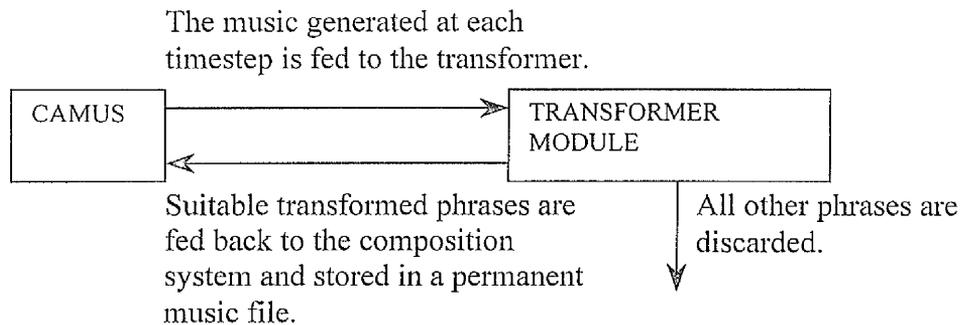


Figure 6.2.1 – The transformer module.

In keeping with the current trend of modularising systems, this functionality would be offered in the form of a “plug-in” program that processes the MIDI output of the system and then passes it back for storage. This also allows for the possibility of using the plug-in system to affect the MIDI output of other software, such as MIDI sequencers and so on.

6.2.2 Fractal orbits

In Section 4.1.6, we discussed the limitations imposed by CAMUS fixed order of cell playback. Here, we present an alternative system of scanning the automaton space to produce a list of cells for playback.

We saw in Section 2 that fractals frequently provide us with natural-looking objects. We propose to harness this property to help the system produce natural-sounding music as follows.

Once the automata have been updated, the algorithm chooses a cell at random. If this cell happens to be live, we choose it as our starting cell, otherwise, we select the nearest cell.

From this point, we calculate the successive quantised orbit points of a 3-dimensional fractal, such as the Lorenz attractor (see Figure 6.2.2). At each stage, the cell is performed if it is live and discarded if it is not. Alternatively, if, as is likely, the

automaton is fairly sparsely populated, the closest cell to the quantised orbit point could be played in order to avoid long periods of silence. This process is then repeated until the user decides to progress to the next automaton timestep.

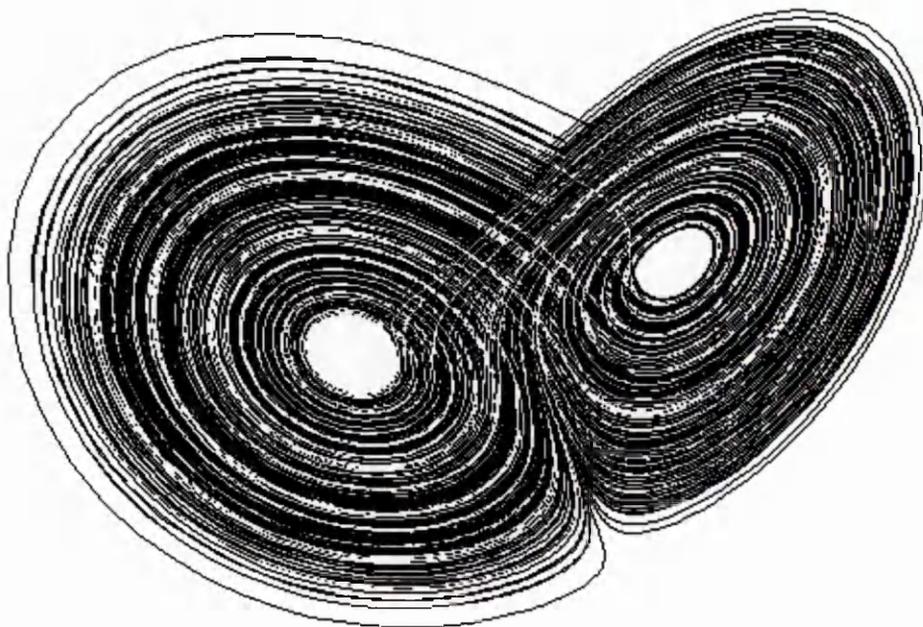


Figure 6.2.2 – The Lorenz attractor.

Of course, this system may be combined with one of the parallel methods of Section 6.1.7 to give an algorithm that combines the benefits of fractally-generated cell orbits – namely the generation of constantly-changing self-similar orbits – with the increased complexity of dual cell performances.

6.2.3 The rhythm of Life

In Section 5.2.5, we examined CAMUS 3D's Markov Chain-based rhythm generation routine. The natural-sounding rhythmic lines that are possible are a testament to the routine's improvement upon the original random number generators used to calculate the note lengths in the previous incarnation of the system. However, there are still areas in which the routine could be improved.

The main problem lies with the long-term trends of the rhythm generator. Although the rhythms sound more natural and have a degree more structure, there is still a general lack of coherence with the rhythmic lines that are formed. For example, it is

possible to engineer certain rhythmic figures, such as the dotted quaver-semiquaver, by manipulating the probabilities of the state-transition matrix. However, it is virtually impossible to specify precisely where and when such events will occur in relation to other rhythmic figures.

Two possible solutions to this problem exist.

The first of these is to utilise higher-order Markov chains in the generation of rhythmic figures. This would benefit the long-term development of rhythmic lines by taking greater account of past events. Moving to, say, a fourth-order Markov chain would allow for the development of fairly complex rhythmic motifs.

However, a problem in interface, similar to that described in Section 4.1.2, arises. By moving to a fourth-order Markov chain, we require a 5-dimensional state transition Matrix to define it. As discussed earlier, there are no simple solutions to the problem of displaying visually the data stored in such an array. Solutions, such as separate pages of data for each step back in time, certainly exist, but these do not provide the user with adequate visual aids to understanding. It is not a trivial matter to visualise such structures, and any lack of visualisation may prove to be a hindrance to usage.

In addition there is the issue of specifying the data in such a large array. Supposing that the number of states in the matrix remains the same, a fourth-order Markov chain would require $8 \times 8 \times 8 \times 8 = 32768$ probability entries.

We should bear in mind that the manual entry of just 64 entries in the first-order case can be very time consuming. Combined with the endless possibilities for getting lost in a 5-dimensional matrix it is clear that some sort of automated method of defining the state-transition matrix is required.

There are several possibilities available to us for this process, most of which, however, prove to be quite unsatisfactory.

The simplest method is to assign random probabilities to the matrix. This would be easy to code and simple for the user to understand. The musical results, however, would be unpredictable, most likely behaving as a random number generator would, thus losing us the benefits offered by the Markov chain.

Another solution is to assign probabilities based on the states of a cellular automaton. Although this solution is slightly more acceptable owing to the degree of structure imposed by the automaton, the user is still faced with the problem of finding a cellular automaton that will give rise to a set of probabilities which match his or her aesthetic criteria.

Perhaps the best approach to this problem lies with fuzzy logic and artificial intelligence. This would allow the user to define the sort of rhythmic figures that he or she would like to be present in the composition using traditional musical language. Indeed, it would be perfectly feasible to use MIDI files as a means of inputting rhythmic figures. Alternatively, the user could tap rhythms directly into the system using a MIDI controller such as a drum pad or MIDI keyboard. AI techniques would then be used to analyse the rhythmic figures and adjust the probability values in the state-transition matrix accordingly.

The main difficulty with this system is, of course, that it is much more difficult to implement than either of the other two techniques. AI is not particularly well understood, and although applications to music theory have been undertaken (see for example [Roads, 1980] and [Baird, Blevins & Zahler, 1993]), this is still very much new territory.

For the end user, however, a well-implemented system of AI should be virtually invisible, and would provide a significantly improved interface for rhythm generation.

The second technique which could be used to improve long-term development of the rhythmic figures generated by the composition algorithm is the use of short rhythm lines to index the rows and columns of the state transition matrix. This would allow the development of complex, coherent rhythm lines whilst still retaining the ease of use of the first-order Markov chain.

The system could be made even more flexible if we allow for variable states. In this situation, each state would offer several choices of rhythmic motif, between which the user would select. By changing the selections as the composition progresses the user would be able to retain musical coherence, whilst introducing a gradual change of rhythmic direction.

The choices for each of the variable states would be user-selectable from a library of pre-defined rhythmic phrases, although the possibility for manual specification as in the AI analysis described above also exists.

7. Conclusion

We begin our final chapter by discussing the degree of success we have obtained in attaining the research goals, which were introduced in Chapter 1. We conclude with a brief debate on the issue of authorship of an algorithmic composition.

7.1 Research goals

In Section 1.3.2 we proposed the following research goals:

- i.) Produce new algorithms for mapping mathematical structures to music.
- ii.) Gain knowledge into what kinds of mathematical structures and mappings produce useful results in sound.
- iii.) Produce new tools for music composition.
- iv.) Produce new compositions.

In the remainder of Section 7.1, we examine how successful we have been in meeting these goals.

7.1.1 Producing new musical algorithms

This research project grew from Dr. Miranda's CAMUS system.

As we have seen from Chapter 3, the original CAMUS system is a composition algorithm that uses the 2-dimensional Game of Life and Demon Cyclic space automata to generate music.

During the first year of this research project, the original system was studied in depth and its strengths and weaknesses were identified. During this time, the algorithm was developed slightly to give a new algorithm that retained the original's best features along with some new ones, which were mainly concerned with interface. These changes were detailed in two conference papers – *Dynamical Systems and Applications to Music Composition: A Research Report* ([McAlpine, Miranda &

Hoggar, 1997a]) and *A Cellular Automata Based Music Algorithm: A Research Report* ([McAlpine, Miranda & Hoggar, 1997b]).

These changes led to CAMUS version 2.0 for Windows 95, which was published on the cover-mounted CD of the Mix magazine, issue 45, and which is presented on the data section of the CD-ROM which accompanies this thesis.

Following a period of testing and experimentation with CAMUS version 2.0, the system was redesigned from scratch to create the CAMUS 3D algorithm.

Although CAMUS 3D is clearly derived from the original algorithm, there are a number of significant differences between it and original. These were detailed in Section 5 and highlighted in Table 5.5.1.

In addition, the CAMUS 3D algorithm has been the subject of two papers: *Music Composition by Means of Pattern Propagation*, ([McAlpine, Miranda & Hoggar, 1998]) and *Making Music With Algorithms: A Case Study System* ([McAlpine, Miranda & Hoggar, 1999]).

In addition to this, we have proposed, though not implemented a number of alternative mappings from cellular automata and fractals to music. Details of these may be found in Sections 4 and 6, and in [McAlpine, Miranda & Hoggar, 1997a] and [McAlpine, Miranda & Hoggar, 1997b].

7.1.2 what mappings produce useful results in sound?

After its completion, we spent a considerable amount of time working with CAMUS version 2.0 in order to see what effect different initial cell configurations had on the resulting output from the system. The results were catalogued in Section 3.3.

This work helped to give some insight into the type of music that is produced by different cell configurations. Further, although it is impossible to predict the actual output from a given cell configuration because of the stochastic methods employed, we were able to develop the “reverse engineering” method of composition described in Section 3.3.7.

The effectiveness of this process was ably demonstrated by its use in the development of a number of new compositions.

Unfortunately, because of the changes in the underlying algorithm it has not been possible to construct a comparable method of working in CAMUS 3D. However, we believe that the improvements to the system have been such that notable musical output can be produced without the need to indulge in such meticulous planning. Further, as we move towards parallel cell checking and the use of Dissonance Networks in the generation of multitimbral music (see Section 6.1), such issues become less relevant.

In any case, we noted in Section 5.5.4, that with a minimal amount of work, the implementation of CAMUS 3D could be altered to trace the playing order of cells.

7.1.3 Producing new tools for music composition

As a direct result of this research, two complete pieces of music software – *CAMUS version 2.0 for Windows 95* and *CAMUS 3D for Windows 95* – have been produced. Both of items of software have been donated to the public domain and are freely downloadable from the research group's web page:

<http://www.maths.gla.ac.uk/~km/research.htm>.

That the former was accepted for publication by the international news-stand magazine, the Mix, suggests that the software is genuinely useful as a music tool.

In addition to this, a partially complete implementation of the Dissonance Network is available for download at the above site. This software is also included on the accompanying CD.

7.1.4 Producing new compositions

CAMUS and CAMUS 3D have been used to compose a number of works.

These are:

Sonatina for Woodwind Ensemble

Kenny McAlpine, 1997

Minuet and Trio for Woodwind Ensemble

Kenny McAlpine, 1997

Calls from across the Ether

Kenny McAlpine, 1998

Four's a Crowd

Kenny McAlpine, 1998

Step by Step by Step

Kenny McAlpine, 1998

Melancholia

Kenny McAlpine, 1999

Untitled Sequence

Campbell MacLean, 1998

Entre l'Absurde et le Mystère

Eduardo Miranda, 1995

Jazz 3

Eduardo Miranda, 1998

It is significant that *Four's a Crowd* was used by Jerome Joy as part of his *Collage Jukebox* project [Joy, 1998].

In addition to this, CAMUS 3D is also being used by Dr. Miranda to compose a commissioned work for solo piano. The piece, titled *Grain Stream* is scored for piano and sampler with electronic sounds. It was commissioned by Studio Forum, Annecy in France and is due to be premiered in February 2000 in Annecy.

All of these compositions, with the exception of *Grain Stream*, can be auditioned on the audio section of the accompanying CD.

7.2 Authorship of algorithmic compositions

During the course of this research the author has necessarily used and become familiar with a number of algorithmic composition systems. Some of these are described in Appendix D.

In addition, the author has also come across several different opinions of algorithmic composition. One of the most common, particularly amongst traditional musicians is that the study of algorithmic composition is not positive because it removes the need for human creativity. However, it has been the author's experience that even the best of algorithmic composition systems require the intervention of a human composer to produce musically coherent works.

Certainly there is always the danger that a lazy or disinterested composer will abuse algorithmic systems, using the formalised methods as a substitute for his or own creativity. Used responsibly, though, algorithmic composers can transform the often laborious tasks involved in the creation of a new composition.

As Curtis Roads points out in [Roads, 1995], humans have long recognised that computers are far better at performing certain types of task, such as repetitive calculation. However, surely if composition was merely a puzzle or formal problem to be solved then the technical superiority of computers would have long surpassed that of humans.

For this reason alone, the author firmly believes that at least an element of human interaction should always be retained. Algorithmic composition systems are useful as a means of generating musical ideas, but it still requires a great deal of skill and creativity to transform these ideas into a coherent whole. This is not a failing of algorithmic composition systems. Rather it is a reflection of the fact that such systems are just one of several musical tools that are available to the human composer.

However, despite the undoubted benefits of using them, algorithmic composition systems give rise to a number of thorny issues concerning the authorship of the musical compositions they create.

Some of the features raised in the following discussion are briefly touched on in Eduardo Miranda's conference paper *Who Composed Entre l'Absurde et le Mystère* [Miranda, 1997].

In his paper, Dr. Miranda makes the following point:

Entre l'Absurde et le Mystère is a piece for chamber orchestra produced by CAMUS, a computer system designed by the author ([Miranda, 1993], [Miranda, 1994]). CAMUS uses cellular automata-based simulations of biological behaviour to produce sequences of music structures (e.g., melodies, chords, clusters, etc.).

The public warmly applauded its performance by The Chamber Group of Scotland in 1995 in Edinburgh. Martyn Brabbins, the conductor, was reluctant to believe that a computer had generated the piece and generally members of the audience found that the piece was pleasant. The general wonder of that evening was: "Was the piece really composed by a computer?"

This question is debatable and has serious ideological implications. In our point of view, a distinction between author and meta-author should be made in such cases. The ultimate authorship of the composition here should be to the person who designed and/or operated the system. Even in the case of a program that has the ability to program itself, someone is behind the design and/or the operation of the system."

The author would like to expand on this slightly.

Certainly, it cannot be doubted that the authorship of algorithmically-composed works is debatable. For example, does the authorship of such a composition belong with the designer of the system, with the computer that generated the music or with the user of the system who set the composition process in motion? Let us examine the case for each.

The original designer of an algorithmic composition system certainly has a legitimate claim to at least part-ownership of any music that is generated by the system that he or she has designed. Clearly, if the designer had not introduced the system it could not possibly have been used to create any compositions. The situation is very similar to that of the commercial 'game designer' packages that have been popular with home-computer users for a number of years (see [Sherman, 1999] for further details).

Such systems frequently offer a number of tools that allow users to create a 'new' game based on one or more base examples of a particular genre. Using these packages fairly complex computer games can be created with little or no programming knowledge.

The limitations vary from system to system, but some allow for the possibility of stand-alone games to retail standards. When such games are distributed commercially, the designers of the system usually demand a royalty payment as part authors.

However, in the case of musical composition, we have seen that frequently, the design of algorithmic composition systems results in a new musical style. Thus the designer of an algorithmic system can be viewed as the creator of a new musical style. As such, the designer has laid the ground rules that characterise this style, and any subsequent compositions that adhere to it are surely not his own unless actually physically created by him.

Similarly, if a computer is used to compose algorithmically it too has a partial claim to the authorship of the work since it was responsible, perhaps wholly, for the generation of the body of the music. However, we have already satisfied ourselves that computers are used in this way to perform the menial tasks of composition, effectively relegating them to the status of composition tools.

There is a strong analogy here to the working practices of traditional composers and their copyists. In the past, copyists were employed by composers to perform menial composition tasks such as transcribing composition sketches into full score notation. In some cases it is believed that the composer would offer a more experienced copyist little more than a rough outline of a work, which the copyist would then use to create a complete composition. Indeed, this practice continues to this day and echoes Stockhausen's technique of collaborative composition ([Worner, 1973]). In this case, the composer holds seminars in which a carefully selected group of composers discuss ideas and work together to develop thematic material provided by Stockhausen.

Further, since computers are, at least at present, machines without sentience, can any claims of computer authorship be taken seriously? After all, who will fight for the computer's claims to intellectual property?

Finally, the user has, at least in the author's eyes, the strongest claim on authorship. After all, it was the user's decision to compose a piece of music. It was the user who decided on the composition parameters and who instigated the composition process. And if, as is likely, the output from the system was subject to any sort of editing, then the user also transformed the musical sketches generated by the computer into a finished musical work.

To conclude then, we cannot hope to answer conclusively the problem of authorship of an algorithmic composition. Both the system designer and the user of the system have claims. So too does the computer itself, although to a lesser extent.

The author believes that the simplest solution is to consider each composition on its own merits. Clearly if a user has done little more than installed an algorithmic composition system on his computer and used the default settings to create a composition which then undergoes no further editing he can surely have little claim on its authorship. He is as much an automaton as the computer that produced it.

If, however, the user invests considerable time and effort planning a composition; setting the composition parameters, and sculpting and editing the musical output into a complete work then he or she is in a much stronger position to claim authorship.

If the designer and the user are one and the same, then much of what has been said here is redundant. However, in order to clarify the situation for the composition systems described in this thesis, the author hereby waives all rights to any compositions created using his results.

Appendix A

Using the CD

Accompanying this thesis is a dual-format compact disc.

The disc is divided into two partitions, a data section, which is located on track 1, and an audio section which fills the remainder of the disc.

It is extremely important that you *do not* play track 1 on a compact disc player. This contains only computer data, which, if played through loudspeakers may cause irreparable damage. Most domestic compact disc players filter out such data, but please err on the side of caution and skip straight to track 2 when listening to the audio tracks.

The audio tracks can all be played quite safely on a compact disc player.

The entire CD can also be used quite freely in the CD-ROM drive of any PC running Windows 95 or above. The data section may then be read as normal using *Windows Explorer*, whilst the audio section can be played through the computer's soundcard using *Windows CD Player*. Each item of software has its own directory on the CD.

In order to make the installation of software as easy as possible, the author has provided a software menu on the CD which should run automatically when the disc is inserted in your computer's CD-ROM drive³⁴. Once the menu has opened, each item of software may be installed by double clicking on each piece of software's menu entry.

The files may also be installed by executing their installation programs directly from Windows Explorer.

The items of software and their installation programs are listed in Table A1 below.

³⁴ This feature will only work in Windows 95 or Windows 98 provided that the *Autorun* feature is enabled.

Software	Installation Program
CAMUS v2.0	CAMUS/disk1/setup.exe
CAMUS 3D	CAMUS 3D/install.exe
Chaosynth	Chaosynth/setup.exe
Dissonance Network	Dissonance Network/install.exe
GAMusic	GA Music/setup.exe
Musinum	Musinum/setup.exe
The Well Tempered Fractal	WTF/setup.exe

Table A1 – List of items of software and their installation files.

Once the installation file is complete, the software should be available to run from the Windows Start menu.

Below, we provide a track list for the audio section of the CD.

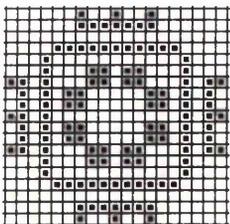
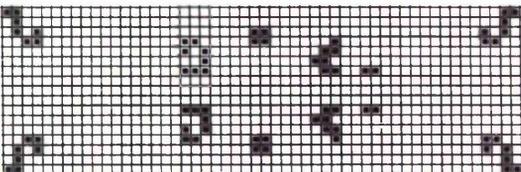
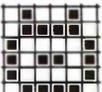
Track Number	Title	Author	Assistance
1	CD-ROM Data	n.a.	n.a.
2	Together in Threes	Kenny McAlpine	CAMUS
3	Sonatina	Kenny McAlpine	CAMUS
4	Minuet and Trio	Kenny McAlpine	CAMUS
5	Calls Across the Ether	Kenny McAlpine	CAMUS
6	Excerpt from <i>Entre l'Asurde et le Mystère</i>	Eduardo Miranda	CAMUS
7	Step by Step by Step	Kenny McAlpine	CAMUS 3D
8	Melancholia	Kenny McAlpine	CAMUS 3D
9	Jazz3	Eduardo Miranda	CAMUS 3D
10 – 14	Selected incidental music from <i>Coeur de Chien</i>	Chris Sansom	Fractal Music ST
15	Twinkle Funk	Kenny McAlpine	Band-in-a-Box
16	Twinkle Rock	Kenny McAlpine	Band-in-a-Box
17	Twinkle Reggae	Kenny McAlpine	Band-in-a-Box
18 - 20	Excerpts from <i>Grain Stream</i>	Eduardo Miranda	CAMUS 3D

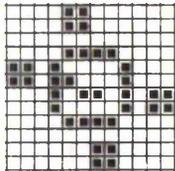
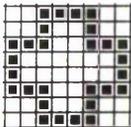
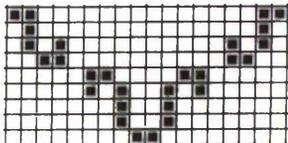
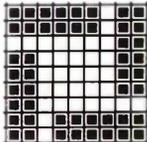
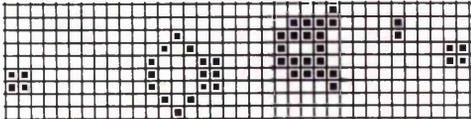
Table A2 – Audio track list.

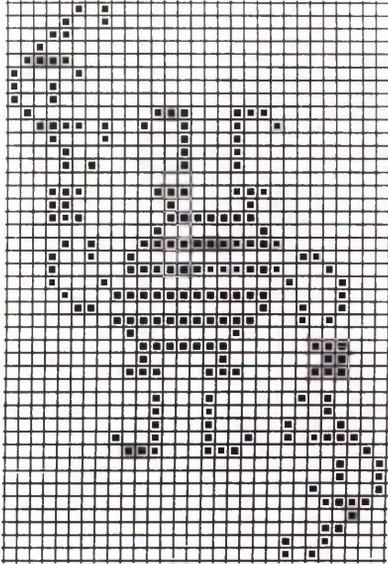
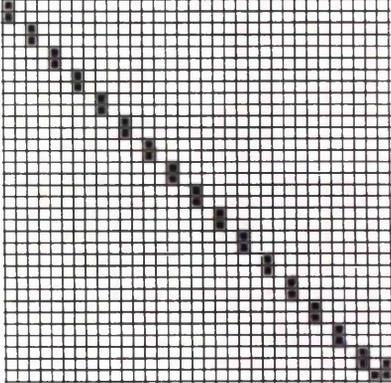
Appendix B

Some common Conway objects

Below we illustrate some common Conway life objects. The list is by no means exhaustive and is provided to assist the reader in the construction of compositions using the Game of Life.

Name	Object
Aircraft Carrier	
Beacon	
Billiard Table	
Bhepto	
Blinker	
Boat	
Bow Tie	
Centinal	
Cheshire Cat	

Clock	
Cross	
Cup	
Eater	
Fence Post	
Galaxy	
Glider	
Glider Gun	
Large Ship	

Max	
Pond	
Pump	
Small Ship	
Snake	
Three Quarter Cross	
Toad	
Zipper	

Appendix C

MIDI

In this section we discuss the MIDI control paradigm. We begin by tracing the early development of the system from the gate/CV system, and showing how it led to a standardisation in communication between musical equipment. We then explain the mechanics of the system, illustrating with connection examples before concluding with a list of commonly used control messages and controllers.

What is MIDI?

MIDI stands for *Musical Instrument Digital Interface*. It is a digital communications protocol that allows the transmission of polyphonic note information on up to sixteen distinct channels using a single cable ([Loy, 1985]). The communication is in one direction only, so if a player wishes an instrument to send and receive MIDI data, it is necessary to connect two cables – one for carrying the data from the MIDI OUT port of the transmitter to the MIDI IN port of the receiver, and one for carrying the data to the MIDI IN port of the transmitter from the MIDI OUT port of the receiver.

At its simplest, MIDI enables musicians to access sounds on a remote module from a master keyboard (see figure C1 below).

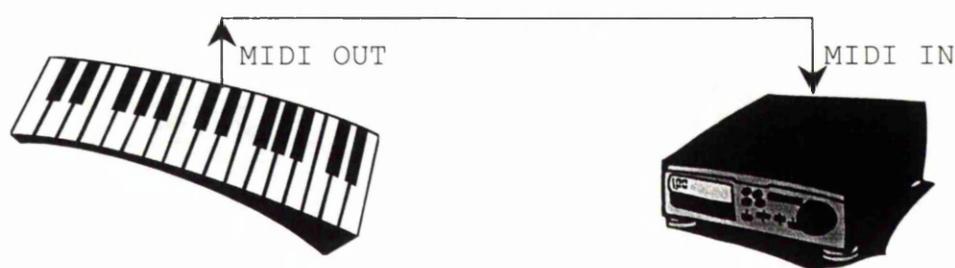


Figure C1 – Simple MIDI connection. The MIDI OUT from the keyboard is connected to the MIDI IN on the sound module, enabling the player to access the module's sounds from the keyboard.

A MIDI connection of the type described in Figure C1 above is quite useful. It allows access to a much broader sound palette and enables the layering of sounds – for example, by selecting a nylon guitar patch from the module and a string patch from

the keyboard one could obtain a warm, sustaining sound with a rich, sharp attack. Automating the layering of sounds in this manner is much easier than attempting to play the melody simultaneously on two separate keyboards.

The history of MIDI

MIDI is not the only musical instrument communications system that is available. Indeed, a control system known as the *gate/CV* system performed exactly this task for many years before MIDI was introduced ([Tucker, 1993]).

The *gate/CV* system was used to connect analogue synthesisers together. It required two cables – one to carry gate information, which was used to switch notes on and off, and one for Control Voltage information, which determined the note that would be triggered.

The main problem with this system is that a single *gate/CV* pair allows only for the monophonic triggering of notes, so if, for example, two-note polyphony is required, four leads (two gate leads and two CV leads) are needed to carry the note information. Similarly, a five-note chord requires ten leads. As the polyphony is increased, the system rapidly becomes unwieldy.

Another problem with this control system is that there was a lack of general standardisation between manufacturers as to how synthesisers responded to the control signals.

In 1982, a consortium of major synthesiser manufacturers, including Roland, Oberheim and Sequential Circuits, proposed a new communications standard, so that any new instrument that adhered to that standard would be compatible with any other. The first MIDI instruments were introduced early in 1983, and the system has subsequently been widely implemented on both professional and domestic musical equipment and computers.

MIDI has undergone several revisions since its introduction, most notably with the introduction of *General MIDI* (GM) in 1990. This was introduced to supply users with a standard setup of 128 preset sounds and two effect types (reverb and chorus),

Hex	Binary	Bytes ³⁵	Command	Range
8X ³⁶	1000xxxx ³⁷	2	Note off	0 – 127
9X	1001xxxx	2	Note on	0 – 127
AX	1010xxxx	2	Polyphonic aftertouch	0 – 127
BX	1011xxxx	2	Control change	0 – 127
CX	1100xxxx	1	Program change	0 – 127
DX	1101xxxx	1	Channel aftertouch	0 – 127
EX	1110xxxx	2	Pitch bend	see below

Table C1 – Some common channel messages

Pitch bend messages are slightly different from other commands. The centre position is set at 2000 (Hex) using a 2-byte 14-bit number (the first two bits of each byte are not part of the data). The bend range is usually set on the instrument itself.

System messages, on the other hand, do not carry any channel information – they apply to the system as a whole. This includes the timing information that is used to synchronise two sequencers³⁸ together. Table C2 below lists some common system messages.

³⁵ The number of bytes following the initial status byte

³⁶ The 'X' here indicates a number between 0 and F (hex) which defines the MIDI channel

³⁷ The 'x' here indicates a binary value.

³⁸ A *sequencer* is a device that stores a sequence of MIDI commands and timing information. The sequencer sends these commands to MIDI instruments at the pre-defined times, in effect telling the instrument how to play the piece of music. Sequencers can be either hardware or software based.

Hex	Binary	Command
FF	11111111	Reset
FE	11111110	Active sensing
FC	11111100	Stop
FB	11111011	Continue
FA	11111010	Start
F0	11110000	System exclusive
F7	11110111	End of exclusive

Table C2 – Some common system messages

An important type of system message is the *system exclusive (SysEx)* message. These are very versatile messages that can be used to set and alter the parameters of specific machines. Because MIDI instruments are so varied, SysEx messages are not defined by the MIDI standard, but are left to manufacturers to implement in any way they wish. Each manufacturer is given a unique identification number that ensures the SysEx messages it creates do not interfere with other equipment.

On receiving a SysEx message, the receiver checks the ID byte that follows. If it recognises this code as its own it will listen to the subsequent data bytes until it receives an End of exclusive message. Otherwise, the message is ignored.

The most common use of SysEx messages is to transfer the data that make up a synthetic sound to an external sound library or editor.

MIDI pitch representation

In the MIDI transmission example of the previous section, the note on message is immediately followed by a data byte that defines the pitch to be note number 60.

Pitches in MIDI are always transmitted as 7-bit³⁹ fields in this manner. This gives $2^7 = 128$ possible pitches that may be addressed by MIDI.

The MIDI specification requires these pitches to be equal-tempered, although it is possible to work around this restriction by, for example, pitch bending notes by the required amount before they are triggered.

The MIDI pitch range runs from note number 0 (C-2 at 8.17 Hz) through to note number 127 (G8 at 12543.89 Hz). Note, however, that MIDI octave numbering is non-standard: middle C is generally taken to be C4 in music theory texts, but is numbered C3 using MIDI octave numbers.

MIDI modes

The way an instrument responds to MIDI messages depends on the MIDI mode that it is operating in. Most often, this will not be a concern – most instruments default to the correct mode when they are turned on – but it is useful to be familiar with the various operating modes in case a MIDI instrument does not respond in the way that it is expected to.

There are four MIDI modes to choose from:

Mode 1 – Omni On/Poly

A MIDI instrument that is operating in Omni mode will ignore any channel information that it receives. In other words, the synth will play all of the information that arrives at the MIDI IN port, regardless of what channel it arrives on.

Poly means that the instrument will respond to notes polyphonically – that is, more than one note at a time.

This mode is not of much use from within a sequencer setup, since all parts will be played simultaneously using the same sound. It is, however, useful when linking two MIDI instruments together with the intention of layering sounds. Rather than matching the output and input channels of the two instruments, all that is required is

³⁹ The MIDI protocol demands that all data bytes start with 0, leaving 7 bits to carry the information.

to set the receiving instrument to MIDI mode 1 and it will play whatever is received at its MIDI IN port.

Mode 2 – Omni On/Mono

This is exactly the same as mode 1, with the exception that the instrument is made to perform as a monophonic instrument, which means that it can only play one note at a time.

This mode is not used extensively, although it can be of use for monophonic synthesisers.

Mode 3 – Omni Off/Poly

This is probably the most useful mode as far as sequencer users are concerned. Instruments in mode 3 will respond to MIDI messages only if they are set to receive data on the track(s) on which the MIDI messages were transmitted, and play notes polyphonically.

Mode 4 – Omni Off/Mono

This is the same as mode 3, except that each channel can only handle one note at a time. Some multi-timbral⁴⁰ synthesisers use this mode, but it is most useful on a MIDI guitar system. Here, the MIDI guitar uses a different MIDI channel for each of the six strings. The receiving synth is set up in mode 4 with the same sound on each channel. This ensures that a player cannot trigger two notes on the same string simultaneously, which would sound unnatural on a guitar part.

The MIDI file format

In order to standardise fully the MIDI format, a file system had to be introduced that would be compatible not only with the instruments themselves, but also with any sequencers or computers that were in the system. Thus, the file format needed to store MIDI data in a way that could be read by different machines.

⁴⁰ A multi-timbral synthesiser is one which can play many different tones simultaneously.

The MS-DOS file format was decided upon, because there was a vast user-base, and because the format was easy to convert to and from, meaning that MS-DOS MIDI files could be read easily by other operating systems (such as MAC OS, AmigaDOS etc.).

There are three main types of standard MIDI file (SMF):

Type 0 MIDI files store all of the MIDI data in one contiguous block, which contains the information for all of the tracks.

Type 1 MIDI files store the MIDI data in a number of discrete blocks, which correspond to the musical tracks.

Type 2 MIDI files store a number of independent sequences or patterns in a single MIDI file. It is similar to a collection of Type 0 sequences stored in a single file and is not as widely supported as either type 0 or type 1 files.

A MIDI file always begins with a header block, also known as the MThd header, because of the four bytes of data that begin the block. An example header block is shown below:

<i>Byte value (Hex)</i>	<i>Description</i>
4D	ASCII code for M
54	ASCII code for T
68	ASCII code for h
64	ASCII code for d
00	The following four values
00	indicate the length of the header.
00	In this case it is
06	6 bytes long.
00	These two bytes indicate that this
00	is a type 0 MIDI file.
00	These two bytes indicate that there
01	is 1 track in the file.
00	These two bytes indicate that the
78	file is set for a 120ppqn ⁴¹ sequencer.

Table C3 – Example of a MIDI header block.

The MIDI header block (after the header and block length information) is always 6 bytes long. It is immediately followed by a track, also known as an MTrk block

⁴¹ The resolution at which a sequencer works is measured in *pulses per quarter note (ppqn)*. This value is the maximum number of clock ticks into which a crotchet can be divided. 120ppqn is a fairly common value, although professional packages work at resolutions of up to 15360 ppqn.

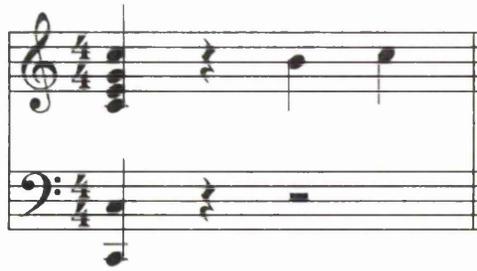
because of the values of the first four bytes. The above MThd header is looking for a single track. This is of the following form:

<i>Byte value (Hex)</i>	<i>Description</i>
4D	ASCII code for M
54	ASCII code for T
72	ASCII code for r
6b	ASCII code for k
00	The values of these four
00	bytes indicate the length
00	of the track.
04	Here, it is 4 bytes.
00	The values of these four
FF	bytes marks the end of
2F	the track. This end of track marker
00	must follow every MTrk header.

Table C4 – Example of a MIDI track block.

In the MIDI file, the above channel commands would be immediately preceded by up to three bytes which define the time that the event is to occur. The MIDI specification uses a technique known as *delta time* – that is, the time specifies the elapsed time since the last event, rather than the absolute time in a composition. Thus simultaneous note events are specified by setting the delta time to 0. The time commands are also stored in variable length format, so that the time is stored as a 1, 2 or 3-byte value.

Figure C2 below shows how a simple one-bar melody translates to a MIDI file.



<i>Byte value (Hex)</i>	<i>Description</i>
4D	ASCII code for M
54	ASCII code for T
68	ASCII code for h
64	ASCII code for d
00	The following four values
00	indicate the length of the header.
00	In this case it is
06	6 bytes long.
00	These two bytes indicate that this
00	is a type 0 MIDI file.
00	These two bytes indicate that there
01	is 1 track in the file.
00	These two bytes indicate that the
78	file is set for a 120ppqn sequencer.
4D	ASCII code for M
54	ASCII code for T

72	ASCII code for r
6b	ASCII code for k
00	The values of these four
00	bytes indicate the length
00	of the track.
44	It is 68 (dec.) bytes long.
00	0 time indicates the start of the track.
90	Note on, channel 1
24	Note number is 36 (dec.), corresponding to C2
7F	Note on velocity is 127 (dec.)
00	0 time indicates that the new event is performed concurrently with previous one.
90	Note on, channel 1
30	Note number is 48 (dec.), corresponding to C3
7F	Note on velocity is 127 (dec.)
00	Next event performed concurrently with the previous one.
90	Note on, channel 1
3C	Note number is 60 (dec.), corresponding to C4
7F	Note on velocity is 127 (dec.)
00	Next event performed concurrently with the previous one.

90 Note on, channel 1

40 Note number is 64 (dec.), corresponding to E4

7F Note on velocity is 127 (dec.)

00 Next event performed concurrently with the previous one.

90 Note on, channel 1

43 Note number is 67 (dec.), corresponding to G4

7F Note on velocity is 127 (dec.)

00 Next event performed concurrently with the previous one.

90 Note on, channel 1

48 Note number is 72 (dec.), corresponding to C5

7F Note on velocity 127 (max.)

78 Next event performed 120 ticks (i.e. one crotchet beat) after the previous one.

80 Note off, channel 1

24 Note number is 36 (dec.), corresponding to C2

7F Note off velocity is 127 (max.)

00 Next event performed concurrently with the previous one.

80 Note off, channel 1

30 Note number is 48 (dec.), corresponding to C3

7F Note off velocity is 127 (max.)

00 Next event performed concurrently with the
previous one.

80 Note off, channel 1

3C Note number is 60 (dec.), corresponding to C4

7F Note off velocity is 127 (max.)

00 Next event performed concurrently with the
previous one.

80 Note off, channel 1

40 Note number is 64 (dec.), corresponding to E4

7F Note off velocity is 127 (max.)

00 Next event performed concurrently with the
previous one.

80 Note off, channel 1

43 Note number is 67 (dec.), corresponding to G4

7F Note off velocity is 127 (max.)

00 Next event performed concurrently with the
previous one.

80 Note off, channel 1

48 Note number is 72 (dec.), corresponding to C5

7F Note velocity is 127 (max.)

78 Next event performed after 120 ticks. Since
the previous event was a note off, this
corresponds to a crotchet rest.

90	Note on, channel 1
47	Note number is 71 (dec.), corresponding to B5
3C	Note on velocity is 60 (dec.)
78	Next event performed after 120 ticks
80	Note off, channel 1
47	Note number is 71 (dec.), corresponding to B5
3C	Note off velocity is 60 (dec.)
00	Next event performed concurrently with the previous one.
90	Note on, channel 1
48	Note number is 72 (dec.), corresponding to C5
3C	Note on velocity is 60 (dec.)
78	Next event is performed after 120 ticks
80	Note off, channel 1
48	Note number is 72 (dec.) corresponding to C5
3C	Note off velocity is 60 (dec.)
<hr/>	
00	The values of these four
FF	bytes marks the end of
2F	the track. This end of track marker
00	must follow every MTrk header.

Figure C2 – One bar of music translated to a MIDI file.

One thing that is not immediately apparent from the above figure is the reason for the note off velocity and indeed many instruments do not respond to this at all. There is, however, good reason for the parameter.

For example, some instruments allow for additional sounds, such as fret noise following a guitar tone) to be triggered when they receive a note off command. The note off velocity determines how loudly (or quietly) this is played.

Connectivity

MIDI enables two or more pieces of equipment equipped with MIDI ports to be connected together and to communicate (see figure C3).

Individual instruments in the system can be addressed using different channels. MIDI channels are distinct data paths along which note information can travel without being affected by any information from other channels. This is a very powerful system, and one in which the power may not be apparent initially.

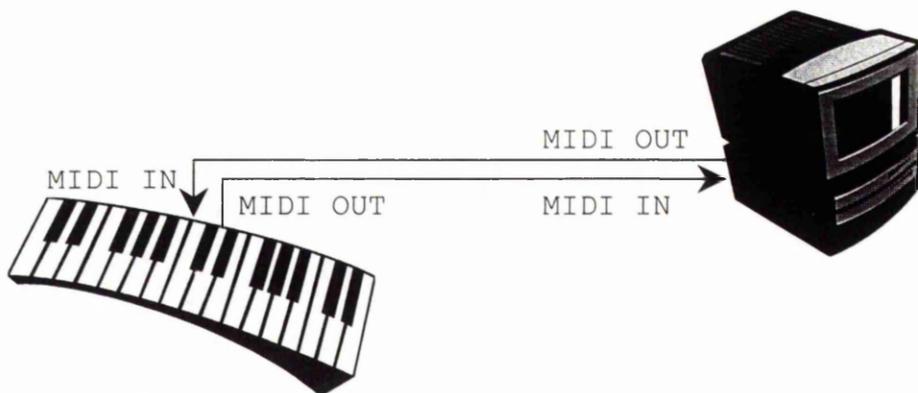


Figure C3 – Simple bi-directional MIDI configuration for a keyboard and a computer running sequencer software.

The configuration of figure C3 is bi-directional. That is, the keyboard can communicate with the PC, and the PC can, in turn, communicate with the keyboard. This allows, for example, a musician to play a keyboard part, record and edit it as MIDI data on the PC and then play it back.

The configuration of figure C1 is said to be uni-directional, because the communication is one-way (from the keyboard to the module).

In addition to MIDI IN and OUT ports, there is a third type of port, known as MIDI THRU. This port monitors the data arriving at the MIDI IN port and re-transmits it, allowing several MIDI instruments to be daisy-chained, or connected in series (see figure C4). Note that it is the rightmost keyboard that is the master in this setup.

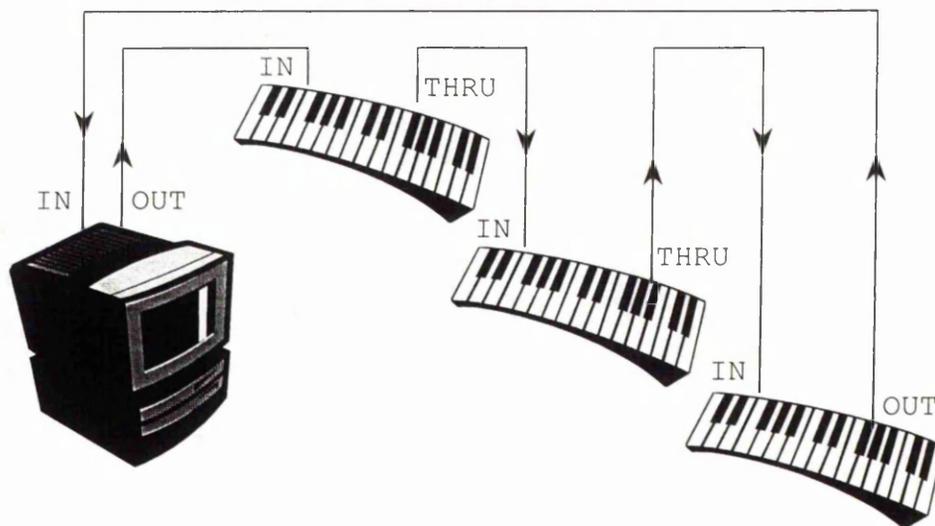


Figure C4 – Several MIDI instruments daisy-chained using THRU connections.

Although theoretically any number of MIDI instruments could be connected together in this manner, there is a practical limit to the number of devices that can be so connected. This is due to slight timing delays that are introduced to the MIDI chain because of the serial transmission of the data. If the total length of the MIDI chain is greater than about 15 metres, there will be a noticeable time gap between the MIDI data reaching the first and the last instruments in the chain, potentially causing synchronisation problems.

Another problem with long MIDI chains (especially if the data are travelling along one long cable) is that the signal can become severely degraded and errors may creep in.

One solution to these problems is to use a multi-port MIDI interface, or a MIDI patchbay to connect the instruments together. These allow the instruments to be permanently wired to their own MIDI port using short cables. Routings can be controlled from the PC, or from the patchbay.

General MIDI and beyond

We have seen that MIDI was initially proposed as a means of standardising musical instrument communications to allow greater interconnectivity between devices. However, although the MIDI instruction set was indeed standardised and widely implemented, the way in which it was implemented often varied from manufacturer to manufacturer.

The main problem was that there was no standard for the relationship between *patch numbers* (the MIDI information bits corresponding to the different instruments) and the order in which preset sounds were actually stored on an external synthesiser. In other words, because there was no standardised order for preset sounds, the same patch number could, and often did, give rise to different sounds on different instruments.

For example, suppose a composer had created a MIDI file that had been configured to use patch number 12 on channel 1. On his own synthesiser this patch might correspond to an electric organ. If he now takes the MIDI file into a recording studio to be played back on a different synthesiser, he may find that patch 12 is a flute, even though this instrument may have a perfectly good electric organ tone at some other patch number. One can appreciate the frustration involved in having to reprogram a MIDI file each time it is required to be played on another instrument.

The General MIDI (GM) specification (see, for example, [Heckroth, 1994a]) was introduced to ensure that end users had a standardised layout of sounds and effects no matter which particular brand of synthesiser they used.

The specification includes a standardised list of General MIDI instruments (see Table C5), which is usually referred to as the *GM Sound Set* ([Heckroth, 1994b]); a standardised list of percussion sounds (see Table C6), referred to as the *GM Percussion Map* ([Heckroth, 1994c]), and a set of performance capabilities, such as total number of available channels and types of MIDI message, which are recognised by the instrument.

The effect of this standardisation is such that standard MIDI file created for use on one GM instrument should play correctly on any other. There will, of course, be

minor differences due to the slightly different sound samples used in the creation of the on-board sounds, but files will not need extensive reprogramming to play back as intended.

Patch	Instrument	Patch	Instrument	Patch	Instrument
1	Acoustic Grand	44	Contrabass	87	Fifths
2	Bright Acoustic Grand	45	Tremolo Strings	88	Bass and Lead
3	Electric Grand	46	Pizzicato Strings	89	New Age
4	Honky-tonk Piano	47	Harp	90	Warm
5	Rhodes Piano	48	Timpani	91	Polysynth
6	Chorus Piano	49	String Ensemble 1	92	Choir
7	Harpichord	50	String Ensemble 2	93	Bowed
8	Clavinet	51	Synth Strings 1	94	Metallic
9	Celesta	52	Synth Strings 2	95	Halo
10	Glockenspiel	53	Choir Aahs	96	Sweep
11	Music Box	54	Voice Oohs	97	Rain
12	Vibraphone	55	Synth Vox	98	Soundtrack
13	Marimba	56	Orchestral Hit	99	Crystal
14	Xylophone	57	Trumpet	100	Atmosphere
15	Tubular Bells	58	Trombone	101	Brightness
16	Dulcimer	59	Tuba	102	Goblins
17	Hammond Organ	60	Muted Trumpet	103	Echoes
18	Percussive Organ	61	French Horn	104	Sci-fi
19	Rock Organ	62	Brass Section	105	Sitar
20	Church Organ	63	Synth Brass 1	106	Banjo
21	Reed Organ	64	Synth Brass 2	107	Shamisen
22	Accordion	65	Soprano Sax	108	Koto
23	Harmonica	66	Alto Sax	109	Kalimba
24	Tango Accordion	67	Tenor Sax	110	Bagpipe
25	Nylon Guitar	68	Baritone Sax	111	Fiddle
26	Steel Guitar	69	Oboe	112	Shanai
27	Jazz Guitar	70	English Horn	113	Tinkle Bell
28	Clean Guitar	71	Bassoon	114	Agogo
29	Muted Guitar	72	Clarinet	115	Steel Drums
30	Overdrive Guitar	73	Piccolo	116	Woodblock
31	Distorted Guitar	74	Flute	117	Taiko
32	Guitar harmonics	75	Recorder	118	Melodic Tom
33	Acoustic Bass	76	Pan Flute	119	Synth Drum
34	Fingered Bass	77	Bottle Blow	120	Reverse Cymbal
35	Picked Bass	78	Shakuhachi	121	Guitar Fret Noise
36	Fretless Bass	79	Whistle	122	Breath Noise
37	Slap Bass 1	80	Ocarina	123	Seashore
38	Slap Bass 2	81	Square Lead	124	Bird Tweet
39	Synth Bass 1	82	Sawtooth Lead	125	Telephone Ring
40	Synth Bass 2	83	Calliope Lead	126	Helicopter
41	Violin	84	Chiff Lead	127	Applause
42	Viola	85	Charang	128	Gunshot
43	Cello	86	Voice		

Table C5 – The GM Sound Set.

Note Number	Drum Sound	Note Number	Drum Sound
35	Bass Drum 1	59	Ride Cymbal 2
36	Bass Drum 2	60	Hi Bongo
37	Side Stick	61	Low Bongo
38	Acoustic Snare	62	Mute Hi Conga
39	Hand Clap	63	Open Hi Conga
40	Electric Snare	64	Low Conga
41	Low Floor Tom	65	High Timbale
42	Closed Hi-Hat	66	Low Timbale
43	High Floor Tom	67	High Agogo
44	Pedal Hi-Hat	68	Low Agogo
45	Low Tom	69	Cabasa
46	Open Hi-Hat	70	Maracas
47	Low Mid Tom	71	Short Whistle
48	Hi Mid Tom	72	Long Whistle
49	Crash Cymbal 1	73	Short Guiro
50	High Tom	74	Long Guiro
51	Ride Cymbal 1	75	Claves
52	Chinese Cymbal	76	Hi Wood Block
53	Ride Bell	77	Low Wood Block
54	Tambourine	78	Mute Cuica
55	Splash Cymbal	79	Open Cuica
56	Cowbell	80	Mute Triangle
57	Crash Cymbal 2	81	Open Triangle
58	Vibraslap		

Table C6 – The GM Percussion Set.

Although General MIDI brought a great improvement in file sharing, it has been accused of being restrictive because it offers only a limited sound palette. In response to this, Roland introduced the GS standard ([Heckroth, 1994d]).

The Roland GS standard offers a superset of sounds and functionality. All the GM sounds are present, but GS also offers a number of additional variations of these sounds. Also included are basic reverberation and chorus effects, the depth of which are adjustable on each of the sixteen MIDI channels.

Yamaha also offered its own extension to GM, known as XG ([Yamaha, 1999]). As with GS, XG is a superset of GM and offers even more sounds and effects than Roland's GS. It has also been implemented in some higher specification computer soundcards, such as Yamaha's own SW1000GX (see [Walker, 1998]).

Appendix D

An alternative approach to algorithmic composition

The algorithmic composition techniques described elsewhere in this thesis represent just one approach to the problem of coaxing musical compositions from a machine. A number of alternative approaches exist.

Some broadly follow the same empirical approach as this research, whilst others attempt to work replicatively to compose in existing musical styles. Below we present a brief review of several of these algorithmic packages.

Please note that this list is by no means exhaustive. We merely intend to suggest to the reader that a number of alternative approaches exist and compare them with our own system.

Chaosynth

Strictly speaking, Chaosynth ([Miranda, 1993]) is not an algorithmic composition package. It was designed by Eduardo Miranda and coded for Windows by Joe Wright of NYR Sound Ltd, and is a software synthesiser that uses cellular automata to generate *granular sounds*⁴².

However, as we shall see, it is possible, through the manipulation of the system parameters, to use the system as a rudimentary melody generator. In addition, the system's reliance on cellular automata for the generation of sounds has particular relevance to our own work.

Chaosynth uses a different cellular automaton to drive the process of sound generation from that of CAMUS. The automaton is known as ChaOs, an n -state automaton that was designed to model a particular neurophysiological phenomenon.

⁴² Granular sounds are composed of many short sonic quanta, which, when played successively form a single complex tone. For further details about granular synthesis, please see [Roads, 1978], [Roads, 1988], [Roads 1996], [Jones & Parks, 1988] and [Truax, 1988].

The cells in the automaton model nerve cells in the brain. The states of these cells are represented by a number between 0 and $n - 1$.

Those cells in state 0 corresponds to a *quiescent* state, whilst a cell in state $n - 1$ corresponds to a *burned* state. All states in between exhibit a degree of *depolarisation*, corresponding to the number of their state. The closer a nerve cell's state number gets to $n - 1$ the more depolarised it becomes.

A transition function, F , is defined by the following three rules, which are selected according to the state of the cell currently under examination.

A quiescent cell may or may not become depolarised at the next tick of the clock. This depends upon the number of polarised and burned cells in its neighbourhood, and the cell's *resistance to being burned*.

A depolarised cell has the tendency to become more depolarised as time progresses. Its state at the next timestep depends on the *capacitance* of the nerve cell and the degree of polarisation of its neighbourhood. The degree of polarisation of the neighbourhood is defined as the sum of the states of the cell's 8 neighbours divided by the number of polarised neighbours.

A burned cell generates a new quiescent cell at the next timestep.

Each sonic particle produced by *Chaosynth* is composed of several *partials*⁴³ – a sine wave produced by an oscillator. Each oscillator requires three parameters to function: the frequency in Hertz; the amplitude in decibels, and the duration in milliseconds of the sinewave.

Chaosynth uses the ChaOs automaton to control the frequency and duration values of each particle. The amplitude values are configured by the user.

The states of each cell in the automaton are associated to a frequency value. In addition, each of the oscillators is associated to a number of cells. The frequency values of the partials at time t are established by calculating the arithmetic mean of the

⁴³ In general, a partial is an arbitrary frequency component in a sound's harmonic spectrum [Roads, 1996].

frequencies associated with the states of the cells of the oscillators. Each particle is then the result of the *additive synthesis* (see [Roads, 1996]) of the component sine waves.

The duration of the entire sound event is determined by the number of iterations of the algorithm and the duration of the sound particles. For example, 50 iterations of the process using particles of 10 milliseconds in length results in a sound event of duration 0.5 seconds.

This mapping method described above can be thought of as modelling the physical properties of some acoustic instruments – the random initialisation of the automaton produces wide distribution of frequency values, which tend to settle to an oscillatory cycle. In many acoustic instruments, particularly those that are plucked or hit, the sound begins with a burst of frequency-dense noise before the higher partials die off and the sound settles into a semi-oscillatory cycle ([Roads, 1996]).

On first executing the program, the user is presented with the main screen, presented in Figure D1 below.

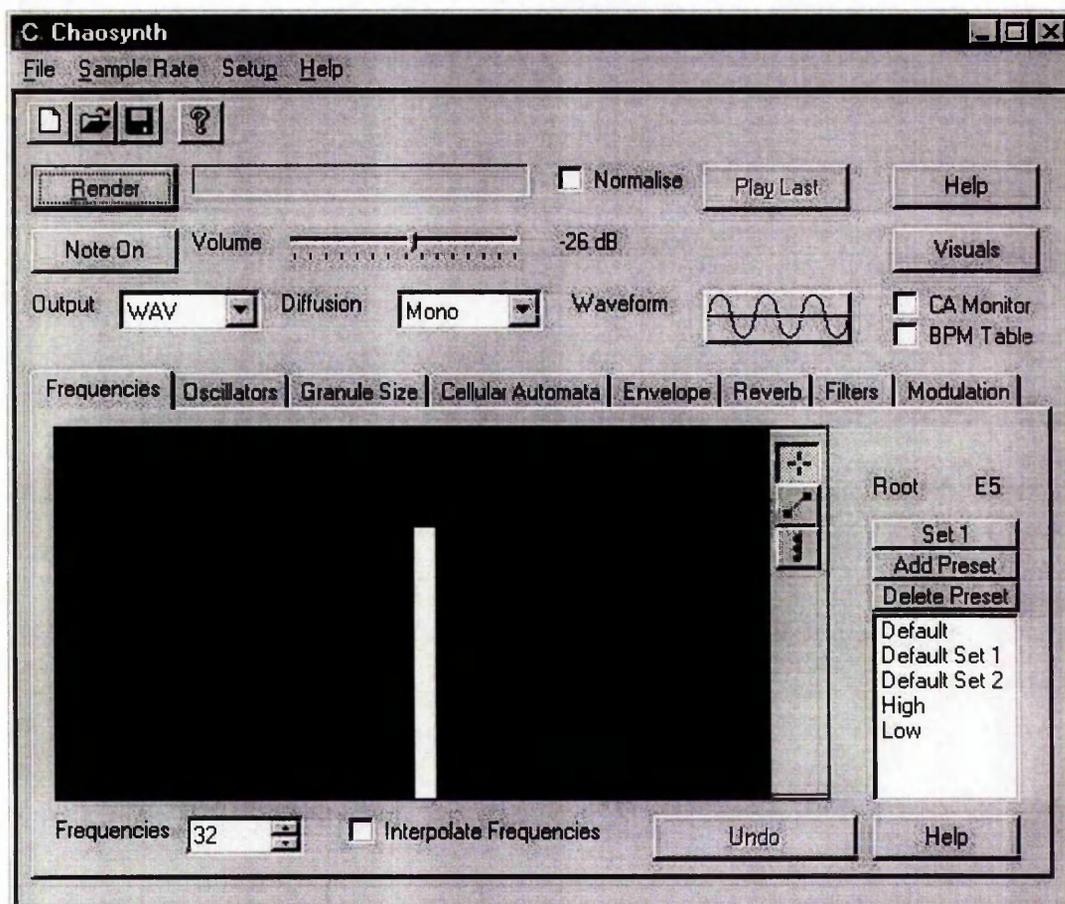


Figure D1 – The main Chaosynth window.

At the top of the window lie the options relevant to the actual creation of sounds. These allow the user to audition a sound (either by clicking on the *Note On* button or by playing a note on a MIDI keyboard); create a sound file and matching video clip of the automaton using the *Render* button, as well as altering the overall volume and makeup of the sound.

Along the bottom is a tabbed window, which allows the user to alter all parameters, detailed description of which we omit. The interested reader is, however, free to examine the help files included with the software for the relevant information.

We do, however, illustrate the usage of the system as a melody generator.

In order to use Chaosynth in this way, the user must first alter the size of the individual granules. This may be achieved by clicking on the *Granule Size* tab.

The Granule Size tab allows the user to control the total number of and the size of each of the granules over the duration of the generated sound.

The size of each of the granules over time is controlled by a graph, illustrated in Figure D2 below. The graph consists of a piecewise linear curve from which the length of each granule, given by the vertical axis, may be calculated as time progresses along the horizontal axis.

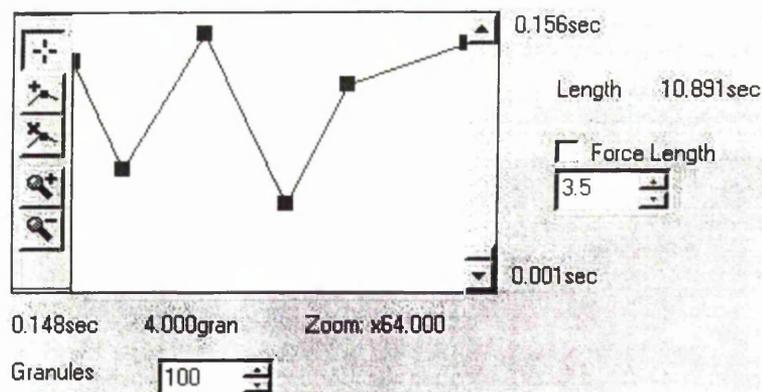


Figure D2 – The Granule Size graph.

In general, long granule lengths result in a more melodic sound, whilst shorter sizes produce ‘bubbling’ noises. Thus, in order to generate a sound which is in any way melodic, the granule lengths must be sufficiently large to be distinguishable as individual tones.

In Figure D2 above, the granules range in size from 0.05 seconds to 0.148 seconds. The sound consists of 100 granules, giving a total length of 10.891 seconds, as illustrated in the top right-hand corner of the screen. The leftmost node is positioned at time $t = 0$, and the rightmost at time $t = 10.891$ seconds.

Now, the user must choose the frequencies that will provide the notes of the melody. This is achieved by clicking on the *Frequencies* tab and is illustrated in Figure D1.

As a sound is generated in Chaosynth, the harmonic spectrum is taken from configuration of this tab. For any given granule, a subset of this spectrum will be used, as determined by the ChaOS cellular automaton.

Each bar corresponds to a different note ranging from C-2 to B8, and up to 64 different pitches may be specified. Now, since we are using Chaosynth as a melody generator, the harmonic spectrum defined by the bars is the set of pitches from which our melody will be constructed. Clearly, defining a greater number of pitches will result in a more diverse melody, but the chances of any individual pitch sounding at any time will be reduced.

Finally, the user configures the system to produce notes of varying amplitude using the *Oscillators* tab (see Figure D3).



Figure D3 – The Oscillators graph.

This is almost identical to the method used by the system for the input of note values. Here, however, the values in the bar graph refer to the amplitude of up to 64 oscillators, each of which can produce any of the pitches defined in the frequency spectrum. More oscillators increase both the volume and the complexity of the sound, whilst the height of each bar determines the volume of the corresponding oscillator in decibels.

The melody may then be auditioned by clicking on the *Note On* button or by playing a note on a MIDI keyboard. If the settings are satisfactory the finished sound is created by clicking on the *Render* button.

In conclusion, as a software synthesiser, Chaosynth is excellent. Not only does it offer the user a chance to experiment with a new and unusual form of synthesis, it allows

for the creation of organic sonic textures that would have been difficult, if not impossible to create using other methods.

Generally speaking, however, the melodies created by the system are quite different in character from those with which the user may be familiar, even those 'alien' melodies generated by other algorithmic techniques. With their 'bubbling' character they are similar to the simplistic attempts to sonify the Mandelbrot set by equating the number of iterations of a colour band to a particular pitch, such as was offered as an interesting aside by software such as Vista Pro.

This is not intended as a criticism of the software, since its functionality as a melody generator is also incidental to its intended role, which is as a software granular synthesiser – a role which it performs admirably. However, it is important to note this fact, because it does limit the system's usefulness as a melody generator.

Another hindrance to the system's use in this capacity is that it offers few composition tools and lacks the ability to export musical data in MIDI format. Thus all editing must be performed on sound files, which is not really a viable method for creating complex musical compositions in the traditional sense, although it undoubtedly works well in an electroacoustic setting, which is, after all, the particular area for which this product is intended.

Musinum

Musinum, the music in the numbers ([Kindermann, 1995]) is a formalised melody generator designed and implemented by Lars Kindermann.

At the heart of the program lies a very simple mapping which takes as input the value obtained from the following formula:

$$\text{Current note} = ((\text{counter} \mathbf{div} \text{ speed}) * \text{step}) + \text{start}.$$

In the above formula, *counter* is a computer-controlled counter; *speed* is a user-defined parameter that controls the rate at which *counter* changes; *step* is a user-defined parameter that controls the amount by which the counter increases, and *start* is a user-defined offset.

In order to map the resulting integer to a musical note it is first expressed in binary notation and the number of 1s counted. By default the pitch is chosen according to the following table:

Number of 1s	Pitch
1	c
2	d
3	e
4	f
5	g
6	a
7	b

Table D1 – The mapping from binary numbers to pitch used by Musinum.

The mapping can, however, be altered so that the resulting pitches belong to one of several scale types, including major, minor, twelve-tone and pentatonic.

Consider as an example the following:

Decimal	Binary	Number of 1s	Resulting pitch
1	1	1	c
2	10	1	c
3	11	2	d
4	100	1	c
5	101	2	d
6	110	2	d
7	111	3	e
8	1000	1	c
9	1001	2	d
10	1010	2	d
11	1011	3	e
12	1100	2	d
13	1101	3	e
14	1110	3	e
15	1111	4	f
16	10000	1	c
...

Table D2 – Examples of the mapping employed by Musinum.

The mapping can be further extended by employing a variety of arithmetic operations to the integer *Current note*.

The software interface is clear and relatively straightforward. It should certainly present no problems to an experienced operator of the Windows system. The main display is presented in Figure D4 below.

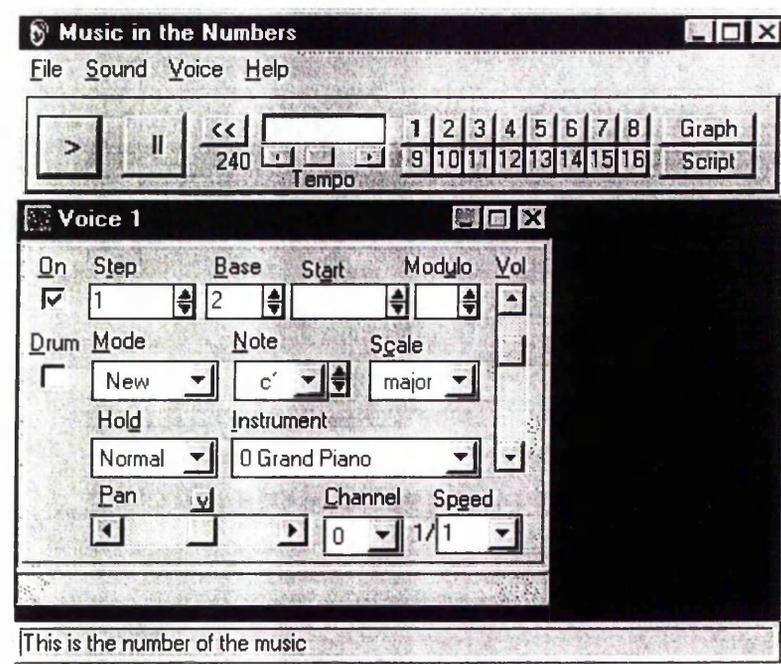


Figure D4 – The main operating window of Musinum.

At the top-left of the main window are three buttons, which are respectively used to begin and pause the composition process, and to reset the counter, which is positioned immediately beside the rightmost of the three buttons.

Below the counter is a slider that is used to control tempo. The indicated range is from 60 to 600 beats per minute, although the author has discovered that the actual tempo is half as fast as that which is indicated.

Towards the top-right of the main window are two more buttons labelled *Graph* and *Script*.

The Graph button displays a graphical display (see Figure D5) of the generated music in a format which closely resembles that of the *Key Edit* window used by Cubase and other software sequencers.

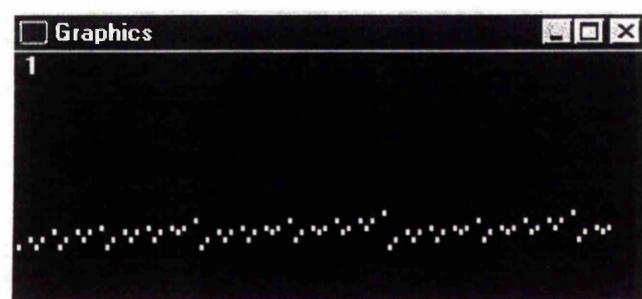


Figure D5 – The Musinum Graphics dialog.

The Script button displays the Script dialog box (see Figure D6), which allows the user to specify a list of counter values. By clicking on these counter values the user can skip to that counter value – effectively creating loops in the music.

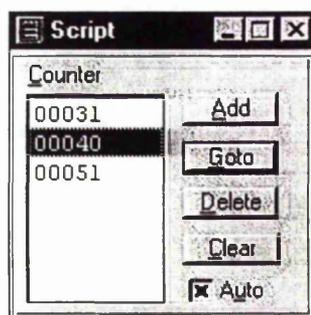


Figure D6 – The Musinum Script Dialog.

The box titled *Voice 1* in the main window corresponds to the first of the 16 available voices. These are selected by clicking on the numbers 1 – 16 in the top-right of the main window.

Voices can be activated and silenced by clicking the checkbox labelled *On* in the top-left of the box. The default setting is mono-timbral note generation on Voice 1 only.

The *Step* and *Start* parameters relate to the values that are substituted into the note generation function described above. In addition, the *Base* and *Modulo* parameters are

applied to the value *Current Note* after the calculation is complete: *Current Note* is taken modulo *Modulo* before being converted to base *Base*.

The *Mode* box allows the user to select between one of five rhythm modes, each of which creates rhythms depending on the previous pitch that was generated. For example, the default setting, *New*, plays a new note only if the result of the formula has changed: two or more successive equal notes are combined to form one long note.

The *Note* box allows the user to specify the root note which is played when the value of *Current Note* has a single '1' digit. The scale onto which the notes are mapped is selected using the *Scale* box.

The *Hold* option allows the user to control the way in which notes are sounded. The options are *Normal*, which is the default setting; *Pedal*, which mimics the effect of pressing the *sostenuto* pedal on a piano, and *staccato*, which performs notes of short duration.

The remainder of the parameters are concerned with the MIDI settings of the instrument that performs the music.

Setting the system in motion with the default settings results in the following:



Figure D7 – 12 bars of music generated using Musinum's default settings.

The result of this process, as can be seen from Figure D7, is reasonably pleasant, although quite repetitive. This could quickly lead to listener boredom, although in its favour the system does offer rudimentary facilities for imposing common musical devices such as repetition (through the use of scripts) and transposition (through changing the *Note* and *Scale* settings).

KeyKit

KeyKit, developed by Tim Thompson, is a music-specific programming language with a graphical user interface ([Thompson, 1996]). The system was designed specifically for manipulating MIDI data. As such, it is perhaps best regarded as a composition environment rather than an algorithmic composition system.

The full KeyKit system is similar in notion to the transformer module proposed in Section 6.2.1, and can be used for both algorithmic and realtime composition.

At the heart of the system lies the KeyKit language, which is both object-oriented and multi-tasking. The latter is particularly useful, since it allows the composer to apply several different functions or work with several different parts concurrently.

The system is fully active at all times, meaning that it will respond to MIDI input without being told explicitly when it will receive such information. This is particularly useful when working with the system in a realtime situation, since it means that if inspiration strikes at an awkward moment – as it so often does! – the computer will capture the performance, which might otherwise have been lost had the composer had to prime the system.

When the system is first loaded the user is presented with the main screen, which is illustrated in Figure D8 below.

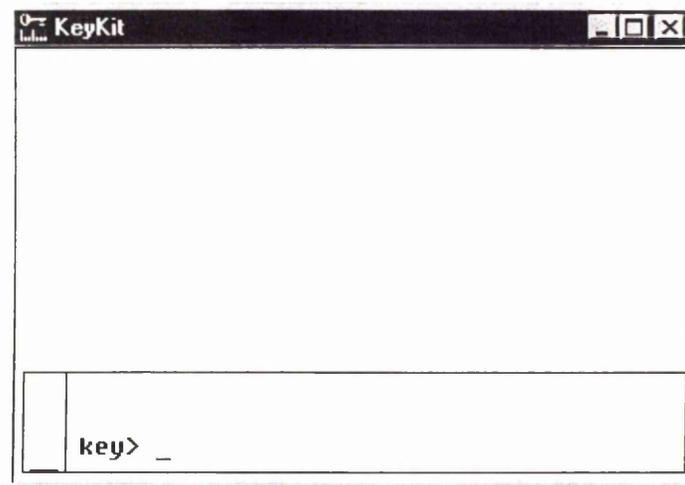


Figure D8 – The main KeyKit window.

The prompt at the bottom of the screen is the KeyKit Console, and contains an interpreter that reads and executes KeyKit statements.

A complete graphical user interface is implemented in the user-defined library of KeyKit. The graphical interface is configured by the user. Elements may be added and removed by means of the KeyKit menu (see Figure D9), which is accessed by left-clicking with the mouse in the blank area of the window.

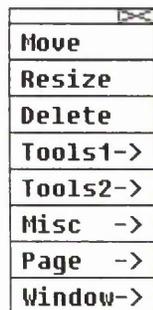


Figure D9 – The KeyKit menu.

Each element of the interface is known as a *tool*, and may be accessed from one of the two Tools options on the menu.

The largest tool in the user interface is a full-featured multi-track sequencer. However, because of the open nature of the system and the large number of tools

available, we cannot offer a complete definition of the package. Instead, we outline four of the tools the author found particularly useful.

The *Blocks* tool, which allows the user to manipulate a linear sequence of musical phrases, which are referred to as *blocks*.

The *Chord Palette* allows the user to play a selection of chords using the mouse. By left-clicking in a given cell, the user can play chords via MIDI. The rows of the matrix are labelled with the chord types, such as major and minor, and the columns are labelled with the chord keys.

The *Controller* tool allows the user to edit and send MIDI controller messages, such as volume and pan. The tool consists of 16 sliders, each of which relates to one of the 16 available MIDI channels. In addition, a pull-down menu allows the user to select the type of controller message that is sent when the slider values are altered.

The *Markov Maker* tool is perhaps the most noteworthy of all the tools available in KeyKit. This allows the user to generate music algorithmically using Markov Chain techniques. The actual implementation of the Markov Chain is quite different to that of CAMUS 3D's. Here it is used to *replicate* existing music rather than *generate* as with CAMUS. The top half of the Markov Maker window displays an existing piece of music that is input to the system. KeyKit then analyses the MIDI data and creates a state-transition matrix which corresponds to this piece of music. Finally, the system uses the Markov Chain to create a 'similar' piece of music which is displayed in the bottom half of the window.

Figure D10 shows the above four tools in the KeyKit window.

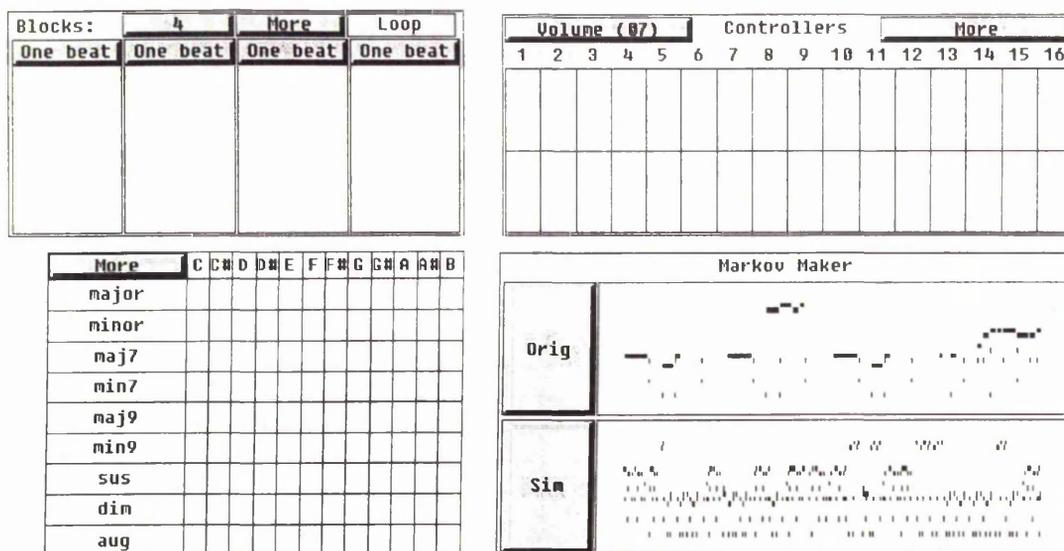


Figure D10 – The Blocks, Chord Palette, Controller and Markov Maker tools.

The results obtainable from the Markov Maker tool are good, although the compositions generally lack the musical structure required to make them truly convincing. The reason for this is almost certainly because the routine effectively models the statistical properties of the input data, but not the musical properties. Thus, although the new composition has similar statistical properties, it wanders aimlessly without having any particular direction.

As we saw earlier, this too is a limitation of CAMUS 3D. KeyKit, however, is more strongly equipped to counter this problem than our own system, since it is equipped with many tools for manipulating MIDI data.

Indeed, because the system operates on MIDI data, it can be used as an external transformer module of the type described in Section 6.2.1. Therefore the author strongly recommends KeyKit as a complementary package to CAMUS 3D. When used in tandem, with CAMUS 3D as a thematic generator and KeyKit as a well-specified arranger, the results can be quite powerful. Melancholia, presented on track x of the accompanying CD was composed in this way.

Unfortunately, due to licensing restrictions, it was not possible to include KeyKit on the accompanying CD-ROM. However, the software is available for download from <http://www.nosuch.com/keykit>.

The Well-Tempered Fractal

The Well-Tempered Fractal ([Greenhouse, 1993]) is an algorithmic composition package that utilises fractals to generate compositions.

In terms of conception this package is undoubtedly the closest to CAMUS 3D. Robert Greenhouse conceived of the software after several years of experiments in generating fractally derived sounds. This led him to the following conclusions ([Greenhouse, 1993]):

- i.) Few people are willing to devote a substantial amount of time to listening to the music generated by the tens or hundreds of thousands of points needed to construct a fractal image. Generally speaking, a fractal image contains far more information than in a typical musical composition if we restrict data to pitch and duration. Thus, there is a degree of redundancy built into the system.
- ii.) Listeners like some degree of comprehensible order – music which is devoid of recognisable pattern quickly becomes uninteresting to most listeners simply because they fail to understand it.
- iii.) Listeners do not like music which wildly jumps around or which is too predictably correlated, even if there are recognisable patterns present. What is needed a careful blend between the predictable and the unpredictable.

These design characteristics are shared to a great extent with CAMUS 3D, which was designed with the sole intention of producing pleasant music which is accessible to the majority of listeners.

Indeed, the similarities extend further to the philosophy of the system. Greenhouse notes:

“Foremost in my mind remains the principle that in the end the composer must retain artistic control over what he or she puts down on paper. Without this principle it is hard for me to accept what the program produces as art.

With WTF it is possible to produce a new full length composition in score notation in about 10 minutes, assuming that the appropriate sequencing and notation software is

available. Possible, that is, if the user does nothing but convert WTF output into musical scores.

That means a total of about 48 new compositions in an average 8 hour day! It would not take long to overtake Telemann as the most prolific composer in history going at that rate. But the quality of the output would be as good as one could expect with only ten minutes work involved.

I have spent as much as perhaps 10 - 12 hours manipulating the data of a single fractal into a final composition before I was satisfied with the result. In the end the music was a composition which I could *probably never have imagined* without the aid of a computer, but which the computer could never have produced in such an elaborated form.”

When the program is first opened, the user is presented with the main screen (see Figure D11).

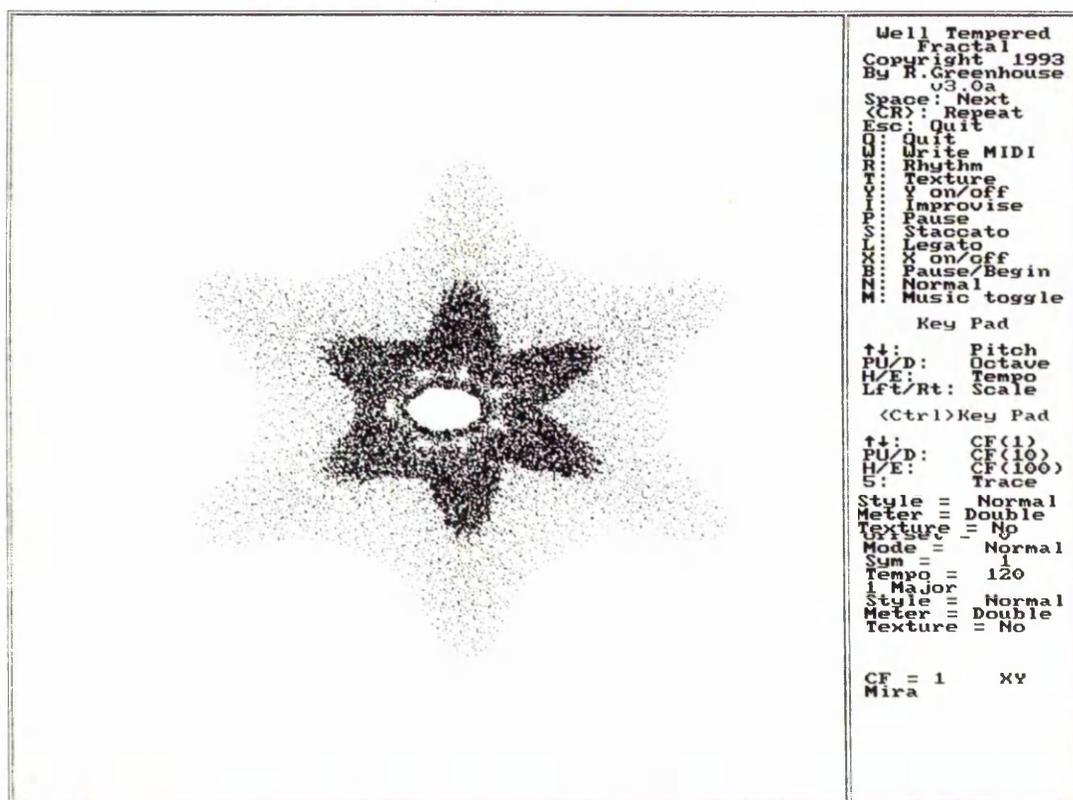


Figure D11 – The Well-Tempered Fractal screen.

The user can choose a fractal type with the function keys F1 - F10. The coefficients for each function are randomised whenever they are selected to give different orbits each time.

The orbit points of the fractals are mapped to one of 21 scales, which include the major; aeolian; dorian; whole tone; diminished, and phrygian, and whose name is displayed in the lower hand corner of the screen. The user may select alternative mappings by using the left and right arrow keys.

The system employs "selective data extraction" in order to reduce the amount of data stored in the fractal image to an amount more suitable for a musical composition. To control this, the user defines a *compression factor*, which is the number of data points skipped by the mapping on each iteration. In other words, a compression factor of j means that only every j th data point will be converted to music.

By pressing the 'M' user may audition the music that is generated in real time. The user begins writing a MIDI file, the maximum size of which is 100,032 bytes, by pressing 'W'.

The system also allows for the alteration of a number of other parameters, perhaps the most notable of which is *Improvise*, which allows the system to elaborate on the basic musical material that is generated. The function is fairly rudimentary, but is a welcome addition and helps to improve the system's overall usefulness.

The music generated by the system is monophonic, which means that a considerable amount of work must be done by the user in order to create a finished composition. This is not necessarily a bad thing, particularly when we consider Greenhouse's comments above. However, the monophonic output does mean that it is difficult to visualise the fractal music in a fuller context.

On an operating level, the program runs under DOS, and as a result, the interface arguably suffers to an extent. To be fair, the system is still very simple to use, and has a rudimentary graphical interface, but the DOS front-end gives the software the look and feel of a much older program, and the author believes this acts detrimentally.

In addition, the software assigns filenames and paths (such as the MIDI output) automatically, which can make it difficult for the new user to find output files.

Band-in-a-Box

PG Music's *Band-in-a-Box* is different from all other systems discussed here in its approach to algorithmic composition.

The system is designed to compose *replicatively*, that is to produce pastiche in a variety of preset and user-definable musical styles, and appears to be built around a knowledge-base of such styles⁴⁴. The software is intended primarily as a means of auto-accompaniment, and thus does not generate melodies.

Upon launching *Band-in-a-Box*, the user is presented with the main composition screen (see Figure D11) from which all the functionality of the system is accessed. Along the top of the screen lies a series of buttons which relate to the instrumentation of the generated music. Below this is a two-tier keyboard, the top tier of which illustrates the melody of the composition in a manner similar to the keys of a player-piano. The other tier similarly illustrates the accompaniment. Next is a set of transport buttons which enable the user to record and playback music. The area in blue below this is a set of composition-specific parameters that relate to the stylistic content of the composition. Finally, lies the musical notepad, where information about the harmonic structure of the piece is entered and displayed.

⁴⁴ *Band-in-a-Box* is a commercial release, and so detailed information about the composition algorithm that lies at its heart is not publicly available. The above discussion is based on the author's experience of the software.

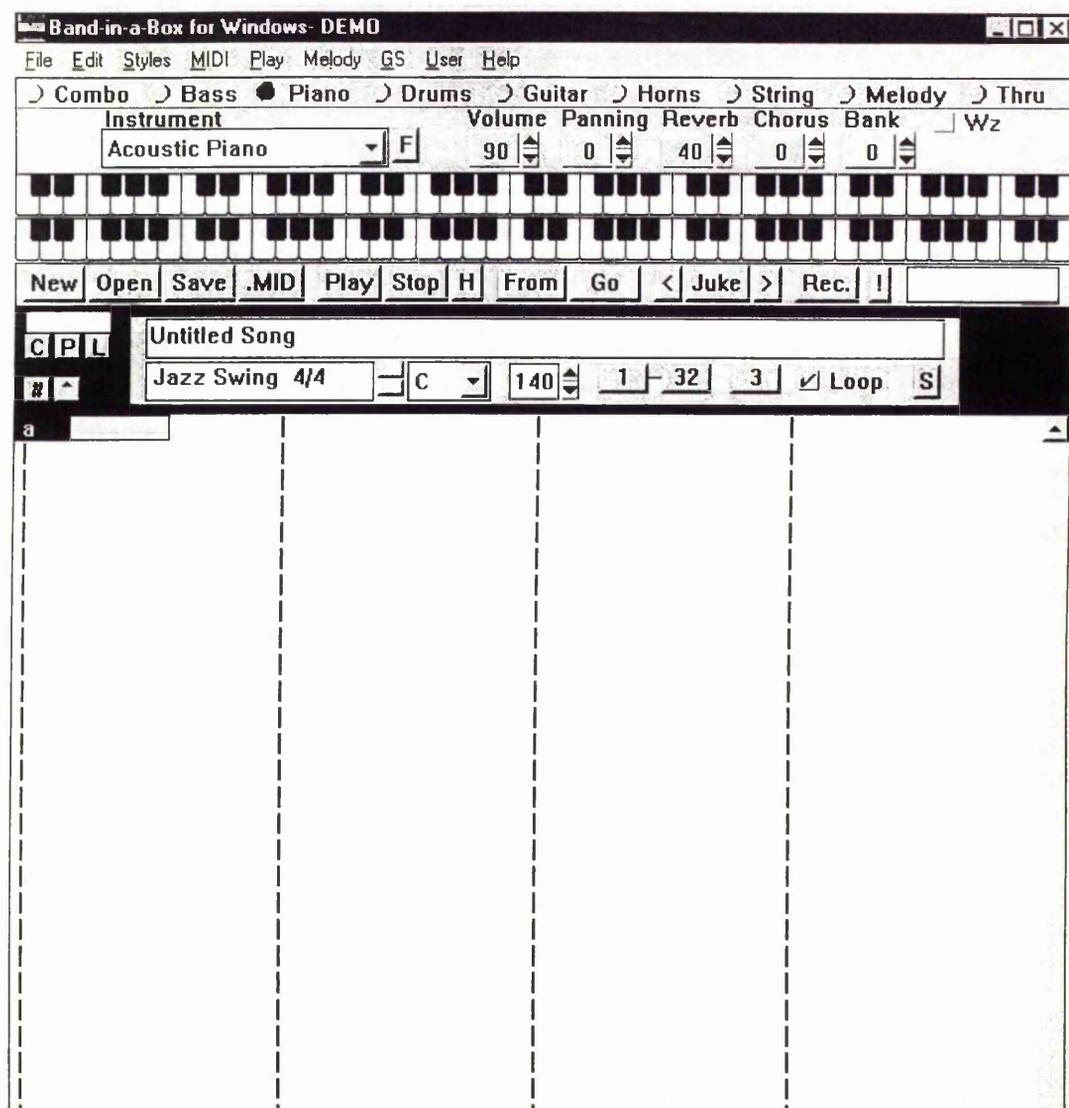


Figure D11 – The main Band-in-a-Box screen.

The interface employed by Band-in-a-Box is excellent and extremely intuitive to use. It is essentially a blend of software sequencer, similar in essence to Cubase, for example, and a musical notepad. Thus, anyone who has experience of music software should immediately feel quite comfortable using the software, and the use of standard chord notation to enter the structure of songs, as compositions within the software are termed, means that a detailed knowledge of musical score is not required to create complex musical pieces. The software has the ability to recognise a wide range of common chords.

Band-in-a-Box then uses the chord chart to create an arrangement of up to 7 parts and depends on the chosen musical style, key and instrumentation. Several styles, which

are mainly geared towards contemporary popular music, jazz and blues are included with the software, as are facilities to edit existing styles and create new ones.

In order to create a new song, the user must first select, or create a musical style. This is done via the Styles menu.

Once a style has been selected for the song, the composer must decide on the chord structure. This is done by clicking on the notepad area of the screen and entering the chords in standard notation. Each box in the notepad corresponds to a complete bar of the music.

With the outline of the music in place, all that remains is for the composer to select an instrumentation and set the process in motion. This will generate a complete backing track using the chord structure and instrumentation that the composer has defined. Additional tools exist to increase the complexity of the music, such as the ability to add lyrics and annotations, and also to specify markers in the song to which the system will respond by subtly varying the accompaniment.

In order to complete the process the composer may wish to record a melody line to be played over the top of the accompaniment. This may be done in real or step time.

Generally speaking, the results obtainable from Band-in-a-Box are excellent, and although it is unlikely that a listener would be fooled into believing that a live band was responsible, convincing music can be created quickly and simply. Indeed, were the system to be used to create printed parts for a live band, it would be difficult to be sure that the piece had not been written in its entirety by a human composer.

The programming of the stylistic features in particular is of an exceptionally high standard. Using this package, it is possible for a composer, armed only with a little knowledge of instrumentation and arranging, to produce a fairly convincing pastiche of almost any type of music.

In addition, because the system offers extensive facilities for stylistic editing, there is plenty of scope for experimenting with hybrid styles, performing existing music in different styles and creating music with a similar stylistic content to that of, for example, CAMUS. Tracks 15 - 17 on the CD were created using a jazz harmonisation

of Mozart's *Ah, Vous Dirais Je Maman!* (Twinkle, Twinkle Little Star), and illustrate some of the stylistic features of the system.

However, the system is not perfect. One problem in particular is that all the given musical styles are fairly pedestrian, and the music tends to lack depth as a result. Of course, this can be overcome by defining one's own styles, but in general, the system responds to similar input in similar ways – rarely, if ever, does the system produce the 'happy accidents' of music that live musicians frequently chance upon when jamming together.

Similarly, although the system can compose music in a number of different styles, the music is still geared towards the small band. It would be very difficult to create an invention in the style of Bach using the system, unless one wished to work à la Emerson, Lake and Palmer!

Overall, though, this type of package is invaluable to composers who must work to tight deadlines and who have in mind a particular style. The system is undoubtedly rooted firmly in jazz music, but is flexible enough to create songs in a much wider range of styles.

Appendix E

Publications and papers

Papers published in the proceedings of a refereed conference

Dynamical Systems and Applications to Music Composition and Sound Synthesis: A Research Report. Published in proceedings of the Journées d'Informatique Musicale '97.

A Cellular Automata Based Music Algorithm: A Research Report. Published in proceedings of the IV Symposium of Brazilian Computer Music.

Music Composition by means of Pattern Propagation. Published in proceedings of XII Colloquium on Musical Informatics.

Papers published in a refereed journal

Making Music with Algorithms: A Case-Study System. Published in Computer Music Journal, Vol. 23, No. 2.

Other publications

Review of Brazilian Electroacoustic Music Concert, University of Glasgow, Thursday 13th November 1996. Published in the Computer Music Journal, Vol. 21, No. 2.

Audio Architect Tutorial. Published in 'Computer Sound Synthesis for the Modern Musician' by Eduardo Miranda (Focal Press, 1998. ISBN 0-240-51517-X).

CAMUS version 2.0 for Windows 95. Published on the cover-mounted CD of The Mix magazine, issue 45.

Death of a Pusher. Published as part of Jerome Joy's *Collage Jukebox* project.

A Scottish Dance. Published as part of Jerome Joy's *Collage Jukebox* project.

Four's Company. Published as part of Jerome Joy's *Collage Jukebox* project.

Electro. Released as part of MIT's CSound CD-ROM collection.

Bibliography

[Abraham, 1982] Gerald Abraham, *The Age of Beethoven 1790 – 1830*. Oxford University Press.

[Abraham, 1990] Gerald Abraham, *Romanticism 1830 - 1890*. Oxford University Press.

[Ames, 1983] Charles Ames, *Stylistic Automata in Gradient*. Computer Music Journal 7(4): 45 – 56.

[Ames, 1989] Charles Ames, *The Markov Process as a Compositional Model: A Survey and Tutorial*. Leonardo 22(2): 175 – 188.

[Autodesk, 1999] Autodesk Inc., *3D Studio MAX R2.5*.

[Bach, 1923] Johann Sebastian Bach, *48 Preludes and Fugues for the Well-Tempered Clavier*. Edwin Ashdown Ltd.

[Baird, Blevins & Zahler, 1993] Bridget Baird, Donald Blevins, & Noel Zahler, *Artificial Intelligence and Music: Implementing an Interactive Computer Performer*. Computer Music Journal 17(2): 73 – 79.

[Bays, 1987] Carter Bays, *Candidates for the Game of Life in Three Dimensions*. Complex Systems 1(2):373 – 400.

[Beyls, 1997] Peter Beyls, *Aesthetic Navigation: Musical Complexity Engineering using Genetic Algorithms*. In proceedings of Journées d'Informatique Musicale, pp. 97 – 105.

[Bharucha & Todd, 1989] Jamshed J. Bharucha & Peter M. Todd, *Modeling the Perception of Tonal Structure with Neural Nets*. Computer Music Journal 13(4): 44 – 53.

[Bolognesi, 1983] Tommaso Bolognesi, *Automatic Composition: Experiments with Self-Similar Music*. Computer Music Journal 7(1): 25 – 36.

[Burger & Gillies, 1989] Peter Burger & Duncan Gillies, *Interactive Computer Graphics*.

[Chaplin, 1936] Charles Chaplin, *Modern Times*. United Artists.

[Chareyron, 1990] Jacques Chareyron, *Digital Synthesis of Self-modifying Waveforms by Means of Linear Automata*. *Computer Music Journal* 14(4): 25 – 41.

[Cope, 1987] David Cope, *An Expert System for Computer Assisted Composition*. *Computer Music Journal* 11(4): 30 – 46.

[Cope, 1991] David Cope, *Computers and Musical Style*. Oxford University Press.

[Corso, 1957] J. F. Corso, *Absolute Judgements of Musical Tonality*. *Journal of the Acoustical Society of America* 55: 292.

[Darwin, 1981] Charles Darwin, *On the Origin of the Species by Means of Natural Selection*. In *A Concordance to Darwin's Origin of Species, First Edition*. Cornell University Press.

[Dewdney, 1987] A. K. Dewdney, *The Game of Life Acquires Some Successors in Three Dimensions*. *Scientific American*, 286(2): 16 – 22.

[Dodge, 1988] Charles Dodge, *Profile: A Musical Fractal*. *Computer Music Journal* 12(3): 10 – 14.

[Dolson, 1989] Mark Dolson, *Machine Tongues XII: Neural Networks*. *Computer Music Journal* 13(3): 28 – 40.

[Ermentrout & Edelstein-Keshet, 1993] G. Bard Ermentrout & Leah Edelstein-Keshet, *Cellular Automata Approaches to Biological Modeling*. *Journal of Theoretical Biology*, 160: 97 – 133.

[Freedman, 1971] David Freedman, *Markov Chains*. Holden-Day.

[Gardner, 1971] Martin Gardner, *On Cellular Automata, Self-Reproduction, the Garden of Eden and the Game 'Life'*. *Scientific American* 224(2): 112 – 118.

[Greenhouse, 1993] Robert Greenhouse, *The Well-Tempered Fractal*.

[Haykin, 1994] Simon Haykin, *Neural Networks: A Comprehensive Foundation*. Macmillan.

[Hearn & Baker, 1994] Donald Hearn & M. Pauline Baker, *Computer Graphics Second Edition*. Prentice Hall.

[Heckroth, 1994a] Jim Heckroth, *A MIDI Tutorial - The General MIDI System*.
http://kingfisher.cms.shu.ac.uk/midi/mt_gm.htm

[Heckroth, 1994b] Jim Heckroth, *General MIDI Patch Map*.
http://kingfisher.cms.shu.ac.uk/midi/gm_map.htm

[Heckroth, 1994c] Jim Heckroth, *General MIDI Percussion Set*.
<http://kingfisher.cms.shu.ac.uk/midi/gmdrum.htm>

[Heckroth, 1994d] Jim Heckroth, *The Roland GS Standard*.
http://kingfisher.cms.shu.ac.uk/midi/mt_gs.htm

[Hogeweg, 1988] P. Hogeweg, *Cellular Automata as a Paradigm for Ecological Modeling*. Applied Mathematics and Computation 27: 81 – 100.

[Hoggar, 1992] Stuart G. Hoggar, *Mathematics for Computer Graphics*. Cambridge University Press.

[Holmes, 1878] Edward Holmes, *The Life of Mozart Including his Correspondence*. Novello, Ewer.

[Holtzman, 1994] Steven Holtzman, *Digital Mantras: The Language of Abstract and Virtual Worlds*. MIT Press.

[Honegger, 1994] Arthur Honegger, *Pacific 231*. EMI Classics.

[Horner, Beauchamp & Haken, 1993] Andrew Horner, James Beauchamp & Lippold Haken, *Machine Tongues XVI: Genetic Algorithms and Their Application to FM Matching Synthesis*. Computer Music Journal 17(4): 17 – 29.

[IRCAM, 1993] IRCAM, *Patchwork*.

- [Jenkins & Sano, 1989] B. Keith Jenkins & Hajime Sano, *A Neural Network Model for Pitch Perception*. *Computer Music Journal* 13(3): 41 – 48.
- [Johnson, 1997a] Derek Johnson, *Atari Notes*. *Sound on Sound* 13(1): 258.
- [Johnson, 1997b] Tom Johnson, *Lab Reports*. In proceedings of Journées d'Informatique Musicale, pp. 56 – 60.
- [Jones, 1980] Kevin Jones, *Compositional Applications of Stochastic Processes*. *Computer Music Journal* 5(2): 45 – 61.
- [Jones, 1996] Andy Jones, *The Ultimate Trinity*. *Future Music*, 51: 30 – 33.
- [Jones & Parks, 1988] Douglas L. Jones & Thomas W. Parks, *Generation and Combination of Grains for Music Synthesis*. *Computer Music Journal* 12(2): 27 – 34.
- [Joy, 1998] Jerome Joy, *Collage Jukebox Project*. <http://homestudio.thing.net/collage/>
- [Kartalopoulos, 1996] S. V. Kartalopoulos, *Understanding Neural Networks and Fuzzy Logic*. IEEE Press.
- [Kindermann, 1995] Lars Kindermann, *Musinum, the Music in the Numbers*.
- [Knuth, 1973] Donald E. Knuth, *The Art of Computer Programming Vol. 1: Fundamental Algorithms*. Addison Wesley.
- [Lehrman, 1997] Paul D. Lehrman, *4 Score! Digidesign Pro Tools 4.0 Software*. *Sound on Sound* 12(9): 162 – 173.
- [Lewis, 1935] Meade Lux Lewis, *Honky-Tonk Train Blues*. Peter Maurice Music Co. Ltd.
- [Lipschutz, 1974] Seymour Lipschutz, *Schaum's Outline of Theory and Problems of Probability*. McGraw-Hill.
- [Little, 1993] David Little, *Composing with Chaos; Applications of a New Science for Music*. *Interface* 22: 23 – 51.

[Lloyd & Boyle, 1963] Llewelyn S. Lloyd & Hugh Boyle, *Intervals, Scales and Temperaments*. Macdonald and Jane's Publishers Ltd.

[Lorrain, 1980] Denis Lorrain, *A Panoply of Stochastic Cannons*. *Computer Music Journal* 4(1): 53 – 81.

[Loy, 1985] Gareth Loy, *Musicians Make a Standard: The MIDI Phenomenon*. *Computer Music Journal* 9(4): 8 – 26.

[Loy, 1989] Gareth Loy, *Composing With Computers – a Survey of some Compositional Formalisms and Music Programming Languages*. In *Current Directions in Computer Music Research*, MIT Press, pp. 292 – 396.

[Mandelbrot, 1982] Benoit B. Mandelbrot, *The Fractal Geometry of Nature*. Freeman.

[McAlpine, Miranda & Hoggar, 1997a] Kenneth B. McAlpine, Eduardo R. Miranda & Stuart G. Hoggar, *Dynamical Systems and Applications to Music Composition: A Research Report*. In proceedings of Journées d'Informatique Musicale, pp. 106 – 113.

[McAlpine, Miranda & Hoggar, 1997b] Kenneth B. McAlpine, Eduardo R. Miranda & Stuart G. Hoggar, *A Cellular Automata Based Music Algorithm: A Research Report*. In proceedings of IV Brazilian Computer Music Symposium, pp. 7 – 17.

[McAlpine, Miranda & Hoggar, 1998] Kenneth B. McAlpine, Eduardo R. Miranda & Stuart G. Hoggar, *Music Composition by Means of Pattern Propagation*. In proceedings of XII Colloquium on Musical Informatics, pp. 105 - 108.

[McAlpine, Miranda & Hoggar, 1999] Kenneth B. McAlpine, Eduardo R. Miranda & Stuart G. Hoggar, *Making Music with Algorithms: A Case-Study System* *Computer Music Journal* 23(2): 19 – 30.

[Meyn & Tweedie, 1993] Sean P. Meyn & Richard L. Tweedie, *Markov Chains and Stochastic Stability*. Springer-Verlag.

[Microsoft, 1999] Microsoft Corporation, *Direct Music*.

[Miranda, 1993] Eduardo R. Miranda, *Cellular Automata Music: An Interdisciplinary Project*. *Interface* 22: 3 – 21.

[Miranda, 1994] Eduardo R. Miranda, *Music Composition Using Cellular Automata*. Languages of Design 2: 105 – 117.

[Miranda, 1995] Eduardo R. Miranda, *Granular Synthesis by Means of a Cellular Automaton*. Leonardo 28(4):297 – 300.

[Miranda, 1997] Eduardo R. Miranda, *Who Composed Entre l'Absurde et le Mystère: An Introduction to Music and Artificial Intelligence*. In proceedings of IV Brazilian Computer Music Symposium, pp. 59 – 71.

[Moore, 1994] Jason H. Moore, *GA Music*. Software Visions.

[Monro, 1995] Gordon Monroe, *Fractal Interpolation Waveforms*. Computer Music Journal 19(1): 88 – 98.

[Murphy, 1991] Ian S. Murphy, 1991. *Probability: A First Course*. Arklay.

[Peitgen, Jurgens & Saupe, 1992] H. O. Peitgen, H. Jurgens & D. Saupe, *Chaos and Fractals: New Frontiers of Science*. Springer-Verlag.

[Peitgen & Richter, 1986] H. O. Peitgen & P. H. Richter, *The Beauty of Fractals: Images of Complex Dynamical Systems*. Springer-Verlag.

[Peitgen & Saupe, 1988] H. O. Peitgen & D. Saupe, *The Science of Fractal Images*. Springer-Verlag.

[Philips, 1999] Philips Electronics N. V., *FreeSpeech 98*.

[Preston & Duff, 1984] K. Preston & M. J. B. McDuff, *Modern Cellular Automata: Theory and Applications*. Plenum Press.

[Res Rocket, 1999] Res Rocket Surfer Inc, *The Res Rocket Network*.

[Roads, 1978] Curtis Roads, *Automated Granular Synthesis of Sound*. Computer Music Journal, 2(2): 61 – 62.

[Roads, 1980] Curtis Roads, *Artificial Intelligence and Music*. Computer Music Journal 4(2): 13 – 25.

- [Roads, 1988] Curtis Roads, *Introduction to Granular Synthesis*. Computer Music Journal 12(2): 11 – 13.
- [Roads, 1996] Curtis Roads, *The Computer Music Tutorial*. MIT Press.
- [Russ, 1992] Martin Russ, *Datamusic Fractal Music*. Sound on Sound 7(8): 114.
- [Sansom, 1992] Chris Sansom, *Fractal Music ST*. Datamusic.
- [Schalkoff, 1997] Robert J. Schalkoff, *Digital Image Processing and Computer Vision*. Wiley.
- [Schoenberg, 1984] Arnold Schoenberg, *Style and Idea: Selected Writings of Arnold Schoenberg*. University of California Press.
- [Schwarz, 1973] Charles Schwartz, *Gershwin: His Life and Music*. Abelard-Schuman.
- [Sherman, 1999] Shane Sherman, *Rust – Gamedesign.net: Your Source for Game Design and Level Design Information*. <http://www.gamedesign.net>
- [Sullivan, 1990] Charles R. Sullivan, *Extending the Karplus-Strong Algorithm to Synthesize Electric Guitar Timbres with Distortion and Feedback*. Computer Music Journal 14(3): 26 – 37.
- [Sutherland, 1994] Roger Sutherland, *New Perspectives in Music*. Sun Tavern Fields.
- [Syntrillium, 1996] Syntrillium, *Wind Chimes*.
- [Thompson, 1996] Tim Thompson, *KeyKit*. AT&T Corp.
- [Tolonen, 1987] Pekka Tolonen, *Symbolic Composer*. Tonality Systems, Inc.
- [Truax, 1988] Barry Truax, *Real-Time Granular Synthesis with a Digital Signal Processor*. Computer Music Journal 12(2): 14 – 26.
- [Tucker, 1993] Tim Tucker, *Get to Grips with MIDI*. Future Music 4:65 – 68.
- [Turing, 1950] Alan Turing, *Computing Machinery and Intelligence*. Mind 59: 433-460.

[Ullman, 1990] Jeffrey D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume 2*. W H Freeman.

[Walker, 1998] Martin Walker, *Sonics and Daughters*. *Sound on Sound* 14(2):210 – 214.

[Warthman, 1993] Forrest Warthman, *Neural Network Audio Synthesizer*. *Dr. Dobbs Journal* 18(2): 50.

[Wentk, 1995] Richard Wentk, *Surf's Up*. *Future Music* 29: 57 – 59.

[White, 1996] Paul White, *G Whizz! Yamaha G50 MIDI Guitar Interface*. *Sound on Sound* 12(2): 242 – 246.

[Wolfram, 1985] Stephen Wolfram, *Cryptography with Cellular Automata*. In *Proceedings of Crypto '85*, pp. 429 – 432.

[Worner, 1973] Karl H. Worner, *Stockhausen: Life and Work*. Faber and Faber.

[Xenakis, 1971] Iannis Xenakis, *Formalized Music*. Indiana University Press.

[Yamaha, 1999] Yamaha, *On-line Guide to XG*.
<http://www.yamaha.co.uk/xg/reading/index.html>