Synthesis of organic sounds for electroacoustic music:

Cellular models and the TAO computer music program

Author: Mark Pearson

Thesis submission for the degree of DPhil in Music Technology

Department of Electronics, University of York

Supervisor: Dr. David M. Howard

November 29, 1996

Abstract

This thesis examines the question of what differentiates naturally occurring sounds from the majority of digitally synthesised sounds. The discussion centres around the notion that sounds may be viewed as *structured auditory information* and examines both the human auditory system's ability to generate perceptual *imagery* from this information, and the underlying principles which govern the creation of *complex* but *coherent* structured information in Nature. A solution to the goal of developing new sound synthesis techniques capable of generating 'organic', 'naturalistic' sounds events for electroacoustic music is proposed. This solution centres around the use of a particular class of computer models, collectively referred to as *cellular models*, which consist of large numbers of simple agents interacting with one another on a local basis, and give rise to complex global patterns of behaviour. A survey of existing sound synthesis techniques is given, including descriptions of some contemporary computer music programs, and a new computer music program called TAO is described. TAO forms a substantial part of the research undertaken and is a working prototype capable of creating a wide variety of organic and naturalistic sound events. It therefore enables the provision of aural evidence for many of the arguments put forward. Other specific topics covered include the spectro-morphological and acousmatic approaches to music, the ecological view of auditory perception, chaos theory, complexity, the study of dynamical systems and emergent behaviour. The thesis concludes with a model of *organic* sounds and comments on the future development of cellular sound synthesis techniques.

Contents

List of Figures

Acknowledgements

I would like to thank my supervisor, Dr David Howard, for allowing me the freedom to pursue my ideas as I saw fit, for his general efficiency and enthusiasm for my work, and also for helping to invent the degree 'DPhil in Music Technology' at the eleventh hour. I would also like to thank Andy Hunt and Richard Orton for interesting discussions on a number of occasions, and Prof David Worrall for showing so much enthusiasm, and potentially providing me with an opportunity for continuing the work.

There are many individuals who should rightfully be acknowledged in this thesis, through having provided friendship and moral support. All have, at one time or another, helped to maintain some level of sanity in the midst of the madness known as a DPhil. It is difficult to remember to mention everybody, but in particular I would like to thank Nick Fells, Sharon Lyons, Andy Holbrook, Tony Hood, Ian Crutchley, Lisa Reim, Paul Yoward, Amina Alyal, Ian Gibson, Giselle Ferreira, Sotiris Missailidis, and Michelle Evans for the good times.

Special thanks to Nick, Sharon and Andy for lots of nice food and days out, to Amina for even more nice food, and to Tony for lots of relaxing coffee breaks and much moral support during the closing stages of the thesis.

Finally, thanks to Maria Holgate and John Munns for many memorable weekends, to Stephanie Freeman for (long distance) moral support and encouragement, and to my family.

Acknowledgements

Declaration

I hereby declare that this thesis contains research which is for the most part my own. In cases where I have drawn upon the ideas of others I have clearly stated so and given references.

I also declare that some parts of the research described have been previously published, and that some are due for publication. These publications are listed below:

1. Pearson, M. (1995). TAO: a physical modelling system and related issues, *Organised Sound* 1(1): 43–50.

2. Pearson, M. and Howard, D. M. (1995). A musician's approach to physical modelling, *Proceedings of international computer music conference*, pp. 578–80.

3. Pearson, M. and Howard, D. M. (1996). Recent developments with the TAO physical modelling system, *Proceedings of international computer music conference*, pp. 97-9.

Chapter 1

Background and thesis structure

1.1 Introduction

Since music is a perceptual phenomenon, the goals of what might be termed musical research, and the ways in which it proceeds, are very different from those of scientific research. One of the goals of musical composition and analysis lies in gaining a greater understanding of the nature of sound in all its variety, and how we perceive it. The phenomenon of sound is two-fold though, depending both on the mechanisms responsible for the pressure fluctuations reaching a listener's ears, and on the perceptual apparatus of the listener. The quest for a greater understanding of the nature of sound ought then to place equal emphasis on both scientific enquiry into the mechanisms which are responsible for the multitude of sounds we hear about us, and on the subjective perceptions of musicians involved in the business of expressing themselves through sound.

This thesis follows in a fairly well established tradition of querying the musical nature of sound, especially prevalent in the field of electronic and computer music, and in particular, deals with the question of how we may take inspiration from natural sounds[1] in order to develop new sound synthesis techniques which are capable of producing more 'naturalistic' or 'organic' sounds. The program of research described in this thesis was initially prompted by some perceived deficiencies in existing approaches to sound synthesis and in the sounds they produce when compared to

[1] sounds produced by vocal or instrumental means, environmental sounds or any other sound produced as a side-effect of some physical process or mechanism.

natural sounds. To be more specific, natural sounds often seem to be:

1. more strongly suggestive of physical causality;
2. more subtle and intricate;
3. more coherent, seemingly possessing stronger identities;
4. and more vibrant and organic than synthesised sounds.

These points are based directly upon aural experience and should be taken at face value as empirical observations rather than concrete facts or fundamental criticisms of digital technology itself. However, the fact that many electroacoustic compositions make use of naturally occurring rather than synthesised sounds as source material, inevitably tells us something about the special resonance which natural sounds hold for us, and also about the amount of time and energy which must be expended in order to create synthetic sounds possessing a similar degree of subtlety. The term *natural sound* is used here to refer to the physical processes or mechanisms responsible for a sound, whereas terms such as *naturalistic* and *organic* are used to refer to specific perceptual attributes possessed by a sound, which seem to suggest that it has been produced by a physical process or mechanism of some description. One of the goals of this thesis is to identify the factors which contribute to a sound being classed as *organic* or *naturalistic*, and also to attempt to provide explanations for other adjectives such as *vibrant* and *lively* which may be applied to any type of sound but seem at first to be rather subjective. A central premise of this thesis is that such terms do have a stronger basis for their use than mere personal taste, and relate to the *structured information* inherent in a sound.

1.2 Hypothesis

The hypothesis of this program of research is that:

> Cellular computer models, inspired by the behaviour of naturally occurring complex dynamical systems, provide an ideal medium for the development of a new generation of sound synthesis techniques, more holistic in their approach than traditional techniques, and capable of producing

complex organic sounds events, whilst simultaneously being sympathetic to the needs of electroacoustic music.

This hypothesis is supported in three ways:

1. At a theoretical level, by examining the notion that all sounds may be viewed as structured auditory information, addressing both the perceptual effects which this information evokes, and the underlying natural laws which give rise to particular patterns of information in the first place.

2. At a practical level, by a variety of visual and sonic examples produced by the TAO computer music program, based entirely upon instruments constructed from cellular physical models.

3. By a comparison of the strengths and weaknesses of the approach taken by TAO with existing synthesis techniques.

The rest of this introductory chapter serves to lay out the background for the research. Section 1.3 describes some key points pertaining to digital sound synthesis for composition. Section 1.4 describes the spectro-morphological and acousmatic approaches to music, for which TAO has been specifically designed. Section 1.5 describes the ecological approach to auditory perception, based on the premise that the environment presents a listener with *structured information* containing all the details necessary for the perception of objects and events. Section 1.6 relates the comments given above about the deficiencies of digitally synthesised sounds to a wider set of views expressed by those involved in the composition and analysis of electroacoustic music. Finally, section 1.7 lays out the plan for the rest of the thesis.

1.3 Digital technology and models of sound synthesis

The composer Edgar Varése first coined the phrase *organised sound* as a general definition of what all music basically is, regardless of genre. Taking this definition as a starting point, the task of musical sound synthesis is actually one of organising sound at various structural levels. The strategies available for organising sound should be as general as possible in order not to interfere with the individual composer's musical

ideas, and whilst digital technology has made it possible for composers to capture and manipulate sounds, and place them in contexts other than that in which they originally occurred, on its own, a digital computer deals with nothing but raw numerical data. Thus in order to create new sounds or manipulate existing ones in ways which are musically meaningful, we have to first develop synthesis models which encapsulate our view of how sound is structured and how it functions. The theoretical claim made that a digital computer is capable of synthesising *any* sound because of its universality and programmability, is meaningless without these synthesis models.

> *Faced by an infinite palette of potential timbres and infinite gradations of frequency and time, the question posed seems to be one of finding appropriate ways of structuring such continuous variables (Windsor, 1995, section 2.1.3).*

When digital computer technology first became a viable tool for sound synthesis, the most obvious model to adopt was based upon the technology used in analogue voltage controlled synthesisers. This model provided digital versions of components such as oscillators, filters, mixers etc. originally appropriated from the discipline of electronic engineering. The majority of sound synthesis techniques has traditionally relied upon an essentially reductionist, frequency domain view of sound as we shall see in chapter 3. The computer music program Csound, which relies upon this approach, is briefly described at the beginning of the same chapter.

Whilst it is not the place of this thesis to *prove* that one approach to sound synthesis is better in every respect than another, it is proposed that with the use of cellular models, it will be possible to develop a whole new generation of techniques which will be more holistic in their approach to the task of synthesising complex and organic sound events for electroacoustic music. These techniques will address specific deficiencies inherent in traditional techniques but will ultimately complement them, giving the composer a wider range of tools for the task of organising sound. The TAO computer music program, described in chapters 4, 5 and 6 provides some evidence for this claim in the form of both sonic and visual examples, but this thesis also addresses the wider implications of the use of cellular models.

1.4 Spectro-morphological and acousmatic music

The work described in this thesis takes account of both the *spectro-morphological* (Smalley, 1990) and *acousmatic* (Windsor, 1995) approaches to music. In traditional 'note' based music the notes are seen as the 'prime carriers of information' whereas timbre is seen as a secondary attribute which merely provides coloration for the notes. The spectro-morphological approach to music does not oppose this view directly but embraces it within a wider context in which all categories of sound potentially have equal musical value. In this context the pitched sounds of traditional musical instruments represent just one possible type of spectral structure in which the partials happen to be arranged with a preference for harmonic relationships. Smalley describes three main categories of sound, in term of their spectral structures, *note*, *node* and *noise*. These are elaborated on later in this section.

One of the most important skills the electroacoustic composer must possess is the ability to listen to sounds *acousmatically*. An *acousmatic* approach to listening involves the apprehension of a sound as an object in its own right, without relation to its source. In everyday circumstances the human auditory system serves its evolutionary purpose of helping us to identify objects and events in our immediate environment. This mode of listening, or rather *hearing* since it is essentially passive and subconscious, reduces sounds to the role of mere triggers for recognition or identification of objects or events. For example when a car drives past, the normal subconscious reaction is to conclude that the sound heard *is* a car rather than to pay any special attention to its timbral characteristics. Although the sound possesses characteristics which in everyday life might suggest that it has been produced by a car, it is possible to suppress the image of the car itself and concentrate on the evolution of the sound as an object in its own right. The normal mechanism of source-recognition is so deeply rooted and automatic that, often, practice is required in order to suppress it in favour of a more active, acousmatic mode of listening.

An analogy may be drawn with visual art, since an artist must learn to see subtle textures and hues of colour which might not be immediately evident to a non-artist, more concerned with recognition and categorisation of objects. In the same way that visual art may be representational or abstract, so then there are two aspects to sound, the *concrete* and *abstract*.

> *All sounds possess this dual potential – the* abstract *and* concrete *aspects of sound – and all musical structures are balanced somewhere between the two, although exactly how they are balanced can vary greatly among listeners (Smalley, 1990, p.64).*

The use of the word 'acousmatic' has led to its being adopted to refer to another genre of music, *acousmatic music*, defined by Windsor (1995) as:

> *a form of music which is presented through loudspeakers to an audience from an analog or digital tape-recording. This music may contain sounds that have recognisably musical sources, but may equally present recognisable sources that are beyond the bounds of traditional vocal and instrumental technology. We are as likely to hear the sounds of a bird, or of a factory as we are the sounds of a violin. Consider also that the technology involved transcends the mere reproduction of sounds. Techniques of synthesis and sound processing are employed which may present us with sounds that are unfamiliar and that may defy clear source attribution. Consider that this form of music may present us with familiar musical events: chords, melodies and rhythms which are easily reconcilable with other forms of music, but may equally present us with events which cannot be classified within such a traditional taxonomy (Windsor, 1995, section 1.0).*

The precise differences between the spectro-morphological and acousmatic approaches to music are not under discussion here, but what is significant to both is the use of a much wider, all-encompassing palette of sounds. One of the themes which appears frequently in the literature concerning both genres of music is the importance of aural perception in the composition, analysis and interpretation of such pieces. Conventional musical theories concerned with notes, rhythms, melodies, phrases etc. are inadequate to explain what a listener makes of such music.

> *Attention to perception affords us some knowledge of which attributes of sound may be perceived by the listener; and as an aural practice, acousmatic composition is as much about listening as it is about abstract technical manipulations of sound. Hence, by studying perception one might*

arrive at descriptions that correspond to what is heard by the composer, and hence, provide a method of describing why particular sounds lead to particular compositional decisions (Windsor, 1995, section 1.1.1).

Spectromorphology reaffirms the primacy of aural perception which has been so heiniously ignored in the recent past, and warns composers, researchers and technologists that unless aural judgement is permitted to triumph over technology, electroacoustic music will attract deserved condemnation (Smalley, 1990, p.93).

...we return to aural discrimination and perception as the supreme musical tools. It is not a scientific knowledge which is required but an experiential knowledge (Smalley, 1990, p.81).

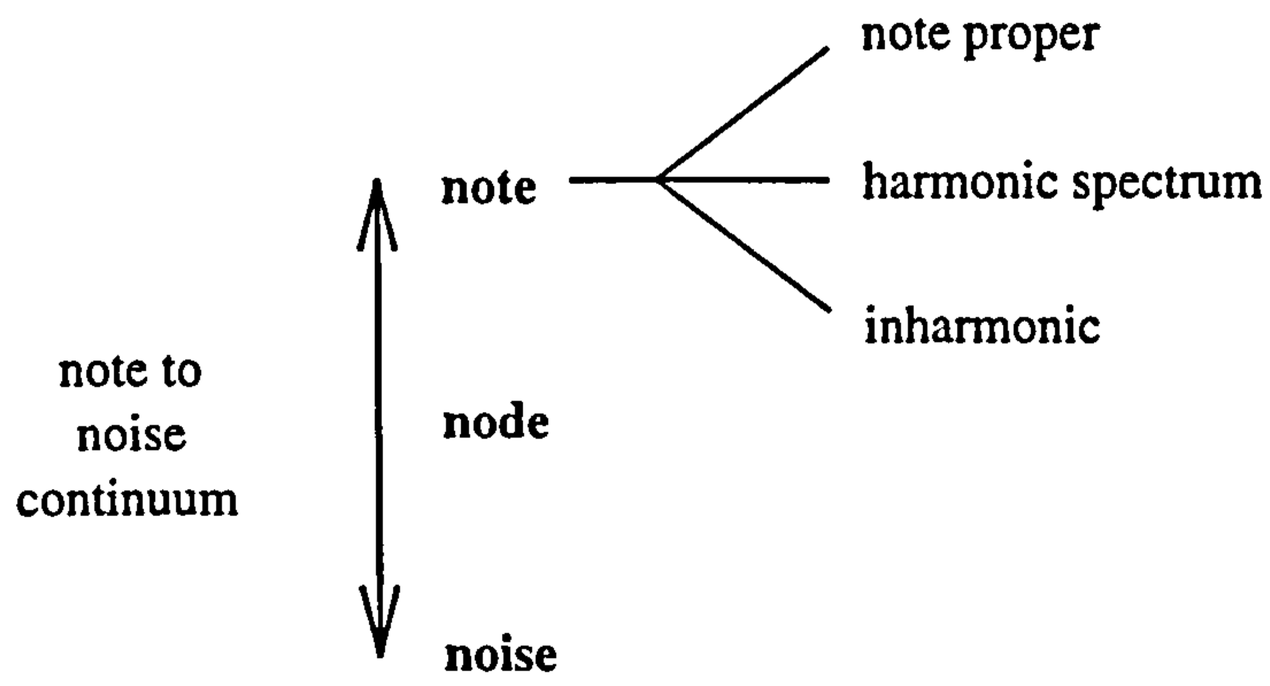


Figure 1.1: Spectral typology of sound (after Smalley 1990)

We return now to Smalley's sound categories: *note*, *node* and *noise*. The *note* category is further subdivided into: *note proper*, where the absolute pitch of the sound is used in the traditional context of intervallic relationships with other pitches; *harmonic spectrum*, in which the fundamental frequency or perceived pitch is of secondary importance to the individual partials and their balance within the sound, i.e. to rephrase in traditional terms, the *timbre* becomes more important than the *pitch* even though there *is* a clearly perceivable pitch; and *inharmonic spectrum*, which includes certain tuned percussion instruments which allude to pitches and yet have non-harmonicly related partials. A *nodal spectrum* is one in which the partials are clustered in such a way that they are perceived as a whole, and yet no pitch

centres are evident in the sound. An example is the sound produced by a cymbal. Finally a *noise spectrum* is one in which it is impossible to perceive any kind of fixed pitches or clusters of partials and yet which still exhibits some kind of structural coherence, such as the sound of the sea or wind.

1.5 The ecological view of auditory perception

As described in the introduction to this chapter, one of the goals of this thesis is to clarify the meanings of words such as *organic*, *naturalistic*, *vibrant* and *lively*. In order to attack this question we first need to make explicit some premises on which the various arguments are based. The first premise is that the qualities described above are not merely subjective and personal in nature, but relate instead to the information contained within a sound, and the way in which it is structured. Gibson (1979) proposed a theory of perception in his book *The ecological approach to visual perception* which supports this notion. The theory is described briefly in this section and although not based directly upon Gibson, this thesis adopts some of the terminology introduced.

Gibson's theory contradicts the view that organisms such as ourselves have to maintain mental models of our environment in order to make sense of the chaotic mass of information generated by that environment. Instead it is proposed that the environment, because of its inherent structure, presents the organism with *structured information* which it is able to *pick up* visually or aurally etc. The perception of objects and events in the environment depends, then, not on internally structured mental models, but upon direct perception of structures which are external to the organism.

> *Rather than assuming that the sensations passed from the sense organs to the central nervous system represent a chaotic source of information that mental processes organise and store in the form of meaningful percepts and memories, an ecological approach assumes that the 'external' world, the environment, is structured and that organisms are directly sensitive to such structure (Windsor, 1995, section 2.1).*

There are two important aspects about the natural environment which relate to

the nature of the structured information it generates: firstly, it is hierarchical both spatially and temporally; and secondly, it always contains elements of repetition and non-repetition, elements which persist and those which do not, once again both spatially and temporally.

> *Just as physical reality has structure at all levels of metric size, so it has structure at all levels of metric duration ... And ... it is important to realise that smaller units are nested within larger units. There are events within events, as there are forms within forms ... (Gibson, 1979, p.12)*

> *The environment normally manifests some things that persist and some that do not, some features that are invariant and some that are variant. A wholly invariant environment, unchanging in all parts and motionless, would be completely rigid and obviously would no longer be an environment ... At the other extreme, an environment that was changing in all parts and was wholly variant, consisting only of swirling clouds of matter, would also not be an environment. In both extreme cases there would be space, time, matter, and energy, but there would be no habitat (Gibson, 1979, p.14).*

An organism possesses a variety of sense organs and *perceptual systems* whose purpose, from an evolutionary point of view, is to enable it to make sense of its immediate environment by perceiving objects and events pertinent to its survival. According to Gibson, perception of these objects and events is made possible by the structured information which they generate, and the process of perceiving them relies on the organism being able to extract the *invariant* features from this continual flow of information. Gibson raises some very subtle points about the nature of *identity*, emphasising that the recognition of an object or event's persistence is more fundamental and direct than the recognition of differences between several objects or events. The former is seen as occurring in a direct, unmediated manner, whilst the latter requires abstraction after the event of perception.

> *In the case of the persisting thing, I suggest, the perceptual system simply extracts the invariants from the flowing array; it* resonates *to the invariant structure or is* attuned *to it. In the case of substantially distinct*

> *things, I venture, the perceptual system must* abstract *the invariants. The former process seems to be simpler than the latter, more nearly automatic (Gibson, 1979, p.249).*

A practical illustration of these ideas can be found in the perception of a substance such as water. We are able, instantly, to recognise water, since the visual information it generates, via reflected light, tends to exhibit certain spatial and temporal patterns. Even though the surface of water is in a continual state of flux, it always behaves in a water-like manner and in this context the adjective *water-like* refers to the invariant features in the structured information caused by the water.

Gibson's theory of perception also extends to the meanings which objects and events have for an organism in terms of its survival. Rather than these events and objects being detached from the organism perceiving them, they *afford* certain possibilities. For example, the sound of a loud explosion, via the particular pattern of structured information which it causes, *affords* the possibility of being seriously injured. Similarly the sound of a sea-shore with gently breaking waves might afford great relaxation, for a human listener anyway. The crux of the theory lies in the fact that an organism evolves an innate ability to attune to, and attach certain meanings to, particular patterns of structured information, in order to enhance its chances of survival, but that these patterns of information are contained within the environment, and are not created by the organism itself.

> *An organism evolves . . . to pick-up information that will increase its chances of survival. It develops perceptual systems that enable it to perceive features of the environment that facilitate continued existence, and hence reproduction. Moreover, the dynamic relationship between a perceiving, acting organism and its environment is seen to provide the grounds for the direct perception of meaning. Gibson's term for this is "affordance". Objects and events are related to a perceiving organism by structured information, and they "afford" certain possibilities for action relative to an organism. For example, a cup affords drinking, the ground, walking.*
>
> *Affordances, "point both ways" (Gibson, 1979, p.129) in that they can neither be explained purely in terms of the needs of the organism, nor in*

> *terms of the objective features of the environment. The affordance is a relationship between a particular environmental structure and a particular organism's needs and capacities (Windsor, 1995, section 2.1.1).*

Affordances are not always so clearly and unambiguously defined as in the case of a loud explosion, since each sound will have its own connotations for each individual listener based upon their own personal experiences. We are not concerned here with analysing the affordances of particular sounds but merely with the fact that such mechanisms operate at a deeply rooted, subconscious level and will therefore influence the musical effect which a sound has upon a listener, even if this effect is not intended by the composer. And, more importantly, we are interested in the fact that such mechanisms need nothing other than appropriately structured information for their evocation. There is nothing in principle which prevents a synthetic sound from possessing utterly convincing natural qualities, and even evoking very real affordances in the listener, provided the information generated by the synthesis model captures the essence of the patterns which would be found in an equivalent natural sound.

Gibson also raises some significant issues relating to the nature of objects and events. Firstly, natural hierarchies are different from man-made hierarchical structures such as machines. In an organic hierarchy, or *holarchy* (Sheldrake, 1988, p.95), there are no divisible levels of structure and no divisible components at any particular scale. Instead there is a smooth continuum of organisation from the smallest scale up to the largest.

> *A living organism ...is not assembled from parts, and its members, although they move, constitute a different sort of hierarchy (Gibson, 1979, p.96).*

Secondly, Gibson's use of the words *duration* and *size* rather than *time* and *space*, emphasise the point that the abstract concepts of space and time described by physicists have little to do with everyday perceptual practice. Our senses of space and time arise from the *objects* and *events* we perceive in our environment, and whilst both occur at all scales from the the atomic to the cosmic, we are only able to perceive a limited 'bandwidth' of the available information.

It is worth emphasising that the information generated by natural environments may be extremely *complex* and yet *coherent*. If a single point source of sound or light generates spherically radiating wavefronts, each of which leads to multiple reflections off of the various surfaces in the immediate vicinity, then each one of these reflected wavefronts will undergo further reflections. Combined with the fact that real surfaces are not perfectly flat mathematical planes but have rough, uneven textures, even a tiny portion of such a surface will lead to wavefronts being scattered in many directions. If we then consider the subsequent reflections of these scattered wavefronts, it quickly becomes clear that the information generated will be extremely complex. An observer placed at a fixed location will only pick up a tiny fraction of the total information generated by the environment, and yet even this fraction will be extremely complex. But if the observer moves about the environment, although the information picked up at each point will be extremely complex, it will bear a *coherent* relationship to the information available at every other point in the environment.

An indication of this complexity may be found in the generation of realistic computer graphics images. Computer graphics techniques such as ray-tracing [2] involve enormous amounts of computation but often produce very realistic images, whereas the use of more simple, short cut techniques often lead to images possessing overtly simplistic and synthetic qualities. Yet somehow, regardless of the complex nature of the information generated by natural environments, even an untrained listener is able to recognise the difference between say, the reverberant characteristics of a church and a small furnished room in a house. Moreover, this occurs in a direct, subconscious manner. The listener can immediately detect the close proximity of a wall or an open doorway etc. Structured information may therefore be very complex, and yet through its coherence, may be capable of providing the perceiver with a clear and consistent 'picture' of the surrounding environment.

Traditional techniques of sound spatialisation used in computer music consisting of systems capable of projecting sounds over multiple loudspeaker arrays, and digi-

[2] A technique which involves tracing individual virtual rays of light back from the flat projection plane represented by the computer screen, through a virtual environment, encountering various reflective and refractive surfaces along the way, to their original source. The technique copes well with smooth reflective and refractive surfaces but cannot be applied to accurately modelling rays reflected off a rough surface, since the associated scattering increases the complexity of the computation to impractical proportions. See Hearn and Baker (1986).

tal reverberation techniques based upon the use of various combinations of filters, whilst providing some spatial cues, lack the coherence associated with information generated by natural spatial environments.

1.6 The musical perception of sound

Our perceptual systems operate at a number of different levels simultaneously without our necessarily being aware of it and, as described in the previous section, they are attuned to certain patterns of structured information, which are able to trigger clearly defined affordances, such as in the case of a loud explosion. In a natural context, if we merely reflected on the quality of timbre inherent in such a sound, the likelyhood is we would not survive the event.

Another example of a deeply rooted core perceptual mechanism may be found in our ability to perform auditory streaming (Bregman, 1990), which enables us to perceptually group together sounds which are similar in timbre, pitch range, loudness etc. or sounds which occur in close proximity either spatially or temporally. These perceptual mechanisms are found in all listeners, not just those with musical training, and are of natural origin. The fact that we humans employ our perceptual systems in activities such as music, which are non-essential to our immediate survival, does not automatically imply that we can simply 'switch off' these mechanisms at will. In the previous section, it was stated that natural sounds, through the complexity and coherence of their information content, are capable of evoking a clear and consistent 'picture' or *image* of an environment. The term *image* is interpreted by Emmerson (1990) as:

> *...lying somewhere between true synaesthesia with visual image and a more ambiguous complex of auditory, visual and emotional stimuli (Emmerson, 1990, p.17).*

Another way of looking at the perceptual imagery evoked in a listener is in terms of *landscape* defined by Wishart (1990, p.43) as *the source from which we* imagine *the sounds to come*. We hear examples of natural aural landscapes all around us in everyday life. For example the aural landscape of a small room in a house is vastly different to the aural landscape of a cathedral, a busy street or the seaside. More

importantly, as Wishart points out, with the use of digital technology, it is possible to create artifical landscapes which may be realistic or surrealistic in nature.

Wishart defines the most important characteristics of landscape as being: I, the nature of the perceived acoustic space; II, the disposition of sound objects within the space; and III, the recognition of individual sound objects (Wishart, 1990, p.45). We are mostly concerned here with the characteristics of individual sound objects, since the TAO computer music program is not yet capable of simulating acoustic spaces or the disposition of sound objects within a space, although the sounds produced often do possess a strong (but limited) sense of space. However, the wider arguments addressed in this thesis about structured information and the use of cellular models in the generation of such information also apply to the problem of simulating acoustic spaces, since the problem is essentially the same: how to develop models which are able to generate coherently structured information which the listener's auditory perceptual system, through its evolved abilities, will *resonate* with, or *attune* to, leading to the evocation of a convincing sense of space.

1.6.1 Mimesis in electroacoustic music

The sounds and structures found in electroacoustic compositions often mimic aspects of everyday human experience. This is by no means unique to electroacoustic music since traditional note-based music has always drawn inspiration from aspects of human experience such as the rhythmic activity of breathing and walking and the regular beating of the heart. Melodies and phrase structures have their origins in the natural limits set by the human vocal apparatus. This aspect of electroacoustic music is termed *mimesis* and, according to Emmerson (1990), denotes:

> *the imitation not only of nature but also other aspects of human culture not usually associated directly with musical material (Emmerson, 1990, p.17).*

> *Music is always related in some way to human experience, which means that mimesis is always at work even in music regarded as abstract, though such mimesis is notoriously difficult to explain (Smalley, 1990, p.64).*

Emmerson highlights the fact that there are two types of mimesis:

> *...'timbral' mimesis is a direct imitation of the timbre ('colour') of the natural sound, while 'syntactic' mimesis may imitate the relationships between natural events ...(Emmerson, 1990, p.18).*

As with the affordances of particular sounds, this thesis does not attempt to analyse the mimetic qualities of specific sound examples, but merely acknowledges that mimesis has an important part to play in electroacoustic music. It *is* concerned, however, with the fact that musical sounds and sound shapes are often mimetic of natural events. Gibson highlighted the fact that natural events are hierarchical in nature, and Emmerson's two types of mimesis might be viewed as lying at two ends of a continuous spectrum, with timbral mimesis referring to the mimicry of a sound's microstructure, and syntactic mimesis referring to the mimicry of an event's macrostructure (not necessarily a sonic event). More will be said of the micro- and macrostructures of natural sound events in the next section.

1.6.2 Sound categories and their perception

Natural sounds are constrained to evolve according to certain patterns due to the underlying physical processes of which they are a side-effect. Maybe the most striking examples of this are sound categories such as smashing, bubbling, scraping, exploding, colliding, shattering etc. When listened to acousmatically, by suppressing the normal mechanism of source-recognition, we are still able to infer the kind of mechanisms or actions responsible for their production. According to the ecological view of auditory perception, adjectives such as *bubbling*, *scraping* and *smashing* etc. refer not only to the perceptual characteristics which we attach to a sound, but also directly to the patterns of structured information which give rise to these perceptual images in the first place.

Windsor (1995, section 2.1.2) describes a number of experiments concerned with relating the macrostructure of a temporal event to its perceived cause. One experiment described involved simulating the sound of breaking glass, through a process of adjusting the temporal relationships between a number of prerecorded glass impact samples (Warren and Verbrugge, 1984). As Windsor relates, the experiment showed that the macrostructure of such events was sufficient for the sound to be perceived as having been caused by 'breaking' . Another experiment described by Windsor

showed that the elasticity of a bouncing ball may be perceived directly, merely by listening to one period between two successive bounces (Warren, Kim and Husney, 1987). In both of these examples the temporal macrostructure of a sound event was shown to be sufficient for a listener to infer a physical cause for the sound, in a direct, rather than analytical manner.

> *It is clear from this research that a lawful relationship obtains between the physical structure of such events and the acoustic or visual information available to the organism. This relationship is not physical, nor is it imposed by the organism: rather it is picked up through our contact with the lawful behaviour of environmental events, and hence can be described as specifying such events directly (Windsor, 1995, section 2.1.2).*

Whilst this may be true, it is probable that the researchers were not concerned with the vibrancy of the sound events or how suited they would be to a musical context, and it is likely that the sounds used, containing coherently structured information at the macroscopic level juxtaposed with predetermined microstructural information, lacked some element of overall coherence which would determine how convincingly the image of breaking or bouncing was evoked. A real breaking sound, whilst providing these temporal cues in its macrostructure, will also inherit microstructural details which may be *attuned to* by a listener in order to perceive the spatial proximity of the event, and possibly the kind of material being broken. Also, digitally sampled sounds possess a disconcerting ability to sound perfectly real the first time they are heard, and yet increasingly synthetic as they are repeated, whereas natural breaking events, no matter how close in character they may seem, will never be repeated exactly.

> *There is always some degree of recurrence and some degree of nonrecurrence in the flow of ecological events. That is, there are cases of pure repetition, such as the stepping motions of the escapement of a clock and the rotations of its hands, and cases of nonrepetition or novelty, such as cloud formations and the shifting sandbars of a river. Each new sunrise is like the previous one and yet unlike it, and so is each new day. An organism, similarly, is never quite the same as it was before, although it*

> *has rhythms. This rule for events is consistent with the general formula of nonchange underlying change (Gibson, 1979, p.101).*

The coherence associated with real shattering and bouncing events, arises from the fact that the microstructure and macrostructure are causally related by the physical evolution of the process [3], and this is detectable in the auditory information which such events generate. This observation supports the notion of objects and events being *holarchies*, and once we accept this aspect of natural sound events, the question arises: is it possible to synthesise sounds with convincing properties which might merit the use of adjectives such as *shattering*, *bubbling* etc., without recourse to models which explicitly simulate, to some degree, the underlying mechanisms?

Once again it is possible to draw analogies with the visual domain, this time in the computer generation of convincingly natural or *photorealistic* images. For example, in a wide ranging discussion on the nature of virtual reality, Woolley (1992) comments that:

> *Most regard [photorealism] ...as a product of higher resolution and more colours. However, a grainy, black and white picture can look more 'photorealistic' than a TV-quality full-colour computer image. The reason seems to be that the realism lies in the image's content, not the quality of its reproduction. A photorealistic image is one that looks as though whatever it depicts is in some sense real - it concerns, in other words, the sophistication of the computer models, of the descriptions of the virtual objects and landscapes, as much as their rendering (Woolley, 1992, p.240).*

These arguments may be transposed quite easily into the domain of computer generated sounds. 'Higher resolution' and 'more colours' could be said to correspond to greater dynamic range, wider frequency response, and better sampling rate, considerations which although important, are often blown out of proportion by hi-fi enthusiasts and technophiles in general, who place more emphasis on clarity of reproduction than on the actual information content of the music they are reproducing.

[3] This process includes acoustic radiation to the air and subsequent reflections and refractions caused by the acoustic environment.

The comments pertaining to 'the sophistication of the computer models' and of 'the descriptions of the virtual objects and landscapes' apply just as well to the computer generation of sounds as they do to images.

All of the arguments introduced in this section apply to traditional musical sounds as well as the noise categories mentioned. For example the sound of a bowed cello string possesses qualities which point to the fact that it has been generated by some mechanism involving dragging and friction. The sonic characteristics imparted to the sound by this physical process become an integral part of its character. Even if we manage to suppress the mental image of a cellist playing the familiar orchestral instrument and treat the sound as an object in its own right, it will still contain coherently structured information which our auditory perception system will interpret, without any conscious effort on our part. Regardless of the particular notes played by the cellist, an *aggressively* bowed sound will afford a different meaning, to a listener, than a *gently* bowed sound.

It is very difficult to attribute a sound's convincing *bowed* qualities to either its micro- or macrostructure in isolation, since they arise out of the coherence and causal relationship between the micro- and macrostructure. It is equally difficult to synthesise such sounds without recourse to physical models of the interaction between a bow and string. We are accustomed to utilising our perceptual systems in everyday life without thinking about the way in which they function, but it is remarkable to think that by using appropriate models, it is theoretically possible to synthesis sound events containing convincing physical and spatial cues, and even attributes which are suitably described by adjectives such as *aggressive*. That is, if the models are capable of generating complex yet coherently structured information.

1.6.3 Perceived energy sources in natural sounds

The ability to surmise the cause of a sound includes surmising the type of energy source, the amount of energy involved, and how that energy builds up, dissipates or changes in general. As with all the other perceptual mechanisms described, this happens in a direct experiential manner, and in this context the word *energy* has a different meaning to the scientific usage of the word. In other words we *feel* sounds building up, dissipating or moving from one state to another, as well as perceiving

the other qualities described.

> *During the execution of a note, energy input is translated into changes in spectral richness or complexity. When listening to the note we reverse this cause and effect by deducing energetic phenomena from changes in spectral richness ... This aural congruence of spectral and dynamic profiles, and their association with energetic phenomena, are the substance of everyday perceptual practice (Smalley, 1990, p.68).*

Smalley relates the energetic, perceptual aspects of a sound to its spectro-morphology in the above quote, but this could be restated in terms of structured information instead, which is preferable since this thesis attempts to move away from the reductionist, frequency domain view of sound in favour of a more integrated approach encompassing all aspects of sound.

> *During the execution of a note, energy input is translated into changes in the structure of the information generated. When listening to the note we reverse this cause and effect by deducing energetic phenomena from changes in the structure of the information ... This coherence of structured information and its association with energetic phenomena, is the substance of everyday perceptual practice.*

Looking more closely at the musical ramifications of this ability to perceive energetic cues in a sound, Smalley (1990) describes two aspects of sounds, the *gestural* and *textural* aspects. Gesture is defined as being:

> *...action directed away from a previous goal or towards a new goal; it is concerned with the application of energy and its consequences; it is synonymous with intervention, growth and progress, and is married to causality. If we do not know what caused the gesture, at least we can surmise from its energetic profile that it could have been caused, and its spectro-morphology will provide evidence of the nature of such a cause (Smalley, 1990, p.82).*

Texture according to Smalley is defined as being:

> *...concerned with internal behaviour patterning, energy directed inwards or reinjected, self-propagating; once instigated it is seemingly left to its own devices; instead of being provoked to act it merely continues behaving. Where gesture is interventionist, texture is laissez-faire; where gesture is occupied with growth and progress, texture is rapt in contemplation; where gesture is carried by external shape, texture turns to internal activity; where gesture encourages higher-level focus, texture encourages lower-level focus (Smalley, 1990, p.82).*

Physical gesture has always had a part to play in music, and although gestures are not capable of conveying concrete facts or ideas in the same way that a natural language can, they *are* capable of conveying musical information. The most obvious examples include a performer's use of physical gestures when playing an instrument, the orchestral conductor's ability to convey musical intentions via physical gestures, and obviously dance. In this context though, the concept of gesture is extended beyond the limits of human gesture to include energy changes in the external environment. The previously used example of an 'exploding' sound illustrates this point quite well. It seems that the gestural qualities of this sound relate to the kind of human movement which would be induced in the listener by the sound, rather than to any human gesture responsible for the sound in the first place.

In asking the question: 'is there a natural morphology of sound?', Wishart (1990) introduces two more terms *intrinsic morphology* and *imposed morphology.*

Intrinsic morphology is defined as follows:

> *Most sound-objects which we encounter in conventional music have a stable intrinsic morphology. Once the sound is initiated it settles extremely rapidly on a fixed pitch, a fixed noise band or more generally on a fixed 'mass' as in the case of bell-like or drum-like sounds with inharmonic partials. Furthermore, most physical systems will require a continual (either continuous or iterative) energy input to continue to produce the sound. Others (such as bells or metal rods), however, have internal resonating properties which cause the sound energy to be emitted slowly with ever decreasing amplitude after an initial brief energy input (Wishart, 1990,*

p.57).

Imposed morphology is defined as that which is imposed by external energy input to a system. These notions are similar in spirit to Smalley's definitions of gesture and texture and once again support the idea that the energetic nature of a sound, both internal and external, and its imagined physical cause are of musical importance. One way of paraphrasing both sets of definitions is to say that *texture* and *intrinsic morphology* somehow reflect the internal identity of an object or system used to produce a sound whilst *gesture* and *imposed morphology* are indicative of external energy applied to the system.

Interestingly, Wishart also states that:

> *Sounds undergoing continuous excitation can carry a great deal of information about the exciting source (this is why sounds generated by continuous physiological human action - such as bowing or blowing - are more 'lively' than sounds emanating, unmediated, from electrical circuits in synthesisers) (Wishart, 1990, p.58).*

Whilst this is partly true, a central idea of this thesis is that *liveliness* does not depend on human intervention but is actually a much more fundamental aspect of the behaviour of naturally occurring dynamical systems. If appropriate synthesis models are used, the sounds produced will possess this liveliness even though they may have been produced out of real time and without the use of any direct, human physical gestures. It is worth adding that the gestural and textural aspects of a sound are often capable of evoking a sense of motion, even if the sound object itself does not physically move.

> *Spectro-morphological design on its own ... in controlling the spectral and dynamic shaping, creates real and imagined motions without the need for actual movement in space (Smalley, 1990, p.73).*

1.6.4 Summary

This discussion has covered some of the main perceptual mechanisms which seem to operate in all listeners, regardless of musical training, including: auditory streaming;

the ability to surmise the physical origin of a sound; the ability to take spatial and energetic cues from sounds; and the evocation of the gestural and textural aspects of sound. It has also touched upon Gibson's notion of affordances. According to the ecological view of auditory perception, all of these mechanisms are due, partly at least, to information which is already contained within the sounds, i.e. the subjective perceptual images evoked by sounds are not purely internal to us but relate directly to the structure inherent in those sounds. This argument supports the personal observations given at the beginning of this chapter on the appealing characteristics of natural sounds, and whilst none of the quotes in this chapter are concerned with explicitly criticising synthesised sounds, the various arguments put forward support the idea that natural sounds, because of the central part they play in our perception of the world around us, have a special resonance with listeners. They are often capable of evoking stronger images than synthesised sounds, unless the synthesis models used are sophisticated enough to generate a similar level of complexity as our auditory perceptual system has come to expect from the natural environment.

1.7 Thesis structure

This introductory chapter has described the background to the thesis and the personal motivation for pursuing it. It has also placed the program of research in context with a number of contemporary ideas relating to the musical and everyday perceptual attributes of sound. The important relationship between natural sounds (and events) and the sounds and sound shapes found in electroacoustic music has been highlighted.

Chapter 2 examines the notion of structured information from the point of view of the underlying laws of Nature which lead to its creation. A survey of contemporary scientific ideas which relate to the behaviour of naturally occurring dynamical systems is conducted, covering such areas as chaos theory, complexity, complex dynamical systems and emergent behaviour. This chapter introduces different types of cellular models, with some examples of their behaviour, and lists their appealing characteristics.

Chapter 3 presents a survey of existing synthesis techniques and describes an existing computer music program, Csound, which is based around the traditional reduction-

ist, unit generator approach to sound synthesis. A number of physical modelling techniques such as modal synthesis and the MOSAIC computer music program, CORDIS-ANIMA, and digital waveguide synthesis are also described. The chapter concludes with a set of criteria by which digital synthesis techniques may be judged on an equal footing.

Chapter 4 describes the cellular computer model which forms the basis for the TAO computer music program. This piece of software was developed as part of the program of research described, and examples are given of the kind of wave phenomena which emerge naturally from the model, and the structural possibilities it affords.

Chapter 5 describes TAO's user interface, a script language, which enables the user to create and control instruments. This language comprises both an *orchestra* language, and *score* language based upon the idea of sounds as hierarchically nested events.

Chapter 6 gives various practical examples of TAO instruments and their associated behaviour. Some of the examples illustrate, graphically, general points relating to the strategies which may be employed when designing instruments, whilst others are supported by sound examples listed in appendix C.

Chapter 7 presents a summary of the key ideas presented in previous chapters, assesses how successfully the hypothesis has been supported, and concludes the thesis.

Appendix A gives a brief user manual, including how to install the TAO system and get it up and running. This appendix is not intended as a tutorial since many practical examples are given in chapter 6 and appendix C.

Appendix B contains a complete reference manual for TAO's script language.

Appendix C describes the sound examples which accompany this thesis, and gives the TAO scripts which were used to produced them.

Appendices D and E describe the implementation of the system in detail, and appendix F gives details of the mathematical model used to simulate the interaction of a virtual bow with an instrument.

Finally appendix G gives a complete listing of the implementation code.

Chapter 2

The complexity of natural systems

2.1 Introduction

In the previous chapter the notion of *structured information* was introduced in the context of auditory perception, and a number of specific perceptual attributes of sound were highlighted. In this chapter we move away from the perceptual aspect of sound and turn instead to a completely different set of questions:

1. What are the main factors affecting the generation of patterns of information in Nature?

2. Is there a way to relate subjective adjectives such as *organic*, *vibrant* and *lively* to the behaviour of natural systems, and the information they generate?

3. If we wish to synthesise sounds possessing the natural characteristics mentioned, what kind of computer models are available?

We will begin to answer these questions with a review of a theory which has had a massive impact on scientific thinking in the last three decades, *Chaos theory*. Chaos theory has a large part to play in the ideas presented in this thesis, although in an indirect way, and it is suggested that attempts to apply it to the organisation of sound have often 'missed the point' of the theory in the past, applying it in inappropriate

contexts. This view is supported by the composer Barry Truax who comments on the popular fractal images which have come to be inextricably associated with chaos theory:

> *What seems to have attracted the most public attention is the fact that [behaviour in non-linear systems] displays fractal properties and self-similarity across different scales. Various composers have attempted to find musical analogies to the famous computer graphic fractals, but most of these attempts have involved a mapping onto macro-level compositional parameters as in the works of Larry Austin ... Charles Dodge ... Bruno Degazio (Truax, 1990a, p.100).*

This thesis places chaos theory in the wider context of attempting to explain how it is that complex dynamical systems such as acoustic musical instruments can exhibit appealing behaviour which might be called *organic*. It is proposed that whilst chaos theory has provided a suitable paradigm shift for the re-appraisal of such questions, it is only one part of a much larger equation. As Truax states:

> *Given that acoustical systems are prime examples of dissipative dynamical systems, it is surprising that more work has not been done to investigate fundamental relationships between chaotic behaviour and acoustical models of sound. The unsolved problems in the field of acoustics seem ripe for such basic re-examination as those in other fields (e.g turbulence) which have been completely reformulated in recent years*
>
> *[Gleick] suggests that scientists have traditionally been trained to think in terms of linear systems and solvable linear differential equations as the norm, and to ignore the irregularities of any complex behaviour that cannot be explained by them. Are not the 'difficult' problems of acoustic phenomena treated similarly? Here we cite onset transients, departures from pure harmonicity, and the complex behaviour of certain types of environmental sounds as examples whose full explanation has eluded researchers (Truax, 1990a, p.100).*

We begin the discussion then with a brief recap on the main features of chaos theory, but other topics covered include complex dynamical systems, emergent behaviour, phase space portraits and attractors, and the central theme of this thesis, cellular models.

2.2 Chaos theory

It will not be necessary to go into too much detail here about chaos theory since it is already well documented, the most widely read and accessible account of its historical development and implications being (Gleick, 1991a). It is, however, useful to briefly recap on the theory at an intuitive rather than mathematical level, since it is central to the questions addressed by this thesis such as what makes sounds *vibrant*, *organic*, and *coherent* and how it is that natural sounds are able to exhibit endless variations of patterned and yet unpredictable behaviour. Chaos theory is unlike previous Western scientific theories in that it deals with explaining the behaviour of natural systems on all scales from the very smallest to the very largest. It is concerned with cloud formations, weather systems, turbulent fluid flow, the formation of mountains etc. It turns out that in all of these systems there are underlying universal laws at work which give rise to similar patterns of behaviour, even if the patterns seem to be unrelated at first glance.

In the words of Gleick (1991a):

> *Where chaos begins, classical science stops. For as long as the world has had physicists inquiring into the laws of nature, it has suffered a special ignorance about disorder in the atmosphere, in the turbulent sea, in the fluctuations of wildlife populations, in the oscillations of the heart and brain. The irregular side of nature, the discontinuous and erratic side – these have been puzzles to science, or worse, monstrosities (Gleick, 1991a, p.3).*

> *The simplest systems are now seen to create extraordinarily difficult problems of predictability. Yet order arises spontaneously in those systems – chaos and order together. Only a new kind of science could begin to cross the great gulf between what one thing does – one water molecule, one cell*

of heart tissue, one neuron – and what millions of them do (Gleick, 1991a, p.8).

Chaos has been defined as:

The complicated, aperiodic, attracting orbits of certain dynamical systems. Philip Holmes

A kind of order without periodicity.
A newly recognised and ubiquitous class of natural phenomena. Hao Bai-Lin

The irregular, unpredictable behaviour of deterministic, nonlinear dynamical systems. Roderick V. Jensen

Dynamics freed at last from the shackles of order and predictability ... Systems liberated to randomly explore their every dynamical possibility ... Exciting variety, richness of choice, a cornucopia of opportunity. Joseph Ford

2.3 The phenomenon of bifurcation

In order to develop a rigorous mathematical understanding of non-chaotic and chaotic behaviour and the transition from one to the other we can focus our attention upon the simplest system which can be made to produce this chaotic behaviour. In mathematical terms the simplest and most frequently quoted example of an equation capable of chaotic behaviour is the *logistic difference equation* $x_{next} = \lambda x(1-x)$. The essential feature of this equation is that it is iterative and involves feedback. An initial value is chosen for x between zero and one, and a fixed value is chosen for λ. Using these values, a new value is calculated for x which is then fed back into the equation to calculate the next value, ad infinitum. For certain low values of λ the successive values of x very quickly settle down to a single number. For slightly larger values of λ the succession of x values eventually oscillate between two alternate numbers after settling down. Increasing the value of λ further leads to the values of x oscillating between four values, then eight, then sixteen etc. Suddenly when a critical value of λ is reached the value of x seems to jump about at random, never settling down to a single value or set of values. This phenomenon is known as *period doubling* or *bifurcation*.

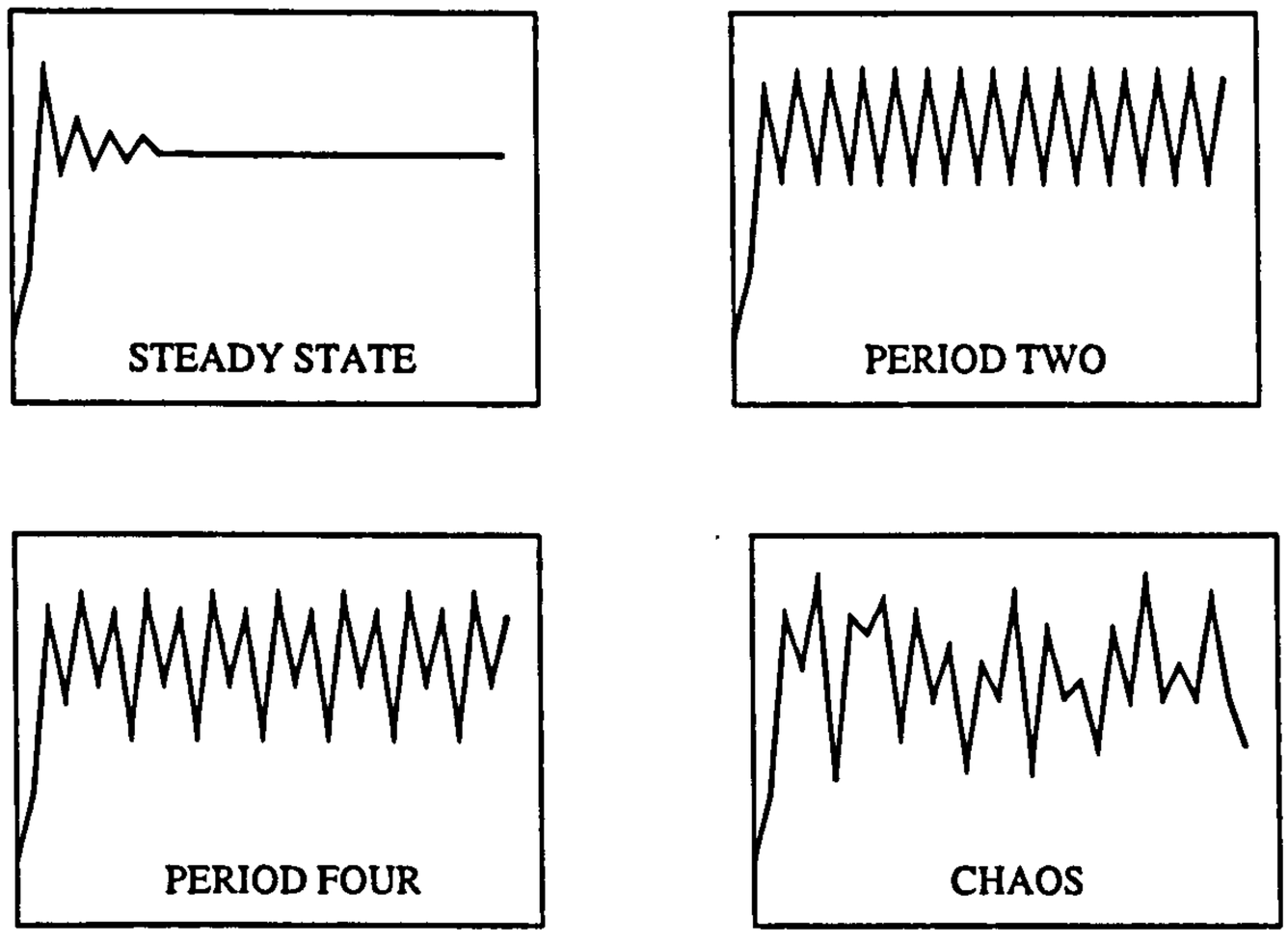


Figure 2.1: The phenomenon of period-doubling or bifurcation

This process is illustrated in figure 2.1 which shows the successive values of x which occur for various values of λ, but there is a way to combine all of the images in this figure into a single diagram: a *bifurcation diagram*. By plotting the final values of x after a fixed number of iterations against various values of λ we obtain the bifurcation diagram shown in figure 2.2. For values of λ up to about 3, the iterations produce one stable value for x. For values of λ between about 3 and 3.45 the value of x alternates between two numbers and so on. For values of λ greater than approximately 3.57 the stream of values produced by the equation never settle down into any kind of pattern, they behave chaotically. The image (a) at the top of figure 2.2 contains a small shaded region which is enlarged in (b). Image (b) then has a shaded region of its own which is further enlarged in (c). This shows the self-similarity of the diagram. Figure 2.3 starts from the same point but shows a different sequence of enlargements, highlighting the fact that islands of order exist within the chaos. Some of these islands possess periods of three, five, seven etc. rather than two, four, eight etc.

The logistic difference equation is useful for distilling the essence of chaotic behaviour out from other distracting elements, since it is the simplest possible system involving feedback. But it must be remembered that chaos theory came about because of the empirical observation of such period doubling in real systems, and often very complex

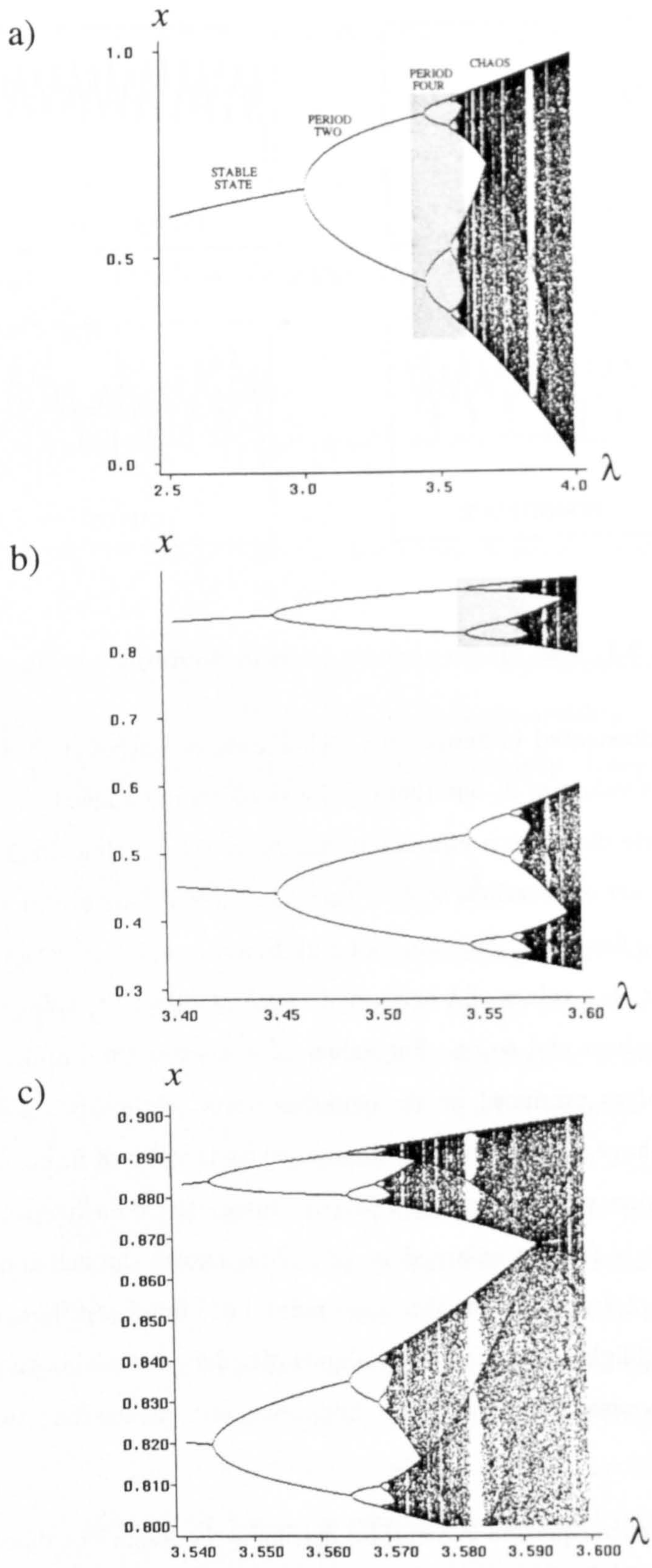


Figure 2.2: Bifurcation diagram with selectively enlarged regions

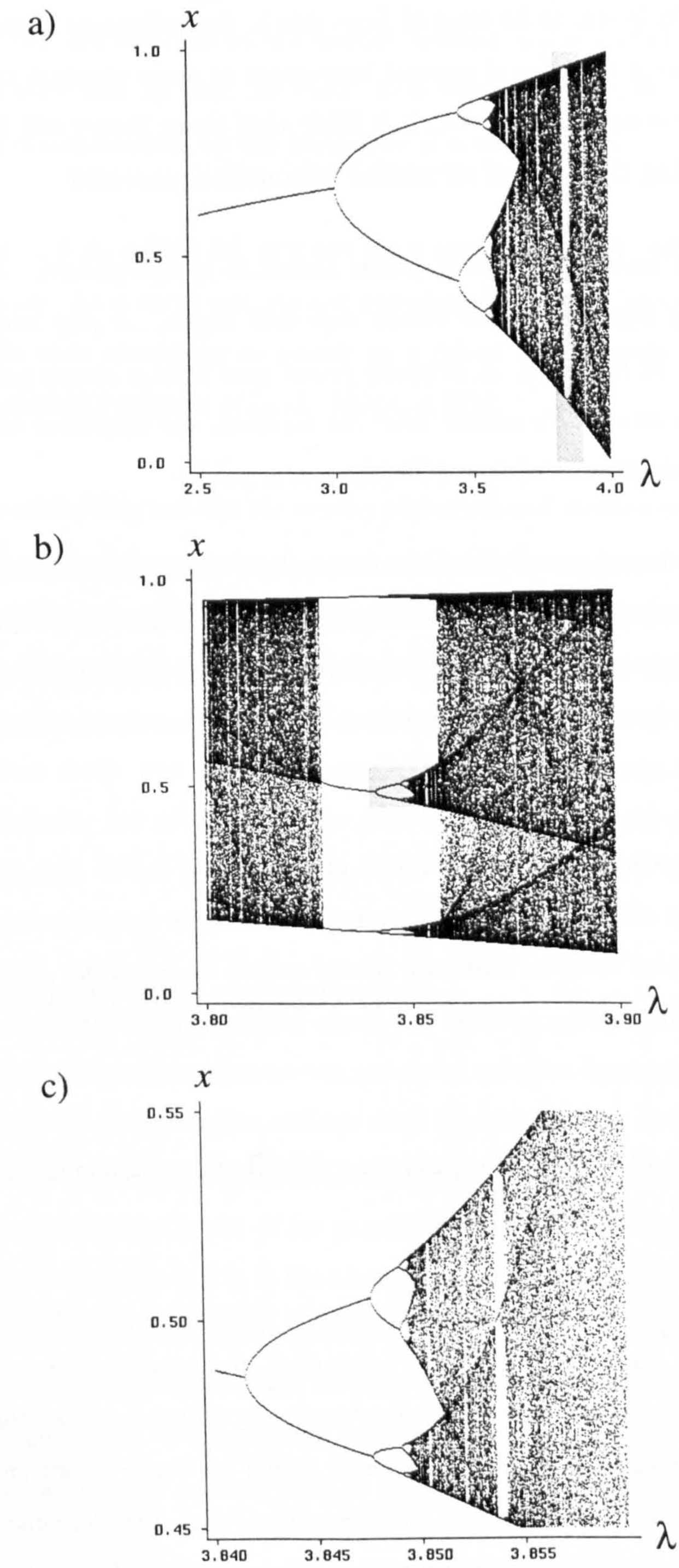


Figure 2.3: Islands of order within chaos

ones. The main lesson to be learned from this is that whatever system we care to observe, whether a traditional musical instrument or some physical process such as bubbling, shattering, scraping etc., it is likely that chaos theory will have a part to play in explaining the kinds of structured information generated.

> *Now that science is looking, chaos seems to be everywhere. A rising column of cigarette smoke breaks into wild swirls. A flag snaps back and forth in the wind. A dripping faucet goes from a steady pattern to a random one ...No matter what the medium, the behaviour obeys the same newly discovered laws (Gleick, 1991a, p.5).*

If we return for a moment to Gibson's observations on the hierarchical organisation of Nature and its mixture of persistence and non-persistence we see that the points which are highlighted as being fundamental from a perceptual point of view correspond very closely to those which are dealt with by chaos theory. For example the hierarchical and sometimes self-similar organisation:

> *Just as physical reality has structure at all levels of metric size, so it has structure at all levels of metric duration ...And once more it is important to realise that smaller units are nested within larger units. There are events within events, as there are forms within forms ...(Gibson, 1979, p.12)*

and the ability of all dynamical systems to exhibit both order and chaos, depending upon the amount of feedback present:

> *The environment normally manifests some things that persist and some that do not, some features that are invariant and some that are variant. A wholly invariant environment, unchanging in all parts and motionless, would be completely rigid and obviously would no longer be an environment ...At the other extreme, an environment that was changing in all parts and was wholly variant, consisting only of swirling clouds of matter, would also not be an environment. In both extreme cases there would be space, time, matter, and energy, but there would be no habitat (Gibson, 1979, p.14).*

Gibson's second comment suggests that there is something fundamental about environments which are situated at the border between order and chaos, something which is intimately tied up with life itself. This view is supported by the following quotes, the first commenting on the behaviour of a water faucet:

> *If you turn it up a little bit, you can see a regime where the pitter-patter is irregular. As it turns out, its not a predictable pattern beyond a short time. So even something as simple as a faucet can generate a pattern that is eternally creative (Gleick, 1991a, p.262).*

> *...unpredictability was not the reason physicists and mathematicians began taking pendulums seriously again in the sixties and seventies. Unpredictability was only the attention grabber. Those studying chaotic dynamics discovered that the disorderly behaviour of simple systems acted as a creative process. It generated complexity: richly organised patterns, sometimes stable and sometimes unstable, sometimes finite and sometimes infinite, but always with the fascination of living things (Gleick, 1991a, p.43).*

2.4 Simplicity and complexity

Before proceeding to a discussion on the nature of complex dynamical systems, it is a worthwhile exercise to examine notions such as *information*, *simplicity* and *complexity* more closely. We often use these words in an everyday context without really being aware of what they mean. If we perceive a stream of structured information, whether visual or aural, what is it that makes one stream more information-rich or complex than another? Gell-Mann (1995) devotes a whole book to such questions, and defines some useful terminology. Broadly speaking, Gell-Mann provides further evidence for the notion that a balance between order and chaos is fundamental for life and *complex adaptive systems* such as humans to exist at all:

> *The environment must exhibit sufficient regularity for the systems to exploit for learning and adapting, but at the same time not so much regularity that nothing happens. For example, if the environment in question is*

the center of the sun, at a temperature of tens of millions of degrees, there is almost total randomness ...nothing like life can exist. Nor can there be such a thing as life if the environment is a perfect crystal at a temperature of absolute zero ...For a complex adaptive system to function, conditions are required that are intermediate between order and disorder.

Conditions in between order and disorder characterize not only the environment in which life can arise, but also life itself, with its high effective complexity and great depth (Gell-Mann, 1995, p.116).

The terms *effective complexity* and *depth* are essentially both derived from computational theories of information which relate the useful content in a stream of information to: (a) the length of the smallest program which would be capable of reproducing the stream in all its original detail; and (b) the length of time that this smallest program would have to be left to run in order for it to compute the stream.

In more intuitive terms, effective complexity relates to the task of identifying the regularities in a stream, and compressing them into some kind of *schema* about the stream's behaviour. Such a schema may be used to make predictions about a system's future behaviour. If a stream exhibits too many regularities, then building such a schema becomes a trivial task. Conversely, building a schema for a completely random system is impossible since there are no regularities whatsoever. The important point is that a system of the former type is often of little interest since it is, by definition, completely *predictable*. Conversely, a system of the latter type is often uninteresting since it is *unpredictable* but in an entirely predictable manner. Its associated schema is therefore also rather trivial. For systems which lie in between the two classes described above, the effective complexity reaches a maximum, since there are enough regularities to make the construction of a schema worthwhile, but there will always be surprises which force the schema to be updated, thus leading to it becoming more lengthy.

There are classes of systems, however, which although very complex, do not operate at 'the edge of chaos' and therefore do not have a high effective complexity. How is it then that such systems may also be labelled 'complex'? This question leads to the idea of *depth*, which is a measure not of how long the schema itself is, but of how long it would take to proceed from the schema to a full blown description of

the original stream of information. This point is equally important since it tells us that some streams of information, although the result of processes which are regular enough for appropriate schemas to be constructed, may rely upon the shear amount of information processing which has gone into their creation for their associated complexity. It also warns us that there may be no short-cuts when modelling natural streams of information, such as those coming from musical instruments, if we wish the result to be truly complex[1].

Gell-Mann introduces both terms not as abstract mathematical concepts, bearing little relationship to our experience of the world, but in an attempt to elicit the precise meaning of the word *complexity*.

2.5 Complex dynamical systems and emergent behaviour

So far we have seen the peculiar structure which lies behind the transition from ordered to disordered behaviour in a simple dynamical system. We have also seen some evidence in support of the hypothesis that dynamical systems act as creative sources of information when operating in a regime balanced at the border between order and chaos. The universality of chaos theory means that the bifurcation diagrams shown earlier in this chapter apply to the behaviour of more complex systems also. A complex dynamical system may exhibit more intricate spatial and temporal patterns, and there may not be a single parameter equivalent to λ, but essentially a complex system holds the same potential for ordered and chaotic behaviour and bifurcation as a simple one.

A complex dynamic system is formally defined by the following characteristics (Beyls, 1989):

- It has a large number of similar simple elements
- All elements evolve in parallel over time
- The same external rule applies to all elements simultaneously
- Any element performs local interactions only

[1] A point which is particularly salient in the field of sound synthesis, since it is very easy for the goal of real-time synthesis to override more musical considerations about the quality of the sounds produced.

- The systems exhibits emergent global properties

Complex dynamic systems occur throughout nature in many different forms and at many different scales. Some immediately observable examples which are caused by such systems of one kind or another include:

- cloud formations;
- the Earth's climate system;
- the swirling patterns occurring in a rising smoke column;
- wave patterns and turbulence in fluids;
- flocks and herds of animals;

Apart from the category of systems described above, consisting of large numbers of identical elements interacting on a local basis, we will extend the definition to include systems such as acoustic musical instruments also, since they are by definition complex and dynamic. Classical science may tell us that they are not complex, and that their behaviour is well understood, but in practice it seems that synthetic sounds based upon classical models rarely possess the same depth as their real counterparts. In practice, musical instruments produce sounds as an emergent behaviour, although they are not quite so clear cut and homogeneous in nature as the examples given above.

Whilst chaos theory makes it clear that even the simplest systems such as the logistic difference equation described in the last section are capable of extremely complex behaviour, it also begs the opposite question: how is it that extremely complex systems are capable of spontaneously organising themselves into highly ordered global patterns of behaviour? This leads to the concept of *emergent behaviour*, i.e. the ability of a system consisting of many agents interacting on a local basis to spontaneously organise itself without outside intervention. Such systems are greater than the sum of their parts from a perceptual point of view, since we are able to perceive the patterns formed as a whole.

If we take the example of cloud formations, in practice we observe some clouds which are highly organised, producing wave patterns which stretch from horizon to horizon,

whilst at other times there seems to be no overall coherence, as on a stormy day, although even in this example, when viewed from a satellite there is still significant self-organisation on a larger scale in the form of depressions or large swirling vortices of air. The important point to note is that although complex dynamical systems and simple dynamical systems both obey the laws of chaos, the simple systems do not have many degrees of freedom, as in the case of the logistic difference equation, and are therefore not capable of expressing the laws of chaos in very subtle and interesting ways. Complex dynamical systems, on the other hand, have a great many more degrees of freedom and therefore manifest the laws of chaos in more interesting ways.

2.6 Phase space and attractors

A concept which appears frequently in the study of complex dynamical systems is that of *phase space.* A phase space is a multi-dimensional map of a dynamic system's behaviour. If a system is characterised at any point in time by a set of *state variables*, then by plotting the changing values of these state variables over time, with each variable having its own axis, an abstract picture of the system's evolution may be produced: a *phase space portrait.*

> *Phase space gives a way of turning numbers into pictures, abstracting every bit of essential information from a system of moving parts, mechanical or fluid, and making a flexible road map to all its possibilities (Gleick, 1991a, p.134).*

A simple, damped harmonic system such as a pendulum will always swing back and forth until it comes to rest at the same point in its phase space, i.e. the point of minimum potential energy, and this point is referred to as the *point attractor* of the system. Conversely, a system which exhibits a tendency to oscillate regularly with a fixed period possesses what is known as a *periodic attractor*, and its phase space portrait appears as a closed loop, when the system is left to its own devices. Once again, if such a system is set in motion from a point in its phase space which doesn't lie on the attractor, the system will eventually settle back into its dynamically stable mode of oscillation. Phase space portraits are useful for eliciting a system's

character, since they provide qualitative, pictorial representations of a system's long term behaviour, rather than quantitative descriptions at specific moments in time.

2.6.1 Strange attractors

If a dynamical system is operating in a chaotic regime then it exhibits what is known as a *strange attractor* (Gleick, 1991a; Hofstadter, 1986). Figure 2.4 gives an example of a strange attractor, the Lorenz attractor, after Edward Lorenz who was the first to discover chaotic behaviour in a simple dynamical system. Lorenz was interested in understanding the behaviour of the weather through simplified and idealised sets of equations which nevertheless captured some of the essence of the convection flows responsible for real weather patterns. He discovered, whilst re-running a computer simulation of one of these mathematical models from half-way through a previous run, feeding the intermediate starting values for the second simulation in by hand from a computer printout, that the model quickly diverged from its previous pattern of behaviour.

According to the classical, deterministic view of the world, this should not have occurred, since small influences on a system were assumed to average out, not affecting the global behaviour of the system. However, Lorenz' discovery showed that dynamical systems may be sensitive to initial conditions and that in a chaotic regime of behaviour, no matter how accurately we capture the initial conditions of a such a system, it will always be impossible to predict its evolution, since small influences will grow larger and larger, making the system's behaviour divergent. This effect is known as the *butterfly effect* since, in theory, a butterfly flapping its wings on one side of the planet could be responsible for the development of a hurricane elsewhere.

Lorenz recognised that the chaotic behaviour his model produced was not purely random but possessed a strange, hidden order. In order to elicit this hidden structure to chaotic behaviour, Lorenz began to look for simpler and simpler sets of equations which would produce the desired behaviour. The phase space portrait in figure 2.4 shows the behaviour of one of these mathematical models, which reduces the process of convection to a simple, water-wheel like model with a single degree of rotational freedom and discrete cells or buckets containing fluid which condenses and evaporates under the influence of a heat source coming from above. The graph

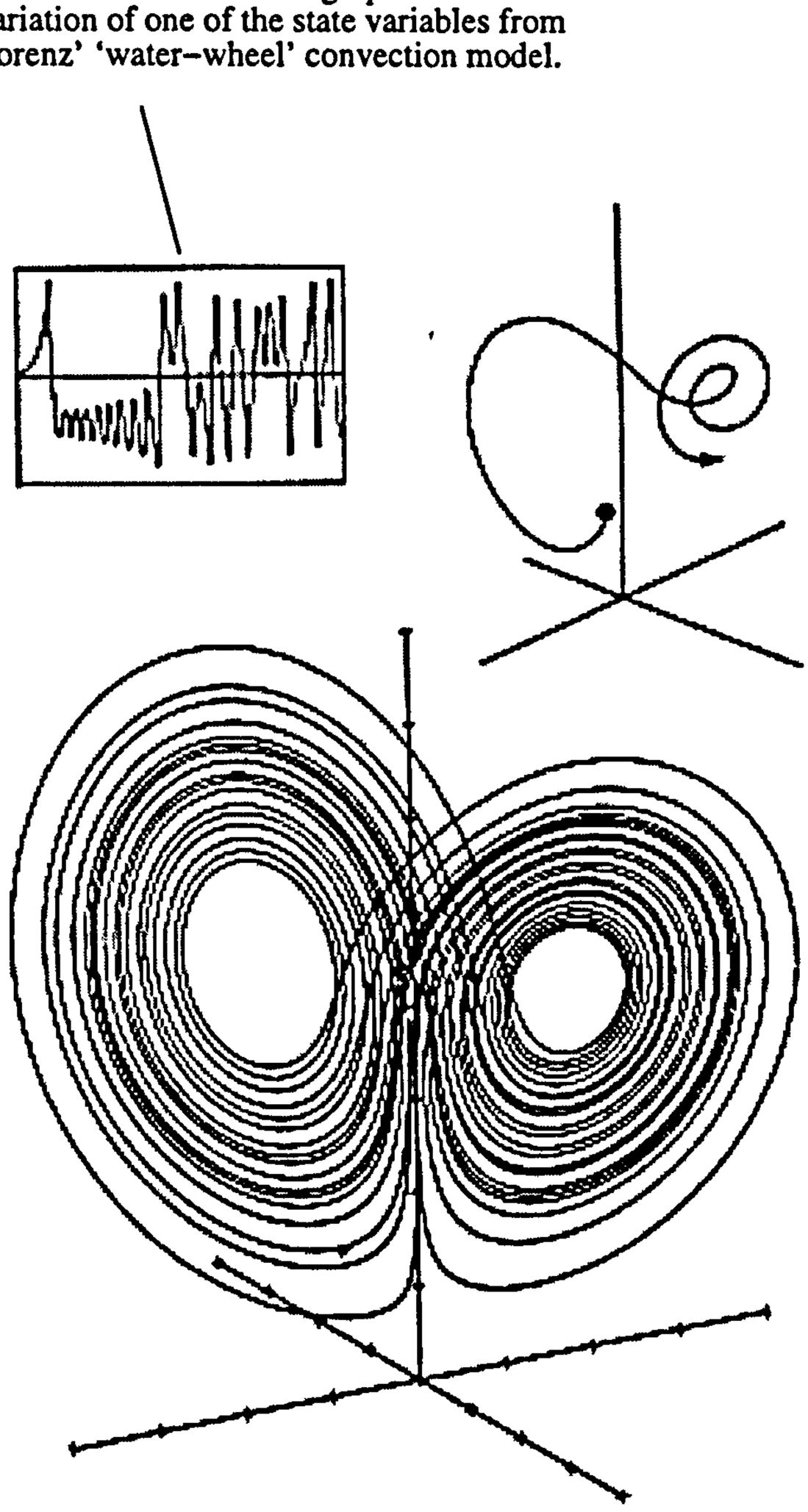


Figure 2.4: An example of a strange attractor (from Gleick, 1991)

situated at the top left of the figure represents a traditional time-domain representation of the behaviour of just one of the variables in Lorenz' model, but it is only when three variables are plotted in phase space over an extended period of time that the strangely ordered and yet aperiodic behaviour of the system becomes clear. The two 'wings' of the attractor illustrate the system's ability to continue flowing in one direction for a period of time and then, without warning, suddenly reverse the direction of convective flow. More significantly, the system never traces the same path twice, and is therefore capable of continually surprising.

It is well understood that the sounds produced by pitched musical instruments are rarely purely periodic in nature, and even if such instruments do not exhibit strange attractors, this implies that they must operate at a regime situated somewhere in between periodic and aperiodic behaviour. The sensitivity of a system to initial conditions and the ability for minute external influences to effect larger changes in its behaviour do have some relevance to acoustic musical instruments as Woodhouse (1992) points out:

> *One is frequently confronted with rather subtle physical effects that result in sounds which our auditory system is able to process with remarkable acuity. It is never safe to assume that because a particular effect is small in terms of physical measurements, it will not be significant to a skilled performing musician (Woodhouse, 1992, p.43).*

2.6.2 Identity and transient behaviour

It has long been understood that the *transients* found in instrumental sounds contribute a great deal to the overall character and expressiveness of the instruments. The language of dynamical systems provides us with some useful terminology for discussing the phenomenon of *transient behaviour.* Firstly, a system's attractor represents a *tendency* towards certain patterns of behaviour, without actually forcing the system to always operate in that way, and therefore represents, in a very deep way, the system's *identity.* Whenever such a system is excited by the application of external energy, transient behaviour occurs. This transient behaviour is caused by the system being pushed to a point in its phase space which is away from the usual path dictated by the system's attractor. Once the excitation has disappeared,

though, the system will return, over a finite time interval, to its usual dynamic equilibrium (static, in the case of a point attractor). More severe excitations will tend to push the system to more remote areas of its phase space and therefore lead to more pronounced transients, but given enough time, these transients will always die away.

The reader will remember that in the previous chapter, the *gestural* and *textural* aspects of sound were described. These very musical concepts have a close relationship to the ideas described here, since the textural aspect of a sound (that which is caused by the sound following its own internal behaviour with no external influence) seems to correspond to the direct perception of a system's identity, through the patterns of structured information it generates. These patterns are, in turn, governed by the attractor of the system responsible for the sound. The gestural aspect (that which seems to have been caused by the application of external energy) seems to correspond to the direct perception of any deviation from the system's attractor, i.e. the auditory perceptual system recognises an interruption in the invariant features produced by the system.

One problem associated with the notion of attractors is that for dissipative systems, i.e. those with point attractors, the attractor only tells us that the system will eventually come to rest at the same point. It tells us nothing about the actual path which the system will take to reach that point. However, the phase space portraits of such systems, e.g. percussive instruments, will still possess a certain character no matter how they are excited and how pronounced the transient behaviour is. But in general, regardless of the particular type of attractor possessed by a system, any deviation from this attractor is perceived by an observer as transient behaviour.

2.6.3 Appealing characteristics of complex dynamical systems

So far we have seen a number of characteristics of complex dynamical systems which make them suitable sources of inspiration for new sound synthesis techniques. Firstly, they have strong identities; secondly, they have the potential to act as creative sources of information, maintaining their identity whilst continually throwing up surprises; and finally, they are compatible with very musical notions such as the *transient behaviour* associated with acoustic instruments, and the *gestural* and *tex-*

tural aspects of sound. In addition to these points, since the behaviour they exhibit occurs as an emergent property, they possess a certain robustness due to their holistic nature. A complex dynamical system may be excited at several different locations simultaneously and may have its characteristics altered on a local basis, and yet it will always retain a strong identity, or create a new one for itself if the alterations are drastic enough. This means that the structured information generated by such systems will always retain a high degree of coherence which, as was proposed in chapter 1, is essential for the evocation of strongly focussed imagery in a perceiver.

2.7 Cellular models: a modelling paradigm

A cellular model is defined in this thesis as:

> A model in which many simple agents interact on a local basis with each other according to well defined rules. Such models are usually updated in discrete time steps and the application of a *cellular update rule* on a local basis both temporally and spatially leads to global patterns of behaviour: *emergent behaviour.*

Cellular models include *cellular automata*, *finite difference models*, *finite element models* and *particle models*, all of which are described below.

At the beginning of this chapter Truax (1990a) commented on the fact that scientists have traditionally been trained to think of solvable linear differential equations as the 'norm'. Toffoli and Margolis (1987) support this view in the introduction to their book *Cellular automata machines - a new environment for modelling*:

> *...the development of mathematics in a certain period of time reflects to a much greater extent than many would suspect the nature of the computational resources available at that time. In the past three centuries, enormous emphasis has been given to (1) models that are defined and well-behaved in a* continuum, *(2) models that are* linear, *and (3) models entailing a small number of* lumped *variables. This emphasis does not reflect a preference of nature, but rather the fact that the human brain, aided only with a pencil and paper, performs best when it handles a small*

> *number of symbolic tokens having substantial conceptual depth ...in this context, one tends to concentrate effort on problems which are likely to yield a symbolic, closed form solution (Toffoli and Margolis, 1987, p.142).*

As computer technology has become faster and computer graphics have become more widely available the classical goal of finding 'closed form solutions' by solving equations has been replaced, to an extent, with the use of digital computers as tools for direct experimentation, leading to an experimental approach lying halfway between conventional laboratory work and classical mathematical modelling. A growing number of researchers in a variety of disciplines have been influenced by the associated paradigm shift (Waldrop, 1994) and have switched to this approach to understanding the behaviour of complex systems. Since it is more intuitive than analytical, more qualitative than quantitative, it has been likened to playing a musical instrument (Toffoli and Margolis, 1987), whereby the experimenter 'plays' the model and 'listens' to the resulting behaviour, thereby gaining a 'feel' for the behaviour of a system, even if its precise behaviour cannot be predicted.

Cellular models, because of their spatial distribution, are ideally suited to computer graphics visualisation, and dispense with the analytical approach to understanding a system in favour of a more direct approach which relies on our ability to perceive patterns in complex evolving sets of data.

2.7.1 Cellular automata

Cellular automata represent the simplest example of cellular models. A cellular automaton consists of a regular array of cells, each cell containing a discrete value. Cell values are updated in discrete time steps according to simple deterministic rules which take account of each cell's previous value and the values of its neighbouring cells. A more formal definition of the characteristics of cellular automata is given by Wolfram (1986):

Discrete in space. They consist of a discrete grid of spatial cells or sites.

Discrete in time. The value of each time cell is updated in a sequence of discrete time steps.

Discrete states. Each cell has a finite number of possible values.

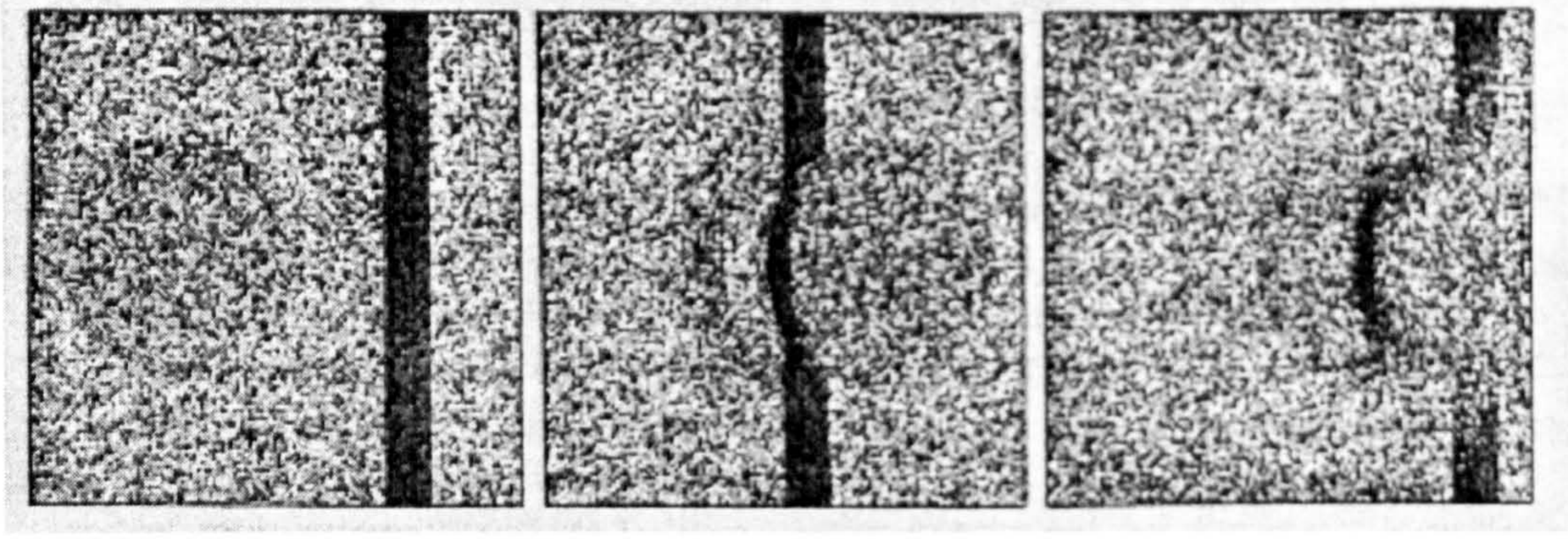


Figure 2.5: CA model of wave optics: refraction through a spherical lens (from Toffoli and Margolus, 1987)

Homogeneous. All cells are identical, and are arranged in a regular array.

Synchronous updating. All cell values are updated in synchrony, each depending upon the previous values of neighbouring cells.

Deterministic rule. Each cell value is updated according to a fixed, deterministic, rule.

Spatially local rule. The rule at each site depends only on the values of a local neighbourhood of sites around it.

Temporally local rule. The rule for the new value of a site depends only on values for a fixed number of preceding steps.

Cellular automata have been used to simulate a variety of natural phenomena such as biological systems (Green, 1990), fluid dynamics (Lakshmi, 1989), crystal growth (Toffoli and Margolis, 1987). They also have applications in digital image processing for feature extraction (Lewis, 1990). For a comprehensive survey of the applications of cellular automata see Toffoli and Margolis (1987) and Wolfram (1986).

Figures 2.5, 2.6, 2.7 and 2.8 show examples of the flexibility and generality of the cellular approach to modelling natural phenomena. Figure 2.5 shows a simulation of wave optics and the refraction of a wavefront by a spherical lens. This model

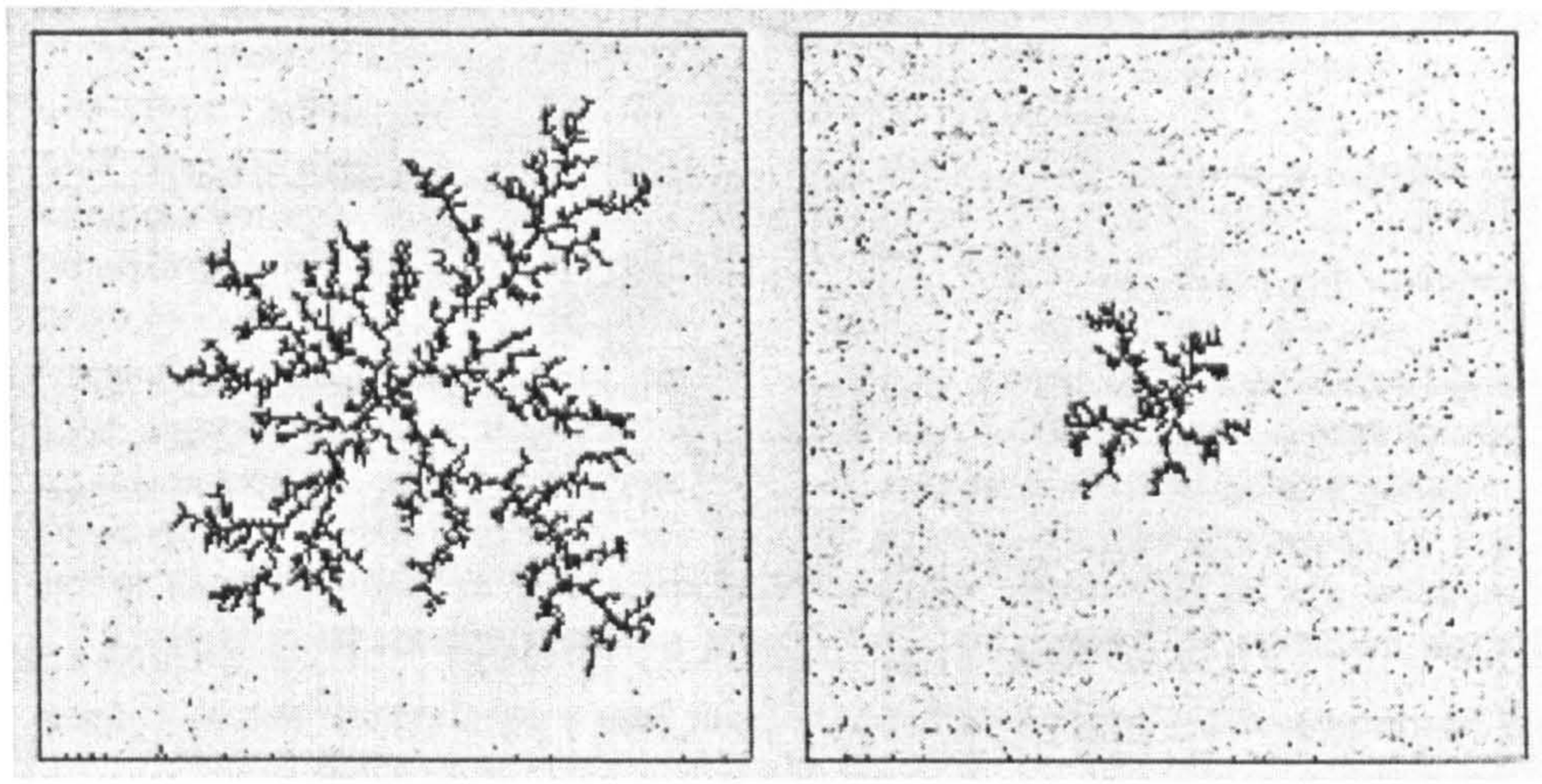


Figure 2.6: CA model of dendritic growth (from Toffoli and Margolus, 1987)

actually consists of a very fine lattice of cells, each capable of containing a 'particle'. The cellular update rule used represents a idealised version of the way in which particles collide. A wavefront is created through the introduction of a short 'burst' of particles at one side of the cellular array [2]. The wavefront propagates automatically as an emergent property of the cellular update rule, and the presence of a lens is simulated by placing obstacles at a random selection of cell sites within the circular region. This makes it more difficult, statistically speaking, for the particles to travel through the darker region, and the effect which this has in global terms, is to refract the wavefront.

Figure 2.6 shows a cellular automata model of dendritic growth, and figure 2.7 shows another model based upon idealised interactions between particles. This time, the velocities of particles in local regions, 96 cells by 96 cells in size, are averaged, and these average velocities are displayed as vectors. Once again a fresh supply of particles are created at the right hand side of the frame in order to set up a steady flow of particles from right to left. The vortex patterns emerge naturally from the local interactions of individual particles.

In figure 2.8 an annealing model is shown. The left hand image shows how by a process involving surface tension, bays are filled and capes eroded. The right hand

[2] Although which side is not entirely clear, since the left image of the three seems at first to be the wrong way round.

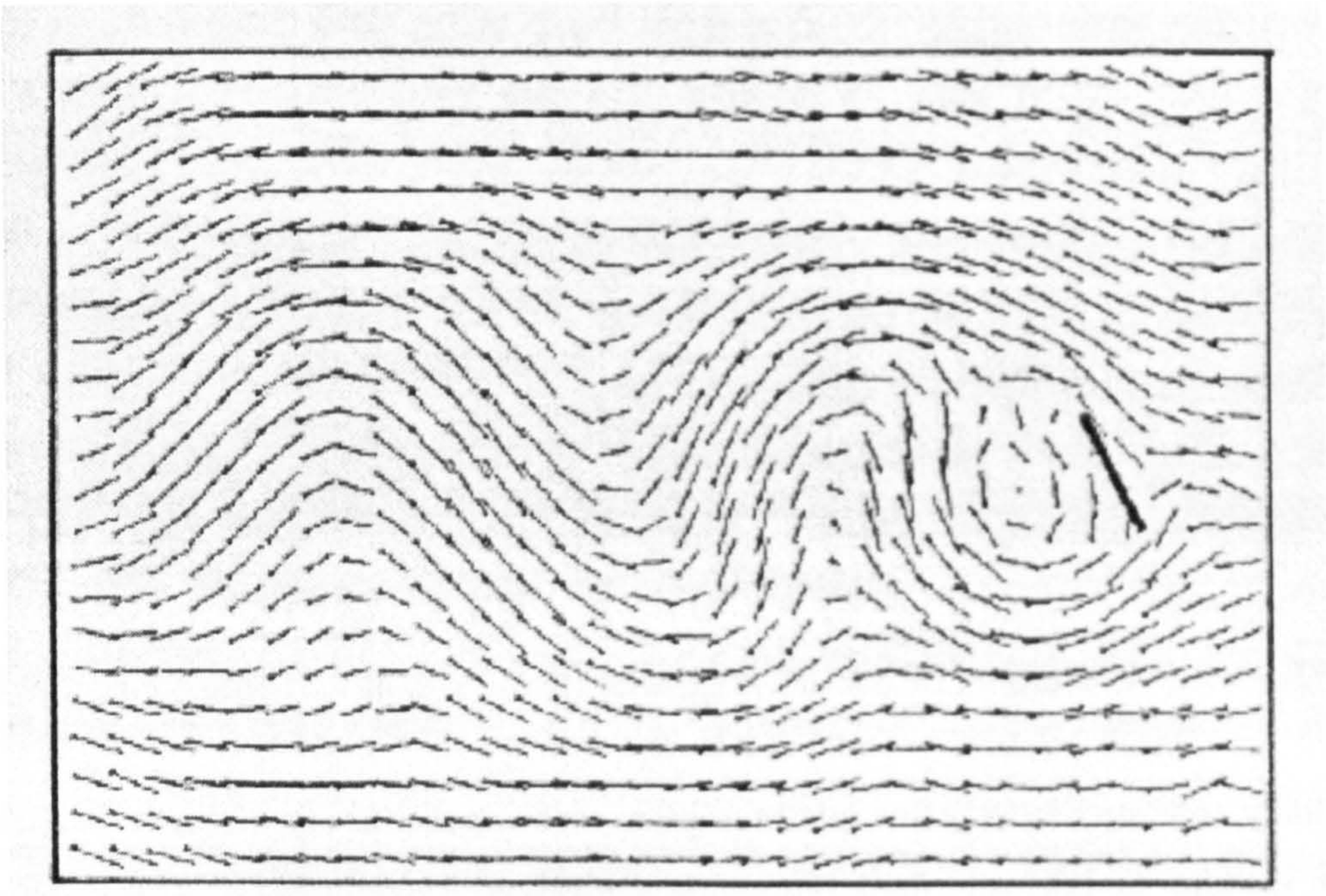


Figure 2.7: CA model of fluid flow around an obstacle (from Toffoli and Margolus, 1987)

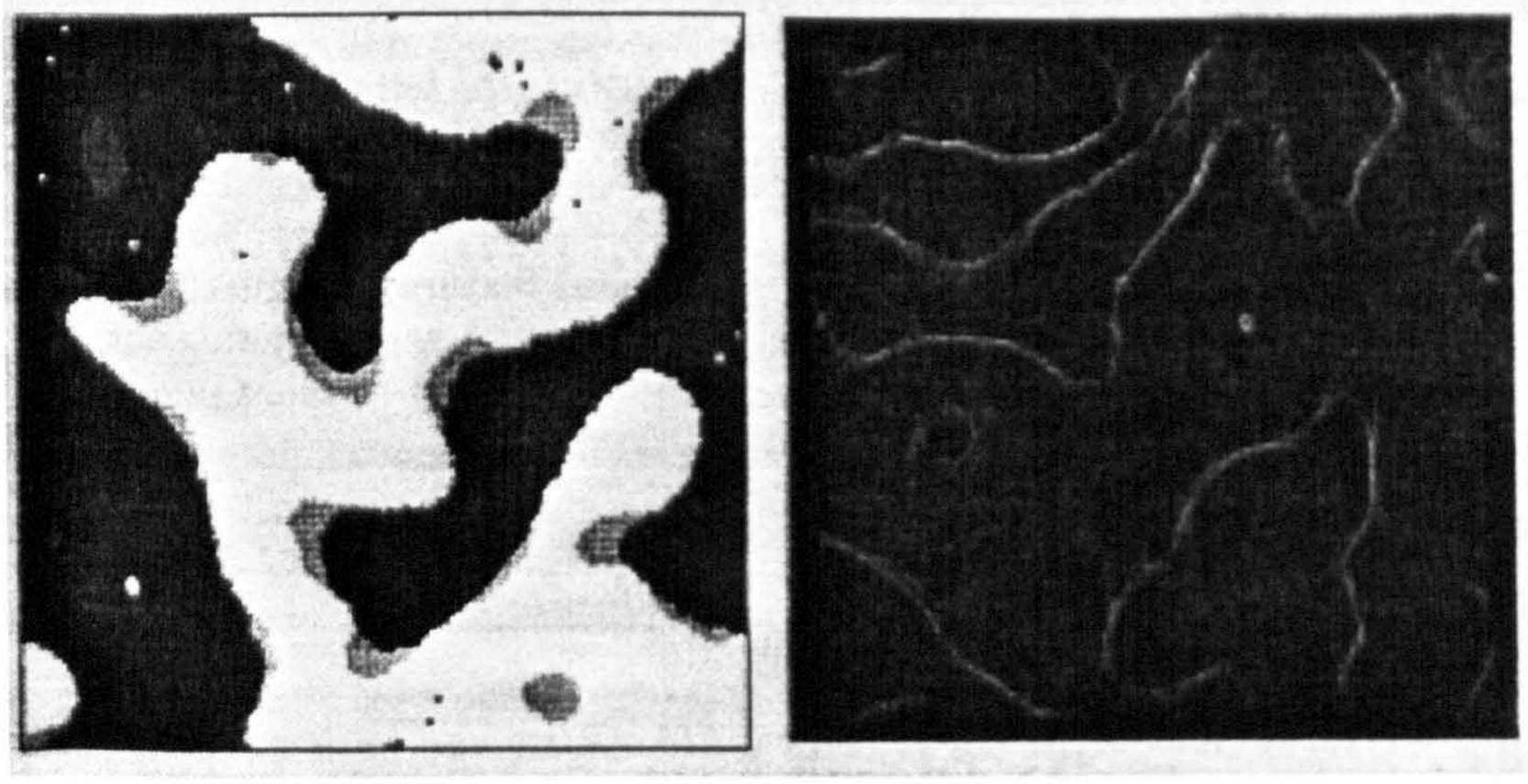


Figure 2.8: CA model of annealing (from Toffoli and Margolus, 1987)

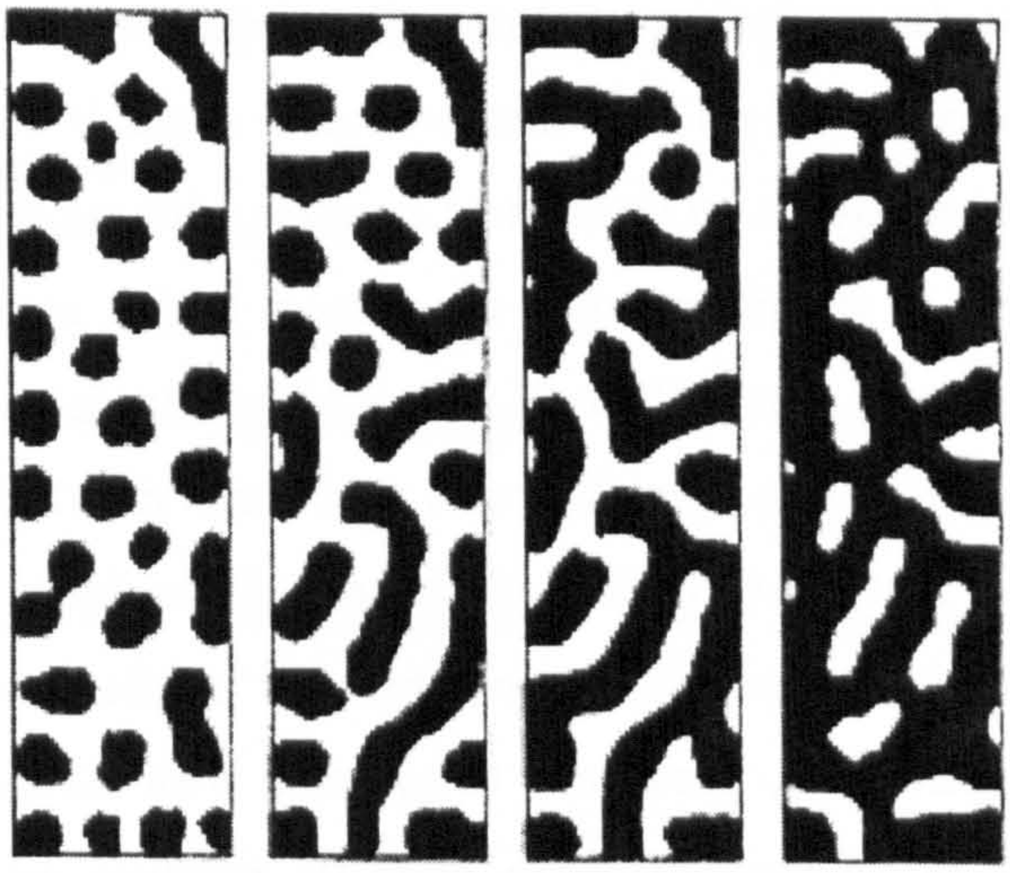


Figure 2.9: CA model of formation of vertebrate skin patterns (from Young, 1984)

image shows how the boundaries wander over a time interval of 400 steps of the model. Finally, 2.9 shows a model of vertebrate skin pattern formation. The four successive images give an elegant example of the robustness of cellular models. By changing the various low level parameters in the model and thus affecting the precise relationship between neighbouring cells, a variety of global forms may be produced, all belonging to a common family. A single model is capable of producing spots and stripes and all manner of patterns in between.

There are several interesting things to note about these examples: firstly, they show the diversity of natural phenomena which even the simplest cellular models are capable of simulating; secondly, each of the examples shows how emergent behaviour gives rise to form and pattern where it is not programmed explicitly; and thirdly, the resulting patterns and forms could be said to be convincingly natural or *organic* in their appearance. They look as if they could have occurred naturally due to some physical process. At a deeper level this statement says something about the structured information which they are capable of generating.

Putting these specific examples to one side for a moment, it is worth saying something about general classes of cellular automata and the kind of behaviour they produce. Wolfram (1986), whilst investigating the properties of a family of cellular automata, found that they could be categorised into four classes (Wolfram, 1984).

Class I automata produce relatively uninteresting behaviour and, regardless of the initial values of the cells, all activity dies out after a small number of iterations. These automata are analogous to dynamical systems with point attractors since no matter at which point in the phase space the system begins at, it will always gravitate back to the same rest state. Class II are slightly more interesting and produce clumps of cells, some of which remain stable and some of which oscillate between two or more states. Automata of this class are said to have periodic attractors because of the oscillation. Class III go to other end of the spectrum and produce completely chaotic behaviour, analogous to systems with strange attractors. The most significant class of cellular automata, class IV, produce patterns which:

> *propagate, grow, split apart, and recombine in a wonderfully complex way (Waldrop, 1994, p.226).*

This result is significant since it shows that even cellular automata, the most machine-like of cellular models, are paradoxically capable of producing patterns which would normally only be associated with natural processes. It also provokes the question of precisely how this behaviour fits in with the bifurcation diagrams of section 2.3, which the reader will remember were produced by a simple dynamical system, the logistic difference equation. The transition between order and chaos seems to be very abrupt in these diagrams, leaving no room for any 'in-between' states. However, if we zoomed in more and more closely on the border-line between the periodic and chaotic region, we would see that in fact the period-doublings continue at an ever increasing rate until, eventually, it would be impossible to perceive any periodic behaviour at all. Combined with this fact, a complex dynamical system poised at the edge of chaos is capable of exhibiting both ordered and chaotic behaviour on a local basis, and also at different scales.

The notion that local islands of order and chaos may exist in a complex dynamical system is supported by the description given in Waldrop (1994, p.234) of the work of another researcher, Chris Langton who was interested in finding analogies to support the hypothesis that life-like behaviour occurs at the boundary between order and chaos. On learning of Wolfram's work on cellular automata classes, and after much previous thought on the subject, Langton drew up the following series of analogies:

Cellular Automata Classes:
I & II → "IV" → III

Dynamical Systems:
Order → "Complexity" → Chaos

Matter:
Solid → "Phase transition" → Fluid

Life:
Too static → "Life/intelligence" → Too noisy

The *phase transition* analogy is the clearest example of a system possessing islands of both order and chaos, since at a phase transition some regions will be solid whilst others will be fluid.

2.7.2 Finite difference models

The finite difference approach to modelling is similar to that of cellular automata, the main differences being that values at each site may be continuous and updating may be asynchronous. Finite difference models are often more *coarsely grained* than cellular automata, e.g. whereas a cellular automaton model of fluid flow might deal with individual particles and their idealised collisions, a finite difference model would deal with averaged quantities such as velocity and pressure, and the model would be updated according to the differences between the values of these state variables at each site.

The cellular model on which the TAO computer music program is based comes under this heading, and apart from using cells with more sophisticated internal states and a more complicated, two-pass cellular update rule, the updating occurs synchronously in the same way that a cellular automata is updated. Precise details of this cellular model are given in chapter 4.

2.7.3 Finite element models

The finite element (Connor and Brebbia, 1978) technique is related to the other techniques described in this section in that it is based on dividing a complex system into regions. However, the other methods involve dynamic simulation, while finite

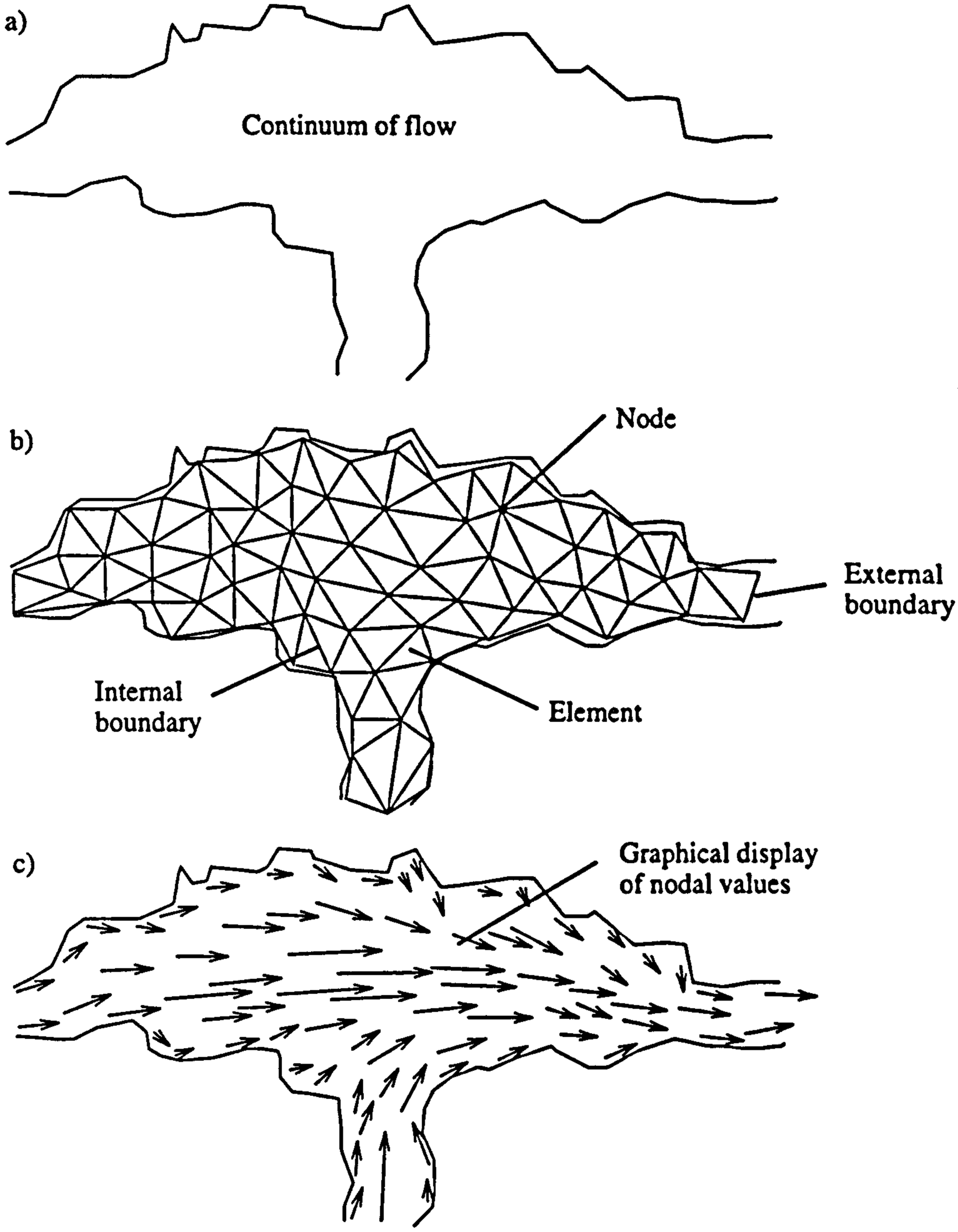


Figure 2.10: Finite element analysis.

element analysis is used to produce values for the internal state of the system which are consistent with external boundary conditions applied at some instant in time.

In a system such as water flowing in and out of a harbour, the flow is spatially and temporally continuous. To perform a finite element analysis of this system the harbour is broken down into a number of regions or elements. Figure 2.10 shows such a system divided up into triangular elements. A local function is chosen for each element and approximates the behaviour within the element. This function is expressed in terms of the unknown values at the nodes of the element, and must satisfy continuity with neighbouring elements and possibly continuity of its derivative with the derivatives of neighbouring elements.

Once a local function has been chosen, the internal and external boundary conditions (figure 2.10) are applied, leading to a set of equations in terms of nodal unknowns to be solved. This set of equations is usually expressed in matrix form and solved by standard matrix manipulation methods. Effectively, the finite element method tries to find a set of values for the nodal unknowns in the system, which are consistent with the local behaviour of the system and the external boundary conditions applied. As with the other techniques described, one of the advantages is that the numeric solutions found can be viewed graphically as in figure 2.10 c) to give an overall picture of the state of the system under given conditions.

2.7.4 Particle models

Particle models are similar to other cellular models except that the particles have continuously variable positions and velocities, instead of operating with a fixed spatial grid as in cellular automata and finite difference models.

> *Particle systems have been used to model fire, smoke, clouds, and more recently, the spray and foam of ocean waves. Particle systems are collections of large numbers of individual particles, each having its own behaviour. Particles are created, age, and die off. During their life they have certain behaviours that can alter the particle's own state, which consists of* color, opacity, location *and* velocity *(Reynolds, 1987, p.26).*

A cellular model of flocks and herds of animals which falls into this broad category is described by Reynolds (1987). The model assumes that a flock is simply the result of the interactions between individual birds and is capable of producing convincing flocking behaviour when the individual agents or *boids* are merely left to their own devices. This kind of model is a good example of one which might have applications in the composition of interesting musical macrostructures, even though it is not directly applicable to the generation of the microstructures, i.e. the sounds themselves.

> *Flocks and related synchronised group behaviours such as schools of fish or herds of land animals are both beautiful to watch and intriguing to contemplate. A flock exhibits many contrasts. It is made up of discrete birds yet overall motion seems fluid; it is simple in concept yet is so visually complex, it seems randomly arrayed and yet is magnificently synchronised (Reynolds, 1987, p.25).*

2.8 Other universal phenomena occurring in dynamical systems

2.8.1 Self organised criticality

Another theory which has a direct bearing on the behaviour of complex dynamical systems is that of *self-organised criticality* (Bak and Kan, 1991). This theory proposes that certain classes of dynamical systems evolve, by a process of self-organisation, to a critical state balanced on the edge of order and chaos. In this state, even small disturbances can cause catastrophic changes in the system's state. The most commonly quoted example of such a system is a pile of sand. If new sand grains are allowed to trickle onto the top of the pile at a steady rate, some grains lead to relatively minor avalanches whilst others cause much larger avalanches. Another example of a critically balanced dynamical system can be found in the geological faults which cause earthquakes.

The theory of self-organised criticality states that such systems are continually poised on the brink of catastrophic events and that there is a close relationship between the magnitude of such events and their frequency of occurrence, regardless of the specific

system being observed. Whenever such a catastrophic event occurs, the system slips just far enough to regain its stability, in which state it is once again poised on the brink of catastrophe. The precise relationship is governed by a *power law*, i.e. the frequency of occurrence of an event of a chosen magnitude is proportional to $1/(magnitude + c)^n$. If we viewed a seismological graph charting the activity of a fault, the largest avalanches would occur much less frequently than smaller ones. The shape of the graph produced by this kind of behaviour is said to be an example of $1/f$ *noise* or *flicker noise* (Bak and Kan, 1991).

Wishart (1990) has proposed that *catastrophe theory*, which relates to self-organised criticality, may have applications in the synthesis of certain sound morphologies.

> *If ...we take sound-objects whose intrinsic morphology is very complex or unstable, how can we relate to these? Are they merely formless or random? I would propose that there are a number of archetypes which allow us to classify these complex sounds perceptually, such as Turbulence, Wave-break, ..., Creak/Crack, Unstable/Settling, Shatter, Explosion, Bubble.*
>
> *...I would suggest that it may even be possible to extend this kind of analysis to phenomena where many individual sound sources are amassed, for example the Alarum (when a colony of animals or birds is disturbed the resulting mass of individual sounds has a very characteristic morphology), or Streaming effects (certain changes occurring in continuous streams of sounds may perhaps be related to models developed in catastrophe theory). (Wishart, 1990, p.60)*

The theory of self-organised criticality would seem, therefore, to have a great deal of relevance to the synthesis of such sound morphologies. Interestingly, Bak and Kan (1991) propose that the *life-like* behaviour of a system when poised at the edge of order and chaos is fundamentally different to truly chaotic behaviour and they refer to this kind of behaviour as being *weakly chaotic*.

2.8.2 Coupled oscillators

The physicist Christiaan Huygens discovered that when two pendulum-driven clocks are placed side by side on a common surface, they synchronise and keep perfect time relative to each other. This is due to the phenomenon of *coupled oscillators* (Strogatz and Stewart, 1993). Even though the physical vibrations transmitted from one clock to the other, and vice versa, are minute, they still lead to the two clocks mutually influencing each other.

Another manifestation of this phenomenon can be found in the gaits of different animals. The different, synchronised patterns of legs movements observed in horses, elephants, giraffes, gazelles etc. are not unique to those species but represent standard modes of synchronised oscillation which may occur in any system consisting of four oscillators coupled together. This phenomenon emphasises the point that many subtle inner rhythms may occur in a dynamical system which is a cohesive whole, whereas such rhythms will not be observed in a system consisting of separate, independent components. It is therefore probable that it has some relevance in explaining the behaviour of polyphonic musical instruments such as stringed instruments and the piano, where many vibrating elements are coupled together via other parts of the instrument.

2.9 Current musical applications of cellular models

Cellular models and dynamical systems have attracted some interest from the computer music community and listed below are some examples from the literature of how such techniques have been applied to various aspects of the music making process.

Beyls (1989) explores the use of cellular automata in the compositional process citing his interest as a composer in models of evolution and growth rather than in theories of structural design. Following on from this work Beyls discusses an approach to composition based on virtual 'actors' interacting in a two dimensional space according to social rules. The rules determine how the actors move and the attributes of each actor can be mapped to musical parameters (Beyls, 1990; Beyls, 1992)

di Scipio (1991) explores the use of simple one parameter maps to control sound synthesis. The equation $x_{n+1} = f(x_n)$ is an example of a one parameter map, where the function is iterated by feeding its output back into its input repeatedly.

Webb (1993) explores the use of a one dimensional cellular automaton as a self-modifying waveform table. The table is filled initially with samples representing a starting waveform. The samples are read out cyclically to produce a continuous waveform and at each cycle the values of the cells are updated according to simple rules, leading to a waveform which transforms over time. Some form of simple gestural control is provided in the form of a computer mouse, the buttons being used to inject random sample values into certain cells to perturb the waveform. Spatial movement of the mouse is also used, to control the frequency and amplitude of the output, leading to a rudimentary musical instrument.

Miranda (1993) discusses the application of cellular automata to pitch based composition and hints at the possibilities inherent in fluid dynamic models with particular reference to the fact that fluid movement may contain vortices. Vortices are cyclic flows usually accelerating or decelerating at a fairly constant rate and produce a constantly evolving source of cyclic material, which Miranda cites as being of musical interest.

Hunt, Kirk and Orton (1991) describes the musical possibilities of the Cellular Automata Workstation, developed at the University of York, concentrating mainly on mapping cell values onto pitch sets and changing the mapping in real time as a means of interactive control over the musical output. The update rule used while the cellular automaton is evolving remains fixed.

2.10 Summary

This chapter has introduced a number of contemporary scientific ideas and theories which relate to the behaviour of naturally occurring complex dynamical systems. The discussion has covered chaos theory and the interesting behaviour which occurs when a system operates poised 'at the edge of chaos'. It has also covered emergent behaviour, the ability of complex dynamical systems to exhibit self-organisation, and a number of other ideas such as self-organised criticality and the phenomenon of coupled oscillators. What this chapter has attempted to convey to the reader is

that Nature exhibits its own subtle 'rhythms' which are governed by the principles described, as well as others which may not have been identified yet, and that complex dynamical systems provide a means for these 'rhythms' to be expressed in fascinating ways.

A recurring theme throughout this chapter has been the special regime of behaviour which leads to *vibrant* or *life-like* behaviour. This regime is referred to in many different ways: as a regime which produces information of maximum *effective complexity* and *depth*; as a regime of *self-organised criticality*; as a regime of *weakly chaotic* behaviour; and as the regime in which Class IV cellular automata operate. The comment was made that many of the ideas addressed by the theories described in this chapter have a direct relationship to Gibson's observations on the nature of the physical world, which he proposed as being fundamental to the act of perception.

Cellular models exhibit many of the appealing characteristics of natural complex dynamical systems, even though they are stylised models, and as such provide a unique opportunity to explore the generation of *naturalistic* or *organic* patterns and forms. The fact that mimesis is cited as being an important aspect of electroacoustic music suggests that the temporally evolving structures produced by cellular models, with their complexity, coherence and organic qualities may possess very real musical qualities.

Chapter 3

A survey of synthesis techniques

This chapter conducts a survey of the most frequently used traditional synthesis techniques. No attempt is made to make subjective judgements about which synthesis techniques produce the 'best' sounds since such a comparison depends on so many factors including the level of skill displayed by the individual user. It is possible, however, to point out practical advantages and disadvantages of each technique which are independent of the particular user. Before proceeding to specific synthesis techniques it is as well to say a little about the building blocks which are common to most of them. This is also necessary in order to understand some of the figures presented in this chapter. The most common means for synthesising sounds before the arrival of powerful digital computers was via voltage controlled synthesisers. These synthesisers provide a number of electronic modules such as oscillators, filters and amplifiers, whose characteristics may be controlled via external control voltages. Early synthesisers of this type such as the EMS VCS3 provide a patch bay, enabling the output of any module to be patched into the input of any other. The provision of a modular approach to synthesis allows for the development of many different synthesis strategies.

In the 1960's the first computer music program, 'MUSIC 3', was created by Max Mathews (Dodge and Jerse, 1985). This program and a series of successors including Csound, which is described below, provide a kind of digital equivalent to the voltage controlled synthesiser. The analogue oscillators, filters and mixers etc. are replaced by algorithmic modules with numerical inputs and outputs which are still capable of

being arbitrarily configured. A language is provided for describing new instruments and a separate score language enables input data to be fed into the appropriate inputs of an instrument causing it to play notes at the correct times.

3.1 The Csound computer music program

One of the most recent examples of a computer music program, and one that is still widely in use is Csound (Vercoe, 1992) which is a general purpose language for audio processing and computer music. What follows is a brief description of the program together with some simple examples.

3.1.1 Unit generators

The algorithmic modules which simulate oscillators and filters etc. have come to be known as *unit generators*. Each unit generator is a single signal generating or processing algorithm and several unit generators may be combined into signal flow networks or instruments, to perform particular tasks. Since many such algorithms need a constant flow of input data to control their behaviour, input signals are often computed prior to performance and stored in tables. A table reader is another example of a unit generator, its input is an index into the table and its output is the signal stored in the table. This signal might be used to control the frequency of an oscillator or the cut off point of a low pass filter etc.

3.1.2 The orchestra

A Csound program is split into two main parts, the *orchestra* file and *score* file. The orchestra file contains descriptions of the instruments in terms of networks of unit generators. Each instrument has a discrete number of input parameters which can affect the pitch, amplitude and timbre of the sound produced when the instrument is requested to play.

Csound provides three different types of variable, *audio rate (a-rate)*, *control rate (k-rate)* and *initialisation rate (i-rate)*. *A-rate* variables are used to represent audible signals which must be updated at full audio rate (e.g. 44.1KHz). *K-rate* variables are used for signals which can be updated less frequently without introducing distortion into a sound e.g. the modulation signal used to create a vibrato effect. *I-rate*

variables are only updated once at the beginning of a new note.

The following example of a Csound orchestra describes an instrument consisting of two oscillators with their outputs summed and sent to a mono output.

```
          instr  1
a1        oscil  10000, 440, 1
a2        oscil  10000, 880, 1
aout      =      a1+a2
          out    aout
          endin
```

The variables in the left hand column `a1`, `a2` and `aout` represent audio signals. `a1` and `a2` represent the outputs of the two oscillators and `aout` is simply the result of evaluating the expression `a1+a2` at audio rate. The two `oscil`'s and the single `out` represent unit generators, and the values situated to the right of these keywords are interpreted as parameters. For example each `oscil` has an amplitude input (`10000`), a pitch input (`440`, `880`), and a table number (`1`), where the particular waveform which the oscillator will produce is stored. The `out` unit generator simply sends its signal to the default output (audio or file output). The line containing `instr 1` labels this as instrument 1. This number is used in a Csound score to specify which instrument some performance data should be sent to. Note that it is the user's responsibility to ensure that amplitudes do not go out of range, i.e. <-32767 or >32767.

3.1.3 The score

The score file contains performance data for the instruments, instructing them when to start and stop playing and specifying the parameter settings to use. In Csound, the score file consists of numeric data specifying these parameters for each instrument. The data is specified in columns called *p-fields*. The first three p-fields `p1`, `p2` and `p3` are 'hard-wired' to represent the instrument number, the start time and the duration of each note. For example the following score plays two notes, each 5 seconds long, using the instrument defined above. The first starts at time 0 and the second starts at time 1. Time is measured in beats, the default tempo being 60 beats per minute, although other score statements allow time to be dynamically 'warped' throughout a performance. Note that when two or more notes are to be played using the same instrument in a score, a new instance of the instrument is created for each note.

```
;         p1      p2      p3
;         instr   start   durat
          i1      0       5
          i1      1       5
```

The score may contain other user defined p-fields to control specific sonic parameters in an instrument. For example, supposing, in the orchestra given above, that the `oscil` lines were changed to read as follows:

```
a1        oscil 10000, p4, 1
a2        oscil 10000, p5, 1
```

Rather than having constant pitches, the oscillators now take their pitches from fields p4 and p5 of the score. The following simple score illustrates how to play notes on this instrument, and causes six notes to be played with frequencies 100 Hz, 200 Hz ...etc. for the first oscillator and 200 Hz, 300 Hz ...etc. for the second.

```
;         p1      p2      p3      p4      p5
;         instr   start   durat   oscil1  oscil2
;                                 pitch   pitch
          i1      0       5       100     200     ; measured in Hertz.
          i1      1       5       200     300
          i1      2       5       300     400
          i1      3       5       400     500
          i1      4       5       500     600
          i1      5       5       600     700
```

Further score commands allow tables of performance data to be created prior to commencement of the performance, and it is also possible to split the score into sections in order to create larger scale musical structures. Another feature is the ability to automatically interpolate or repeat p-field values with the use of the > and . characters. The example given above could be rewritten as:

```
;         p1      p2      p3      p4      p5
;         instr   start   durat   oscil1  oscil2
;                                 pitch   pitch

          i1      0       5       100     200     ; measured in Hertz.
          i1      1       .       >       >
          i1      2       .       >       >
          i1      3       .       >       >
          i1      4       .       >       >
          i1      5       .       600     700
```

The duration p-field has the same value repeated for each note and the pitches or rather frequencies of the two oscillators are interpolated linearly between the initial value and the final value. Further examples of orchestras and scores are given in (Vercoe, 1992).

Having introduced the notion of unit generators and the modular approach to instrument design, and having described Csound, we now move on to a survey of specific sound synthesis techniques.

3.2 Additive synthesis

Additive synthesis is based on the premise that any periodic waveform can be represented as a sum of sinusoidal components. In practice natural sounds continually evolve and in order to create an evolving frequency domain representation of a sound both frequency and time have to divided up into discrete intervals. The audible frequency range is divided up into finite width frequency bands or *channels* and time is divided up into finite length intervals or *windows*. Once a sound has been analysed via a fourier transform, a set of data is produced for each window representing the frequency and amplitude of each channel at each instant in time.

This data is then available for resynthesis by using it to control the frequencies and amplitudes of a set of sinusoidal oscillators. The sound may also be transposed or time stretched before resynthesis. The results of this resynthesis can be indistinguishable from the original sound, even to the ears of a trained musician (Dodge and Jerse, 1985). But of course straight resynthesis does not tap into the potential of this technique. Another possibility is for hybrid sounds to be synthesised, and one approach is to analyse two different natural sounds, and interpolate between the frequency and amplitude envelopes of each sound. In this way, sounds which have some of the characteristics of both the original sounds can be synthesised. There are, however, some important factors to take into account when performing this interpolation.

If sound A has a very short attack and sound B builds up much more slowly, then the partials for each sound will reach peaks at different points in time. For this reason, interpolation between the amplitude values of corresponding partials in each sound is not sufficient. The time axis values have to be interpolated also. The resulting

sound will not only possess spectral characteristics somewhere between those of A and B, but also has temporal qualities somewhere between the two. One problem associated with additive models of musical instruments is that the envelopes for each partial in a real instrument vary as we move up and down the instrument's pitch range. For example, analysing a middle 'C' note played on a piano produces a set of partial envelopes which will only be suitable for resynthesising pitches within few semitones either side of middle 'C'. To convincingly build an additive model of a whole instrument requires envelopes for each partial and register, an enormous amount of information to cope with.

Once constructed, an additive model can be very rigid in its behaviour, since changing the characteristics of a sound requires that all of the envelopes be changed in a coherent manner to produce the desired result. This reductionistic view of an instrument makes additive synthesis quite cumbersome unless special tools for analysis and resynthesis are provided. Fortunately, some tools are provided, and the composer Trevor Wishart has contributed a great deal in terms of both software and expertise to this area. In particular, he has developed a suite of programs which are capable of manipulating spectral data produced by the 'Phase Vocoder' frequency analysis program (*Composer's Desktop Project manual*, 1994) ready for resynthesis (Wishart, 1994).

3.3 Subtractive synthesis

Fixed periodic waveforms such as the sawtooth, square, and triangle waves have harmonic spectra with many partials and are very simple to generate electronically. As they stand, they are not very useful from a musical point of view, since they sound harsh and too bright and lack any sense of movement or evolution due to their purely periodic nature. However, with the use of low, high, and bandpass filters, parts of the spectrum can be filtered out allowing them to be sculpted into a slightly more musically useful form. A common technique employed in subtractive synthesis which adds some movement to the sounds, is to mix the output from several slightly detuned oscillators together, and use the result as the source for filtering. A sound may theoretically be built up from an unlimited number of mixed and filtered waveforms, but once again this requires an increasingly large amount of control data.

Subtractive synthesis, unlike additive synthesis, allows sweeping changes to be made to the spectrum of a sound simply by changing one or two parameters. This makes it more manageable to control, but the sounds produced tend to be more synthetic than those of additive synthesis, and fine control over individual partials is not catered for.

3.4 Frequency modulation

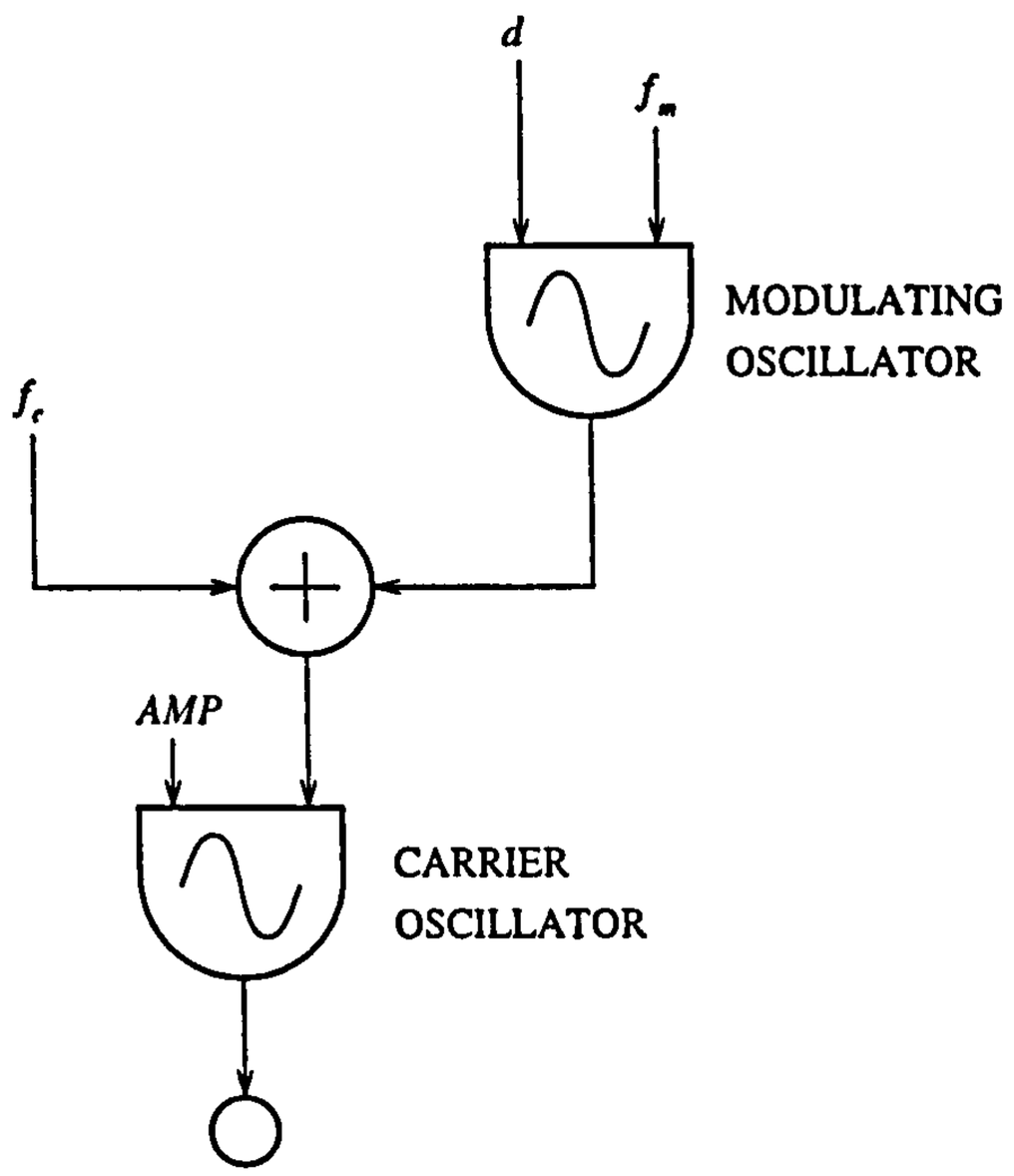


Figure 3.1: A simple FM instrument

A simple frequency modulation (FM) instrument is shown in figure 3.1. If the output from one oscillator (the modulator) is used to modulate the frequency of a second oscillator (the carrier), a complex spectrum arises. The simplest form of FM occurs when both carrier and modulator are sinusoidal. If f_c is the carrier frequency and f_m is the modulator frequency then the spectrum of the resulting sound is centred at f_c and contains regularly spaced sidebands at $f_c \pm k f_m$ where k is an integer. The distribution of power amongst these sidebands is proportional to the amount of modulation. The higher the amount of modulation, the more the power is spread over the sidebands. Even in this simplest case the resulting sounds have quite complex spectra and since the frequency of the modulator and the amount of modulation can

be varied dynamically, it is a simple matter to produce rich, dynamically varying spectra. In practice FM is particularly good at generating metallic, bell like sounds, especially when the ratio between f_c and f_m is non-integer. The most appealing points about FM are that it is simple to implement, and computationally inexpensive, and it generates complex time varying spectra. One of its shortcomings is that it is very difficult to correlate and predict the effect that a change in the frequency or amplitude of the modulator will have on the overall timbre of the sounds produced. A very slight change in f_m can drastically alter the overall sound.

FM was incorporated into Yamaha's DX range of musical keyboards in the 1980's and although reputedly the most popular electronic keyboards ever sold, one criticism frequently heard at the time was that although good at producing inharmonic, metallic sounds such as bells and chimes, FM seemed to lack a certain warmth which was present in the earlier generations of analogue voltage controlled synthesisers based on subtractive synthesis. This was attributed to the 'organic' nature of voltage controlled components. Magazine reviews of commercial synthesisers of the time often talk about the quality of the filters and oscillators in the same way that a luthier might talk of the quality of a particular piece of seasoned wood for a guitar. Another criticism which could be made of the bell and gong-like sounds produced by FM is that whilst they produce appropriate time-varying spectra, the resulting information generated lacks many other dimensions of coherence which are partly responsible for the expressive character of real bell and gong sounds.

3.5 Amplitude modulation

Amplitude modulation is similar in concept to FM in that a carrier and modulating oscillator are needed. There are two main kinds of amplitude modulation: classical modulation; and ring modulation.

3.5.1 Classical amplitude modulation

Figure 3.2 gives a flow chart for classic AM. The value AMP gives a default amplitude for the carrier when there is no modulation. The modulation index m can take a value between 0 and 1. When it is equal to 1, the amplitude of the carrier fluctuates between AMP and zero giving total modulation. Classical amplitude modulation

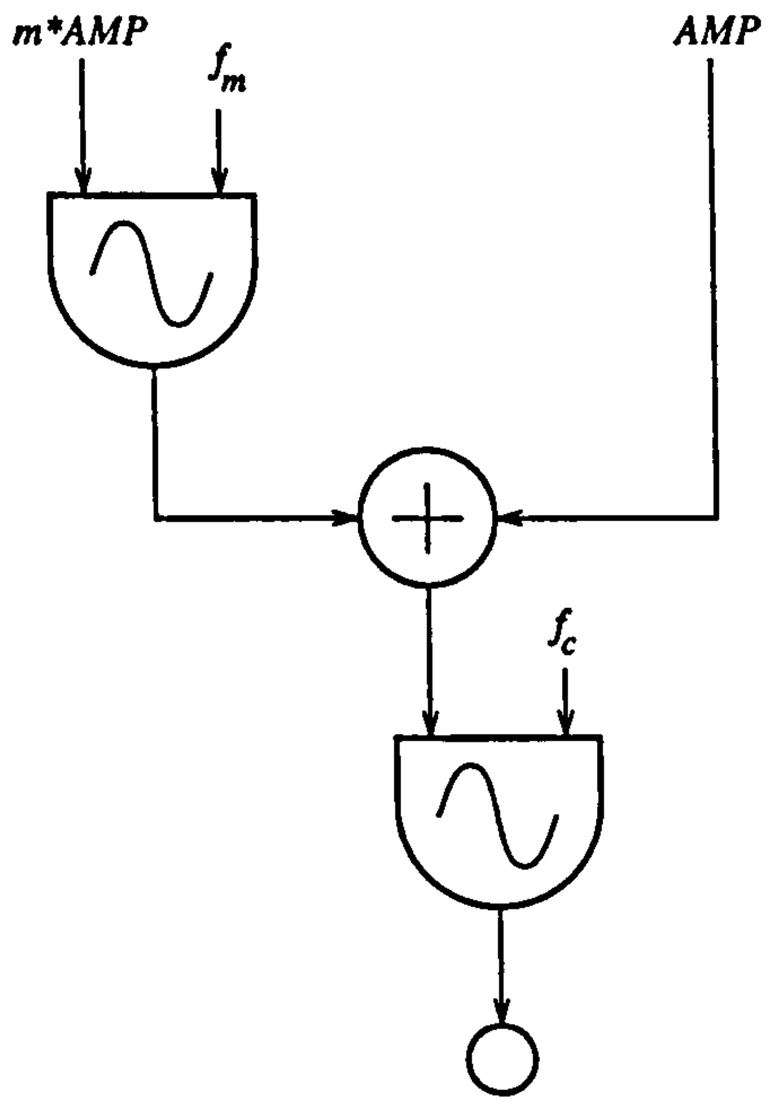


Figure 3.2: Classical amplitude modulation.

simply produces two new sidebands in the output spectrum, with frequencies $f_c \pm f_m$.

3.5.2 Ring modulation

Figure 3.3 gives a flow chart for ring modulation. The modulating oscillator is allowed to modulate the amplitude of the carrier directly. With no modulation there is no output but as the modulation increases, two sidebands at $f_c \pm f_m$ begin to appear. The difference between ring modulation and classical AM is that the carrier frequency f_c does not appear in the spectrum of the modulated sound. The carrier and modulator signals do not have to be sinusoidal and, in general, multiplying two signals together gives rise to ring modulation. If two sounds A and B are multiplied in this way, the resulting spectrum contains frequencies that are the sum and difference between the frequencies of each partial in sound A and those of each partial in sound B. This provides a fairly simple way to produce complex, inharmonic spectra, but once again as with FM, exercising fine control over the resulting spectra for musical purposes is not a trivial task.

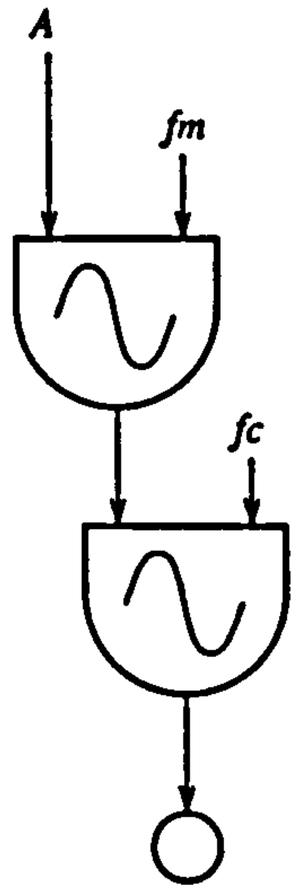


Figure 3.3: Ring modulation.

3.6 Granular synthesis

Granular synthesis (Roads, 1987; Truax, 1986; Truax, 1987; Truax, 1990b) provides a means for combining the frequency and time domain views of sound. A grain of sound is a pure tone of fixed frequency which is modulated by a finite (very short) envelope, and represents a kind of acoustic quanta. Sounds are synthesised by combining many thousands of such grains. The frequencies, amplitudes and patterns of temporal spacing between grains all contribute to the overall quality of the textures produced. The composer Iannis Xenakis was one of the first to explore the compositional possibilities of granular synthesis, and chose a fixed duration of 40ms for the grains (Roads, 1987), concentrating instead on their frequencies and amplitudes.

Granular synthesis presents a similar problem to that of additive synthesis in that a large amount of control data is needed in order to realise musical macrostructures. Xenakis proposed a system of *screens* and *books of screens* as a means of controlling the evolution of the granular texture. Each screen is a two dimensional plane, representing frequency versus amplitude. Grains of sound are scattered across this plane and several such 'screens' can be combined into a book representing the temporal evolution of the sound. The screens are separated by Δt, where 1ms$< \Delta t <$10ms.

This technique, whilst going some way to providing a higher level control strategy, still does explicitly deal with the question of how to create books and screens which lead to perceptually and musically effective sounds and exhibit coherence.

The strengths of granular synthesis lie in its ability to synthesise aperiodic noise sounds which have traditionally been beyond other frequency domain techniques, such as splashing, crunching, and shattering sounds etc. It is also effective for the synthesis of sounds such as those produced by waves crashing. One of the ongoing concerns of granular synthesis is in developing control strategies capable of mapping a small number of input parameters to a larger number of output parameters which actually control the individual grains. Truax (1990a) proposes non-linear chaotic systems as a potential control source and the cellular models introduced in the previous chapter seem ideally suited to this task with the advantage of being more flexible in their patterned behaviour.

3.7 Digital waveshaping

The principle behind waveshaping synthesis is to pass a waveform through a module which alters its shape in some way and hence its spectral content. The conventional way to achieve this is by means of a transfer function, a function which relates the amplitude of the output signal to that of the input signal.

Figure 3.4 shows the way in which a transfer function is used to alter the wave's shape. In this example the amount of distortion introduced by the waveshaping process is linked to the overall amplitude of the input waveform, since the transfer function only deviates from a straight line near its ends. This means in practice that with a sinusoidal input waveform, the number of partials introduced into the output waveform and hence the brightness, increases with amplitude. This corresponds to what we intuitively expect to happen in an acoustic instrument (e.g. blowing harder, plucking harder), although, the partials are not necessarily introduced in a smooth fashion as amplitude increases. Depending on the transfer function they may come and go as distortion increases. Dynamically varying the amplitude, or distortion index, produces a dynamically varying spectra. It is desirable to obtain this without significantly altering the amplitude of the output waveform. For this reason the output is often multiplied by a scaling factor which restores the amplitude to a

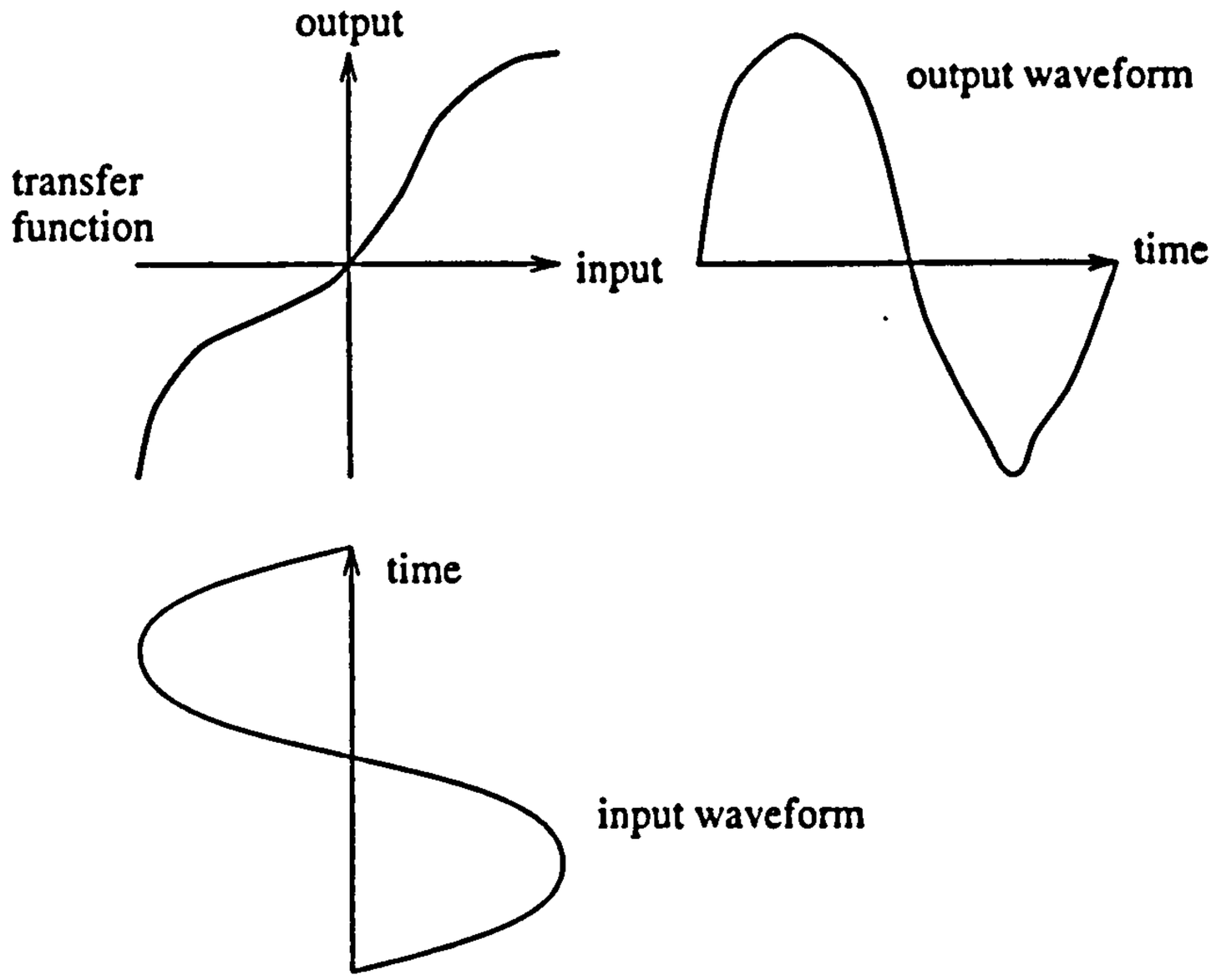


Figure 3.4: Waveshaping using a transfer function.

constant level regardless of the modulation index. The choice of transfer function has a large effect on the timbre of the output waveform and there are techniques which allow it to be chosen according to some desired spectral characteristics. These techniques are described in Dodge and Jerse (1985).

An important point to note is that with a sinusoidal input, the output waveform only contains harmonically related partials. A standard technique for producing inharmonic sounds involves amplitude modulating the output with another sinusoid, to give sidebands which are not necessarily harmonically related (as described in section 3.5). As with FM and AM, waveshaping synthesis is computationally very inexpensive to implement since all that is required is one oscillator and a look-up table containing the transfer function. The most difficult aspect is in calculating an appropriate transfer function to provide musical control over the partials at different amplitudes. This technique formed the basis for Casio's CZ range of commercial synthesisers which appeared in the 1980's as an answer to Yamaha's DX synthesisers. The sounds produced by digital waveshaping (or *phase distortion* as it was referred to by Casio) were often likened to those produced by FM, although they seem not to be quite as 'fluid' in nature.

3.8 Vocal synthesis

The human voice uses a form of subtractive synthesis in order make vocal sounds. Vibrations of the vocal folds produce a pulse-like waveform with a rich harmonic spectrum. This sound, along with noise generated when air is forced past constrictions in the vocal tract is filtered by the shape of the vocal tract, which runs from the glottis to the lips, and the nasal tract if it is coupled in. The vocal tract exhibits natural resonant peaks in its frequency response. These resonant peaks, or *formants*, may be moved up or down the spectrum by changing the shape of the vocal tract and mouth, and by moving the tongue. The combined frequency response of the different parts of the vocal tract (and nasal tract) provides us with the mechanism used to produce vowel sounds. Using a pulse wave and noise source, sounds with a vocal quality may be synthesised by constructing filters which mimic the formants of the human voice. A clear account of the established methods for achieving this is given in Dodge and Jerse (1985) and will not be repeated here, but in the next section one of the most recent systems for vocal synthesis, SPASM, is described. This system is based around a physical model of the vocal tract.

3.9 Synthesis by physical models

The synthesis techniques described so far have all concentrated on the spectral characteristics of sound, treating synthesis as an abstract, frequency domain process, and ignoring the physical origin of naturally produced sounds. An alternative approach is to model the behaviour of musical instruments and then generate sound via these physical models. There are a number of general strategies for achieving this which are discussed in Borin, De Poli and Sarti (1992). The principle methods for sound synthesis by physical modelling are described below.

3.9.1 Modal analysis and synthesis and MOSAIC

Physical objects such as a strings, bars, plates, bells etc. exhibit natural modes of vibration when allowed to vibrate freely. Each mode represents a single standing wave of fixed frequency. One way of simulating the vibrations of such an object is to determine, either by experimental or finite element analysis, precisely what the modes are. Once the modes have been determined, each can be modelled as a

mass-spring-friction combination, a *modal oscillator*. The whole vibrating structure can be represented as a set of modal oscillators coupled together. This is the basic premise of modal synthesis (Adrien, 1991; Djoharian, 1993).

Once an object has been represented in this way, all interaction with the object is translated into calculations involving the modal oscillators for the purpose of synthesis. For example if an external force is applied at a given point, this single force and coordinate must be translated into a set of modal forces which are applied to the various modal masses within the model. Similarly the amplitude of a given point on the surface of the object is determined by looking at the amplitudes of each modal oscillator and then combining them in the correct proportions, determined by the geometrical coordinate of the point chosen. To effect this translation from single geometrical coordinates to sets of modal coordinates a *prismatic window* is used. This window mediates between the outside world and the internal structure of the modal model.

In practice, acoustic instruments consist of many vibrating structures coupled together, each with their own modes of vibration. Modal synthesis supports the coupling together of separate modal objects to form more complex vibrating structures. Once again a connection between two points on different vibrating structures must be translated into a set of connections between the individual modal masses of the two structures. This process can be quite complicated and is described in detail in Djoharian (1993).

Figure 3.5 gives an example the modal decomposition of a surface, in this case a conical surface. This surface is represented as a set of circular sections joined together (b), where each section is basically a closed line. The overall vibrational modes of this structure are a combination of the circular section modes and radial modes. The vibration at a point on the surface can be modelled using the assembly shown in (c). The smaller masses linked to ground by very stiff springs represent the higher sections of (b) and the larger masses with less stiff springs represent the lower sections. MOSAIC [1] (Morrison and Adrien, 1993) is a language for creating and playing modal synthesis instruments. Standard modal models such as strings, rods, acoustic tubes (with one or both ends open), rectangular and circular membranes,

[1]Recently renamed MODALYS although at the time of writing no references could be found.

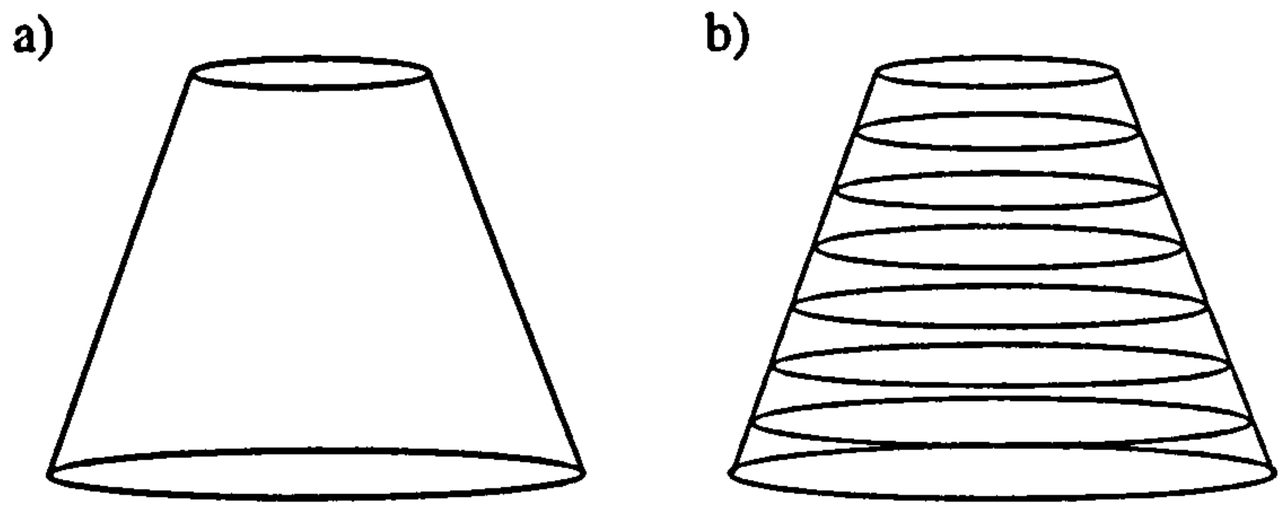


Figure 3.5: Modal decomposition of a conical vibrating structure.

rectangular and circular plates, and violin or cello bridges are available within the system. Other models can be created by experimental or finite element analysis, and once the modal data has been stored in standard file format, the new models can be integrated seemlessly into the system.

MOSAIC provides an interface based on the computer language 'Scheme' (Abelson, Sussman and Sussman, 1985) which is a pure dialect of the functional language 'Lisp'. The language allows instruments to be constructed by combining the modal models available into more complex structures. The following piece of code gives an example of how a modal object is created:

```
(define my-string
   (make-object 'bi-string
      (modes 40)
      (length 0.5)
      (tension 150)
      (density 1000)
      (radius 0.001)))
```

This takes a template for a modal object called **bi-string** and creates an instance of it called **my-string**. The length, tension, density and radius are measured in standard physical units of measurement, meters, newtons, kilograms, etc. Connections between resonant structures are made by creating 'access points' on the structures and connecting these access points together. Standard types of connection exist such

as:-

Adhere. Glues two access points together.

Bow. Simulates alternate frictional sticking and sliding associated with bow/string mechanism.

Pluck. The plucked point is dragged until some tensional limit is reached, at which point it is released.

The following example creates access points on two strings, at positions 60% along the length of the first, and 40% along the length of the second, and glues them together:-

```
(define access-point1
   (make-access string1 (const .6) 'trans0))

(define access-point2
   (make-access string2 (const .4) 'trans0))

(make-connection 'adhere acc1 acc2)
```

3.9.2 Digital waveguides

A waveguide is any medium in which wave motion can be characterised by the one dimensional wave equation (Smith, 1987). Examples of naturally occurring waveguides include the bore of a clarinet, and the vocal tract (see below). One approach to modelling such a medium is to sample its behaviour both spatially and temporally, giving a set of waveguide sections. Each waveguide section has an impedance which can vary with time, but is constant across the section, and propagates two waves one leftgoing and one rightgoing. The sections may have different impedances and as a wave propagates across the junction between two sections this change in impedance causes some of the energy to be transmitted forward and some to be reflected back. The amount of reflection and transmission is determined by a coefficient of reflection.

Figure 3.6 shows a *digital waveguide* (Smith, 1992). The model is divided into sections and each section consists of a scattering junction with reflection coefficient $k_n(t)$ and two delay lines with delays of T seconds, which represent the section

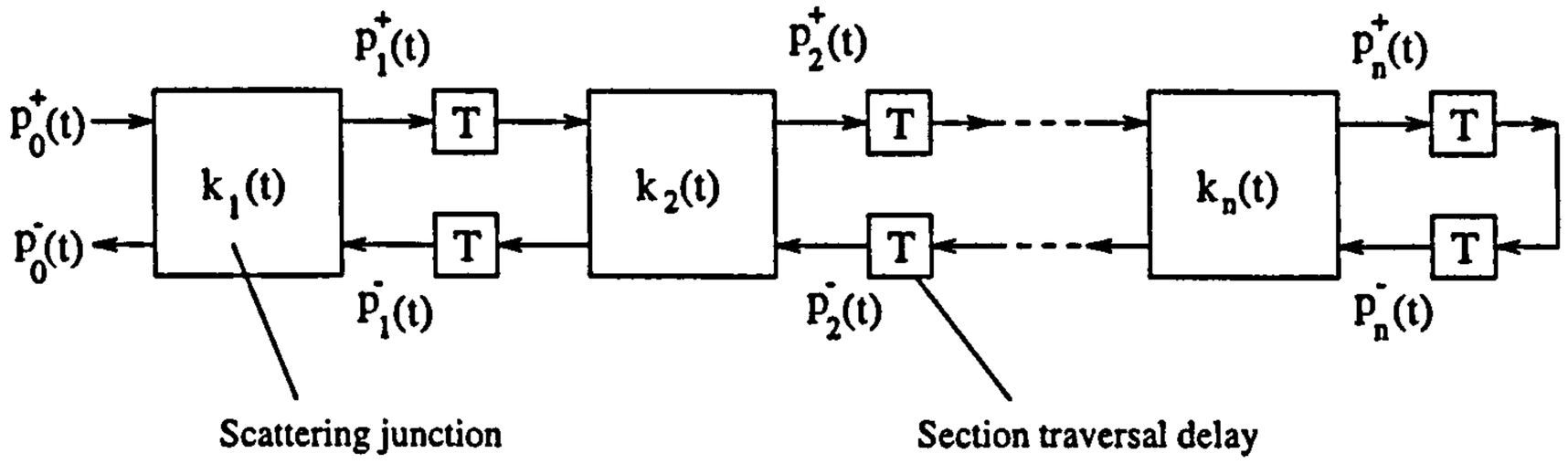


Figure 3.6: A waveguide filter network.

traversal delays for left and right going waves. By calculating the impedances of various parts of such real waveguides, networks of digital waveguides, such as the one shown, can be constructed which model the behaviour of the real systems. Impulse-like excitations are modelled by filling the appropriate delay lines up with initial values. Once left to its own devices, waves will bounce back and forth in the model, constantly being modified by the effect of the scattering junctions. Non-linear excitations such as bowing are modelled with the use of special junctions which allow energy to be introduced into the model and also allow the waves to be 'read' at the same point. These junctions contain appropriate mathematical models which simulate the particular excitation mechanism. The Yamaha Corporation bought the commercial rights for this synthesis technique from Stanford University in America, labelling it 'Virtual Acoustics'. Their first commercial product to use the technique was a keyboard based synthesiser, the VL-1.

The SPASM[2] vocal synthesis system (Cook, 1993), mentioned in the previous section, makes use of digital waveguides as illustrated by figure 3.7. The smoothly changing cross-sectional area of the vocal tract as we travel from the glottis (at the left of the figure) to the lips is sampled and modelled with a finite number of acoustic tube sections, each with constant cross-sectional area. The diameters of these sections may be altered in real time, thus altering the formants produced by the model. The figure also shows the internal structure of the scattering junctions. The nasal tract is modelled in a similar fashion and has to be coupled to the vocal

[2]Singing Physical Articulatory Synthesis Model

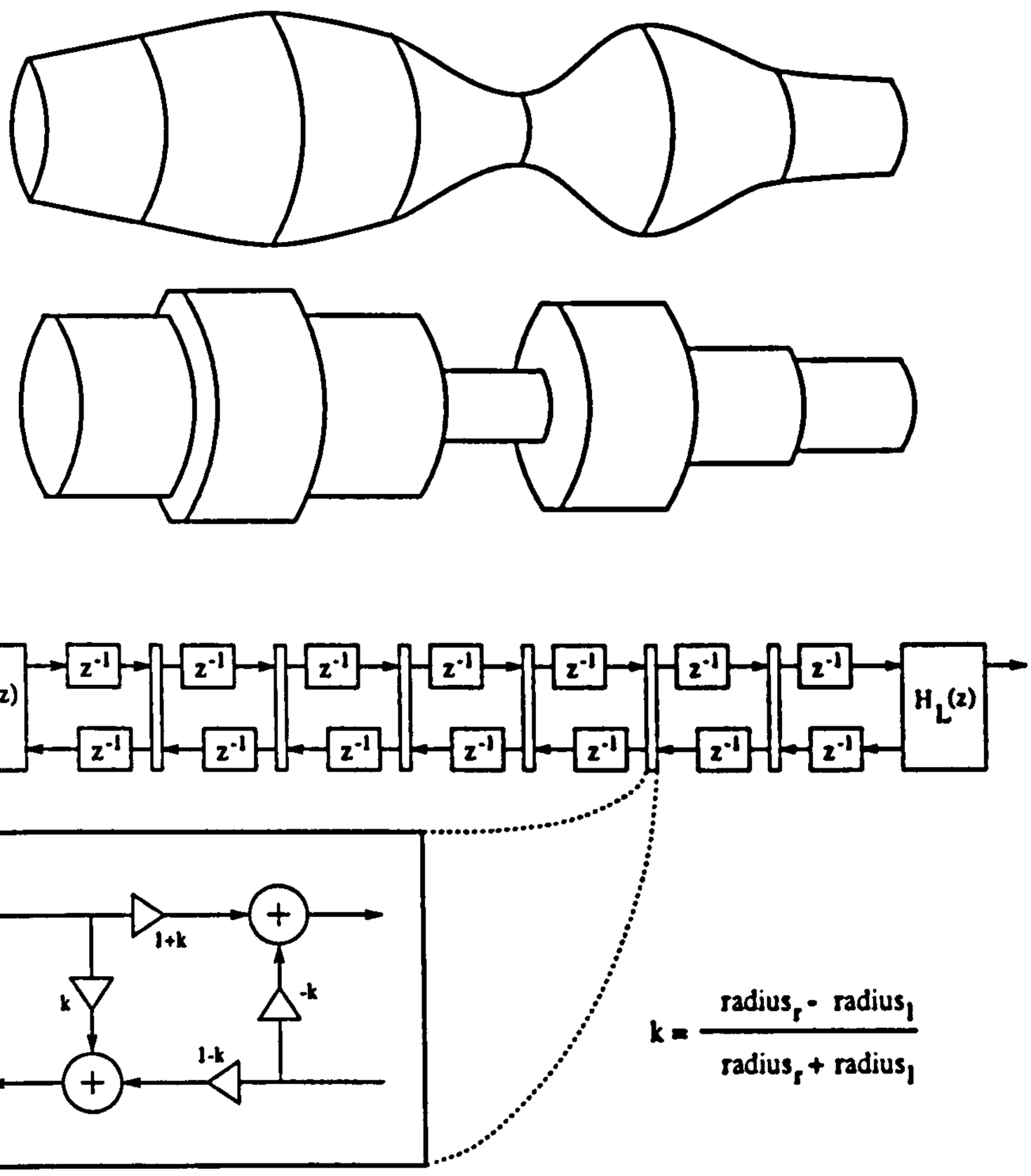


Figure 3.7: A waveguide filter model of the vocal tract (after Cook, 1993).

tract with the use of a multi-way scattering junction. The details of the model are described in the above-mentioned reference.

The system also provides a high level script language called 'Singer' which enables the multitude of vocal parameters to be controlled, making it possible to synthesis spoken or sung words, or any other vocal-like sounds. An example is given below of a 'Singer' script, taken from Cook (1993). This script synthesises the word 'Shiela':

```
// This example sings the word "Shiela"
singer(int fd) {

// initialise the singer model
   init();
// initialise setup for performance
   setup("shh","soft",400.0,0.0,0.0,0.0);
```

```
//              time,   shape,   glot,   freq,    gAmp,    nAmp,   vibr,   file

   synthesise(  0.3,    "shh",  "soft",  400.0,   0.0,     0.3,    0.00,   fd);
   synthesise(  0.1,    "eee",  "soft",  430.0,   0.2,     0.3,    0.04,   fd);
   synthesise(  0.7,    "eee",  "soft",     a4,   0.2,     0.0,    0.07,   fd);
   synthesise(  0.2,    "lll",  "soft",  440.0,   0.4,     0.0,    0.04,   fd);
   synthesise(  0.2,    "ahh",  "soft",  400.0,   0.3,     0.0,    0.00,   fd);
   synthesise(  0.2,    "ahh",  "soft",  400.0,   0.3,     0.0,    0.00,   fd);
   synthesise(  1.5,    "ahh",  "soft",  400.0,   1.0,     0.0,    0.08,   fd);
   synthesise(  0.1,    "ahh",  "soft",  400.0,   0.0,     0.0,    0.08,   fd);
   silence(0.5, fd);     // Write some silence
   return;
}
```

3.9.3 CORDIS-ANIMA

CORDIS-ANIMA (Florens, Razafindrakoto, Luciani and Cadoz, 1986; Cadoz, Luciani and Florens, 1993; Incerti and Cadoz, 1995) is a physical modelling system for the synthesis of sounds and visual images. It has similarities to modal synthesis in that it makes use of masses and links between them as the basic building blocks of the system. CORDIS-ANIMA is a formal framework for creating and interacting with digitally simulated, multi-sensory physical objects, rather than just a software system or synthesis technique. One of the main aims of the project is to provide *total simulations* of physical objects from the real world, including their gestural, tactile, acoustic, and visual aspects. The project therefore also includes the development of gestural transducers capable of providing a two-way physical dialogue with the objects created, i.e. the user is able to 'feel' the reactive force of an object via the transducer, as well as applying external forces to it.

The CORDIS-ANIMA formalism provides two primitive building blocks, M (matter) points and L (link) points. An M point is basically an algorithm which, given a force returns a position, and an L point is an algorithm which, given a position returns a force. M and L points are combined into atomic modules which serve as the basis for constructing simulated objects. Figure 3.8 shows a single M point and L point in (a) and (b). It then goes on to show how the schematic representation of a two way connection between two points (c) is actually implemented by two separate channels of communication, with each point's output being fed into the other's input (d). In (e) and (f) we see the simplest atomic unit consisting of a single mass and a channel of communication with the outside world and in (g) we begin to see how to build vibrating structures such as a string.

In order to create objects for sound synthesis, more complex structures need to be created. In practice, vibrating structures are constructed from M points connected

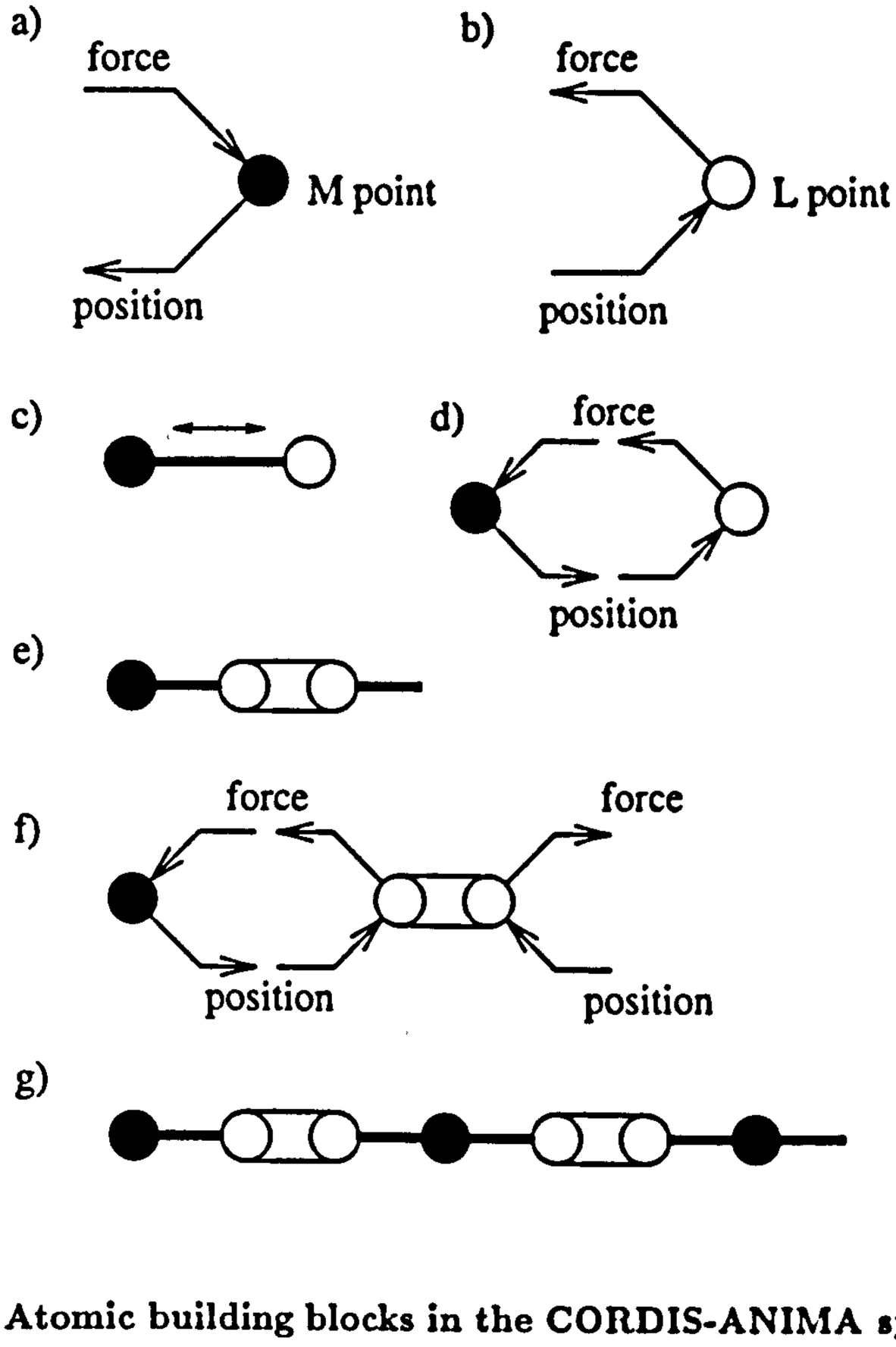


Figure 3.8: Atomic building blocks in the CORDIS-ANIMA system.

Figure 3.9: Representing a string in CORDIS-ANIMA.

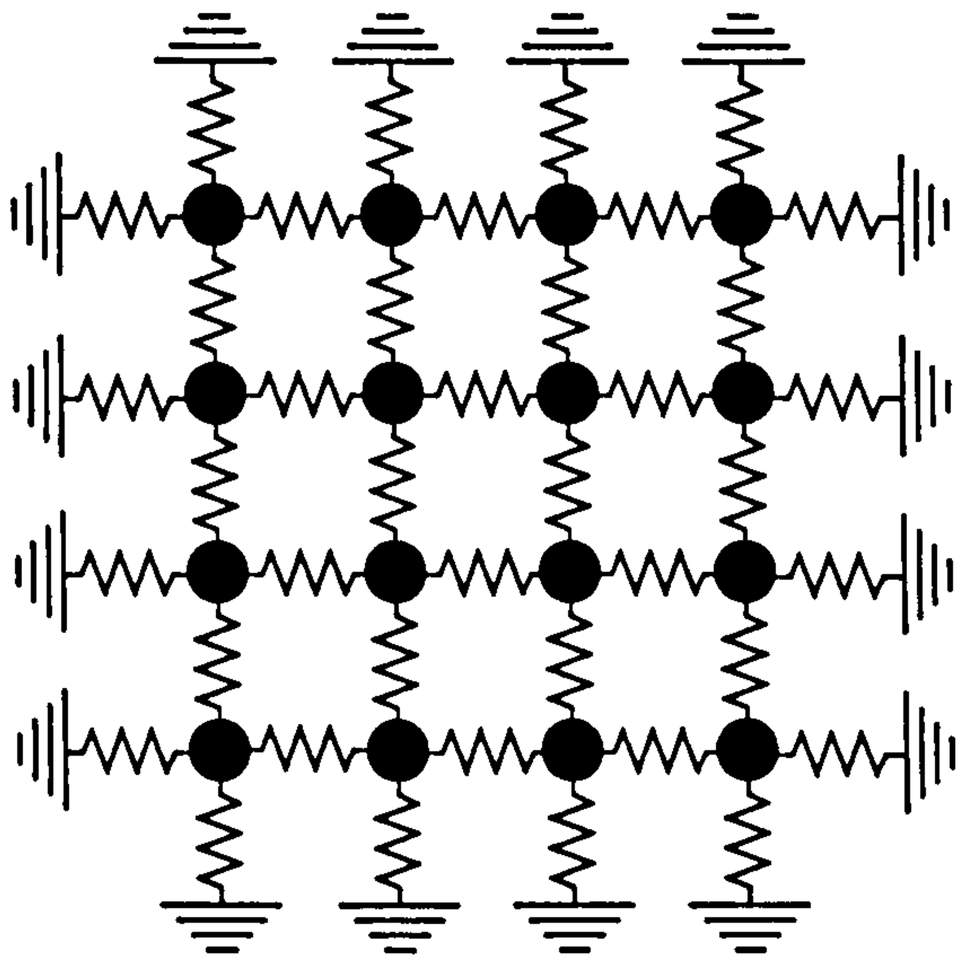


Figure 3.10: Representing a rectangular membrane in CORDIS-ANIMA.

together by visco-elastic *link elements*, comprising two indivisible L points. Movement can be in one, two or three dimensions although for sound synthesis it is usually limited to one dimension. Two examples are given in figures 3.9 and 3.10. The link elements are represented here by spring symbols and the ground symbol represents a special kind of M point called a *ground point.* This is an M point which is fixed in one position. Note that we are effectively looking at these vibrating structures along the axis of vibration, i.e. in and out of the paper. They do not move left to right or up and down. Apart from the two basic building blocks just described, two other modules are provided in the formalism to allow for: (a) dynamic variation of the structure of an object; and (b) dynamic control of other synthesis parameters such as the mass of an M point or the stiffness of a link element. Both of these modules take control data derived either externally from gestural transducers or internally

from some aspect of the state of an M or L point.

To sum up, the basic modules provided are listed below:

M point An algorithm which takes a force as input and returns a position.

L point An algorithm which takes a position as input and returns a force.

Dynamic structural variation module Basically an *if..then* module which makes a link between an M and L point conditional. Used to simulate transitory connections in a combined excitor-resonator system such as when a string is plucked. It takes its input either from an external source driven by the user's interaction with a gestural transducer, or internally from an M or L point. This could be used to make a link break when a force became too large etc., and allows the instrument to modify itself depending on its own behaviour.

Dynamic parameter variation module Allows arbitrary control over algorithmic parameters. Once again takes its input either from an external source, or internally from an M or L point. Examples of algorithmic parameters are the stiffness of a spring, the mass of an M point etc.

CORDIS-ANIMA does not provide a script language for describing either instruments or performances in the manner of Csound, MOSAIC and SPASM. Instead, emphasis seems to have been placed firmly on real time gestural performance with the instruments, and although it is claimed in the literature that there is no need for such a language since the building blocks provided constitute the elements of a language in their own right, it is not always entirely clear from the published literature precisely how a user moves from the abstract formal framework to concrete, practical examples. One answer to this criticism has been provided more recently in the form of a graphical user interface called GENESIS (Cadoz, Florens and Luciani, 1995) which provides standard facilities such as cut, paste, group, ungroup, erase etc. and a menu of objects including all the module types described above.

Moving away from technical details for a moment and considering strategies for instrument design in CORDIS-ANIMA, the main considerations are the topologies in which the masses and visco-elastic links are arranged, and the ways in which parameters such as stiffness, viscosity, and mass are distributed across these topologies.

Cadoz et al. state that one of their research goals is in understanding the relationship between topology and perceptual qualities, and gaining an understanding of which attributes of a sound are due to the mode of excitation and which are due to the topology and physical characteristics of the vibrating structure itself.

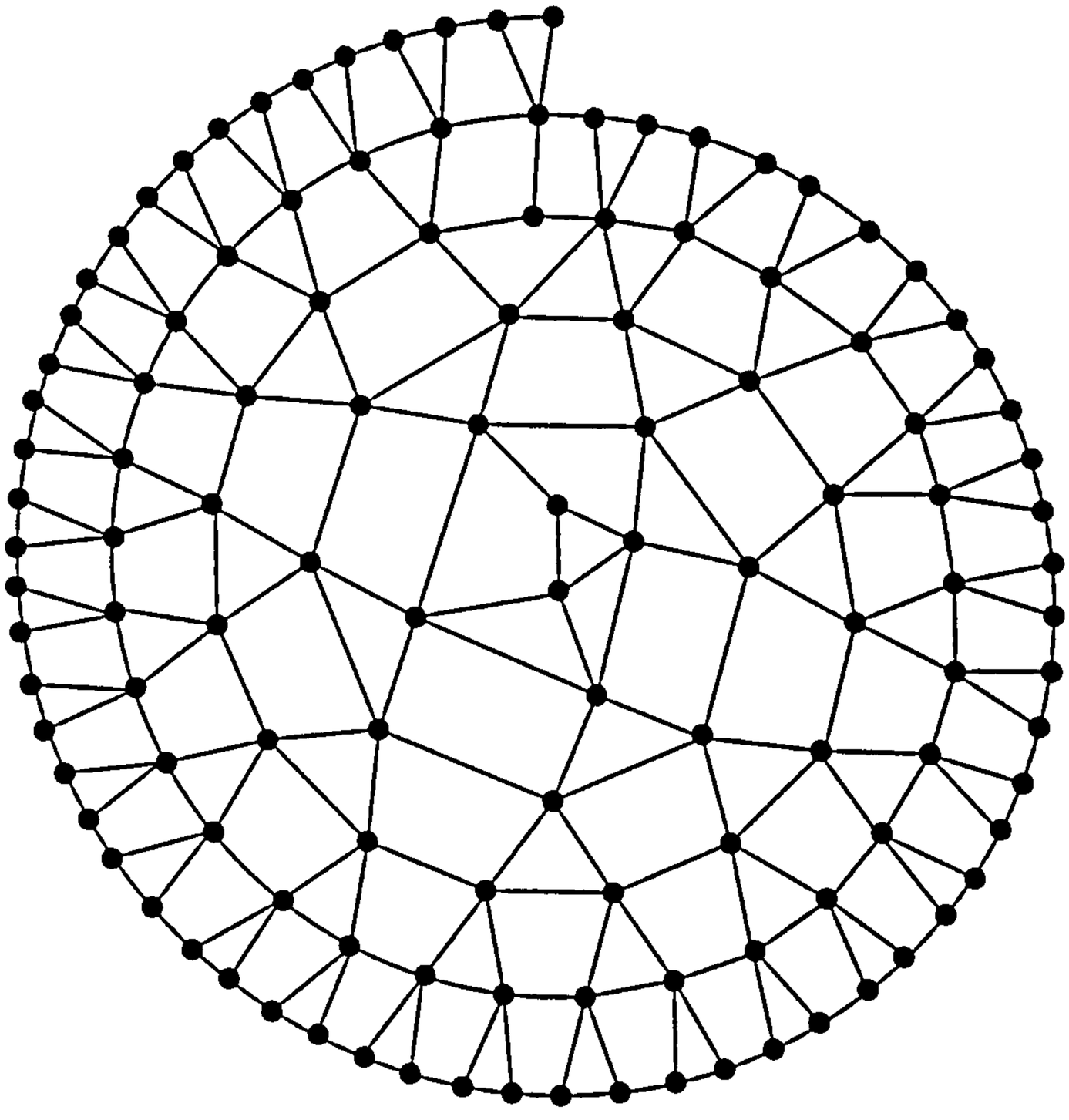


Figure 3.11: An example of a CORDIS-ANIMA topology - a spiral.

Figure 3.11 shows a complex topology, once again viewed along the axis of vibration. By choosing different values for the masses, and the viscosities and stiffness of each link a variety of sounds ranging from bells to gongs and cymbals can be produced. Since there are potentially a large number of parameters in such an instrument, Incerti and Cadoz (1995) discuss specific strategies for simplifying matters, such as giving all radial links a common stiffness whilst all spiral links are given a different common stiffness. This leads to certain modes of vibration being preferred over others and is an effective way of fine tuning the timbre of an instrument without having to control every individual link separately. Work is also under way to develop a graphical interface for capturing, manipulating and re-using gestural signals (Cadoz

et al., 1995). The ideas behind this work are described elsewhere in the literature (Cadoz, 1988; Cadoz and Ramstein, 1990).

3.9.4 Other physical models

Examples of other physical models of acoustic instruments can be found in the literature including bowed strings (Woodhouse, 1992), wind instruments (Keefe, 1992), bar percussion instruments (Serra, 1986), strings (Adrien, Causse and Rodet, 1987), and piano sounds (Garnett, 1987).

It is interesting to note that several recent articles begin to acknowledge the relevance of chaotic behaviour and dynamical systems to the behaviour of natural sounds. Examples include the use of pitch synchronised noise, generated by a chaotic oscillator, to simulate the noise produced by vortex shedding when a flute is played (Chafe, 1995). The technique described leads to a simulated instrument which behaves more naturally in that the periodic and chaotic regions of behaviour in the sound are not separate components which are superimposed, but form an integrated whole in which the output waveform has many intricacies.

Another example is in the use of nonlinear dynamics in the analysis and resynthesis of sounds (Mackenzie, 1995). The phase space portrait of a sound is analysed, and a simpler nonlinear equation is found, by an automated process, which reproduces a phase space portrait capturing most of the character of the original. The system has been tested with sounds as diverse as the rumble of a ventilation fan; a tuba tone; the sound of the wind; and a gong sound, and produces phase space portraits which convincingly capture the essence of the originals. It is obviously important to test the results aurally, but nevertheless the four sounds chosen are radically different in structure and yet the use of non-linear equations allows the technique to exhibit a degree of universality.

3.10 Criteria for comparing sound synthesis techniques

With so many sound synthesis techniques it is important to have some general criteria by which they can be compared. Jaffe (1995) has compiled such a list of criteria which are summarised here, since they may provide a useful focal point for all the ideas presented in this thesis and also allow the TAO computer music

program described in chapters 4, 5 and 6 to be compared and contrasted with the other synthesis techniques described in this chapter.

How intuitive are the parameters? Do they map intuitively to musical parameters such as dynamics and articulation or are they abstract mathematical variables with little correlation to real-world perceptual or musical experience.

How perceptible are parameter changes? When a parameter is changed, how perceptible is the change in timbre?

How physical are the parameters?

How well behaved are the parameters? A change in a parameter's value should produce a comparable change in timbre. If a small change produces a large unpredictable change in timbre then the parameter is not very well behaved.

How robust is the sound's identity? Does it maintain a coherent character regardless of the parameter settings, or does a parameter change lead to the sound seeming to move into a completely new timbral or perceptual category?

How efficient is the algorithm? Efficiency is obviously an important consideration but it should be remembered that a computationally expensive algorithm can be implemented efficiently and a computationally inexpensive algorithm can be implemented inefficiently. Efficiency is not the same as computational expense, which should be considered in context with the other criteria listed here.

How sparse is the control stream? How much control data is needed to produce complex sounds? It may be possible with techniques such as additive synthesis to produce *any* sound, but if the amount of control data needed to control all of the partials is too large, the technique will only be of limited use in practice. The whole timbral space of an acoustic instrument is made available through the use of a small number of physical parameters.

What classes of sounds can be represented? Is the technique only good for percussive sounds, string sounds, bell sounds, etc. or is it general enough to cope with many categories of sounds.

What is the smallest possible latency? Does the technique take a certain amount of time after some input data to synthesis the output. Most techniques are capable of responding within one sample to input data but some such as those involving some kind of fourier analysis will take a finite number of samples before any output can be produced.

Do analysis tools exist?

3.11 Summary

This chapter has attempted to give the reader a feeling for the multitude of techniques which are applicable to sound synthesis. Most of the traditional unit generator techniques suffer from one basic problem: the essentially reductionist approach taken makes it difficult to synthesise sound events which are complex and yet coherent, possessing micro- and macrostructural details which are causally related.

Of the physical modelling techniques digital waveguides are by far the most computationally efficient, although it would appear to be almost impossible for a lay-person to create a completely new waveguide based instrument since a working knowledge and understanding of differential equations and digital filters is required. The main building blocks of these models, delay lines, do not possess *any* physical characteristics at all, and all the 'interesting' physical behaviour must be built into the junctions between the delay lines, a non-trivial task.

CORDIS-ANIMA provides atomic building blocks from which an infinite variety of vibrating structures may be assembled. Whilst being more intuitive than digital waveguides and also being capable of producing very naturalistic and coherent sound events, one criticism might be that there are simply too many parameters which the user must control even in the initial construction of an instrument. The first decision to make involves choosing an appropriate topology for a new instrument, although this problem has been partially addressed by the ongoing development of the GENESIS graphical user interface which provides facilities for the automatic generation of topologies. Incerti and Cadoz (1995), in discussing this problem, make the following comments:

> *...it would be possible to go further and build very sophisticated networks*

> *with highly complex topologies designed to describe specific physical properties. But the numerous experiments that we [have] made tend to prove that it would be [the] wrong way [to proceed] ... Furthermore, some topological properties which are of great interest from a mathematical point of view, or even which may appear in nature (crystal symmetries, growth processes ...) may have no meaning from an acoustic point of view. Here is the difficulty: we have to determine which physical properties are relevant in a simulation for sound synthesis (Incerti and Cadoz, 1995, p.102).*

MOSAIC provides a framework for instrument design which seems to be more focussed and pragmatic than either the digital waveguide technique or CORDIS-ANIMA since equal emphasis seems to have been placed on both the synthesis engine and the use of an existing high level language Scheme for an approachable user interface. It is more computationally efficient than CORDIS-ANIMA because of the use of a smaller number of masses and springs but suffers in that it is not capable of producing informative graphical animations which allow the user to actually see what is happening to an instrument, aiding the debugging process.

> *In practice, it seems to be relatively straightforward to specify complex synthesis scenarios in MOSAIC once the control values are worked out. However, it remains difficult to choose good control values for some synthesis situations (notably those involving reed and bow connections). Debugging a synthesis is also difficult, and though control of the physical synthesis scenario is often straightforward ...control of the actual sound (spectral features, etc.) remains very difficult (Morrison and Adrien, 1993, p.55).*

Chapter 4

The TAO computer music program and its associated cellular model

4.1 Introduction

This chapter describes the cellular physical model at the heart of the TAO[1] computer music program (Pearson and Howard, 1995; Pearson, 1995; Pearson and Howard, 1996), which has been developed from first principles in support of this thesis.

TAO has been specifically designed as a compositional tool for music which is spectro-morphological or acousmatic in nature, and apart from being a useful working system capable of producing a wide variety of organic sounds, it also serves as a practical case study for many of the ideas introduced in this thesis. TAO provides a script language, described in the next chapter, which enables a user to create complex vibrating structures from pieces of cellular elastic material coupled together, and then enables these instruments to be excited and damped in a variety of ways. The system also provides a facility for generating graphical animations depicting the wave-propagation behaviour of instruments.

[1]The name 'TAO' is not an acronym but comes from the Chinese word 'Tao' which originally meant *the 'Way' or process of the universe, the order of nature* (Capra, 1992, p.116), and was chosen because it reflects the philosophical stance taken in this thesis.

The instruments behave according to physical laws operating on a local basis between neighbouring cells and are played by the application of external energy, whether by striking, plucking, bowing or some other excitation. The sounds produced often possess natural transients and physical, energetic and spatial characteristics which make them strongly suggestive of gesture and texture. The rest of this chapter describes the cellular model on which TAO is based; its emergent properties; the structural possibilities it affords; and how sound output is generated via the use of virtual microphones. Section 4.6 describes the underlying mathematics involved in the cellular update rules, and, although accessible to the non-mathematical reader, is not essential reading for those interested only in a user's perspective of TAO.

4.2 The cellular elastic material at the heart of TAO

From a physical perspective the material from which TAO instruments are constructed consists of point masses arranged in a regularly spaced two dimensional grid, connected to their eight immediate neighbours by springs, and constrained to have one degree of freedom, as if they were mounted on frictionless slides all pointing in the direction of the z axis. Figure 4.1 shows a close-up of a typical portion of the material, whilst figure 4.2 shows an individual point mass connected to its eight neighbours. The masses are referred to as from now on as *cells*.

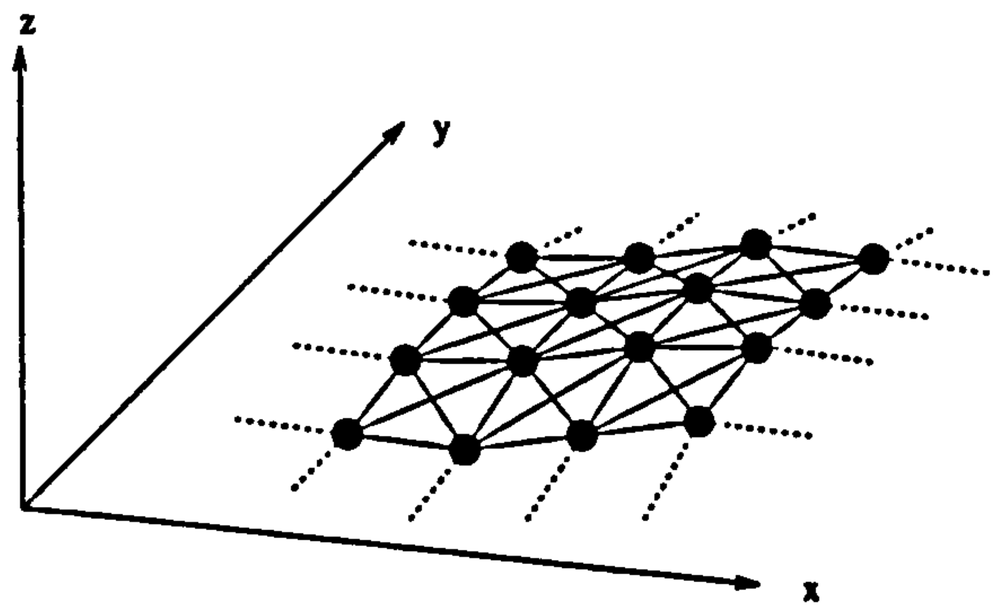


Figure 4.1: A close up view of TAO's cellular material.

Each cell contains variables representing its *position*, *velocity*, *force*, *mass* and amount of *damping*. They are arranged in the xy plane and are free to move only in z direction. The position, velocity and force acting upon a cell are always measured in the

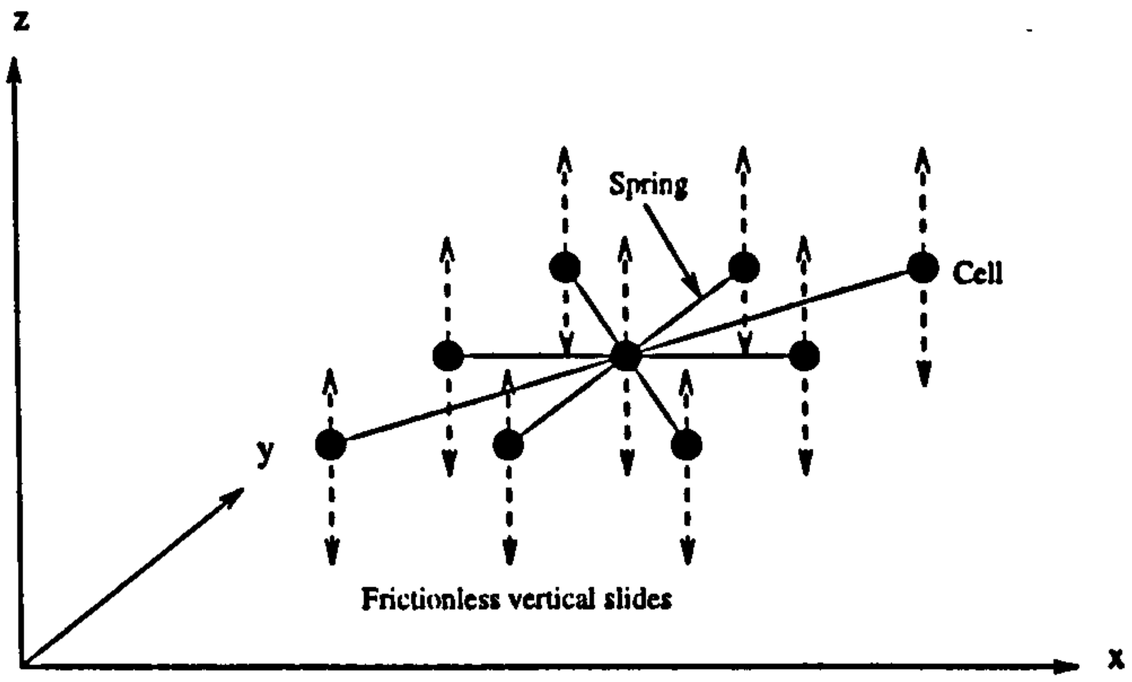


Figure 4.2: A single cell with its eight neighbours.

direction of the z axis and a cell's position is measured relative to the $z = 0$ plane.

The springs connecting the cells together are not modelled as separate entities, instead each cell maintains a set of pointers to its eight immediate neighbours and whenever two cells point at each other, the presence of an elastic spring between them is implied. The springs are represented implicitly in the cellular update rules used. Another feature of the springs is that although they are represented in figure 4.1 by diagonal lines connecting neighbouring cells, they do not actually stretch diagonally between neighbouring cells. Each spring actually exerts a restoring force which is proportional to the vertical distance between the two cells, i.e. measured in the z direction. Two cells connected by a spring may be placed anywhere in the xy plane without affecting the force exerted on both by the spring connection, which relates only to their relative positions in the z direction.

By default every cell is free to move in the z direction but individual cells or groups of cells may be locked in one position. For the purposes of sound synthesis it is usually necessary to lock at least one cell at the $z = 0$ position in order to force the rest of the cells to oscillate about this point. All interaction with the material is via external forces applied to individual cells or groups of cells. Each cell also has a *damping* coefficient associated with it, which causes a cell to be subjected to a frictional force proportional to its velocity, and thus has the effect of continually slowing the cell down. Since every cell maintains information at all times about its position, velocity and the forces acting upon it, excitation algorithms can make

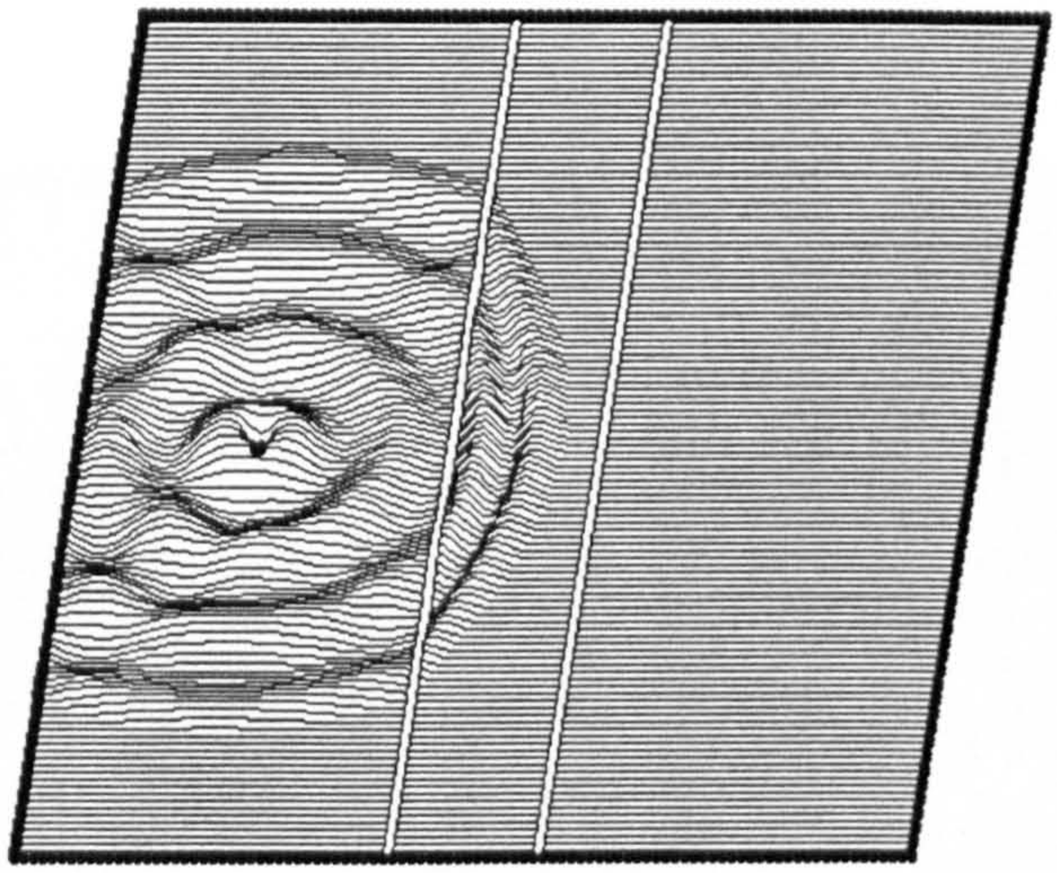
use of feedback from the instrument, very easily. For example an object colliding with an instrument, apart from exerting a force on the instrument, can be made to feel the force of the impact and react accordingly. This kind of facility provides the potential for sound events in which the structured information generated, because of its derivation from physical laws, will be suggestive of realistic physical causes.

Viewed from a distance, the material exhibits wave phenomena such as reflection, refraction (figure 4.3) and diffraction (figure 4.4). The refraction is achieved by giving the cells in the central portion of figure 4.3 higher masses than the rest of the cells, effectively making that region more dense. The diffraction effect is caused by locking the cells in the heavier black region of figure 4.4, leaving just a few 'slits' where the cells are free to move, thus creating a diffraction grating. One of the advantages of using a cellular model is that we can alter the shape of the piece of material or locally alter its properties without losing the coherent, macroscopic wave behaviour. It will also cope just as easily with the most complicated scenarios as with simpler ones.

The use of masses and springs in synthesis is by no means new since it forms the basis for both the CORDIS-ANIMA and MOSAIC systems which were described in chapter 3, but unlike CORDIS-ANIMA, TAO uses a fixed topology of masses and springs which has the effect of making the model more consistent, the material more uniform, and the cellular update rules simpler, without significantly affecting the creative possibilities offered by the system.

In order to generate sound output from a piece of material, *virtual microphones* are provided. In the context of TAO, a microphone is a device which takes arbitrary numerical values and writes them to a file as sound samples. The numerical values are usually generated from mathematical expressions involving the positions of individual cells. The samples in this file are then normalised to fit the maximum amplitude range provided by the sample format and are written to a soundfile (in this case an audio interchange file format or *.aiff* file). In this way, *any* vibrations occurring within an instrument, no matter how large or small their amplitude, may be used as sources for the generation of soundfiles. The Csound language described in section 3.1 places the responsibility for keeping sound samples within the range dictated by sixteen bit integer samples firmly with the user, and in comparison, the

a)



b)

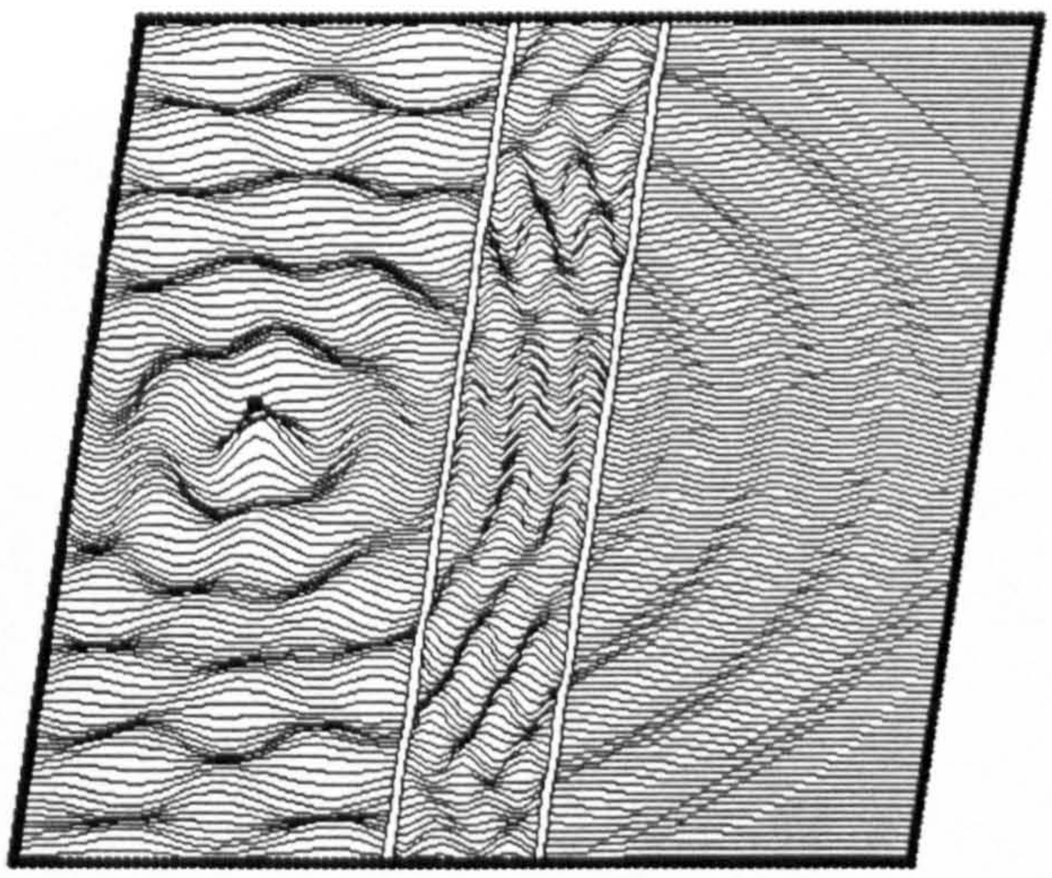
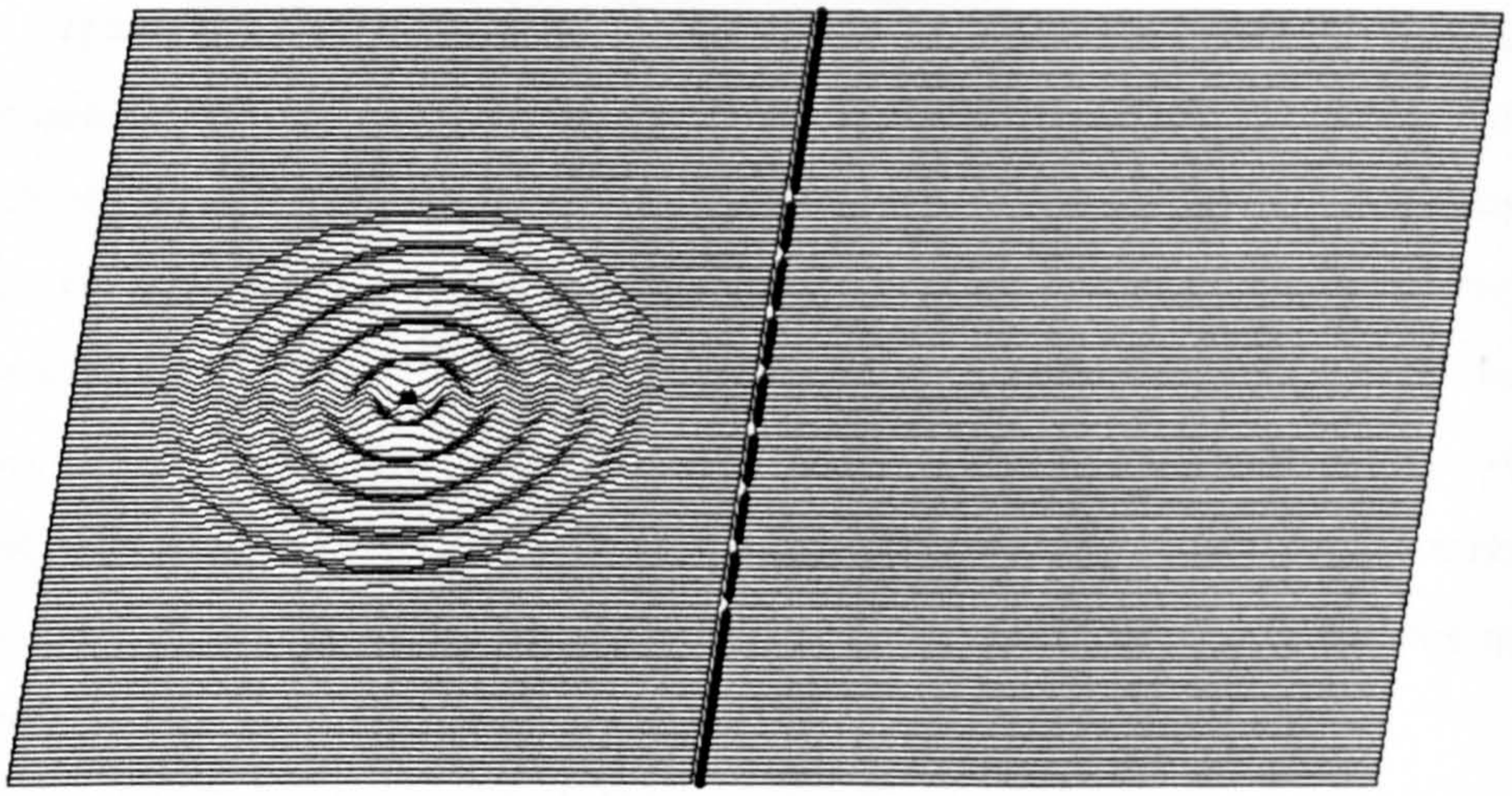


Figure 4.3: Simulating refraction and standing waves.

a)



b)

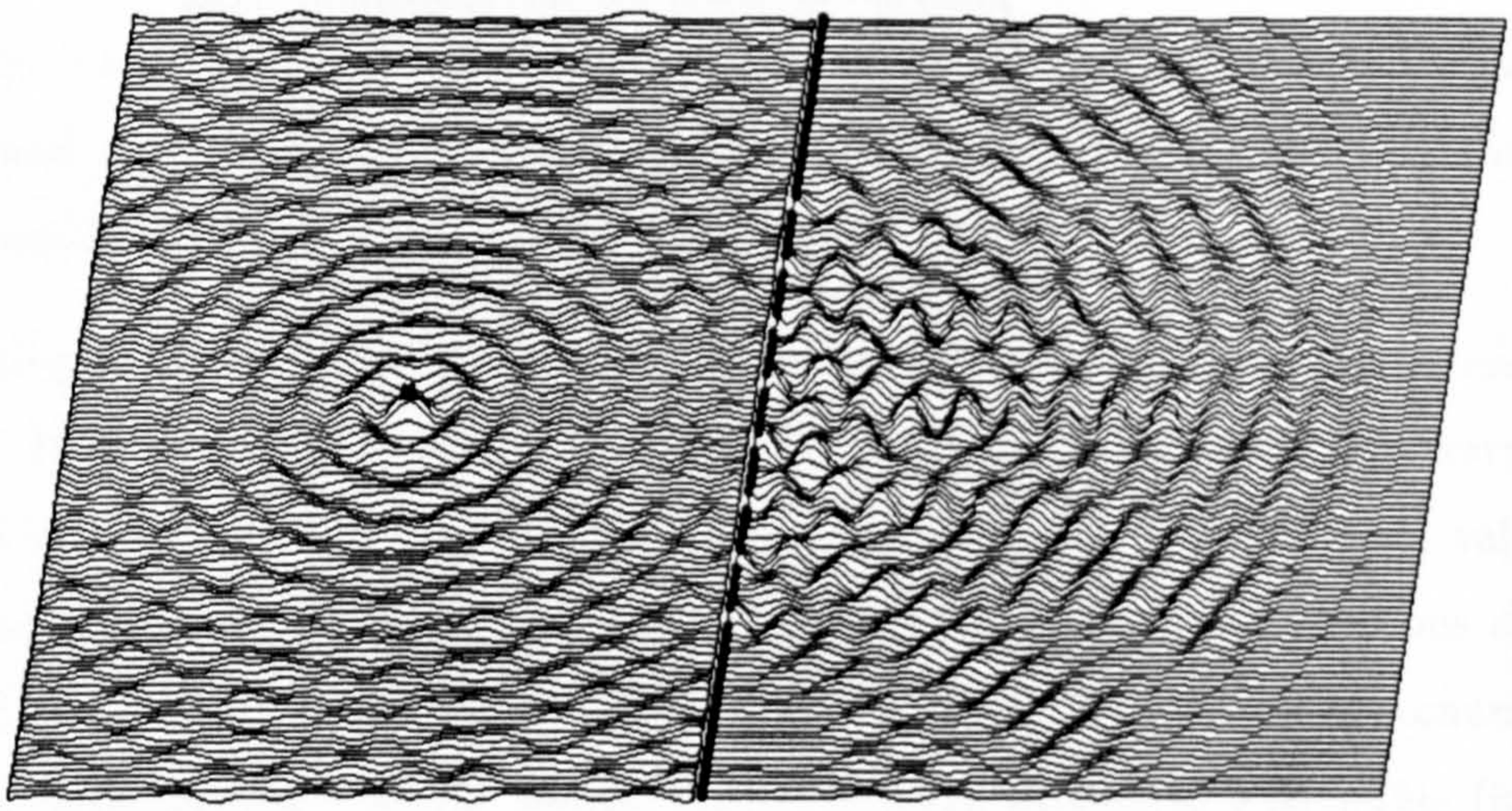


Figure 4.4: Simulating diffraction

approach taken by TAO ensures that a synthesis will never go 'out of range'.

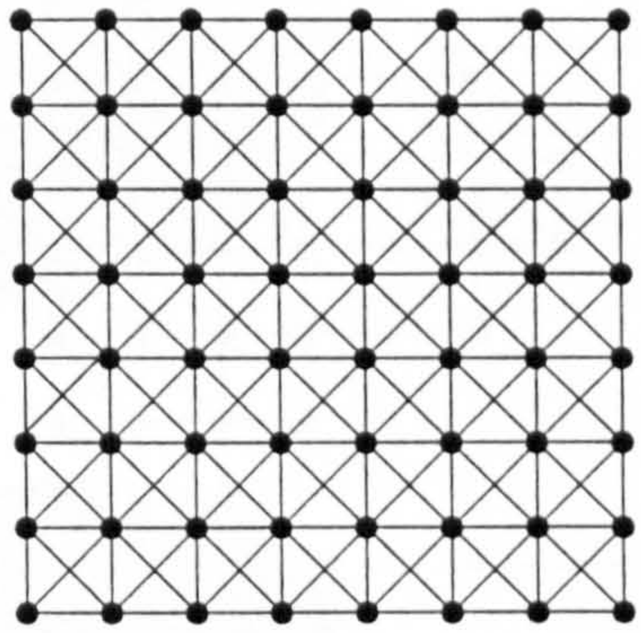


Figure 4.5: A plain square piece of material

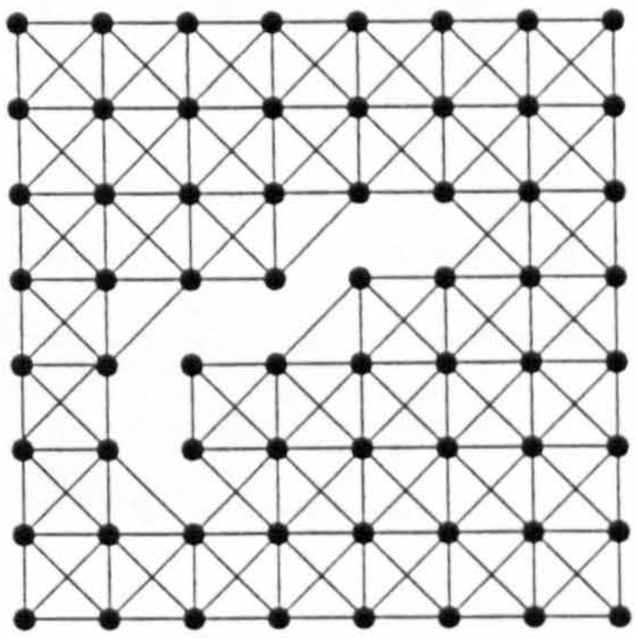


Figure 4.6: The same piece of material having been torn

Figures 4.5, 4.6, 4.7 and 4.8 give some examples of the structural possibilities offered by this cellular model. Figure 4.6 shows how by selectively removing some of the connections between cells, the material can effectively be torn, leading to new modes of vibration which a plain square piece of material would not have.

The model supports the creation of arbitrarily shaped pieces of material, as in figure 4.7, and it is feasible to make shapes which contain holes as in figure 4.8. The shape of a piece of material will have a direct influence on its natural modes of vibration and is therefore one of the most significant parameters made available to the user. We will see in chapter 5 that TAO supports the creation of various simple geometrically shaped pieces of material, although irregular shapes and shapes containing holes are not supported in the present implementation.

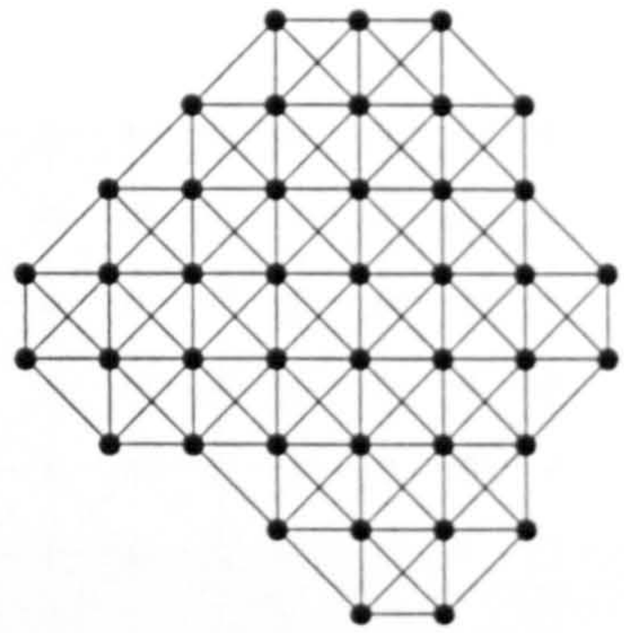


Figure 4.7: Making irregular shapes from the material

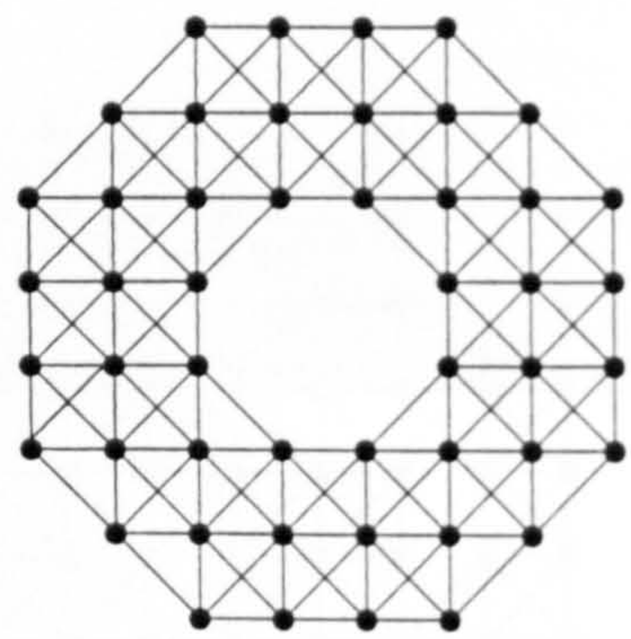


Figure 4.8: Making shapes of material with holes

4.3 Coupling pieces of material together to form instruments

Two methods of coupling separate pieces of material are provided, *glueing* and *joining*. The former allows individual points on two pieces of material to be glued together forcing them to move in unison in the z direction, whilst the latter allows two pieces of material with straight edges to be joined seemlessly, making them act as if they were one continuous piece.

4.4 An example of a TAO instrument

To put all the ideas introduced thus far into context, figure 4.9 shows an example instrument consisting of four strings whose left ends are glued to four points **a**, **b**, **c**

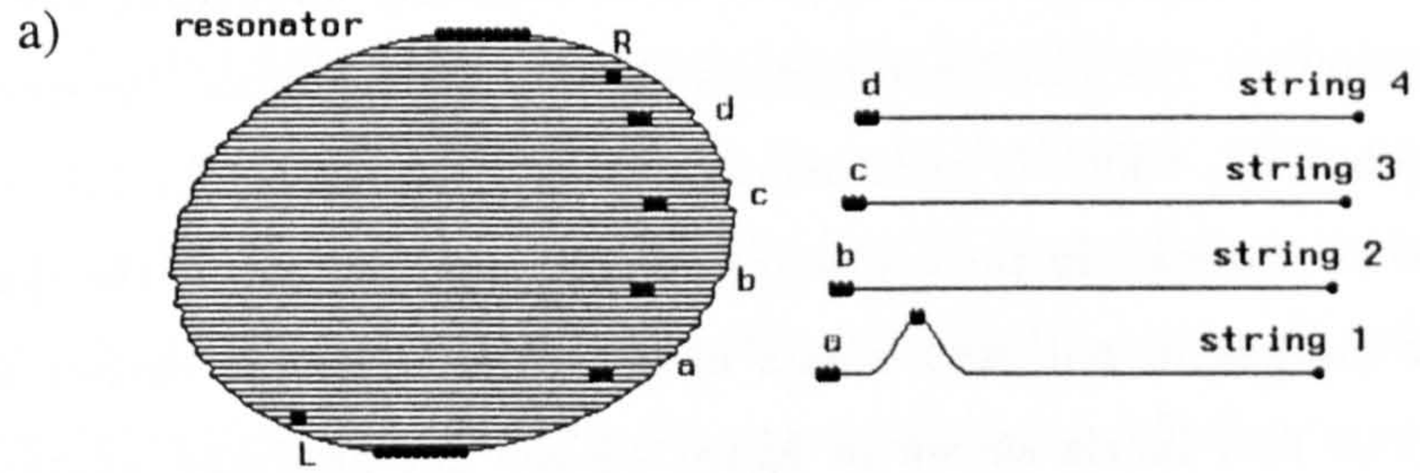


Figure 4.9: A stringed instrument with a circular resonator.

and **d** on a circular sheet. In (a) we see a force being applied to **string 1** which leads to two wavefronts, one left-going and one right-going. In (b) the left-going wavefront has been partially reflected off the end of the string, but some of its energy has also been transmitted to the circular resonator, causing another propagating wavefront. In (c) the energy which was initially imparted to **string1** increasingly spreads throughout the whole instrument making the other strings vibrate in sympathy. Note that the strings are constructed from the same virtual elastic material as two dimensional sheets and simply consist of a long line of cells linked by springs.

Graphical instrument animations form an integral part of TAO's user interface, and this image serves to introduce the visual format used, since it is actually a screen snapshot of a typical animation produced. Note that the individual cells are not

visible, and instead we see what appear to be smoothly undulating and continuous pieces of material. The user's attention is thereby focussed upon the emergent wave patterns produced, which in turn helps to focus attention on the behaviour of the instrument as a whole. In the computer animations produced by the system, glued points are marked in red, and points of excitation or sound output are marked in blue. The text labels shown in figure 4.9 are not produced automatically by the system, although script functions are provided enabling the user to label an animation manually.

4.5 Information needed to create a piece of material

Before moving on to a description of the mathematics involved in animating the model, we will take a closer look at the information which must be supplied in order for a new instrument to be created. Obviously, the quality of sounds which a particular synthesis technique is capable of producing is of paramount importance, but equally important is the question of how easy or difficult it is for a user to begin creating and using instruments. A delicate balance must be struck between, on the one hand, overloading the user with too many parameters in an attempt to cover every eventuality, and on the other, making the system too limited and inflexible as a result of trying to reduce the number of parameters which the user has to deal with.

In order to create a single piece of TAO material, three pieces of information are required: the shape of the component (currently circular, rectangular, one-dimensional, triangular, or elliptical); the size of the component; and the amount of damping initially applied to each cell. When a piece of material is initially created, all cells are given the same *mass*, *position*, *velocity*, *force* and *damping* coefficient. In this uniform state the instrument exhibits some properties which are predictable, such as the overall decay time and the fundamental frequency of vibration [2].

Acoustic instruments usually rely upon a variety of acoustic components coupled together, each one possibly making use of a different acoustic medium, but with TAO's cellular material all components initially exhibit a constant wave propagation

[2] In the case of a two-dimensional, inharmonic instrument such as a rectangular sheet, there may be no perceivable fundamental frequency

velocity. Therefore, in order to create a string of a given frequency f, for example, we can calculate its length from the wave propagation velocity v of the material, and the period of vibration $1/f$. There is no need to specify other non-musical information such as the string's mass per unit length, its tension or its length. As stated above, an instrument with uniform damping applied across all cells exhibits a predictable decay time depending on the exact damping coefficient used, and this holds true for an instrument of any size or shape. The initial damping coefficient chosen for any instrument may therefore be specified as a decay time, measured in seconds.

For circular sheets the information required also consists of a single frequency, used to determine the diameter of the sheet, and a decay time. For other two-dimensional instruments, an x and y frequency are required. The x frequency is used to determine the width of the instrument (at its widest point) and the y frequency is similarly used to determine the height. Note that these frequencies are only intended as a rough guide to the kind of spectrum produced. A rectangular sheet 200 Hz by 300 Hz will not produce clearly perceptible pitches at these frequencies, but we will have a good idea of the region of the audible spectrum this instrument will occupy as opposed to a sheet 700 Hz by 2 kHz.

Once we begin to upset the uniformity of the material things are not always as simple and predictable, but at least we have some simple starting point which allows instruments to be created with a minimum of information. Altering the masses of individual cells changes the fundamental frequency and modes of vibration of an instrument, and altering the damping coefficient changes the decay time and also the spectral evolution of the sound produced by the instrument. In practice the damping coefficient is set either as a decay time, or a percentage, where 0% means that the cell is totally undamped and 100% means that the cell is rigidly fixed in one position.

At this point the reader not interested in the inner workings of the cellular model should proceed to chapter 5 where the description of the system continues from a user's perspective.

4.6 Animating the model

In this section we see how the cellular model actually functions internally. There are a number of different levels at which we can view the model: as an abstract physical device; as a cellular model with update rules based on certain mathematical equations; as a set of data structures and algorithms; and finally, at the lowest level, as implementation code. We have already seen the physical structure of the model and now we turn to the cellular update rules and associated mathematics. Descriptions of the data structures and algorithms used are left to appendix D.

In order to animate the model the following steps are iteratively repeated as many times as is necessary, depending upon the number of output samples required:

1. The forces acting upon each cell due to the springs connecting it to its neighbours are calculated.
2. Any external forces due to excitations are applied to the appropriate cells.
3. The velocities and positions of each cell are updated according to the forces acting upon each.

These steps are described in more detail below.

4.6.1 Calculating all the internal forces within the material

At any instant in time each cell has an overall force exerted on it by the springs attaching it to its eight neighbours. The equation used to calculate the force exerted by one of the springs on a particular cell is based on Hook's law:-

$$F = -\lambda \frac{l' - l}{l}$$

where l is the equilibrium length of the spring, l' is the actual length at a particular time and λ is the coefficient of elasticity. The negative sign indicates a restoring force. However since the cells only move up and down relative to each other we can simply make the spring force a restoring force which is proportional to the difference between the positions of the two cells.

$$F = -\lambda(s_c - s_n)$$

where s_c is the position of the cell we are calculating the force for and s_n is the position of the neighbouring cell whose spring is exerting the force.

4.6.2 Applying any external forces

External forces are applied in order to simulate plucking, hitting, bowing or any other physical interaction with the material. At any instant in time any number of external forces can act upon any number of cells within a piece of material. To apply a force to a cell all that is needed is to add this external force to the cell's internal force which has already been calculated.

4.6.3 Updating the cell positions

Once the total force acting upon each cell has been calculated, Newton's second law of motion can be used in conjunction with the equations relating position, velocity and acceleration to update their velocities and positions. If F, a, v, s, m, are a cell's force, acceleration, velocity, position and mass respectively, then:-

$$F = ma$$

$$v = \frac{ds}{dt}$$

$$a = \frac{dv}{dt}$$

4.6.4 The discrete equations used to animate the model

In order to approximate the continuous equations given above for a discrete time domain simulation, the equations are rewritten:-

$$a_t = F_t/m$$

$$v_{t+1} = v_t + a_t \delta t$$

$$s_{t+1} = s_t + v_{t+1} \delta t$$

where a_t and F_t are the acceleration and force at time t respectively, v_t and v_{t+1} are the velocities at times t and $t+1$, and δt is the length of a discrete time step. The instantaneous acceleration for a cell is calculated from the force acting upon the cell

and its mass. This acceleration is then used to calculate the new velocity of the cell. Finally the velocity v_{t+1} is used to calculate the new position.

In practice energy is lost in any vibrating structure due to air resistance, acoustic radiation, internal friction etc. This is simulated by modifying the equation above to give:-

$$v_{t+1} = D(v_t + a_t \delta t)$$

where D takes a value between zero and one and represents losses due to damping. The velocity of each cell is multiplied by D on each time step. This leads to an overall exponential decay in the amplitude of oscillations. Each cell has its own value of D independent of other cells.

4.6.5 Improving the efficiency of the model

In order to make the material appear continuous rather than made up of discrete masses and springs, a large number of cells are required. Instruments may contain tens of thousands of cells and this makes the efficiency of the calculations very significant. For this reason some simplifications are made to the equations to reduce the number of arithmetic operations per cell, per time step.

If it is assumed that all the variables in the model are measured in arbitrary numerical units, then it is possible to eliminate some constants from the equations. If we assume that $\delta t = 1$ then the equations above become:-

$$a_t = F_t/m$$

$$v_{t+1} = D(v_t + a_t)$$

$$s_{t+1} = s_t + v_t$$

Also if we assume that the coefficient of elasticity $\lambda = 1$ then the equation for calculating the force between two cells becomes:-

$$F_c = -1(s_c - s_n) = s_n - s_c$$

Using this equation, the force acting upon cell c due to the spring connecting it to a neighbouring cell n can be calculated simply by subtracting the position s_c of c from the position s_n of n. If a particular neighbour is absent, no force is exerted on c from the direction of that (non-existent) neighbour.

As an aside, it may seem at first that by fixing the elasticity of all springs, serious limitations are placed upon the ability to control the material's physical characteristics. However, one of the objectives inherent in the design of TAO was to limit the number of parameters the user would have to deal with, without necessarily limiting the creative scope of the system. In practice, there are still many ways in which the characteristics of the material can be altered: by changing the masses and damping coefficients of each cell; by locking regions of cells; and by creating different shaped pieces of material, with different natural modes of oscillation. Combined with the ability to use all manner of different excitation models at arbitrary points on an instrument; to couple different pieces of material together; and to take sound output from any position on an instrument, there are still plenty of useful parameters to explore.

The sound examples described in appendix C also show that in practice losing the stiffness of each spring as a controllable parameter does not significantly reduce the range of sounds which TAO is capable of producing.

4.6.6 Altering the cellular update rules to cope with glued cells

In order to glue two cells together one is nominated as a *master* cell and the other as a *slave* cell. The master cell acts as if it were connected to the slave cell's neighbours as well as its own, giving a total of sixteen spring connections[3]. The total force acting on the master cell due to all of these springs is calculated in the usual way, comparing its position with the positions of the neighbouring cells. Once the force has been calculated and modified by the application of any external forces, the master cell's new velocity and position are updated in the usual manner, and are then simply copied to the slave cell. When the cellular update rules reach the slave cell no calculations actually take place since all the necessary calculations are carried out for the master cell. With this strategy, it doesn't matter whether the

[3] Assuming that neither the master or slave cell lie at a boundary in which case the number of neighbours will be reduced.

master cell or slave cell is updated first. The end result will always be the same.

4.6.7 Joining pieces of material by the installation of new springs

The cells along the boundary of a piece of material indicate the presence of the boundary with the use of null neighbour pointers. However if we take two such pieces of material with straight edges, by redirecting the null pointers along the edge of each piece of material so that they point at the appropriate cells along the edge of the other piece of material, we can effectively join the two pieces of material together seemlessly. Waves will now flow across the boundary between the two components as if the boundary never existed. This is achieved in practice by a process similar to stitching two pieces of material together. Two points on the respective edges of the two pieces of material are chosen as reference points to be lined up with each other. The cells along the two edges are then joined with newly created springs, gradually migrating along the join until the two edges begin to diverge, at which point the joining stops. This process is described in detail in section D.2.6. No modifications to the cellular update rules are needed since in order to install a new spring between two previously unconnected cells we simple redirect the appropriate neighbour pointers (previously null) so that the cells now point to each other.

Chapter 5

TAO's user interface

5.1 Introduction

In this chapter we move away from the underlying synthesis model and instead focus on how the user actually accesses it in practice. Whilst the synthesis model holds future potential for direct, real-time gestural control of instruments, the present (non real-time) implementation makes use of a text based script language. A TAO script contains all the information required to create and play instruments and generate soundfiles.

A TAO script is contained within one file but is conceptually split into two parts, the orchestra and score. The orchestra part of the script contains descriptions of all the instruments, microphones and performance parameters which are to be used in the score and the score enables complex events to be scheduled throughout the duration of the performance. This is similar to the Csound language described in section 3.1 but differs in that a TAO score takes the form of an algorithmic performance language rather than a set of pre-composed numerical performance data.

This chapter familiarises the reader with the general form of a TAO script and briefly describes the features available with the aid of examples. A more detailed reference manual can be found in appendix B. In addition more sophisticated script examples are given in appendix C and show how the sound examples which accompany this thesis were created.

5.2 The object oriented nature of TAO

TAO is implemented in the object oriented (OO) language C++ and whilst implementation details are left to appendices D and E there are certain ramifications to this choice of language, which directly affect the way in which the user approaches the task of constructing and interacting with instruments.

For readers not familiar with the OO programming paradigm, it encourages the design of well structured modular programs. An OO program defines a set of objects (data and algorithms grouped together) which map well onto the chosen problem domain, and then allows a problem to be described in terms of interactions with those objects. This fits the requirements of a synthesis system quite well since we can view instruments, microphones and cells etc. as objects with their own internally defined behaviour. A certain synthesis scenario is then described in terms of a particular configuration of objects and some sort of score which causes time domain interaction with those objects.

In OO terms, an object consists of a set of variables representing its internal state and a well defined, robust interface to the outside world which allows this internal state to be altered or interrogated. Objects are divided into *classes* and each individual object is referred to as being an *instance* of a particular class. For each object class a set of valid *messages* are defined which constitute the interface to the outside world. In C++ terminology messages are referred to as *member functions* but for the purposes of this thesis we will continue to use the more intuitive term *message*.

A message is sent to an object in C++ by appending the message with its arguments, if there are any, after the object's name separated by a period, i.e. *object_name.message_name(arg1, arg2, .., argn)*. TAO inherits this mechanism and various others from C++ which means that interaction with instruments in a TAO script is expressed in a syntax which is very close to that of C++. In practice, instruments are provided with messages for locking and damping parts of the material and selecting individual cells for input or output. For the cell object class the interface includes messages for applying forces or virtual bows to any cell, and for microphones it includes messages for sending sound samples to an output file.

5.3 The general form of a TAO script

The following example gives the reader an idea of the general form of a TAO script. It creates two rectangular instruments `rect1` and `rect2` and a stereo microphone `m`; locks the left hand side of `rect1` and the right hand side of `rect2`; joins the right hand side of `rect1` to the left hand side of `rect2`; applies an impulse to a point on `rect1`; and then eight seconds into the performance, damps a region of `rect1`. The left and right channels of sound output are taken from two points, one on either rectangle.

```
Rectangle rect1:  400 Hz, 600 Hz, 10 secs; ...
Rectangle rect2:  600 Hz, 400 Hz, 10 secs; ...

rect1.lockleft; rect2.lockright;

Join rect1(right,top) to rect2(left,0.5);

Microphone m:  outfile, stereo;

Parameter x=1/2, y=1/3;

Score 10 secs:
At 0 secs for 1 msecs:
     rect1(x,y).applyforce(10.0);
     ...

At 8 secs:
     rect1.setdamping(left,0.1,bottom,top,5%);
     ...

m.leftout:  rect1(0.1,0.9);
m.rightout:  rect2(0.9,0.1);
...
```

Without having introduced any of the language features yet, it should be clear from this example that the information contained within a TAO script is quite straightforward. Every attempt has been made to make the keywords used as clear and self-explanatory as possible. Note the use of the instrument messages `lockleft`, `lockright` and `setdamping`; the microphone messages `leftout` and `rightout`; and the cell message `applyforce` which, as described above, are appended onto the name of an object of the appropriate class, separated by a period. In the case of the line: `rect1(x,y).applyforce(10.0)`, the `(x,y)` operator selects a single cell from

instrument `rect1` at the specified coordinates, and this cell is then sent the message `applyforce(10.0)`.

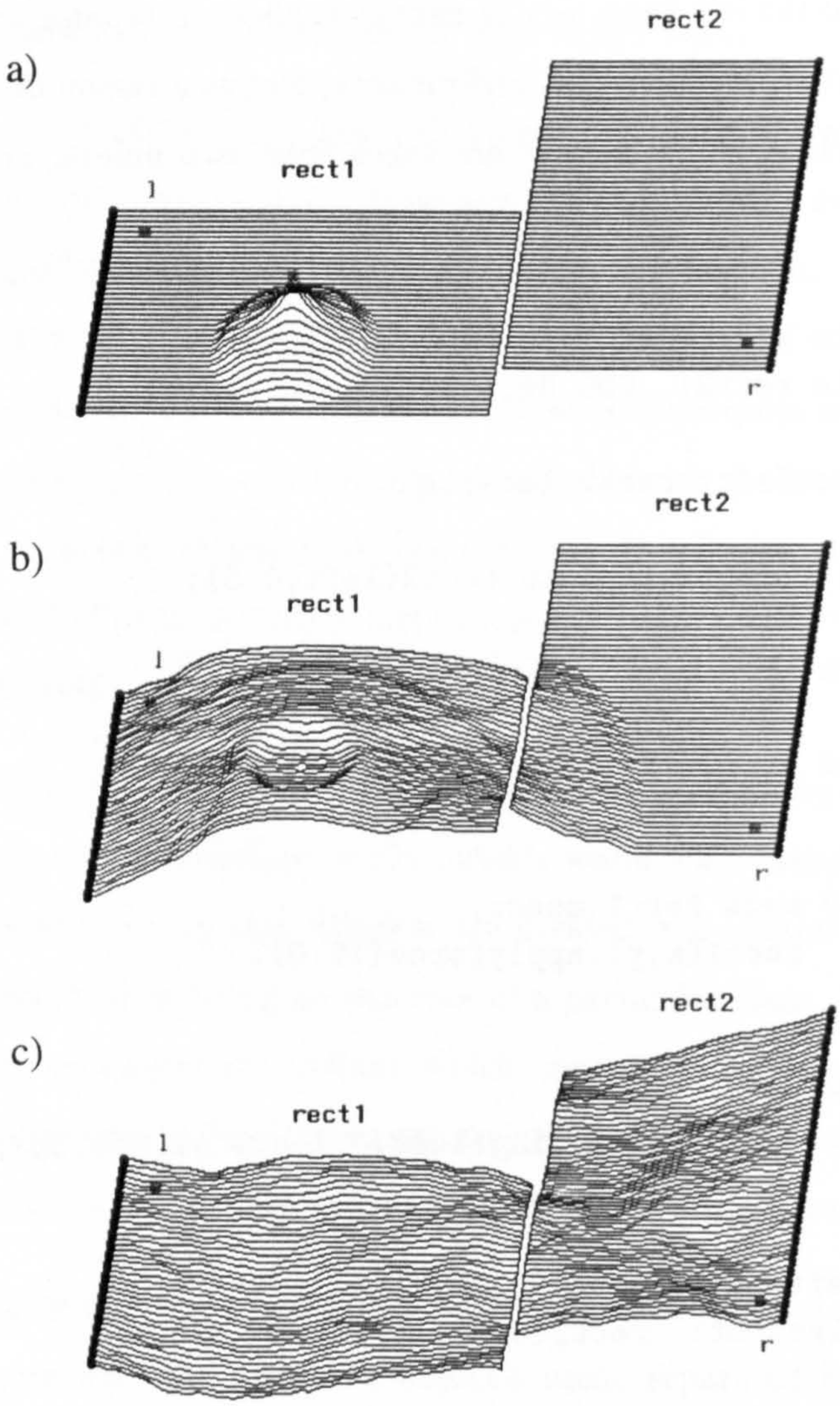


Figure 5.1: A simple instrument created from a TAO script

The instrument created by this script is shown in figure 5.1. The positions of the left and right output sources for the stereo microphone are shown at the points marked **l** and **r** and the heavier black regions of each rectangle represent the locked cells. Apart from the text captions in this figure, the rest of the graphics are produced automatically by the system from the information given in the script.

5.4 The orchestra part of the script

We will now work through the various features of a TAO orchestra one by one with the aid of examples.

5.4.1 Instrument declarations

The instrument declarations given below illustrate some key language features such as the use of pitches, frequencies and decay times given in seconds. They also show how messages can be sent to an instrument as soon as it is created in order to give it certain characteristics from the beginning of its life in the script.

```
(1)  String        string1:  100 Hz, 5.6 secs;
                   lockends;
                   setdecay(left, 1/10, 0.5 secs);
                   ...

(2)  Rectangle     rect1:  250 Hz, 760 Hz, 25 secs;
                   setdamping(left,1/5,bottom,1/5, 15%);
                   lockcorners;
                   ...

(3)  Ellipse       ellipse1:  C#8, Eb7, 1 min + 20 secs;
                   lock(0.3, 0.5);
                   ...
```

Each declaration is split into a *head* and *body* separated by a colon, with the body terminated by an ellipsis. This kind of syntax is used throughout the script whenever sets of instructions need to be grouped together into conceptual blocks and is also used in the score control structures which are introduced in section 5.5.

Declaration (1) creates a string called `string1` whose fundamental frequency and decay time are 100 hertz and 5.6 seconds. The messages `lockends` and `setdecay` do not need to specify the instrument to which they are being sent in this context (i.e. within an instrument declaration) as it is obvious which instrument is being referred to. The messages contained within the body of this particular declaration lead to the cells at the ends of the string to be locked and the decay time being altered for a region extending from the left hand side of the string to a point one tenth of the way along its length. The acoustic consequences of such local damping are explored more fully in chapter 6.

Declaration (2) creates a rectangular sheet of material 250 hertz by 760 hertz [1] with a decay time of 25 seconds and sets the damping coefficient to 15% in the bottom left hand corner, i.e. a rectangular region stretching from the left hand side to a point one fifth of the way across, and from the bottom to a point one fifth of the way up, before locking all four corners.

Declaration (3) creates an elliptical sheet of material, but this time instead of specifying the frequencies in hertz, a conventional pitch notation is used where `C#8` means the C♯ above middle C and `Eb7` means the E♭ just below middle C. Microtonal pitches are allowed by adding or subtracting a fraction of a semitone from the pitch given, e.g. `C#8+1/2, Eb7-1/3`. The instrument is given a decay time of 1 minute 20 seconds and a single point three tenths of the way from the left hand side and halfway up is locked.

Either of the pitch or frequency notations can be used for any of the frequencies required by instrument declarations.

5.4.2 Microphone declarations

The following microphone declarations show how *mono* and *stereo* microphones are created, and how they can either have fixed sources, where each channel takes its output from a single cell, or dynamically changing sources in which case the samples for each channel are calculated from arbitrary mathematical expressions given by the user in the score.

```
(1)  Microphone     mic1:  narrowsound, mono, string1(0.1);

(2)  Microphone     mic2:  widesound, stereo,
                           rect1(left, 1/2), rect1(right, 1/2);

(3)  Microphone     mic3:  sound1, mono;

(4)  Microphone     mic4:  sound2, stereo;
```

`mic1` takes its output from instrument `string1` at a point one tenth of the way along the string and writes its mono output samples to a file called *narrowsound.tao*. `mic2` takes its left and right channels of output from halfway up the left and right

[1] For an explanation of the meaning of the two frequencies see section 4.5

hand edges of instrument `rect1` respectively and writes its output to the file *widesound.tao*. The *.tao* files created by microphones contain raw floating point samples which are normalised and packaged into a standard `.aiff` soundfile by the `float2aiff` program described in appendix A.

Note that in examples (3) and (4) above, all that is given in each microphone declaration is the name of the microphone, the name of the output file and the number of channels. This indicates that the sources for each output channel will be specified later in the score. This feature, apart from allowing sound samples to be created from arbitrary expressions as described above, is also included for future compatibility with versions of TAO which will allow microphones to move around an instrument under algorithmic control during a performance.

5.4.3 Performance parameter declarations

Performance parameters are basically floating point variables which allow the user to store and mathematically manipulate data anywhere within a script. Any number of parameters may be declared. A parameter declaration consists of the keyword `Parameter` followed by a list of parameter identifiers with optional initial values. For example:

```
(1)  Parameter x;

(2)  Parameter x=0;

(3)  Parameter bowvelocity, bowforce, a=5, b=10;
```

5.4.4 Damping parts of an instrument

It is possible to damp individual points or regions of an instrument in a TAO script. For example if we want to damp an individual point one third of the way along a string called `string1` with a damping coefficient of one percent we can say:

```
string1.setdamping(1/3, 1%);
```

The next four examples illustrate how the instruments depicted in figure 5.2 may be damped as indicated by the shaded regions:

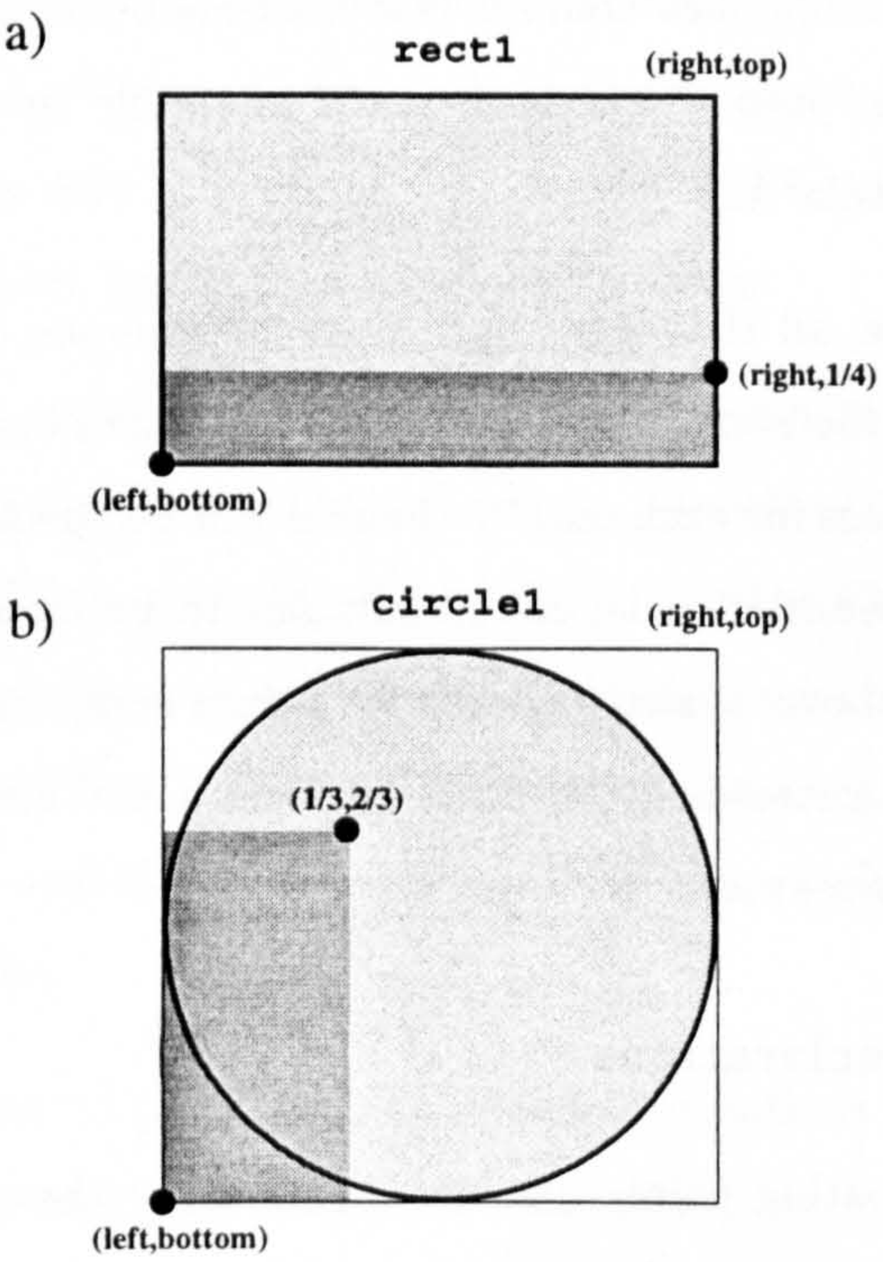


Figure 5.2: Damping local regions of instruments

```
(1)  rect1.setdamping(left, right, bottom, 1/4, 1%);

(2)  circle1.setdamping(left, 1/3, bottom, 2/3, 1%);

(3)  ellipse1.setdamping(0.2, 0.8, 0.33, 0.66, 1%);

(4)  string1.setdamping(left, 0.25, 1%);
```

In the present implementation only rectangular regions can be specified for two-dimensional instruments. Each rectangular region is specified by two vertices (x_l, y_l) and (x_r, y_r). The order in which the arguments x_l, y_l x_r, and y_r are passed to the `setdamping` message is `setdamping`$(x_l,\ x_r,\ y_l,\ y_r,\ ..)$. Note that `string1` only requires a pair of x coordinates specifying the endpoints of the damped region.

Messages like these can be sent to an instrument anywhere in a TAO script. Observing certain conventions, however, leads to more legible scripts. For example, messages which deal with the static structural properties of an instrument should be placed in the orchestra, whilst temporary changes used as part of a performance should be placed within the score. For more details about the different forms of the

`setdamping` message and related messages see appendix B.

5.4.5 Locking parts of an instrument

A number of messages are provided for locking single points or regions of an instrument. These include `lockleft`, `lockright`, `lockbottom`, `locktop`, `lockcorners`, `lockends` and `lockperimeter`. These are described in detail in appendix B. most of these messages are very straightforward in their behaviour with rectangular instruments, locking whole sides of the instrument, or, as in the case of `lockends`, the left and right sides simultaneously. However, for other shapes of material only the extremities of the instrument are locked except in the case of `lockperimeter` which works for any shape of instrument.

5.4.6 Stringing instrument messages together

It is possible to string messages destined for the same instrument together in the following manner:

```
(1)  string1.lockends.setdecay(left, 1/10, 0.5 secs);

(2)  rect1.setdamping(left,1/5,bottom,1/5, 15%).lockcorners;

(3)  ellipse1.lock(0.3, 0.5);
```

5.4.7 Glueing and joining instruments

To recap briefly there are two mechanisms for coupling instruments together. These are made available through the `Glue` and `Join` commands. Glueing forces two single points to move in unison as if they really were glued together. Any forces experienced by the first point will be experienced by the second and vice versa. Joining allows two sheets of material with straight edges to be sewn together by linking the cells along the two opposing edges with springs.

To illustrate the `Glue` command, supposing we have two instruments `string1` and `ellipse1` and we wish to glue the left hand side of `string1` to a point halfway across and one third of the way up `ellipse1`. We can achieve this with the following code:

```
Glue string1(left) to ellipse1(0.5,0.333)
```

Note that the coordinate system used in this example, and in cases where individual points on an instrument are accessed for input or output, is different to the system used for `setdamping`. For `setdamping`, $x = 0$ and $x = 1$ always indicate the left hand and right hand extremities of the instrument, respectively, i.e. the coordinates are measured relative to the bounding box which surrounds the instrument. Conversely, the coordinate system used for glueing and input/output always returns a point lying within the perimeter of the instrument for values in the range $0 \leq x \leq 1$ and $0 \leq y \leq 1$. This coordinate system is described in section 5.4.8.

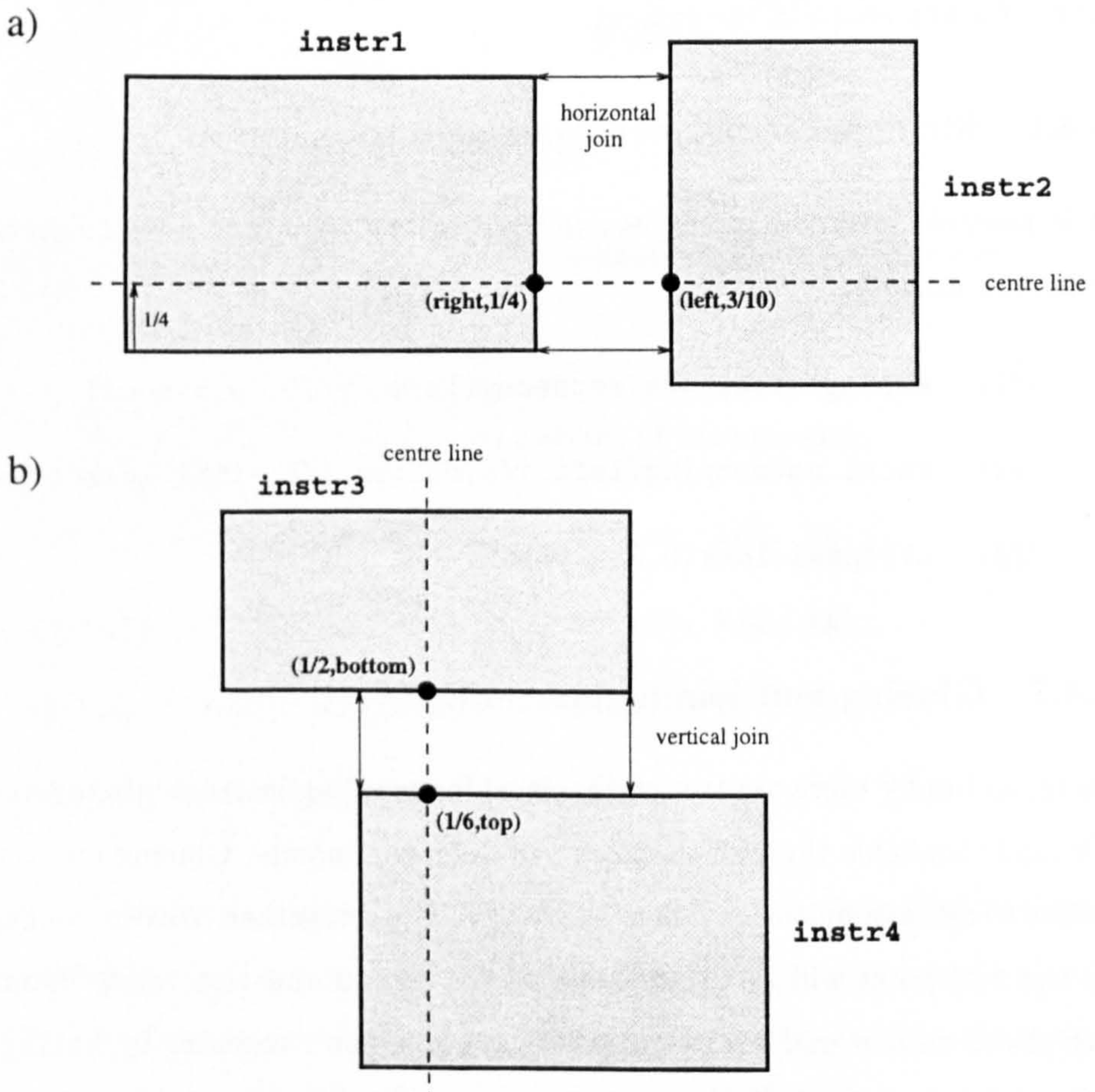


Figure 5.3: An illustration of the join facility

Figure 5.3 shows two examples of the use of the `Join` facility involving rectangular instruments. In the first example **instr1** and **instr2** are joined horizontally, and in the second **instr3** and **instr4** are joined vertically. These two 'joins' are achieved with the following script code:

(a) `Join instr1(right,1/4) to instr2(left,3/10)` or
`Join instr1(1,1/4) to instr2(0,3/10)`

(b) `Join instr3(1/2,bottom) to instr4(1/6,top)` or
`Join instr3(1/2,0) to instr4(1/6,1)`

In (a) the right hand side of **instr1** is joined to the left hand side of **instr2**. The two rectangles are lined up such that the point one quarter of the way up **instr1** lines up with a point three tenths of the way up **instr2**.

In (b) the bottom of **instr1** is joined to the top of **instr2** such that a point halfway across **instr3** lines up with a point one sixth of the way across **instr4**.

It is possible to join the left hand side of one instrument to the left hand side of another, or the bottom of one instrument to the bottom of another etc. It is also possible to join two opposing sides of the same instrument together, either with the same centre line for each side, in which case it is as if the instrument has been wrapped round to form a cylinder, or with different centre lines in which case the instrument is slightly twisted as well as being wrapped round. If we take this to its logical limit, we can join both pairs of opposing sides on a rectangular instrument in which case we end up with an instrument with the modes of vibration of a toroidal shaped piece of material.

It is, however, not possible in the present implementation to join the bottom of one instrument to the left hand side of another, for example, or to join instruments with curved edges, although with careful thought and some clever algorithms such variations are possible in principle since all that joining does is to add in new springs between individual cells.

5.4.8 Simulating physical interaction with instruments

Before describing the structure and function of a TAO score we now move on to the most important feature of the script language: the ability to simulate physical interaction with instruments. The general notation used to access a point on an instrument is shown below:

(1) *instrument*(*x*)

(2) *instrument*(*x*,*y*)

Notation (1) is used for strings and notation (2) is used for all the other two-dimensional instruments. The two coordinates x and y are always normalised such that $x = 0$ indicates the left hand side of the instrument and $x = 1$ indicates the right hand side. Similarly $y = 0$ always indicates the bottom of the instrument and $y = 1$ indicates the top. With a rectangular sheet of material the interpretation of these coordinates is straightforward but things are slightly more complicated for other shaped pieces of material.

The coordinate system is designed such that, regardless of the shape of an instrument, values of x and y lying between 0 and 1 will always specify a point which lies somewhere within the perimeter of the instrument. The way in which this is achieved is shown in figure 5.4. The y coordinate is referred to first to see how far up the instrument to move. Then the x coordinate system is adjusted to fit the left and right edges of the instrument at that y position. One of the advantages of taking this approach is that we can change the shape and size of the instruments defined in the orchestra without affecting the validity of the score.

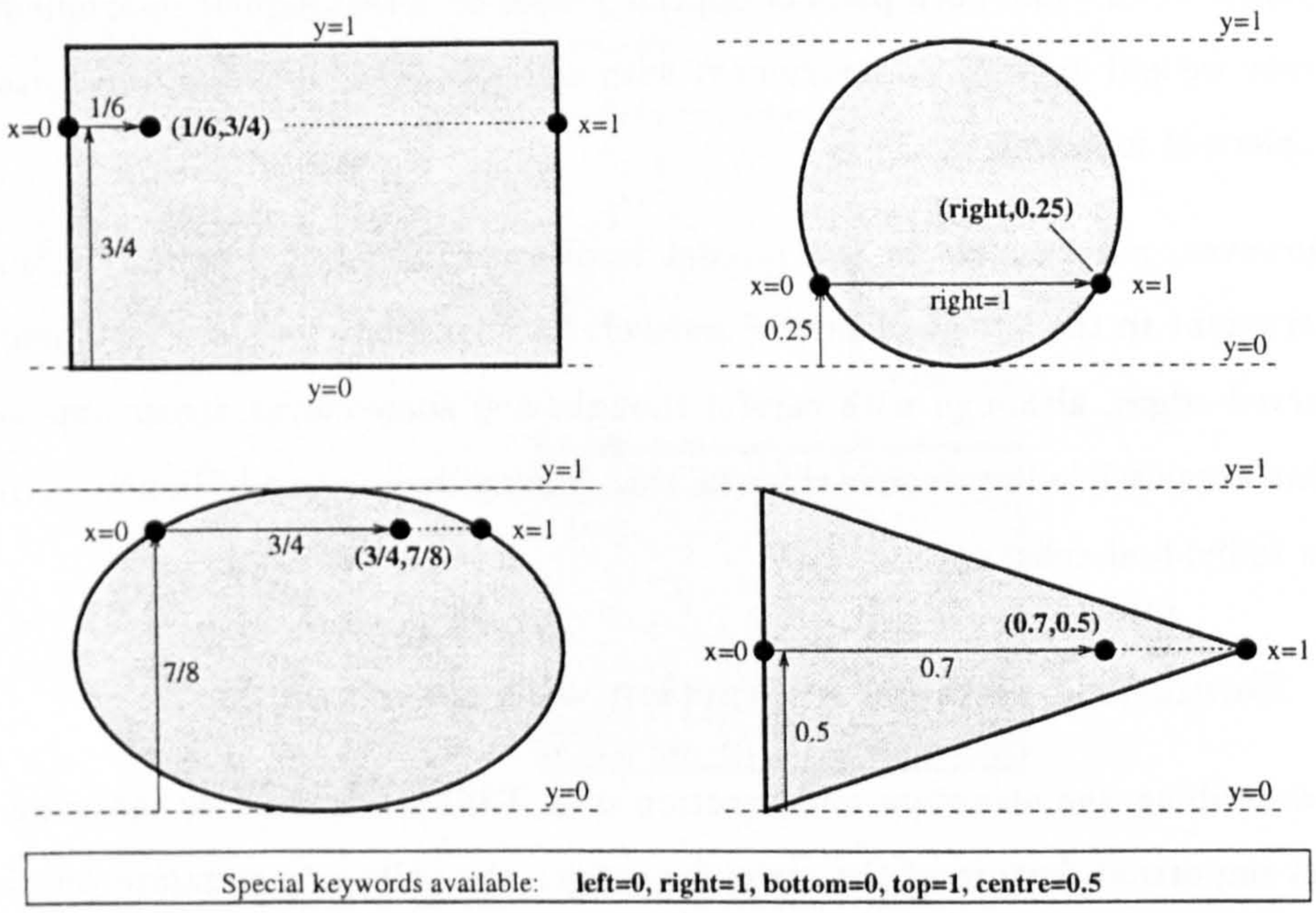


Figure 5.4: The instrument coordinate system

Once we have selected a point on an instrument using this notation we can gain access to the individual variables stored within the cell at that point, in other words

we can apply forces to it or find out its position or velocity etc. Attributes of a cell which are of direct interest to the user include `position`, `velocity`, `force` and `mass` and they are accessed as in the following examples which assume the existence of four instruments `circle1`, `rect1`, `ellipse1` and `triangle1`.

```
circle1(x,y).position;

rect1(x,y).velocity;

ellipse1(x,y).force;

triangle1(x,y).mass;
```

One of the most significant differences between the approach to sound synthesis taken by TAO and more traditional unit generator based languages such as Csound is that the variables representing the internal state of a cell can be used either for input or output. In contrast data always flows in one direction in a network of unit generators. In some cases cell variables are used both for input and output simultaneously, such as when a cell is bowed. The bowing model, described in appendix F needs to apply forces to the cell but at the same time needs to get feedback about the cell's velocity etc. in order to calculate the force to apply.

Some examples are given below of the kind of expressions, involving cell attributes, which might be found in a TAO script:

```
string1(1/3).position=10.0;

If string1(0.5).velocity > 10.0:   do something ...

string1(1/10).force += 5.0; add 5 to the cell's force

string1(left).mass=50.0; sets a single cell's mass to 50
```

Two messages `applyforce(`f`)` and `bow(`f_{bow}`,` v_{bow}`)` are provided for use with cells. For example, assuming the existence of a string called `string1` we can apply a force f to a point one third of the way along the string with the following script code:

```
string1(1/3).applyforce(f);
```

Of course this single line on its own does not contain enough information to say when and for how long the force should be applied. This is where the score language comes in. For example, to specify that the force should be applied at zero seconds for one tenth of a second (i.e. at the start of the performance) we can say:

```
Score 10 secs:
At 0 secs for 1/10 secs:
     string1(1/3).applyforce(f);
     ...
...
```

The `At..for` *control structure*[2] has a head and a body separated by a colon, with the body terminated by an ellipsis. The `Score` control structure is mandatory and simply specifies how long the performance is to last.

5.5 The score

So far we have seen how the user creates the objects which serve as the material for the synthesis and now we turn to how the user actually describes a performance. TAO's score language enables a performance to be described in terms of hierarchically nested *events*. The term *event* is generic in nature and is used to refer to both very simple events such as initialising a parameter or sending some text to the output window, and to complete sound events potentially consisting of hundreds of nested sub-events.

Events may occur at an instant in time, repeatedly at regular intervals or over some interval of time. It is also possible for several events to overlap in time. In order to cope with all this variety a set of control structures is provided, including the `At..for` structure introduced in the previous section. They may be nested within one another and in combination make it possible to describe all events from the simplest to the most complex.

---

[2] A term borrowed from conventional programming languages such as C and C++ where control structures usually include `for` and `while` loops and conditional statements such as `if` and `if..else`

5.5.1 The score control structures

The control structures are listed below:

(1) `At` *start time* `for` *duration*: *body* ...

(2) `From` *start time* `to` *end time*: *body* ...

(3) `Before` *end time*: *body* ...

(4) `After` *start time*: *body* ...

(5) `At` *time*: *body* ...

(6) `Every` *interval*: *body* ...

(7) `ControlRate` *interval in samples*: *body* ...

(8) `If` *condition*: *body* ...

(9) `If` *condition*: *body 1* ...
`Else`: *body 2* ...

(10) `If` *cond 1*: *body 1* ...
`ElseIf` *cond 2* : *body 2* ...
`ElseIf` *cond 3* : *body 3* ...
.
.
`Else`: *body n*

Each control structure has a *head* and *body*. The body contains instructions which specify *what* to do and the head specifies *when* to do it. The syntax is similar to that used in the instrument declarations already introduced in that the head and body are separated by a colon and the body is terminated by an ellipsis. On each time step the whole score is executed from top to bottom and the control structures have the job of ensuring that certain sets of instructions are executed only at the times specified.

`At..for`, `From..to`, `Before` and `After` allow the instructions which form the *body* to be executed only during a specified time interval. In the case of `At..for` and `From..to` both the start and end time are explicitly given, but `Before` and `After` only specify one of these times. The unspecified time is implicitly calculated according to the position of the `Before` or `After` structure within the score.

`At` allows the instructions in the *body* to be executed just once at the specified time. `Every` allows a set of instructions to be executed periodically and is most often used to schedule text output during a performance. This is useful for debugging or simply producing a profile of a performance, detailing how the various parameters change and when the various events occur.

`ControlRate` is similar to `Every` but allows the interval to be specified in time steps rather than seconds. This allows signals to be updated less frequently than at full audio rate and is reminiscent of the *k-rate* signals provided by Csound (see section 3.1).

In the case of the `If` and `If..Else` control structures, the *body* is executed at *any* time so long as the condition contained in the head evaluates to *true*.

5.5.2 The special variables `start` and `end`

It often occurs that we need to execute some instructions just once at the very beginning or end of a time interval. This is analogous to Csound's i-rate evaluation whereby certain parameters are only evaluated once at the beginning of a new note in the score. In order to achieve this in a TAO score we can use two special variables `start` and `end` whose values change throughout the score depending on their context. For example we can say:

```
Score 10 secs:
     At 0 secs for 5 secs:
          At start: at 0 seconds do X ...
          At end:   at 5 seconds do Y ...
          rest of body
          ...
     ...
```

which makes the hierarchical nature of the events explicit unlike the following equivalent example which places all three events at the same level of scope:

```
Score 10 secs:
     At 0 secs for 5 secs:  body ...
     At 0 secs:  at 0 seconds do X ...
     At 5 secs:  at 5 seconds do Y ...
     ...
```

Another (implicit) use of **start** and **end** occurs when the two special time varying functions **linear** and **expon** are accessed within a score. Both functions take two arguments, an initial value and a final value, and return a value which gradually changes from the initial value to the final value over a time interval which depends on where they are accessed within the score. In order to work out the time interval over which they are supposed to change, both functions access the **start** and **end** variables. In the following example, two parameters **x** and **y** are declared. **x** is made to change exponentially from 100 to 1 over the first four seconds of the score whilst **y** is made to change exponentially from 200 to 5 over the whole duration of the score:

```
Parameter x, y;

Score 10 secs:
     y=expon(200.0, 5.0);
     Before 4 secs:
          x=expon(100,1); ...
     Every 1 secs:
          Display "At time", Time;
          Display " x=", x;
          Display " y=", y, newline;
          ...
...
```

This produces the output shown below. The two invocations of the **expon** function in the example above are said to be at different levels of *scope* within the score since the first is at the top level of the score, whilst the second is nested within the **Before** control structure. At the top level of the score **start** always takes the value 0 seconds, and **end** always takes the value specified by the **Score** control structure.

```
At time 0.0000    x=100.0000   y=200.0000
At time 1.0000    x=31.6236    y=138.3006
At time 2.0000    x=10.0000    y=95.6352
At time 3.0000    x=3.1622     y=66.1320
At time 4.0000    x=1.0000     y=45.7305
At time 5.0000    x=1.0000     y=31.6228
At time 6.0000    x=1.0000     y=21.8672
At time 7.0000    x=1.0000     y=15.1213
At time 8.0000    x=1.0000     y=10.4564
At time 9.0000    x=1.0000     y=7.2306
At time 10.0000   x=1.0000     y=5.0000
```

We can place both expon function calls inside the Every control structure, thus evaluating x and y only once per second, without affecting the time interval over which they change:

```
Parameter x, y;

Score 10 secs:
     Every 1 secs:
          y=expon(200.0, 5.0);
          Before 4 secs:
               x=expon(100,1); ...
          Display "At time", Time;
          Display " x=", x;
          Display " y=", y, newline;
          ...
...
```

For more details on the scope facility and start and end see appendix B.

5.5.3 Mathematical functions provided

All the standard mathematical functions such as sin, cos, tan, sqrt etc. are available for use within a script since they are provided by the underlying implementation language C++. There are also two functions random and randomi which return a random number between two specified limits inclusive. random takes two real numbers as arguments and returns a real number and randomi is an integer version. The two special time varying functions expon and linear have already been introduced.

5.5.4 Generating sound output

Once we have created a microphone we can send sound samples to it throughout a performance. The microphone collects these samples together and writes them to a file in chunks. If we have declared a mono microphone m1, a stereo microphone m2, and a two-dimensional instrument instr1, we can generate sound samples in the following way:

(1) `m1.output: instr1(x, y);`

(2) `m2.leftout: instr1(x, y);`

(3) `m2.rightout: instr1(x, y);`

These microphone output messages are subject to the same scope rules as any other instructions, i.e. if they are placed at the top level of the score, then samples are generated on every time step throughout the performance, but if they are placed within the body of a control structure, they can be made to generate samples only at certain times or when certain conditions arise. All three messages can take arbitrary mathematical expressions as arguments. For example:

```
Parameter theta, amplitude, x, y, x1, x2, y1, y2, position

m1.output:  sin(theta)*amplitude*rect1(x, y);
m2.leftout:  rect1(x1, y1) + string(position);
m2.rightout:  rect1(x2, y2) + string(1-position);
```

5.5.5 Generating iterated events

Many sound events require excitations to be applied to an instrument on an iterative basis. For example, suppose we wish to describe a sound which simulates the effect of an object repeatedly bouncing on an instrument and losing energy on each bounce. How would we describe this scenario? The following example demonstrates one approach to this problem, showing how iterative events may be generated. Note that there is no orchestra, and the score merely produces text output rather than actually playing any instruments.

```
Parameter which_string, string_position;
Parameter now=0 secs, interval, force;

Score 10 secs:
     ControlRate 100:
          interval=expon(2 secs, 0.3 secs);
          ...

     At now for interval - 1 msecs:
     At start:
          which_string=randomi(1, 4);
          force=random(1, 10);
          string_position=random(left, right);
          Display newline,"At", Time;
          Display " strike string", which_string;
          Display " at (",string_position;
          Display ") with force", force, newline;
          ...
```

```
            At end:
                 now += interval;
                 Display "Next strike to occur at", now, newline;
                 ...
            ...
        ...
```

At the very beginning of the score the only parameter which is initialised is `now` which represents the start time of each strike. The parameters `which_string`, `force` and `string_position` are randomly determined once at the beginning of every strike within the given limits. The parameter `interval` is used to determine the interval between strikes and is made to change exponentially over the duration of the performance from a value of 2 seconds to a value of 0.3 seconds.

At the beginning of every (simulated) strike a number of parameter values are displayed, including: the time; the string number; the position on the string; and the force. At the end of every strike, the start time for the next strike is calculated using the `+=` operator which adds the result of the expression situated to its right to the parameter specified, in this case `now`.

This example produces the following output:

```
At 0.0000 strike string 3.0000 at (0.8916) with force 1.3463
Next strike to occur at 1.5040

At 1.5040 strike string 2.0000 at (0.3571) with force 7.7562
Next strike to occur at 2.7022

At 2.7022 strike string 4.0000 at (0.1037) with force 5.3566
Next strike to occur at 3.6948

At 3.6948 strike string 2.0000 at (0.3987) with force 3.0201
Next strike to occur at 4.5403

At 4.5403 strike string 3.0000 at (0.8630) with force 2.7436
Next strike to occur at 5.2756

At 5.2756 strike string 3.0000 at (0.5316) with force 1.7555
Next strike to occur at 5.9256

At 5.9256 strike string 4.0000 at (0.5754) with force 1.8750
Next strike to occur at 6.5078

At 6.5078 strike string 2.0000 at (0.2082) with force 7.2068
Next strike to occur at 7.0345

At 7.0345 strike string 3.0000 at (0.3587) with force 5.0737
Next strike to occur at 7.5153

At 7.5153 strike string 4.0000 at (0.5864) with force 8.1805
Next strike to occur at 7.9574

At 7.9574 strike string 4.0000 at (0.2555) with force 8.1723
Next strike to occur at 8.3665

At 8.3665 strike string 1.0000 at (0.0074) with force 3.9198
Next strike to occur at 8.7471

At 8.7471 strike string 3.0000 at (0.3450) with force 5.2898
Next strike to occur at 9.1029
```

```
At 9.1029 strike string 4.0000 at (0.3496) with force 2.2259
Next strike to occur at 9.4368

At 9.4368 strike string 4.0000 at (0.2843) with force 3.0886
Next strike to occur at 9.7514

At 9.7514 strike string 2.0000 at (0.9956) with force 8.5943
```

It is important to understand that the structure of a TAO score is completely independent of the orchestra and a TAO script need not deal with instruments at all. This is sometimes useful for developing complex score algorithms without incurring the computational overhead associated with the synthesis engine. Another significant point is that events do not have to be ordered chronologically from top to bottom in the score but may be placed in any order and may overlap in time. The ordering of events is usually only significant when one event depends on some parameter values which are calculated within another event. In this situation, the event which performs the parameter calculations must be executed before the other event and should therefore be placed nearer the beginning of the score, text-wise.

5.5.6 The use of C++ code fragments within a script

A TAO script is actually a fragment of C++ code in disguise and is translated into C++ before being compiled and linked with the library of TAO objects and functions. It is therefore perfectly acceptable to use standard C++ code within a script. This includes most usefully `for` and `while` loops, declarations of variables with types such as `int` and `char`, and arrays.

Since TAO's script language contains features such as the instrument declarations and score control structures which involve the grouping together of sets of instructions, the standard curly bracket syntax of C++ could have been adopted, but a conscious decision was made to avoid this, in order to emphasise that the semantics of a TAO script are not those of a conventional algorithmic language. This also has the advantage of making any C++ code included within a script stand out more clearly.

This feature need not concern the inexperienced user but gives the advanced user the opportunity to express more sophisticated concepts in a script. Also, since the system is under constant development this facility is useful for developing and testing new TAO features before inclusion in the system.

An example of the use of C++ code within a TAO script is given below:

```
Score 0.1 secs:
     ControlRate 100:
          for (int i=0;i<10;i++) C++ code fragment
          If i > 5:
               Display i, "is greater than 5", newline;
               ...
          ...
```

This repeatedly displays the following lines of output on every 100th time step (a valid but rather trivial script):

```
6 is greater than 5
7 is greater than 5
8 is greater than 5
9 is greater than 5
```

A useful side effect of the fact that C++ underlies TAO's script language is that lines can be commented out in a script by placing a // at the beginning of a line or by enclosing a group of lines between a /* and */.

5.6 Summary of script features

We will end with a summary of script features. For a more detailed reference manual see appendix B.

Instrument creators

The keywords `String`, `Rectangle`, `Circle` and `Triangle` are provided for the creation of instruments. `String` and `Circle` require one frequency to be specified determining the length of the string or the diameter of the circle respectively. `Rectangle` and `Triangle` require two frequencies to be specified, one for the x direction and one for the y direction.

Pitch nomenclature

The frequency of an instrument can be specified in Hertz, conventional notation (including micro-tonal adjustments) or by using a numerical notation of the form

`pitch(`*octave*`.`*semitone*`)`. The conventional notation consists of a note name (e.g. `C`, `C#`, `Bb`, `G`) followed by an octave number (middle C is in octave 8) and an optional microtonal adjustment consisting of a `+` or `-` and a fraction representing the fraction of a semitone that the pitch should be sharpened or flattened by. For example, `C#7`, `Eb6+1/2`, `pitch(8.02)` and `500 Hz` are all valid ways to specify the fundamental frequency of an instrument.

Instrument modifiers

The keyword `lock` allows a single point on an instrument to be locked. It appears in the form `instrument.lock(x,y)` or just `lock(x,y)` in the body of an instrument declaration. The keywords `lockleft`, `lockright`, `locktop` and `lockbottom` allow whole sides of an instrument to be locked. `lockcorners` and `lockperimeter` are self-explanatory.

The `setdamping` and `setdecay` keywords allow the damping factor to be set at a single cell or over a region of cells. Damping can be set in terms of a percentage, where zero per cent means no damping at all and one hundred per cent means that the cell/cells are fixed rigidly in one position, or in terms of a decay time. The keywords `resetdamping` and `resetdecay` allow the damping to be reset to the default value defined when the instrument was created.

Accessing points on an instrument

Points may be accessed on an instrument using notation like `instr(x,y)` for two-dimensional instruments or `instr(x)` for strings. Coordinates are normalised to lie between zero and one and regardless of the shape of the instrument and within these limits the point specified will always lie inside the perimeter of the instrument. The keywords `left`, `right`, `top`, `bottom` and `centre` are provided and evaluate to numerical constants, i.e. 0, 1 or 0.5.

Physical interaction with instruments

The physical attributes of a cell can be accessed using the keywords `force`, `velocity`, `position`, `mass` and `damping`. In order to access these attributes a cell must first be selected using the notation introduced above.

Microphone keywords

The keyword `Microphone` declares a virtual microphone. It is followed by the name of the microphone, a colon and then the name of the file to which sound output is to be written. The output sources for a microphone can be specified within the microphone declaration by specifying one or two cells (only mono and stereo microphones are currently supported) after the soundfile name. Alternatively one of the keywords `mono` or `stereo` can follow the soundfile name in which case the appropriate number of channels worth of output samples are generated via arbitrary mathematical expressions in the score.

Sound output

The `output`, `leftout` and `rightout` keywords allow sound samples for each microphone channel to be generated within the score via arbitrary mathematical expressions usually involving at least one point on an instrument.

Time nomenclature

The keywords `secs`, `msecs` and `mins` are provided. These are placed after a numerical value and automatically convert the value to a numerical constant, measured in seconds, according to the units chosen. The keyword `Time` can be used anywhere within the score to refer to the time elapsed since the beginning of a performance (real-time when the sound is played back but non-real-time during synthesis).

Score control structures

The score is a hierarchical structure comprising nested control structures with the `Score` control structure at the top of the hierarchy. Control structures include `At`, `Before`, `After`, `From..to`, `At..for`, `Every` and `ControlRate`. The `If`, `If..Else` and `If..ElseIf..Else` control structures allow for conditional execution of sets of instructions.

Mathematical functions

In addition to standard mathematical functions such as `sin`, `cos`, `sqrt`, `random` etc., two special time varying functions `linear` and `expon` are provided. When placed in

the body of a control structure, these functions return a value which automatically changes linearly or exponentially over the time interval specified in the head of the control structure. When placed inside an `Every` or `ControlRate` structure they are evaluated less frequently, but the time interval over which they change is determined by the control structure containing the `Every` or `ControlRate` structure.

Performance parameters

The keyword `Parameter` allows the user to declare any number of performance parameters with optional initial values.

Screen output

The `Display` keyword enables text output during a performance and is most useful for debugging a synthesis since the values of various performance parameters can be displayed as the performance evolves. It is also useful when developing a complex score as it allows the macro-structure to be fine tuned before the score is actually used in conjunction with some TAO instruments.

Chapter 6

Practical examples of TAO's capabilities

6.1 Introduction

Having learnt about the abstract structure of the synthesis model and the script based interface to the system we are now in position to forget such internal details and instead concentrate on the behaviour of TAO instruments and the various strategies which may be employed in designing new instruments. This chapter gives some instrument examples, highlighting particular structural details which have a significant effect on the acoustic properties of the instruments. It also discusses how these characteristics may be controlled with the careful use of the damping facility, and other factors such as the placement of microphones. We begin by looking at some of the appealing characteristics of the synthesis model.

6.2 Transient behaviour of a circular sheet

It is well understood that the transients inherent in instrumental sounds are extremely important from a perceptual point of view. If the transient portions of various recorded instrumental sounds are removed, leaving relatively steady state portions of sound, it can be difficult to distinguish between different instruments. Even though a transient may only last for a fraction of a second, the information it creates is sufficient for us to judge in a very direct way the kind of mechanism

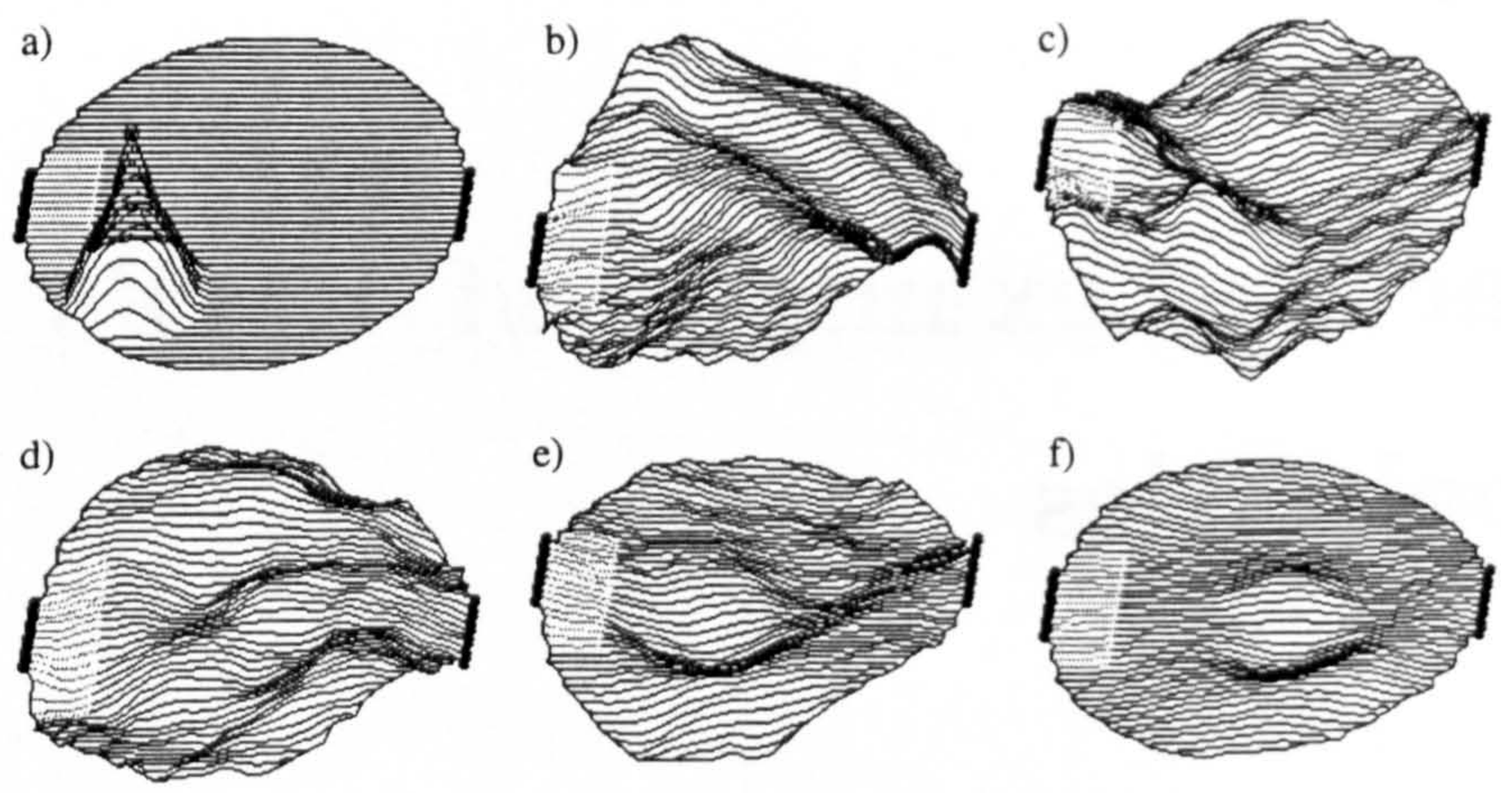


Figure 6.1: Attack transients in a circular sheet

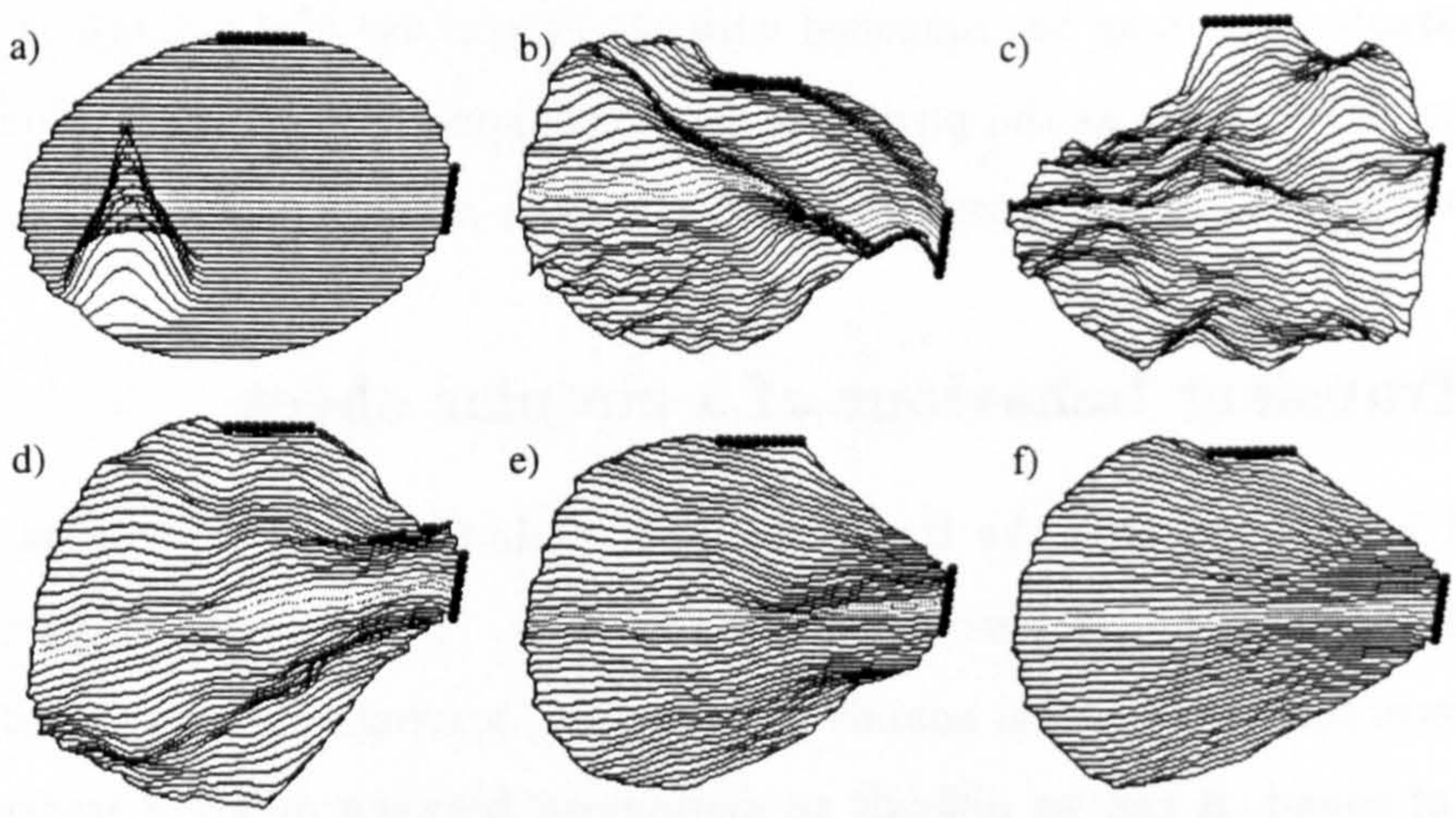


Figure 6.2: Attack transients in a different circular sheet

responsible for the sound (see section 1.6.2).

Figures 6.1 and 6.2 show how two slightly differently configured circular sheets progress from initial transient states caused by a single impact, through various intermediate vibrational patterns, to more steady patterns of vibration. The shaded regions in both figures represent regions of local damping. In both cases the instruments find a natural path from their initial excited state to a smoother, lower energy vibrational pattern which is compatible with the region of damping. We can lock and damp any part of an instrument, or even change the conditions as the sound unfolds and yet, because of the inherently holistic nature of the cellular model, and the physical nature of the underlying cellular update rules, the resulting sounds will always exhibit a certain 'solidity' or coherence.

6.3 An instrument comprising joined rectangular sheets

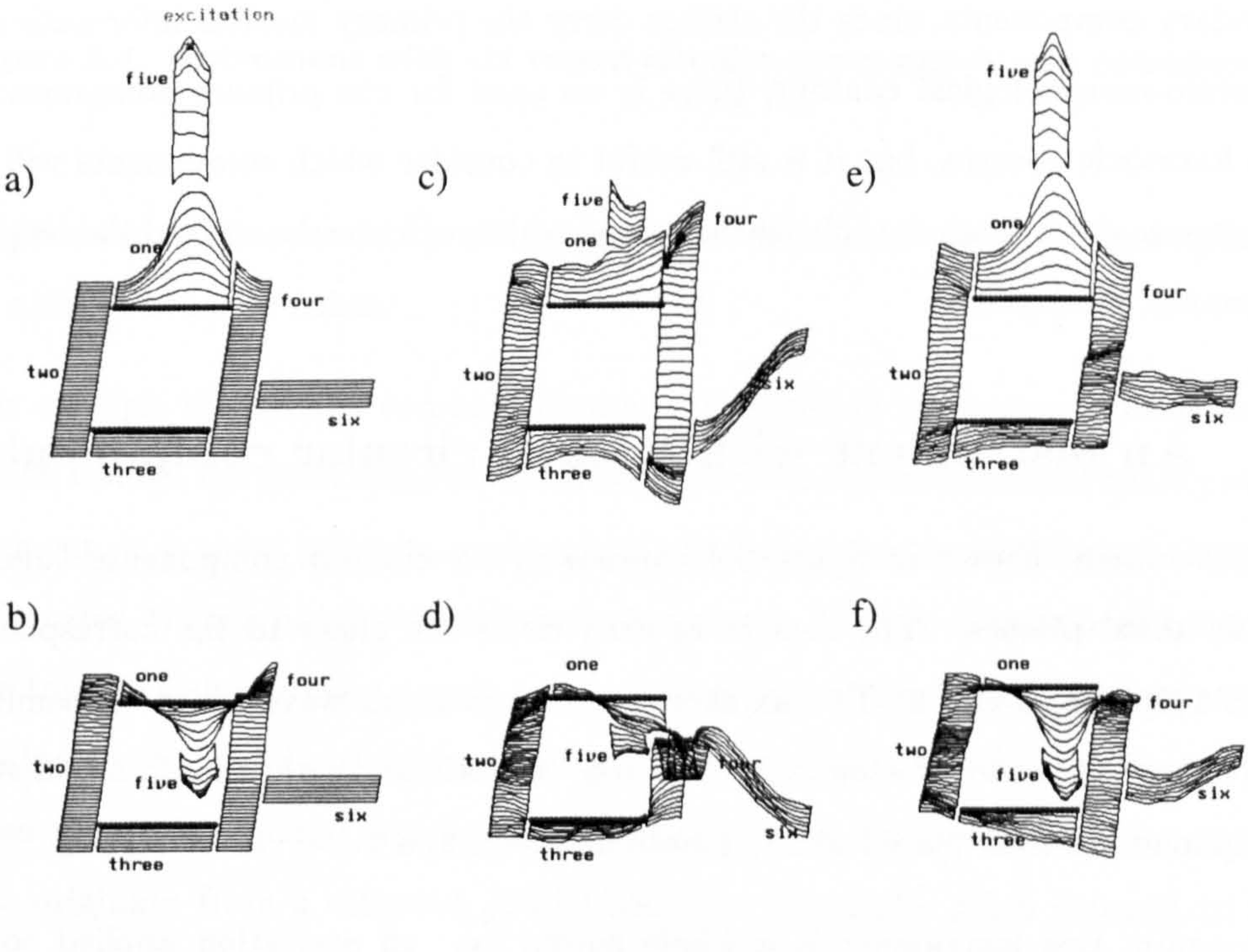


Figure 6.3: An instrument consisting of rectangular sheets joined together

Figure 6.3 shows an instrument consisting of several rectangular sheets joined together, and its typical models of vibration. Instruments such as this produce inharmonic sounds in general, but the precise nature of their spectral structure can be controlled by changing the sizes and characteristics of the various components. If the

components are given closely related frequencies then the instrument will be prone to beating effects, as in figure 6.3, where different components alternate between low and high amplitudes of vibration. If all of the rectangular components are given the same decay time, then none of them will 'stand out' in the resulting sound. If, on the other hand, one component is made significantly larger or smaller than the others, or is given a significantly longer or shorter decay time, then that component will begin to make its mark on the identity of the sounds produced, more than any other.

In practice, since instruments like the one shown in figure 6.3 are whole entities, there are no clear dividing lines between what is contributed to a sound by any one component and what is contributed by another. Having said that, it is sometimes useful to think in terms of *primary* and *secondary* components when designing TAO instruments. In a traditional musical context we might say that the strings of classical guitar are the primary components whilst the body and air cavity are secondary components, since the strings carry the primary musical information. In a spectro-morphological context, there is no need for the primary components to have harmonic spectra, but it is still useful to consider which components will give an instrument its most notable features, and which will merely add subtle details to the sounds produced.

6.4 An instrument with pitched circular components

The instrument shown in figure 6.4 consists of six circular components tuned to specific tonal pitches. The centres of each circle are glued to the corresponding points **1**,**2**, **3**, **4**, **5** and **6**. The six short resonators act as waveguides, transmitting some energy to the long resonator at the top. The left and right channels of a stereo microphone are then placed at the points marked **l** and **r**.

Once again, this instrument is a whole entity, i.e. an excitation applied to any part of the instrument has a very subtle, knock-on effect on the other parts of the instrument. The one-dimensional resonators behave here as abstract one dimensional waveguides capable of physically transmitting energy back and forth between other components. Although they are referred to as 'strings' in the script, in practice they do not have to behave in a string-like manner. In certain situations and with

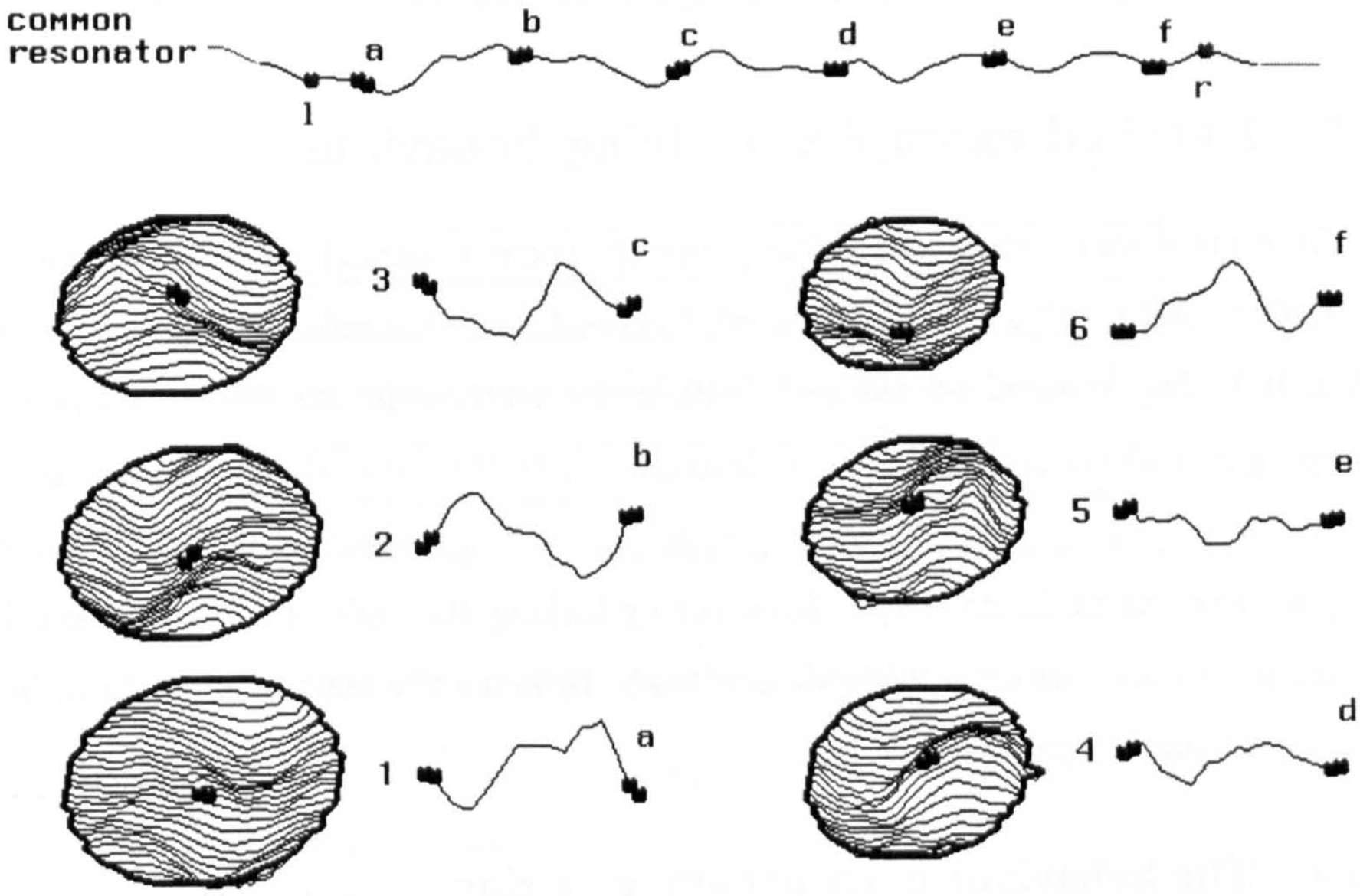


Figure 6.4: Instrument with six tuned circular components and resonators

appropriate excitation models a 'string' can be made to produce sounds perceptually more akin to acoustic tubes.

In this example the circular components are very definitely the *primary* components, and even though the other components contribute to the overall timbral quality of the instrument, it is these primary components which give the sound its most recognisable features, the six *notes* with *inharmonic spectra.* The one-dimensional resonator at the top of the instrument serves to couple together all the other components and by taking stereo sound output from the points marked **l** and **r**, the resulting sounds possess convincing spatial cues with the sounds of each circular component appearing to originate from a different spatial location, especially when listened to over headphones.

Instruments similar to this one form the basis for several of the sound examples listed in section C.10. The decay times of the various components are altered from example to example, achieving a wide range of sounds from bright metallic instruments, through more highly damped almost cow-bell type sounds to almost wooden sounds. The use of circular components gives the instrument more clearly defined pitches

than would be produced with say rectangular components.

6.5 Detailed examples of string behaviour

In this section we examine more closely how the acoustic behaviour of strings may be controlled. Although a one-dimensional TAO instrument is referred to as a 'string', when it is first created all the cells within the instrument are free to move and damping is uniform across the whole instrument. In this state the instrument is not really like a string at all. It has no tension and if left un-excited will simply hang in space and maintain its shape. However by locking the ends of the string and by damping various regions of cells we can begin to make the instrument behave in a suitably string-like manner.

6.5.1 The behaviour of an undamped string

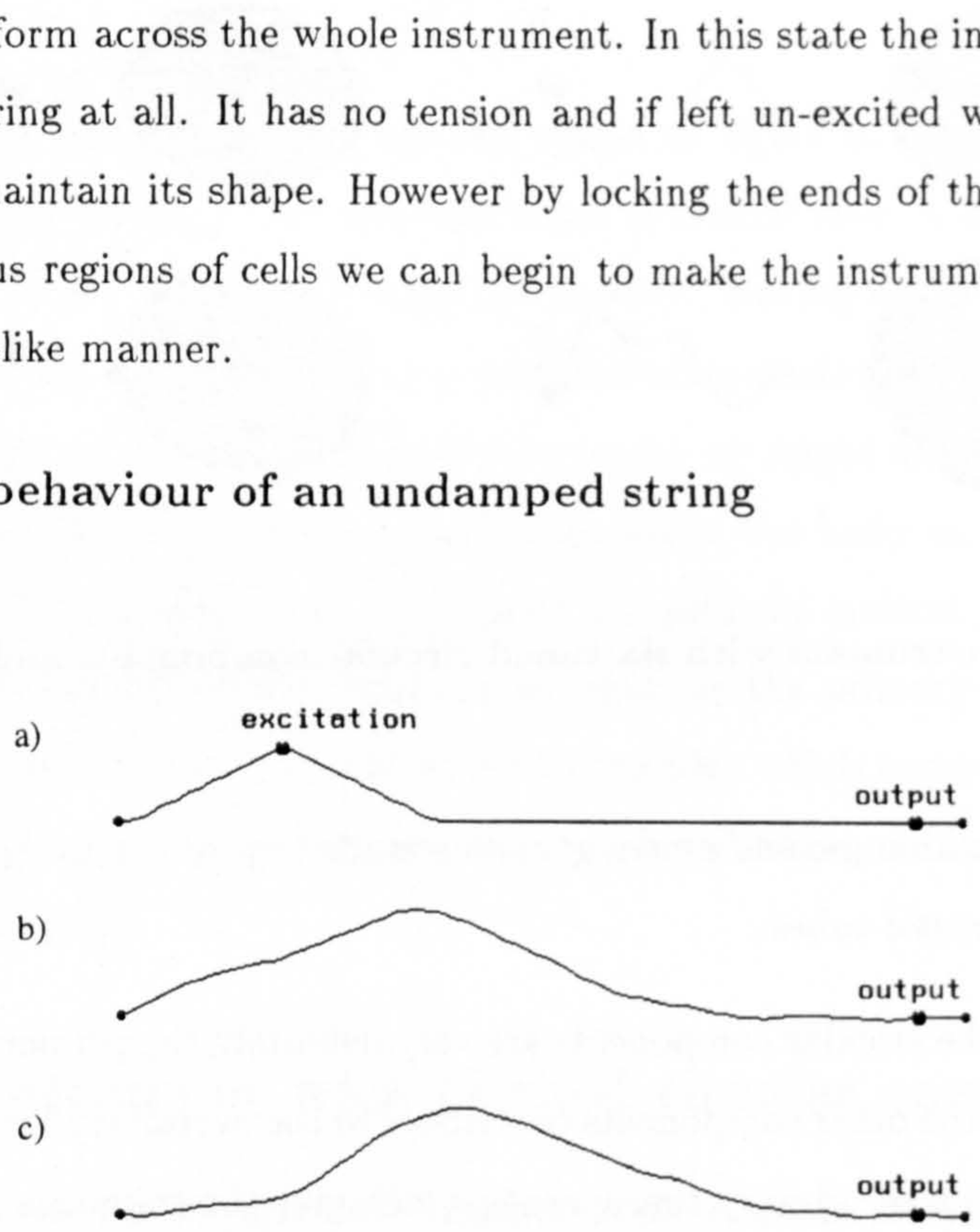


Figure 6.5: Behaviour of an undamped string with locked ends

Figure 6.5 shows a single string with locked ends and with a uniform damping coefficient of 0%. In (a) a constant force is applied to the string for an interval of half a millisecond. In (b) and (c) the state of the string is shown after fifty cycles and one hundred cycles of vibration, respectively. Figure 6.6 shows the spectral evolution of the signal taken from the position marked **output**. The spectrum is harmonic and does not change throughout the duration of the sound since there is nothing to dissipate energy from the system.

This scenario was created with the following script:

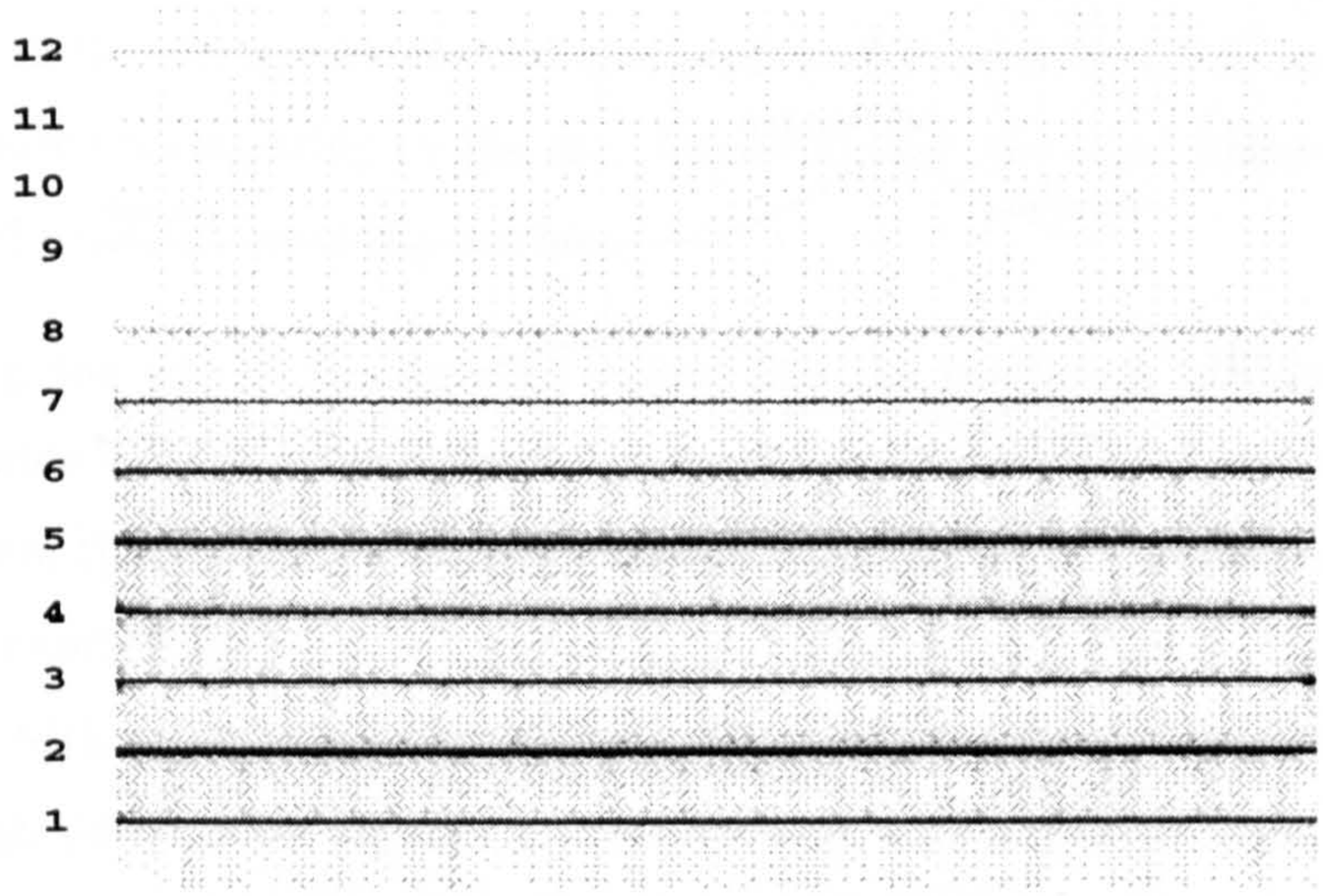


Figure 6.6: Spectral evolution of undamped string

```
String s:
    220 Hz, 0 secs;
    lockends;
    setdamping(left, right, 0%);
    ...

Microphone mic1: undamped_string, mono;

Score 10 secs:
    At start for 0.5 msecs:
        s(0.2).applyforce(10.0);
        ...
    mic1.output:  s(0.95);
    ...
```

6.5.2 The effects of damping the ends of the string

In figure 6.7 the same string now has some damping applied at one end. In (a) the string is excited as in the previous example but now in (b) after ten cycles the higher frequency ripples have been smoothed out, and in (c) and (d), once again after fifty cycles and one hundred cycles, this trend continues. The damping at the end of the string affects the evolution of the spectrum as in figure 6.8. In time domain terms, every time a sharply defined pulse travels through a damped region it becomes a little more smoothed and spread out. Eventually the pulse becomes so spread out that its energy is contained within only the lowest partials.

This example was realised with the addition of one line to the orchestra, which has

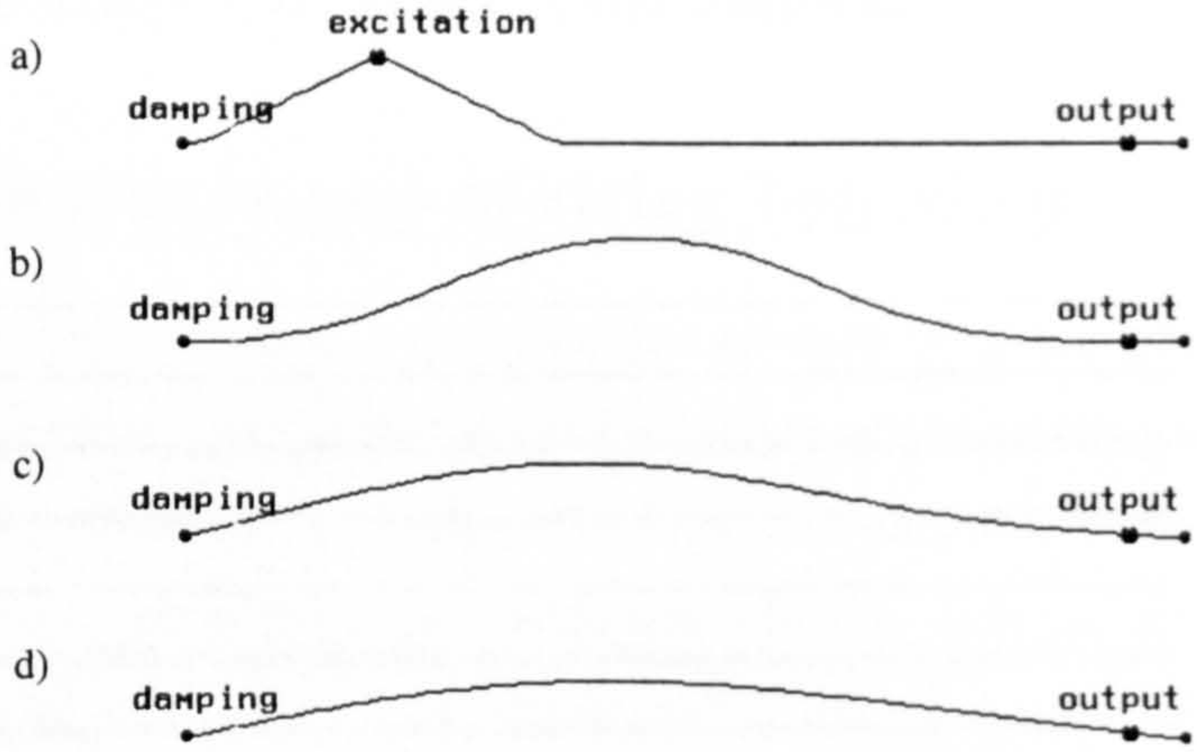


Figure 6.7: Damping one end of the string

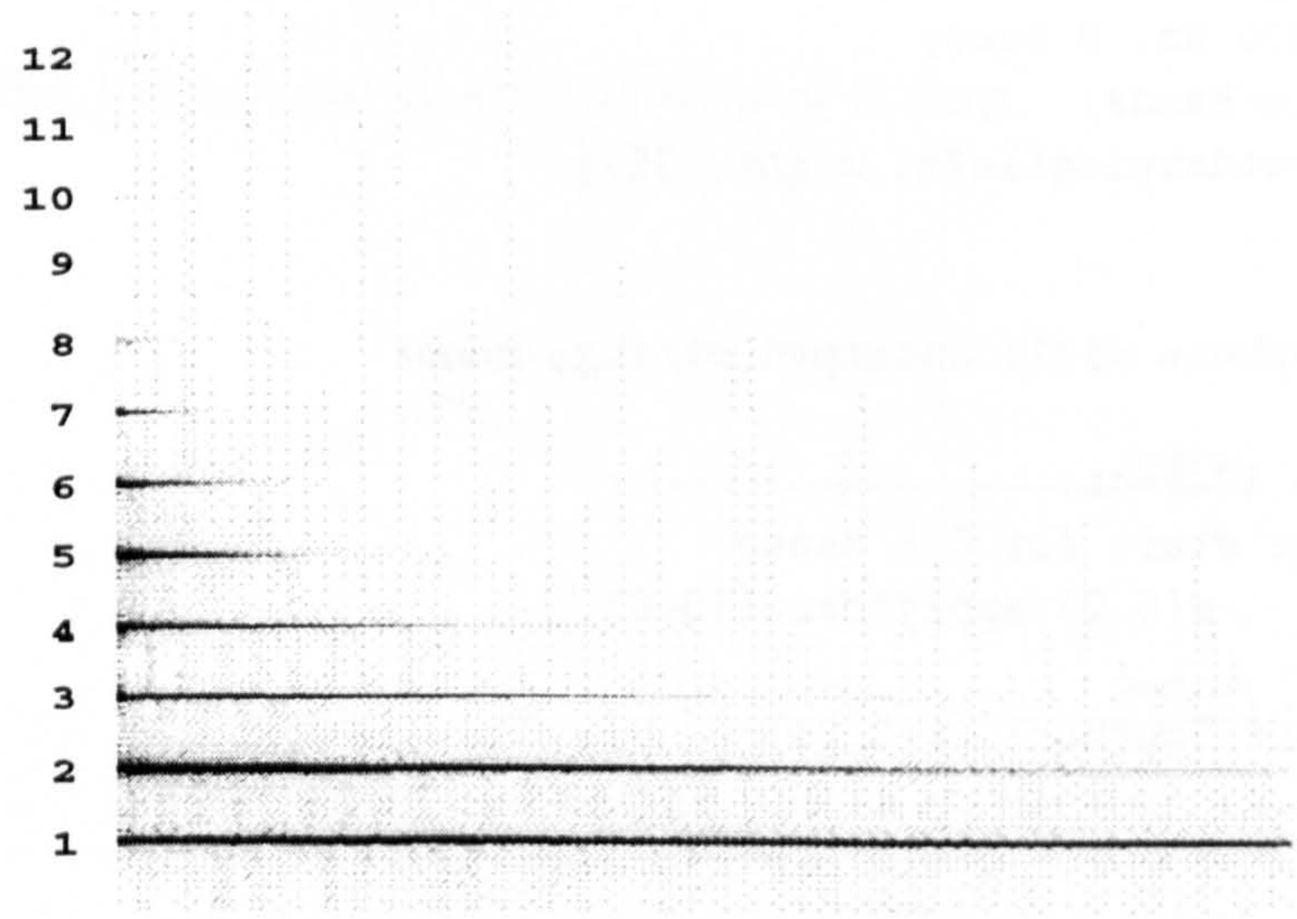


Figure 6.8: Spectral evolution of string with damping at one end

the effect of setting the damping coefficient to 2% over a region extending from the left hand side of the string to a point one twentieth of the way along its length:

```
s.setdamping(left, 1/20, 2%);
```

Damping one end or both ends of a string immediately leads to sounds which have much more realistic string-like qualities, although a single string still sounds relatively synthetic on its own. The ability to control a string's characteristics in this way is an integral part of accurately simulating bowed string sounds (see section 6.7.1) since the smoothing effect of the damped regions helps to dissipate some of the energy imparted to the string by the bow. Too little damping can lead to chaotic

behaviour in the string whereas too much can lead to sounds which are too periodic and therefore uninteresting to the ear. Achieving just the right balance is essential to the fluidity of the resulting sounds.

By varying the size of the damped region and the coefficient of damping we can obtain a wide variety of characteristics. For example, if we wish to obtain a string sound where the highest partials die away very rapidly but the low to mid partials ring on for some time, this is achieved by damping a small region at the end of the string but with a high coefficient. Conversely in order to obtain a sound in which the mid to high partials are affected but die away at a more gentle pace, we can damp a larger region at the end of the string but with a lower coefficient. If we take this process to its logical limit, damping the whole string, then all partials are equally affected and only the overall amplitude decays.

6.5.3 Obtaining harmonics by damping other points on the string

It is possible to damp other points on the string in order to obtain harmonics. Figure 6.9 shows the same string with the end-damping removed and with the midpoint damped instead.

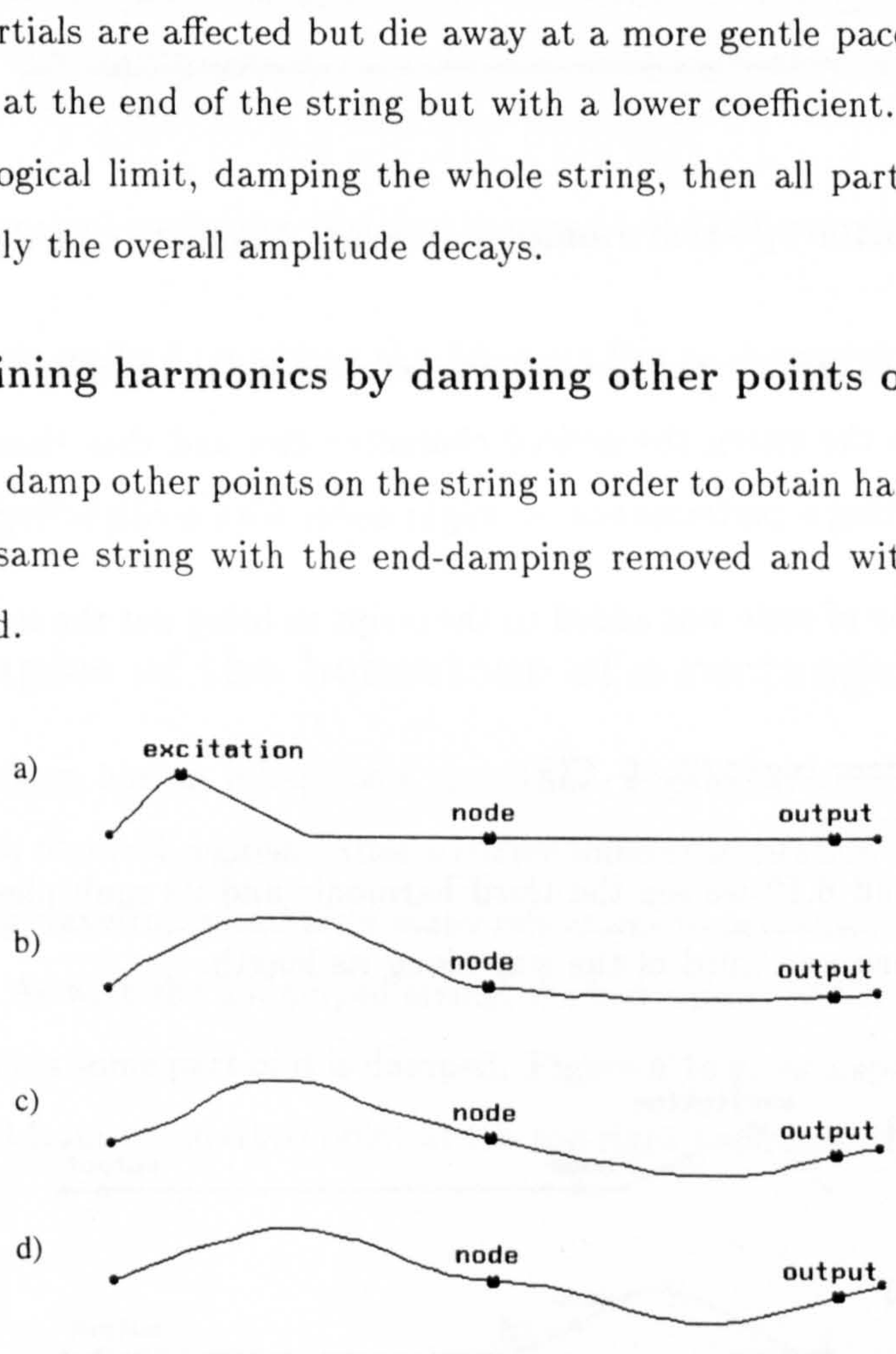


Figure 6.9: Damping the string at its midpoint

The damping forces a node at that point only allowing the second harmonic and its multiples to continue vibrating. The spectrogram in figure 6.10 illustrates this clearly. Note that without damping at the end of the string, the modes of vibration which are unaffected by the newly created node will continue to vibrate ad infinitum.

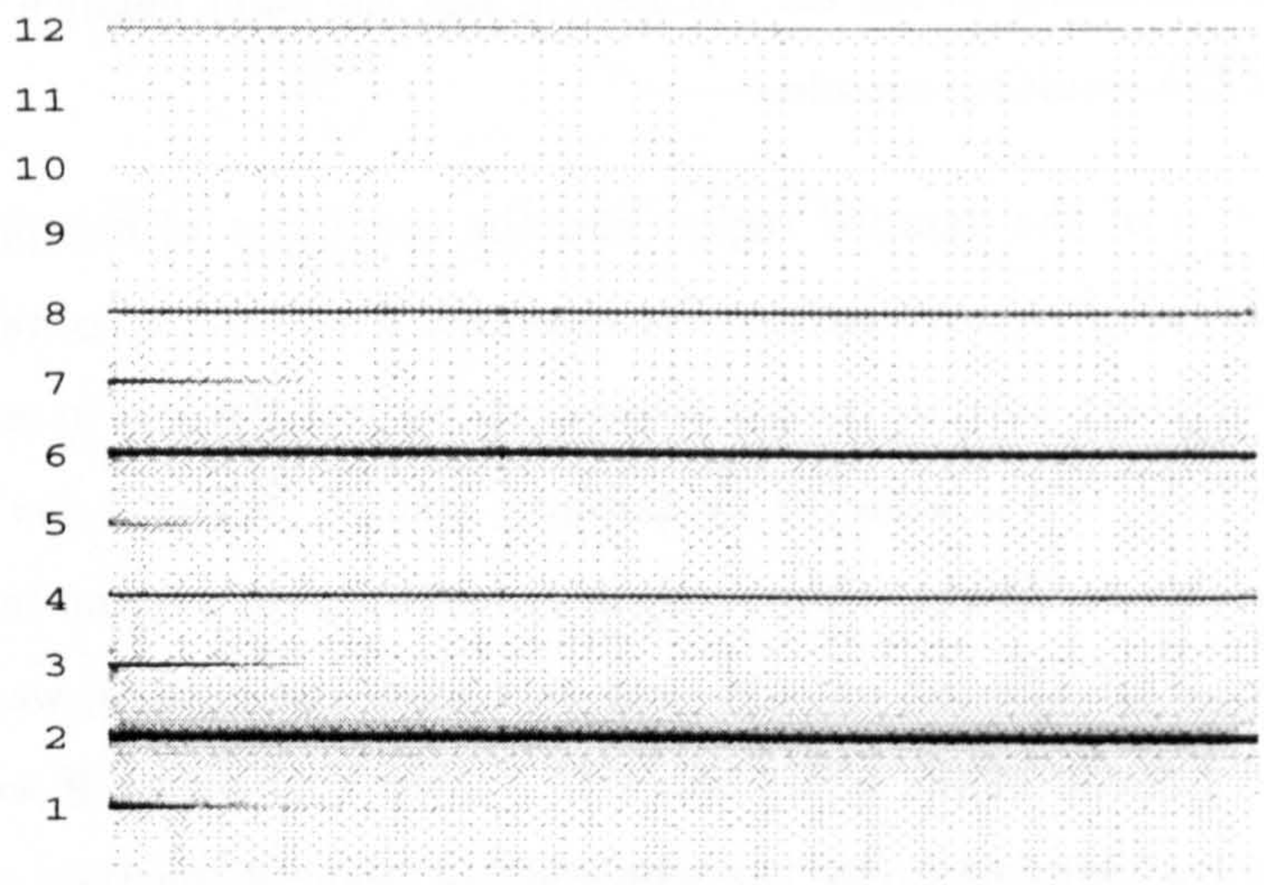


Figure 6.10: Spectral evolution of string damped at its midpoint

If we want a more realistic string-like response we can use a combination of the end-damping to give the string the desired characteristics and then damp other points on the string during a performance, as might occur with a real stringed instrument. The following line of code was added to the script to bring out the second harmonic:

```
s.setdamping(1/2, 0.5%);
```

In figures 6.11 and 6.12 we see the third harmonic and its multiples appearing by damping the string one third of the way along its length.

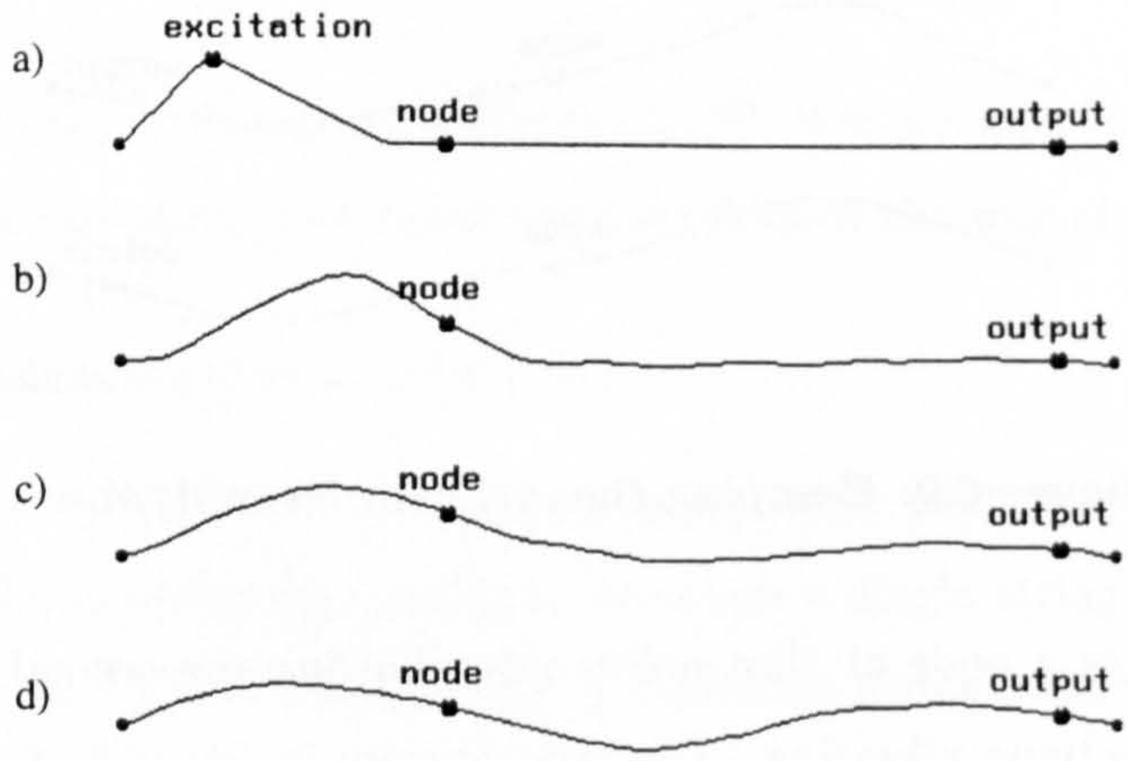


Figure 6.11: Damping the string one third of the way along its length

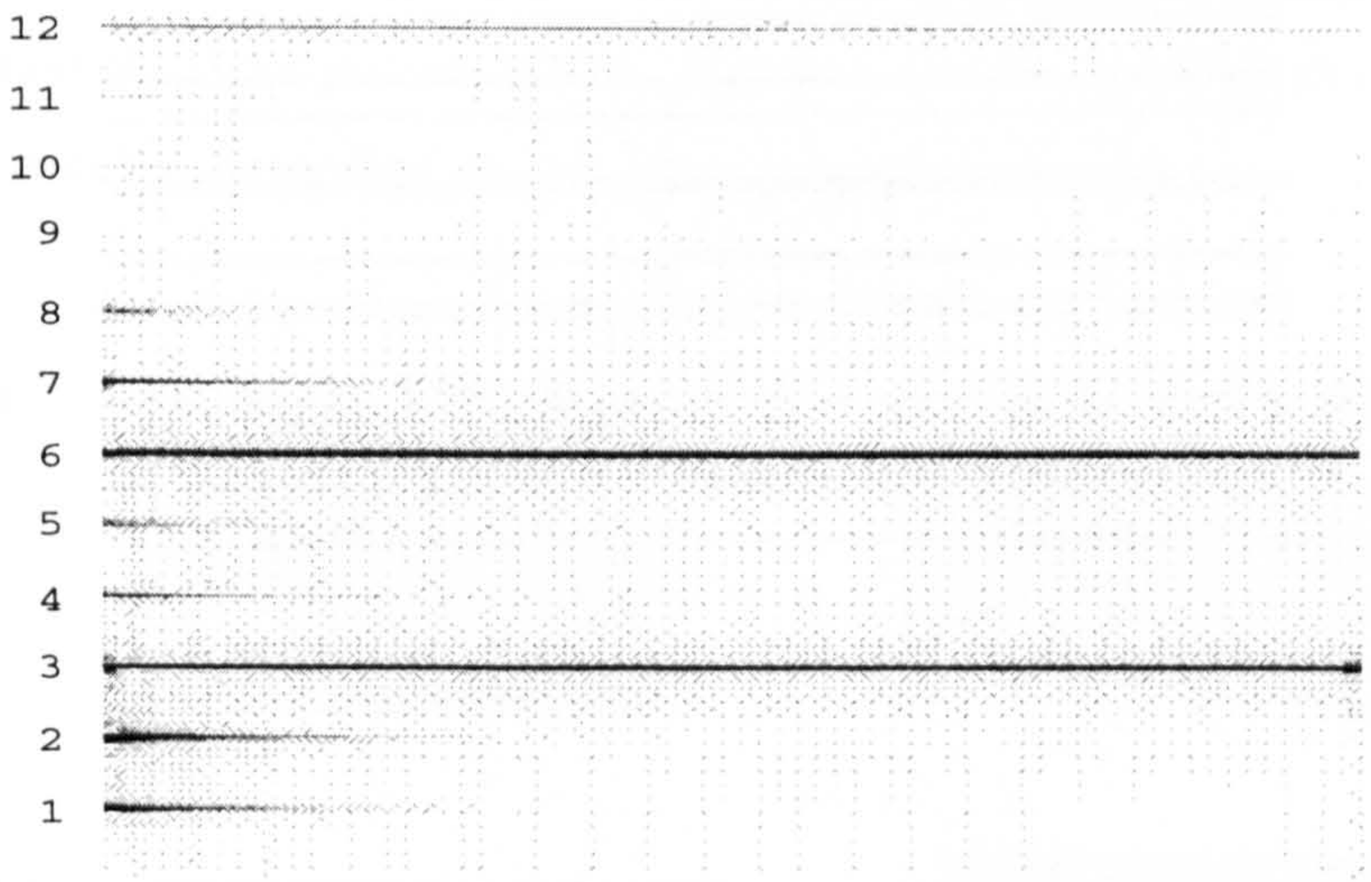


Figure 6.12: Spectral evolution of string damped 1/3 of the way along its length

This example was realised by adding the following line to the script:

```
s.setdamping(1/3, 0.5%);
```

6.6 Examples of the behaviour of a rectangular sheet

A rectangular sheet has an inharmonic spectrum. Figure 6.13 shows such a sheet with no locked or damped regions. After a simple impulse excitation has been applied (a), the resulting wavefronts lead after many reflections to patterns of vibrations such as those in (b). As with the undamped string, the instrument will continue vibrating ad infinitum unless some part of it is damped. Figure 6.14 gives a spectrogram of the output obtained from the marked point at the top right hand side of the instrument.

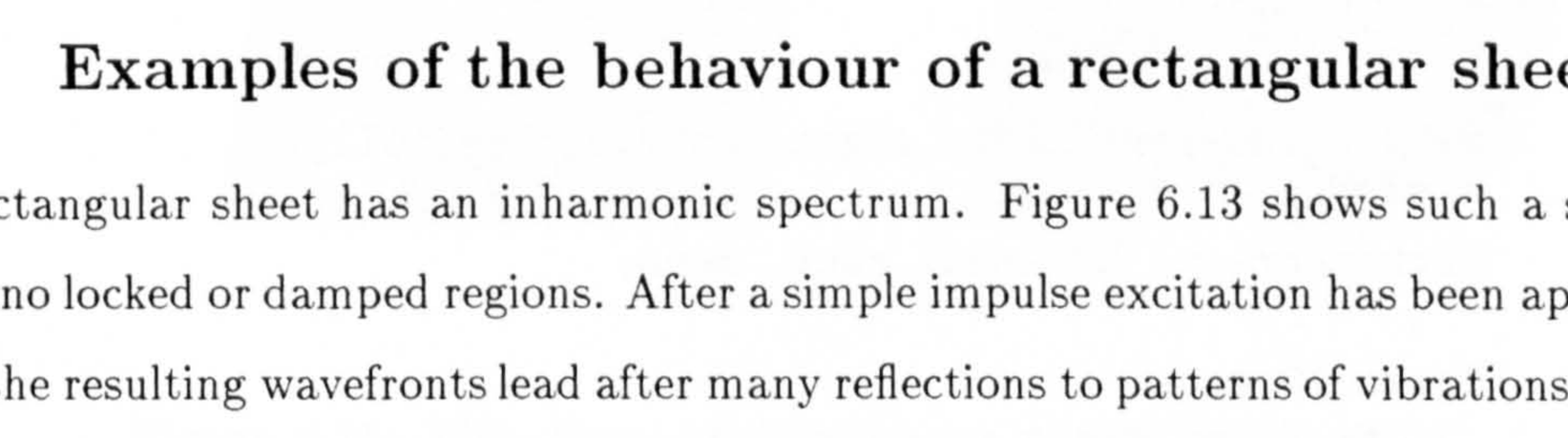


Figure 6.13: An undamped rectangular sheet

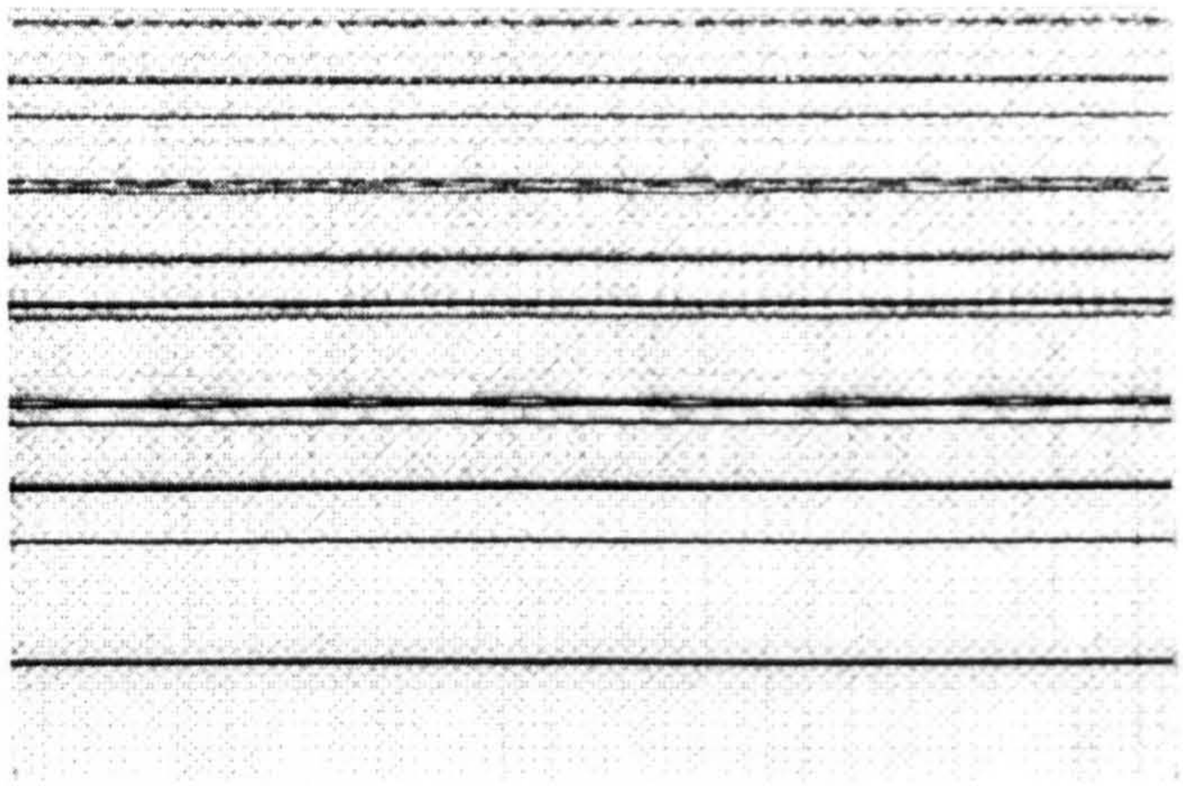


Figure 6.14: Spectrogram of undamped rectangular instrument

This scenario was created with the following script:

```
Rectangle rect:
    470 Hz, 600 Hz, 0 secs;
    lockcorners;
    ...

rect.setdamping(left, right, bottom, top, 0%);

Microphone m: undamped_rect, mono;

Score 10 secs:
    At start for 0.1 msecs:
        rect(0.1, 0.1).applyforce(10.0);
        ...

    m.output: rect(0.95, 0.95);
    ...
```

6.6.1 The effects of damping the rectangular sheet

Damping local regions elsewhere on the rectangle leads to the creation of nodes, which cause some partials to die away whilst others are left to continue. The modes of vibration are more complex than a string's but the same principles apply. Figure 6.15 contains three pairs of images showing the effects of various damped regions on the modes of vibration of the rectangular sheet. The damped regions are shown in light grey and in each case the final pattern of vibration leaves the damped region

almost standing still. The amplitude of the waves is exaggerated somewhat for clarity. The elapsed time interval between the first and second images of each pair is not important since the process can be made to occur over a few tenths of a second or several minutes, depending on the damping coefficient used.

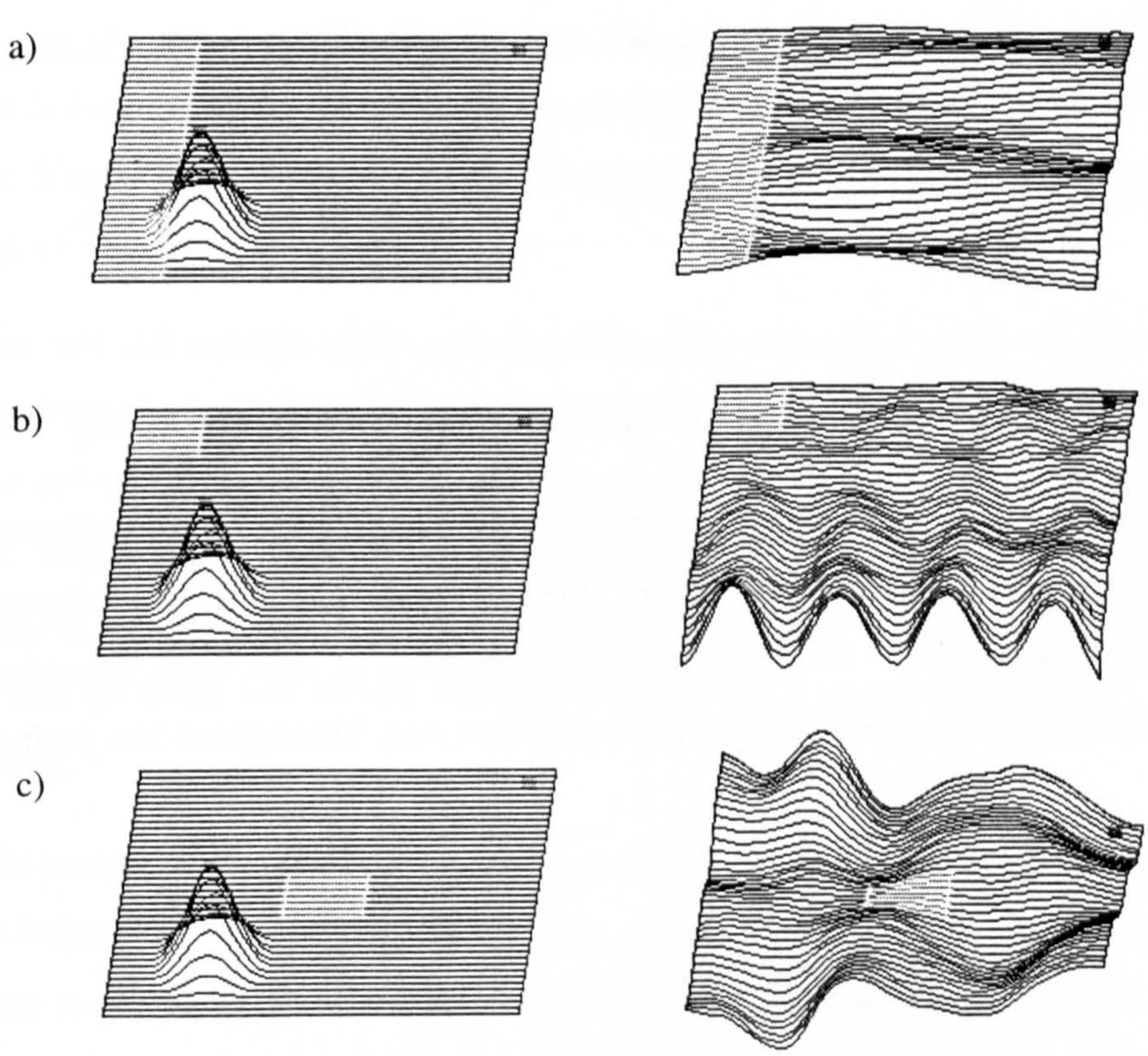


Figure 6.15: The effects of damping on a rectangular sheet

6.7 Examples of the use of other excitations

So far we have concentrated on the structural and vibrational characteristics of many different TAO instruments, but until now the only excitation used in the examples has been a very simple impulse consisting of a fixed force applied over a finite duration. This section explores some more interesting excitation models.

6.7.1 A bowed string

TAO provides a mathematical model for simulating the interaction of a virtual bow with an instrument. The model is described in detail in appendix F but we concen-

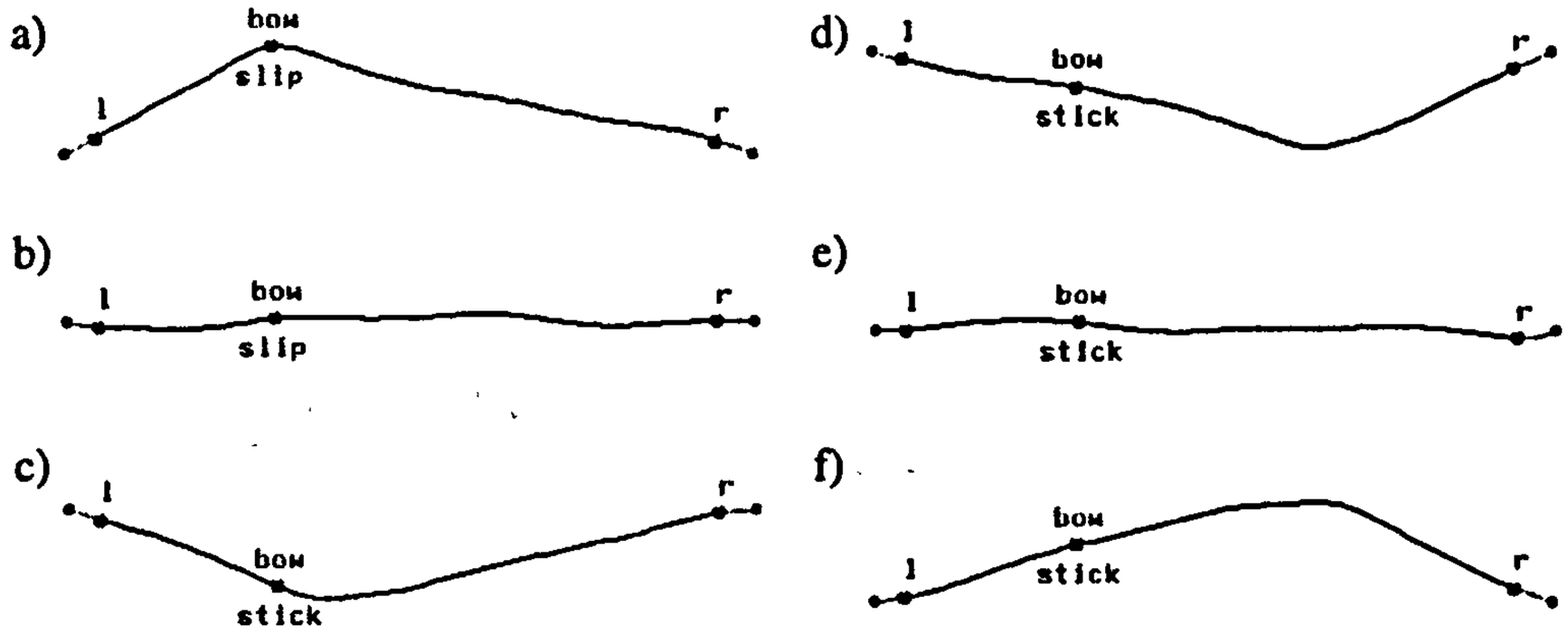


Figure 6.16: Helmholtz motion in a TAO bowed string

trate here on some examples of its application. The following script creates a single string, adjusts the damping at either end to give it roughly similar characteristics to that we would expect from a stringed instrument. It then bows the string with a virtual bow whose velocity and downward force vary throughout the performance:

```
String s: C8, 5 secs; ...

s.lockends;
s.setdamping(left,1/20,0.7%);
s.setdamping(19/20,right,0.7%);

Microphone m: bowedstring, stereo;

Parameter bowforce, bowvelocity;
Parameter vibratodepth;


Score 9 secs:
    From 0 secs to 2 secs:  vibratodepth=linear(0, 1/100); ...
    After 2 secs:  vibratodepth=linear(1/100, 0); ...

    s.vibrato(5.5 Hz, vibratodepth);

    At 0 secs for 0.2 secs: bowforce=expon(2.0, 1.0); ...
    At 0 secs for 0.1 secs: bowvelocity=expon(0.01, 1.0); ...
    From 0.1 secs to 4 secs: bowvelocity=linear(1.0, 4.0); ...
    From 4 secs to 8 secs: bowvelocity=linear(4.0, 0.5); ...

    At 0 secs for 8 secs: s1(0.3).bow(bowforce, bowvelocity); ...

    m.leftout: s1(0.05);
    m.rightout: s1(0.95);
    ...
```

The bow's velocity starts off almost at 0 and rises to a value of 1 after one tenth of a second. Then it gradually increases up to a value of 4, after which it decreases again to a value of 0.5. Meanwhile the downward force exerted by the bow starts off

at a value of 2 and decreases to 1 over the first two tenths of a second, after which it remains constant. With this score, the string settles down into a steady Helmholtz motion (see appendix F) after a few tenths of a second and this motion is shown in figure 6.16.

The concept of a phase space portrait was introduced in section 2.6 and, since the behaviour of the string continually changes throughout the duration of the performance, it is interesting to see what the transients at the start and end of the sound look like. Figure 6.17 shows two phase space portraits of the string's behaviour as produced by the above score.

In (a) the first 0.2 seconds of the sound produced by the above script are depicted. Working along the top row of images from left to right and then along the next row down, we see the string: being dragged away from its rest position at the centre of the cube; beginning to make small slips as the frictional force required to maintain the dragging action becomes too great; falling into a more established pattern of sticking and slipping; and finally settling down into quite a clearly defined pattern of vibration.

In (b) the images depict the behaviour of the string from 7.8 seconds to 8.2 seconds, i.e. just before and after the bowing ceases. At the beginning of this time interval the string is vibrating with a stable pattern of behaviour as illustrated in figure 6.16. As the bowing suddenly ceases at eight seconds, the images show the attractor beginning to collapse leading to the cyclic pattern gradually shrinking down to the original point attractor characterising the natural decay of the string to its rest position.

These images elegantly convey the idea that TAO instruments are *real* instruments, i.e. physical entities with their own time domain behaviour which we interact with via a (simulated) physical dialogue. Another way to convey this is by imagining an instrument as possessing a certain character which will always be perceivable in the sounds it produces, but which we can stretch in different directions by experimentation with different parameters in much the same manner as an instrumentalist may produce different sounds from the same instrument via the use of extended techniques.

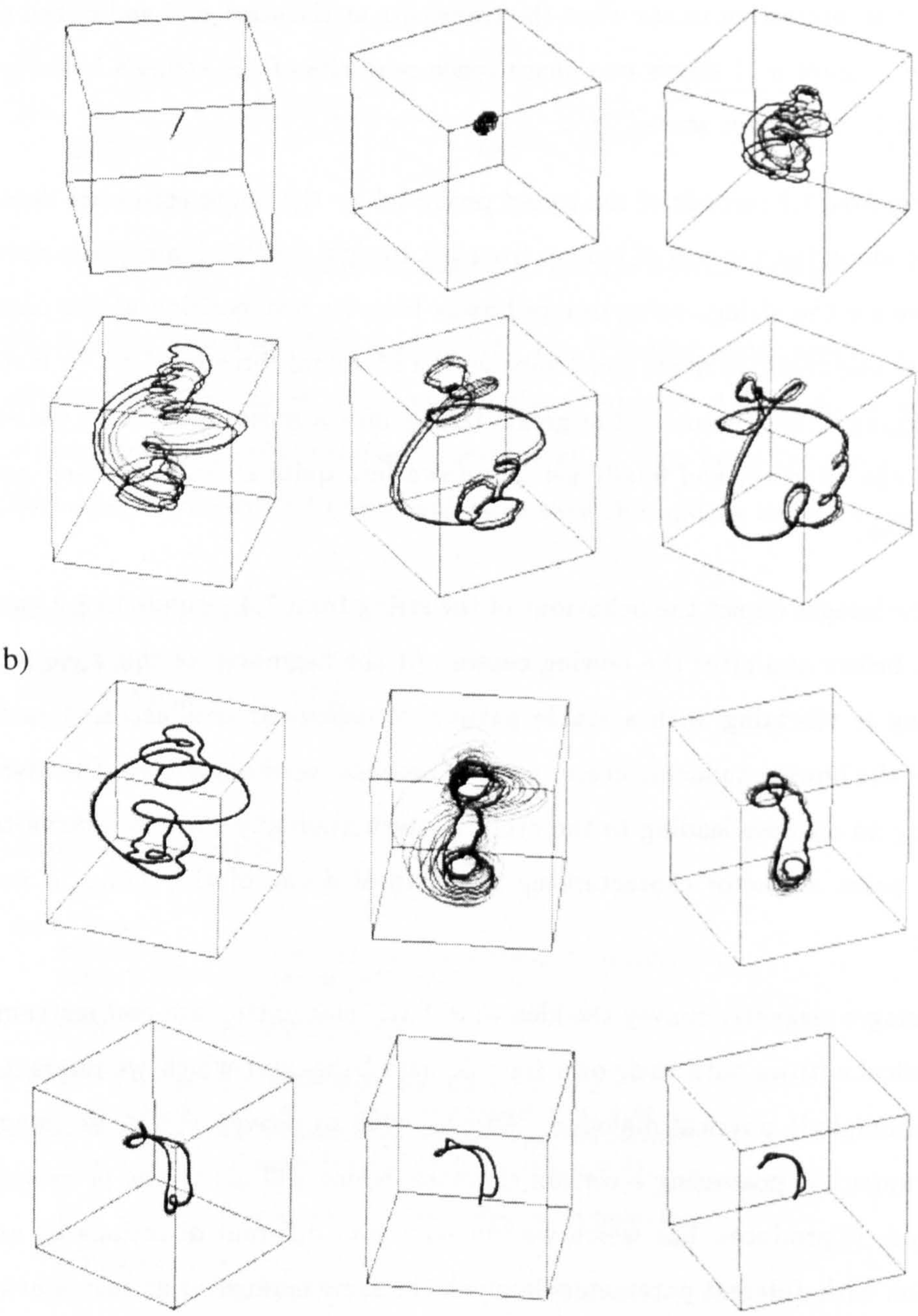


Figure 6.17: Phase space portrait of a bowed string

The sound produced by a similar script is given in section C.8.

6.7.2 A more complex bowed instrument

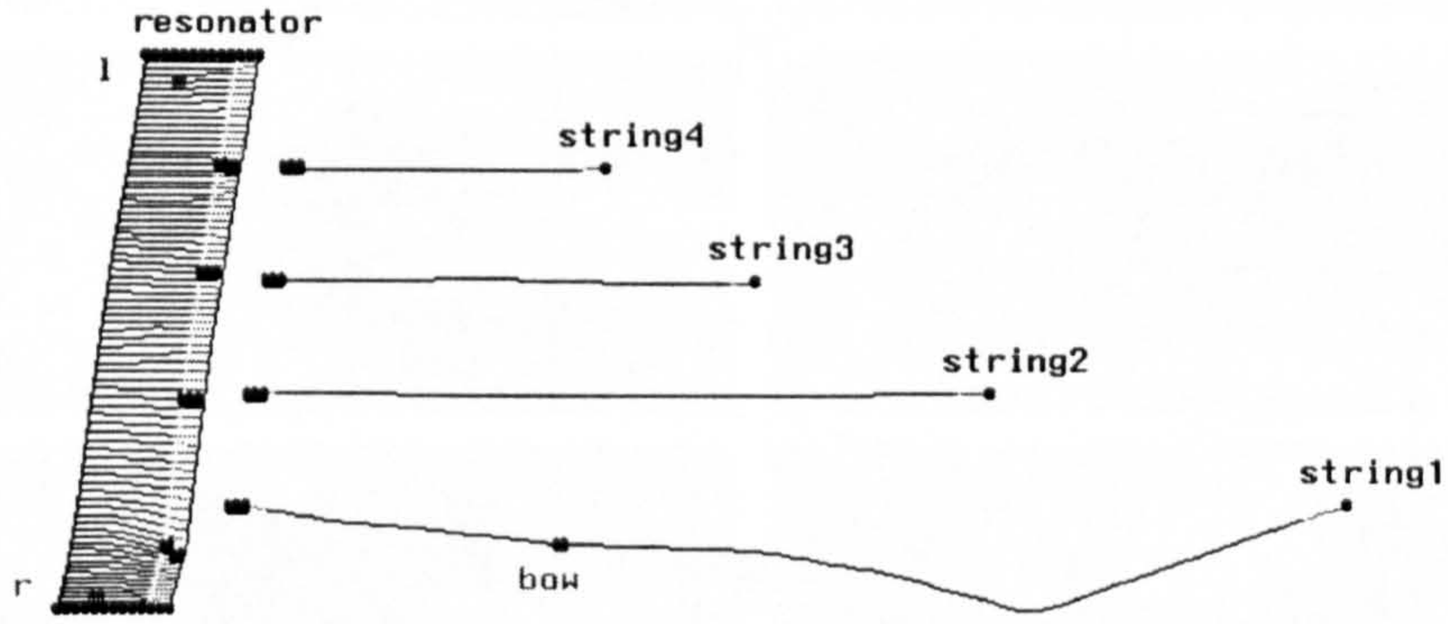


Figure 6.18: A four-stringed instrument with a rectangular resonator

Figure 6.18 shows an instrument with four strings which are glued to a rectangular resonator. The strings are tuned in fifths, hence their respective lengths. Remember that since the cellular material has a constant wave propagation velocity, the frequency of a string is altered by changing its length and not its tension. The component **string1** is bowed at the point marked. Sound output is taken directly from the movement of the points marked **l** and **r**.

An instrument of this family is used for the sound example described in section C.9. The most important feature of this instrument, as with any other TAO instruments comprising several coupled components is that it behaves as a whole entity, i.e. the strings feed energy into the resonator which in turn feeds energy back to the strings. It is often assumed that the body of a stringed instrument does not significantly affect the stable Helmholtz motion of a bowed string and that it is therefore acceptable to model it as a filter which merely colours the sound produced by the string, after the motion has been physically simulated.

This thesis refutes that claim and instead acknowledges that even the slightest changes to a string's motion, due to energy being fed back into it from a resonator, may affect the precise moment at which the slipping and sticking occurs with the bow. Taken over a longer time frame these seemingly tiny physical effects can radically alter the timbral qualities of the resulting sounds.

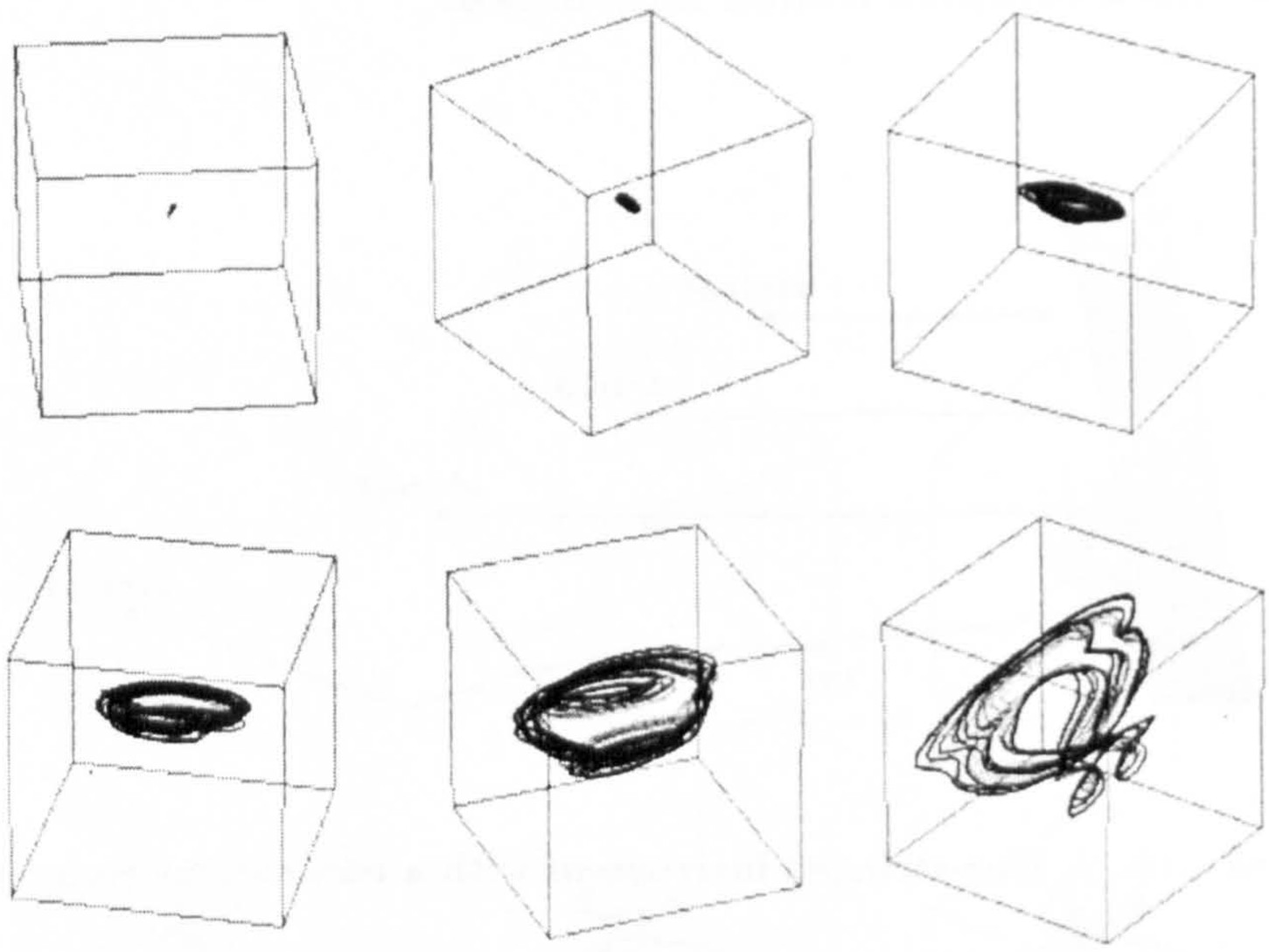


Figure 6.19: Phase space portrait of a bowed string connected to a resonator

In order to demonstrate this we can put the instrument of figure 6.18 to one side for a moment and construct a similar one consisting of a single string, identical to the one used in section 6.7.1 but glued to the same rectangular resonator. Only one end of the string is locked now whilst the other is glued to the resonator. Figure 6.19 shows that the shape of the instrument's attractor as it is bowed (once again depicting the first 0.2 seconds), and hence the character of vibrations in the string, are different from figure 6.17(a) even though the various score parameters are left exactly as the were for the single string example.

The addition of a resonator to a stringed instrument adds interest and depth to the sounds produced especially when microphone output is taken from the resonator instead of from the strings. The organic nature of the sounds produced can be seen from figure 6.20 which shows a portion of the output waveform, taken from the resonator rather than the strings, produced by the sound example given in section C.9. In the top left image we see the top two strings being bowed together and then the middle two. The rest of the images zoom in on the sound at ever smaller scales, and confirm that at every level of structure the sound does indeed evolve organically, even from one cycle of the waveform to the next. The effect of this in perceptual

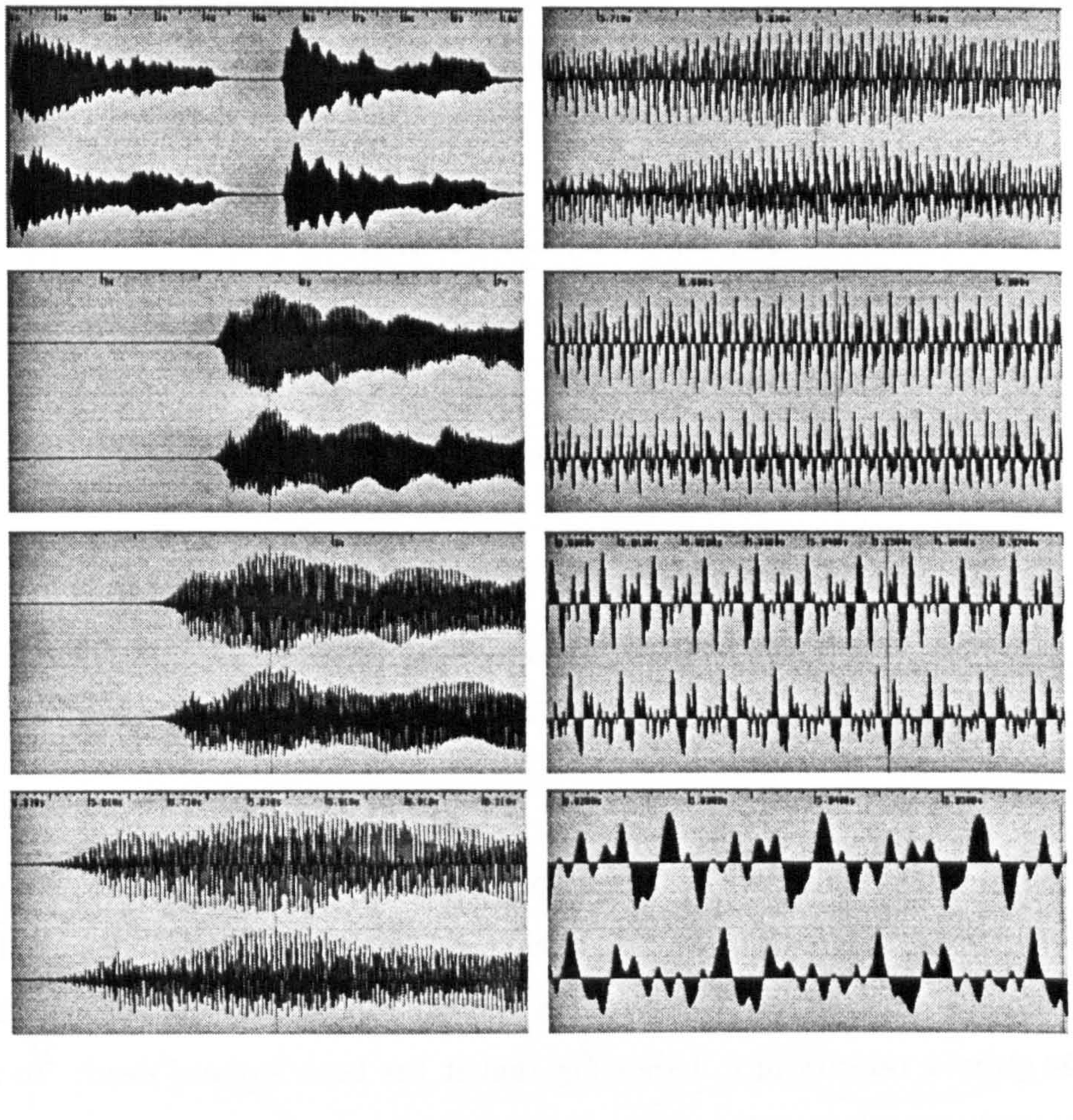


Figure 6.20: Organic evolution of a bowed sound at all levels of structure

terms is to give a strong sense of movement and continual flux in the sound, and also to give it a stronger overall identity.

More generally, whenever we couple several components together, we end up with an instrument which is *greater than the sum of its parts* through the phenomemon of emergent behaviour. The complexity inherent in the vibrational modes of such instruments has a direct effect on the strength of character of the sounds produced. A direct aural comparison between the bowed string sound examples described in sections C.8 and C.9, or between the other sound examples should convince the reader of this point.

6.7.3 Restricting the vibration of an instrument with an obstacle

Although not strictly an excitation, since no energy is injected into the instrument, another technique which produces interesting sounds is to place a virtual obstacle

in the way of an instrument, thus upsetting its natural modes of vibration. The following TAO script shows how this can be achieved:

```
String s: C7, 2 min; ...

s.setdamping(left, 1/40, 0.02%).lockleft;
s.setdamping(39/40, right, 0.02%).lockright;
s(3/4).mass=50.0;
s(1/4).mass=50.0;

Microphone m: test, stereo;

Parameter obstacle_position = 1.0;

Score 20 secs:
    At start for 1 msecs:
        s(0.1).applyforce(1.0);
        ...

    If s(3/10).position > obstacle_position:
        s(3/10).position = obstacle_position;
        s(3/10).velocity = 0;
        ...
    ...
```

This script places an obstacle in the way of a string at a position three tenths of the way along its length and at a vertical position of 1. Whenever the string's amplitude at that point becomes greater than 1 it is immediately limited and the cell at that point is given a velocity of 0, indicating that it has been stopped dead. To make matters even more interesting, two cells in the string are also given higher masses than the rest, leading to inharmonic behaviour even without the obstacle. The sound examples described in sections C.4 and C.5 make use of this technique.

Figure 6.21 shows the instrument in motion. In (a) the instrument is excited with a single impulse. In (b) it comes into contact with the obstacle. In (c) and (d) we can see that the rest of the time the instrument is free to move as it would normally do, and in (e) contact is made again. Every time the instrument hits the obstacle, fresh wavefronts are sent out by the impact. Contact will only be made intermittently and will eventually cease altogether. This can lead to long, evolving, naturalistic sound events.

Part of the beauty of this technique is that a whole family of sounds can be produced by changing the various parameters such as the weight of the masses 'pegged' onto the string, the string's basic characteristics and the obstacle's position, which can even be varied dynamically during a performance. Although cellular models may seem computationally expensive for simple scenarios such as an isolated plucked string,

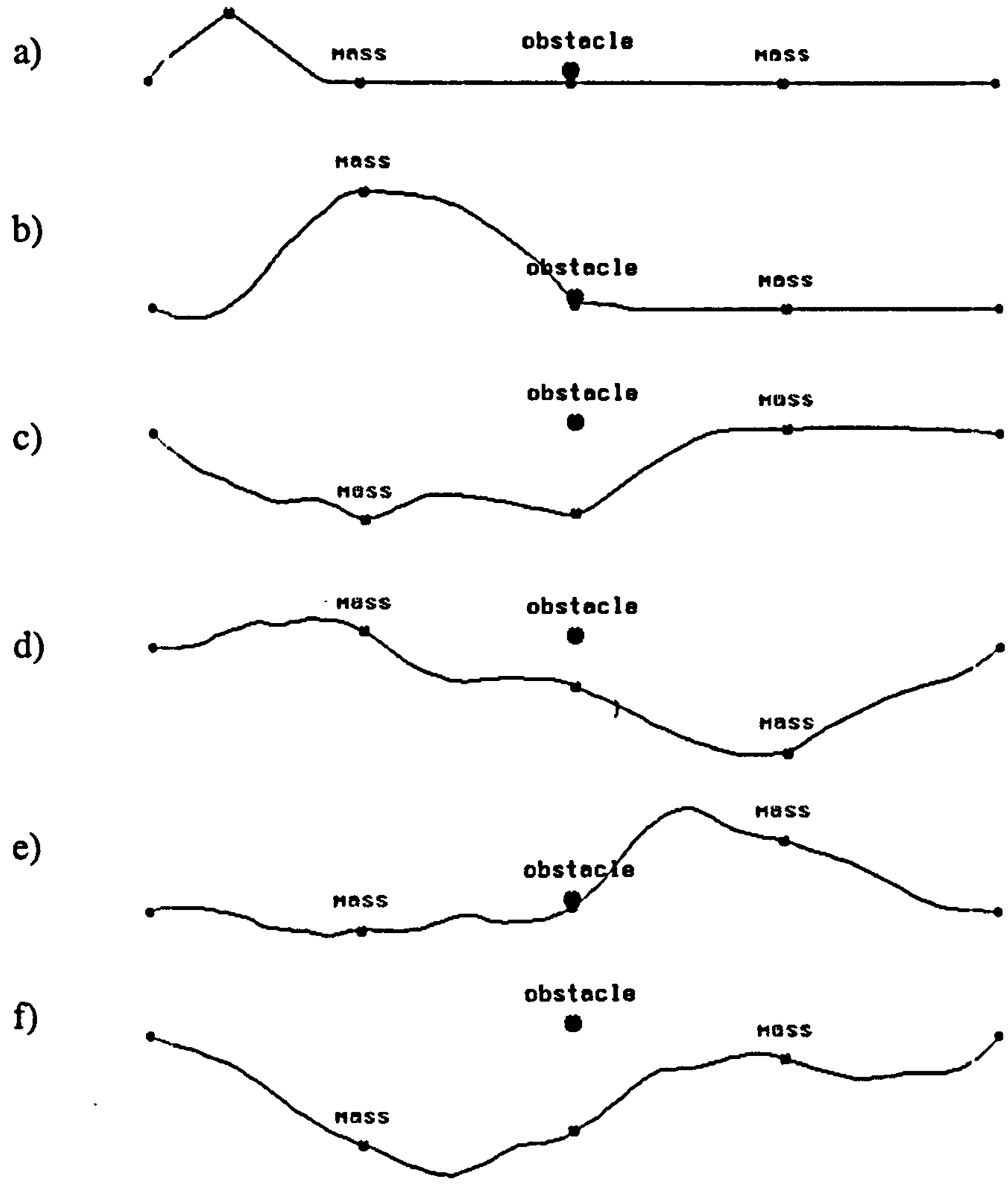


Figure 6.21: A prepared string buzzing against an obstacle

they cope just as easily with much more complex scenarios such as the one described here, at no significant extra computational cost. In addition, such scenarios are often beyond classical closed-form solutions because of their non-linearity.

6.8 A comparison between TAO and other physical modelling systems

A number of comments were made at the end of chapter 3 relating to the various physical modelling techniques which are currently available. The point was made that digital waveguide synthesis, although computationally efficient, is not suitable

for composers having little knowledge of differential equations to begin building new instruments.

TAO takes a fundamentally different approach to physical modelling than taken by digital waveguides synthesis. According to Smith (1992, p.74) the approach taken by TAO falls into the category of "brute force" modelling. However it must be borne in mind that models are often simplified and idealised in order to make them suitable for the computational resources available at the time and, as Toffoli and Margolis (1987) pointed out, the emphasis in such models "does not always reflect a preference of nature" (see section 2.7). In short, it is often impossible to simplify a model without losing some aspect of the behaviour of the real system.

Turning to CORDIS-ANIMA, the main problem associated with the technique was described in section 3.11 as being one of choosing appropriate topologies for the cells and link elements. This problem does not occur with TAO since the cells and springs are arranged in a fixed topology. The images of section 4.2 depicting refraction and diffraction in TAO's cellular material, and the various examples which have been given in this chapter, show that this limitation placed upon topology does not limit the creative potential of the system in any way, and if anything makes TAO easier to get started with.

One of the major problems associated with MOSAIC was stated as being the difficulty of finding appropriate parameters for certain excitations such as reeds and bows. A more general problem was in debugging a synthesis scenario. TAO addresses both of these problems with its informative graphical animations, which allow the user to see, very directly, the effects of certain excitation parameters, as the images from this chapter have shown. TAO's script language takes a very different approach from MOSAIC's, although both are based upon an object oriented view of instruments. MOSAIC provides better abstraction capabilities for building both instruments and excitation scenarios from re-usable modules which have been developed previously. TAO could be developed in order to include a facility of this kind. The addition of a full graphical user interface would also improve TAO's overall user-friendliness.

The major disadvantage with TAO as it stands is in the amount of computational power it requires for complex synthesis scenarios. However, since digital technology

continually increases in power, year by year, with improved speed and memory capabilities, and since the nature of TAO's model makes it suitable for implementation on parallel processors, this problem will ultimately be solved given enough time.

Chapter 7

Summary and Conclusions

7.1 Summary of the key ideas introduced

Electroacoustic music and the perception of sound

In chapter 1 the spectro-morphological and acousmatic approaches to music were described. In both of these musical genres, all sound categories, including those which have been traditionally regarded as 'noises' rather than 'musical' sounds, potentially have equal musical value. Such 'noise' sounds included environmental sounds produced by humans and animals and by physical events or processes. The central importance of aural perception and judgement for both genres of music was highlighted, since traditional theories of music are inadequate when applied to the combination of such a diversity of potential sound sources. The comment was made that, at its most general level, the process of musical composition is one of *organising sound*.

The ecological view of auditory perception was introduced, in which the perceptual attributes associated with a sound are seen as resulting from a combination of the *structured information* contained within the sound, and the listener's perceptual system *resonating with* or *attuning to* the invariant features in that information. This was followed by a discussion of the perceptual attributes of sounds which are considered significant to their use in a musical context, including their *spatial*, *physical* and *energetic* qualities and their ability to suggest a sense of *motion*, *gesture* and *texture*, as well as *affording* certain meanings for individual listeners. We also learnt

that the sounds and sound shapes found in electroacoustic music are often *mimetic* of natural sounds and events. Two types of *mimesis* were described: *timbral* and *syntactic*, although the comment was made that in reality mimemis may occur at any scale from the micro- to the macrostructural.

The complexity of natural systems

In chapter 2 the notion of structured information was examined from the point of view of the natural laws which govern its creation. We learnt about the phenomenon of *bifurcation* or *period-doubling* which occurs universally in *dynamical systems* containing an element of feedback. We also learnt that such systems often exhibit *lively* or *vibrant* behaviour when operating in a regime poised 'at the edge of chaos'. Such behaviour is due to the continual creation of new information. Graphical examples of bifurcation were given, courtesy of the *logistic difference equation*: the simplest equation containing feedback.

The notion of *phase space* was introduced as a graphical way of capturing the identity of a system. This led to the notion of an *attractor*: the general tendency of a dynamical system to follow certain patterns of behaviour, which only become clear when the system is observed over a period of time. The *transient behaviour* exhibited by a system was related to its attractor. Three different types of attractor were described: *point attractors*, associated with dissipative systems; *periodic attractors*, associated with periodicly oscillating systems; and *strange attractors*, associated with chaotic systems.

Complex dynamical systems, and the notion of *emergent behaviour* were introduced: highly structured global behaviour arising in a system consisting of many similar agents interacting on a local basis. We learnt that such systems, although following the same universal laws of chaos as simple ones, are able to express those laws in much more complex and interesting ways, giving rise to intricately evolving spatial and temporal patterns. The resulting structured information potentially has great *depth*, due to the sheer amount of information processing which goes into its production, and high *effective complexity* or a balance between variant and invariant features when a system operates at 'the edge of chaos'.

Other phenomena occurring in dynamical systems were described, including: *self-*

organised criticality, in which a complex dynamical system evolves to a point where it is critically poised at the edge of chaos, and even a small event can trigger a catastrophic change of state; and *coupled oscillators*, in which a set of oscillators, coupled together in some manner, lock into each others vibrational patterns giving rise to to characteristic rhythms, examples of which may be found in the gaits of various animals.

Cellular models were introduced, and in particular the main existing categories of cellular model were described: *cellular automata*, *finite difference models*, *finite element models* and *particle models.* The chapter finished with the comment that: taken in combination, the phenomena described point to the fact that Nature has its own characteristic 'rhythms' both spatial and temporal which are often fascinating sources of pattern and form; and concluded that cellular models provide a unique opportunity for exploring these 'rhythms'.

Existing synthesis techniques and computer music programs

In chapter 3 the most commonly used synthesis techniques were reviewed. The chapter introduced Csound, a computer music program based around the concept of *unit generators*: algorithmic modules which simulate the functionality of analogue components such as oscillators, filters etc.

We learnt that the majority of traditional synthesis techniques is based upon a reductionist, frequency domain approach to sound, making use of combinations of unit generators, in order to create interesting sound textures. Instruments constructed from such components often require macrostructural form to be imposed manually via the use of envelope generators, function tables and numerical performance data specified in the score, making the task of creating sounds with convincing gestural and textural attributes more laborious, although not impossible. The microstructural details of sounds generated with these techniques are also often precomposed and stored in wavetables and the sounds produced, whilst having their own characteristic strengths and weaknesses, usually lack the coherence and subtlety associated with natural sounds.

Of the physical modelling systems introduced, CORDIS-ANIMA could be described as the most cellular in nature, although more emphasis seems to placed on the

creation of non-homogeneous structures, and on the individual masses and springs in an instrument, than on the appealing properties which emerge naturally from a model consisting of large numbers of identical elements interacting on a local basis. The technique of 'digital waveguide' synthesis was described, in which a waveguide is modelled as a set of delay lines connected by *scattering junctions*, which contain all the 'interesting' mathematics. An example was given of a synthesis system which makes use of digital waveguides: the vocal synthesis program SPASM. The technique of modal synthesis was also described, in which analysis of vibrating structures for their natural modes of vibration leads to models in which these modes are represented by sets of *modal oscillators.*

Csound, MOSAIC, and SPASM all provide script languages for describing synthesis scenarios whereas CORDIS-ANIMA does not.

The TAO computer music program

In chapters 4, 5 and 6 the TAO computer music program was described. TAO relies entirely upon the emergent properties of a particular cellular model, consisting of masses interconnected with springs, for the production of its characteristically physical and organic sounds. Visual examples were given of the emergent behaviour of the model, including its ability to simulate wave propagation, reflection, refraction and diffraction. Other visual examples given included the ability of the model to produce transient behaviour, and to support the construction of complex vibrating structures by way of coupling several pieces of the cellular material together.

TAO Instruments are played by exciting and damping individual cells or groups of cells and because of the holistic and attractor-driven nature of the cellular model, regardless of how the instruments are excited and damped, the sounds always retain a certain coherence and identity, being bound together by physical causality. The script language described in chapter 5, although not relating directly to the hypothesis put forward in this thesis, nevertheless gave an example of how a cellular model may be controlled in non-real-time. Unlike the precomposed numerical scores of Csound, TAO's score takes the form of an algorithmic language, with the provision of specific features aiding the description of hierarchically nested time domain events.

7.2 Conclusions

This thesis began with a set of criticisms levelled at digitally synthesised sounds and a hypothesis which proposed that:

> Cellular computer models, inspired by the behaviour of naturally occurring complex dynamical systems, provide an ideal medium for the development of a new generation of sound synthesis techniques, more holistic in their approach than traditional techniques, and capable of producing complex organic sound events, whilst simultaneously being sympathetic to the needs of electroacoustic music.

7.2.1 Sound synthesis as the creation of structured information

This hypothesis has been supported by an in depth examination of the notion of *structured information*. Many examples were given in chapter 1 of specific perceptual mechanisms and musical attributes of sound which arise as a direct result of particular patterns of information, usually generated by physical processes or mechanisms. Terms such as *complexity*, *coherence*, *organicity*, *vibrancy* etc. have been used throughout this thesis, both in relation to the perceptual effects they are capable of evoking, and in terms of the information generating properties of dynamical systems and natural environments.

The process of sound synthesis has been viewed as a process of creating coherently structured auditory information. This view is radically different from frequency domain approaches which concentrate on the spectral content of a sound and its temporal evolution, without providing a consistent framework for the description of the macrostructure and microstructure of a sound and their mutual relationship. Whilst it would be unfair to dismiss frequency domain techniques out of hand, which is not the purpose of this thesis, they are often inappropriate for describing certain classes of sounds with complex temporal patterns of evolution.

Of the more recent synthesis techniques to appear, granular synthesis seems to be alone in its ability to cope with all manner of structured noise sounds. The main problem with the technique is that it does not actually include control strategies for arranging grains of sound into macrostructures. Since natural events are often

arranged in a holarchic manner, in order to convincingly mimic sounds such as the breaking of waves quite sophisticated control strategies are required. Cellular models have already been applied successfully to simulating this phenomenon in the visual domain (see section 2.7.4), and it is highly likely that the structured information generated by such models could be applied to controlling the microscopic grains of sound, or clusters of grains, leading to sounds possessing qualities convincingly mimetic of breaking waves.

This thesis argues that both the complexity and coherence of the information contained within a sound are extremely important to the imagery it is capable of evoking and the sound examples listed in appendix C provide some aural evidence in support of this argument. The simpler instruments comprising few components often produce fairly 'synthetic' sounds, whilst the more sophisticated instruments suddenly seem to 'spring to life'. In particular the sounds produced by these more sophisticated instruments often seem to originate from tangible physical objects, and even when the sounds are more abstract in nature, they still retain a coherence which makes them feel 'solid'. The sounds possess spatial, physical and energetic cues due to the underlying laws governing the cellular update rule.

It is interesting to note that the percussive sounds described in section C.10, since they are generated by dissipative dynamical systems which are linear in their behaviour, probably do not exhibit truly chaotic behaviour. The phenomenon of chaos requires some element of non-linearity, such as is provided by the bowing model. However, the sounds are often still quite vibrant and complex. This is a good example of the notion of *depth*, i.e. complexity arising from the shear amount of information processing occurring. On the other hand, the bowed sound described in section C.9 does rely upon external energy being continually applied to the instrument in a non-linear fashion, and whilst also exhibiting depth of information, it also exhibits a different kind of complexity due to this element of feedback, giving it subtle inner rhythms.

7.2.2 Why the emphasis on chaos?

A recurring theme throughout this thesis has been the balance between order and chaos observed in Nature. Much evidence has been presented supporting the notion

that dynamical systems operating in a regime poised at the edge of order and chaos seem to act as creative sources of information. The amount of emphasis which has been placed on the significance of chaos theory in this thesis might be criticised on the grounds that it only relates to the physical world and not to something as subjective as human perception. There are two answers to this criticism though.

Firstly, the human brain is itself a complex dynamical system, although the fact that it evolves throughout its lifetime by modifying its own internal configuration places it into the special category of *complex adaptive systems.* Because of this, it is subject to the same laws of chaos as all other complex dynamical systems, a view supported by the following quote:

> *A physicist thinking of ideas as regions with fuzzy boundaries, separate yet overlapping, pulling like magnets and yet letting go, would naturally turn to the image of a phase space with "basins of attraction". Such models seemed to have the right features: points of stability mixed with instability, and regions with changeable boundaries. Their fractal structure offered the kind of infinitely self-referential quality that seems so central to the mind's ability to bloom with ideas, decisions, emotions, and all the other artifacts of consciousness. With or without chaos, serious cognitive scientists can no longer model the mind as a static structure. They recognise a hierarchy of scales, from neuron upward, providing an opportunity for the interplay of microscale and macroscale so characteristic of fluid turbulence and other complex dynamical processes (Gleick, 1991a, p.298).*

More direct evidence of the presence of chaotic behaviour in the human brain may be found in the observation of unusual eye movements in patients suffering from a variety of neurological disorders [1]. Regardless of the various arguments concerning the nature and origin of consciousness, intelligence, and emotion, it is evident from the brain's internal structure that chaos, complexity and emergent behaviour must have some part to play in our perception and awareness of the world around us.

[1] An observer's eyes are usually capable of tracking a moving object such as a swinging pendulum with remarkable smoothness of movement, but occasionally the eyes are seen to oscillate with a variety of periods or even move chaotically. See Gleick (1991a) p.276-277

Secondly, according to the ecological view of perception, we humans and other organisms have evolved perceptual systems which are capable of attuning to or resonating with information already present in our environment. Since this information is generated according to patterns which are always consistent with the laws of chaos, it is reasonable to assume that these characteristic patterns are somehow intimately tied up with our perception of events and objects and to our sense of how *naturalistic* or *organic* they are. It is interesting to note that Gibson's book *The ecological approach to visual perception*, in giving examples of the way in which visual information is structured by various surface textures, reproduces photographic images which are remarkably similar to those found in another book *Nature's Chaos* (Gleick, 1991b) which provides many photographic examples of the occurrence of chaos in Nature.

7.2.3 Designing cellular models for the generation of auditory information

Auditory information is constrained by a different set of criteria than is visual information, and cellular automata models such as those represented by the visual examples given in chapter 2, depicting physical processes such as dendritic growth, annealing, fluid flow and reaction/diffusion, are not directly applicable to the task of sound synthesis as they stand. Auditory information must always contain an elements of oscillation occurring at frequencies lying within the audible spectrum. The model used by TAO shows one way in which this can be achieved, but the use of masses interconnected by springs is not the only strategy. If we wish to develop other cellular models for sound synthesis, we have to decide first upon the level at which they will operate. It would be perfectly possible to use a cellular model such as the flocking model described in section 2.7.4 to generate coherent macrostructures which would then be filled in with microstructures generated by traditional synthesis techniques. Conversely, it is *already* possible with TAO to generate microstructures which are then arranged into arbitrary macrostructures through the use of the score language. However, it has been claimed throughout this thesis that the strongest images will only be evoked in a listener when the microstructure and macrostructure exhibit an overall coherence due to some causal connection between them.

It is possible within TAO to implement this causal connection between microstructure and macrostructure by designing score algorithms whose behaviours depend

in part on the microstructural behaviour of the instruments played by the same algorithms. This process is depicted in figure 7.1. Since period doubling or bifurcation and chaotic behaviour are universal phenomena observed in all systems involving feedback, the sounds produced by such a technique might begin to take on some of the 'natural rhythms' described in section 7.1, and by a process of experimentation with the nature and amount of feedback, it would be possible to create information-rich macrostructures, not arbitrary in nature, but intimately linked to the microstructures produced.

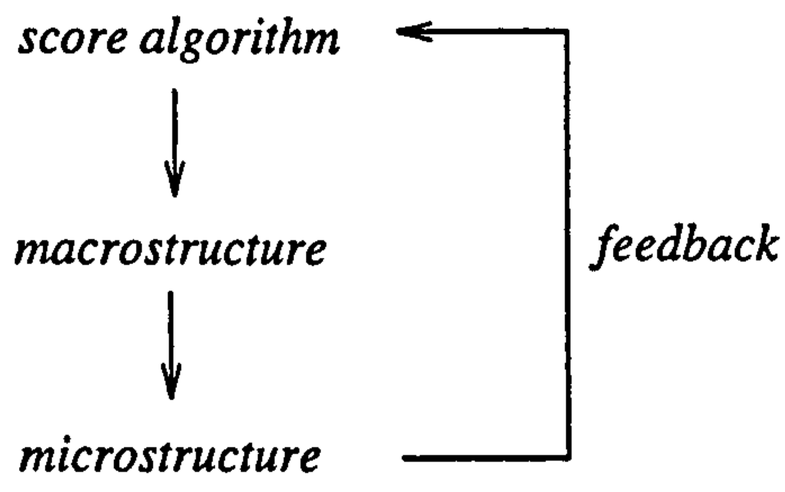


Figure 7.1: Using feedback from the microstructure of a sound event in order to influence the macrostructure.

For example, a script was given in section 5.5.5 for simulating an obstacle bouncing on an instrument. This simulation was based on the approach of using an exponential function in order to calculate the ever decreasing interval between impacts, and the force exerted on the instrument by each impact. This model does not rely, however, upon explicit simulation of the physical mechanism of bouncing. Since TAO provides direct access to parameters such as *force*, *velocity* and *position*, it would be possible to simulate the bouncing interaction properly, the advantage being that if several objects were bounced on the same instrument simultaneously, the system would act as a coherent whole, i.e. each force exerted by one object bouncing on the instrument would affect the forces exerted on all the other objects, and hence the intervals between their successive bounces. Since it has been shown, conclusively, that the human auditory system is capable of inferring the physical cause of a complex sound event from the temporal relationships between the macrostructural elements (see section 1.6.2), this technique would improve the perceived coherence of the sounds produced.

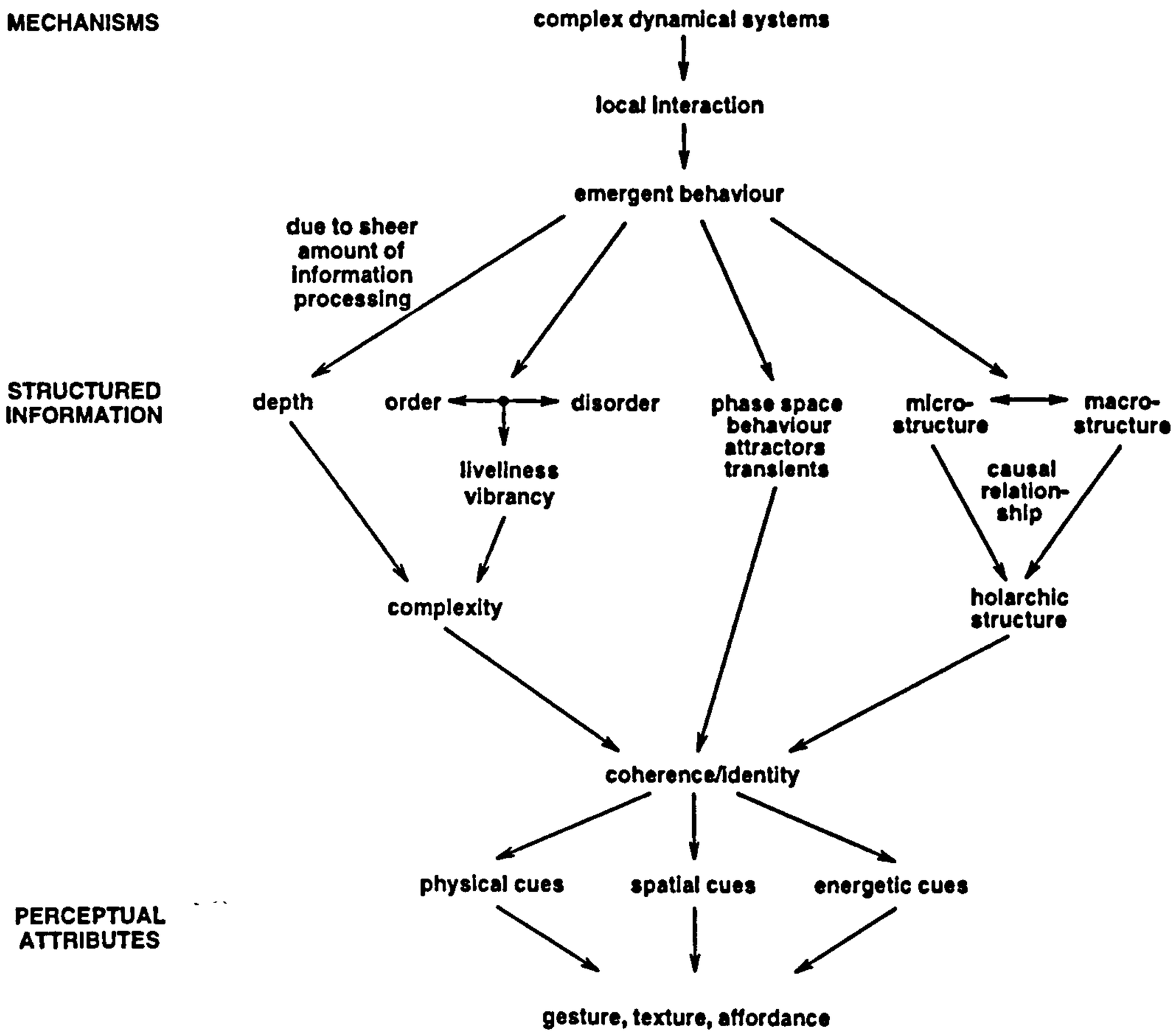


Figure 7.2: The relationship between complex dynamic systems, coherent structured information, and perceptual attributes.

7.2.4 A model of 'organic' sounds

A variety of terms has been used frequently throughout the thesis, including *organic*, *vibrant*, *lively*, *coherent* etc. It has been proposed that these adjectives refer to certain attributes of structured information, which are due both to the mechanisms used to generate the information, and to the perceptual abilities of the listener. The proposed relationship between these adjectives and various other terms introduced is clarified by figure 7.2. The diagram begins at the top with the mechanisms which create structured auditory information, complex dynamical systems; lists the various attributes of this structured information; and finishes at the bottom with the various perceptual effects which the information is capable of evoking in the listener.

The structured information generated by a complex dynamical system has four attributes: firstly, it has *depth*, due to the shear amount of information processing which takes place in a complex dynamical system; secondly, it may exhibit patterns of behaviour lying anywhere along the spectrum between order and disorder; thirdly it is governed by the attractor and associated transient behaviour of the system; and finally, it is holarchical in nature, i.e. the information consists of a nested hierarchy in which there are no clear dividing lines between different scales or between different structures at the same scale.

The first two attributes, taken in combination, describe the overall *complexity* of a system, and complexity relates directly to how information-rich its behaviour is. When taken in combination with the other attributes, the structured information produced may be said to be *coherent* and possess a strong *identity*. Moving back into the sonic domain, this leads to sounds possessing strong *physical*, *energetic* and *spatial* cues, which suggest a sense of *gesture* and *texture*. Coherently structured sounds are also more likely to evoke a sense of *affordance* in a listener.

7.3 Closing comments

According to the ecological view of perception, an organism evolves to pick out features of the environment, both objects and events, which are pertinent to its survival. Ordinarily we take for granted our ability to recognise a scraping sound or a shattering sound since the process of recognition is so subconscious and immediate. If we begin to think about the variety of environmental events which we are able to recognise in this way, the list begins to expand endlessly. Gibson's explanation for this is not that we carry mental models around with us, one for each type of event, but that the events themselves create structured information which we are able to *attune* to or *resonate* with.

An intriguing question to ask then, following on from this observation, is: what happens when we present the auditory perceptual system with patterns of structured information which, although complex and coherent, nevertheless do not conform to any patterns encountered in Nature? If Gibson's view of the process of perception is indeed correct, then such sounds might still be capable of strongly evoking imagery in the listeners mind, although the precise mimetic qualities of this imagery would

be difficult to predict or explain.

The ability of cellular models to create completely artificial but coherently information-rich environments, offers the exciting possibility of exploring synthetic aural landscapes, which nevertheless appear to be completely *organic* in their structural coherence. This process has already begun with the design and exploration of TAO, but there are a multitude of other ways in which cellular models could be applied to the sonic domain, potentially providing the electroacoustic composer with many new approaches to the challenge of *organising sound.*

Appendix A

A brief user manual

A.1 Installation

The file `Tao1.0.tar` contains an archived version of the following directory structure and may be unpacked by typing:

```
tar xvf Tao1.0
```

```
Tao1.0:

   bin/
   lib/
   src/
   translation/
   README

Tao1.0/bin/

   tao                  Command for compiling a TAO script into an executable.

   float2aiff           Executable for translating a raw floating point data
                        file into a .aiff soundfile.

Tao1.0/lib/

   libtao.a             Compiled library of TAO objects classes and associated functions.

Tao1.0/src/
```

`Cell.h`	C++ source code for library `libtao.a`
`Cell.cc`	
`Circle.h`	
`Circle.cc`	
`Ellipse.h`	
`Ellipse.cc`	
`Instrument.h`	
`Instrument.cc`	
`Microphone.h`	
`Microphone.cc`	
`Rectangle.h`	
`Rectangle.cc`	
`String.h`	
`String.cc`	
`Triangle.h`	
`Triangle.cc`	
`main.cc`	`main` and other global functions
`tao_scriptfile`	Intermediate file used in the translation of a TAO script into C++ code.

`Tao1.0/translation/`

`circle_sed_script`	These files all contain Unix `sed` scripts which are used in the general translation of a TAO script into a valid fragment of C++ code. Whilst being commented and fairly bug-free, they were only ever intended as an interim measure and should be replaced with a proper parsing and translation program.
`ellipse_sed_script`	
`rectangle_sed_script`	
`string_sed_script`	
`triangle_sed_script`	
`tao_sed_script1`	
`tao_sed_script2`	
`tao_sed_script3`	
`tao_sed_script4`	
`tao_sed_script5`	

Having unpacked `Tao1.0.tar`, the next step is to set up the environment. In order to do this the full pathname of the file `Tao1.0/bin` should be added to your path, and an environment variable `TAOPATH` should be created with its value set to the full pathname of the directory `Tao1.0/`, which will depend on where you have chosen to install the system. The latter can be achieved by adding the following line to your `.login` file:

```
setenv TAOPATH <full path>/Tao1.0/
```

You are now in a position to start creating and compiling TAO scripts.

A.2 Getting started

TAO scripts are stored in files with a `.script` suffix. Once a synthesis scenario has been described in a script it is ready for compilation. Note that TAO is a compiled language rather than being interpreted like Csound. This is because (a) a TAO script is actually a piece of C++ code in disguise; and (b) the synthesis model is computationally expensive and is therefore made as efficient as possible by relying on compiled code.

Supposing we have a `.script` file and we want to compile it, how do we achieve this? The `tao` command takes the name of a script (without the `.script` suffix) and compiles it, leaving an executable file with a `.exe` suffix in the current directory. For example if we have a script called `myscript.script`, then typing `tao myscript` leads to the creation of a file called `myscript.exe`. This file is an executable program which will carry out the synthesis scenario described in `myscript.script`. There are two ways of invoking a `.exe` file:

`myscript.exe`	⇒	No graphics, just do the synthesis.
`myscript.exe -g N`	⇒	Open a graphics window and display animations of the instruments described in the script. `N` is a real number and specifies the factor by which the amplitudes of any waves are exaggerated graphically. This has no effect on the sound output.

All microphones in a TAO script write their output samples to files with a `.tao` suffix. For example, in the following microphone declaration:

```
Microphone mic1:  outfile, stereo;
```

`mic1`'s output will be sent to the floating point soundfile `outfile.tao`[1] which may be converted into a `.aiff` file during or after the file's creation with the use of the `float2aiff` program. This program expects three arguments: the full pathname of the `.tao` file; the full pathname of the `.aiff` file; and the sampling rate required: 44100, 32000, 22050, 16000, 11025 or 8000. For example:

[1] In the current implementation the microphone `mic1` will actually write its output to a file called `/var/tmp/outfile.tao`, since raw floating point soundfiles can be very large.

```
float2aiff /var/tmp/outfile.tao outfile.aiff 44100
```

A.3 Mouse functions for use in the graphics window

If a `.exe` file is invoked with the `-g` option a graphics window appears. There are a number of mouse functions associated with the graphics window and they are listed below:

- Holding the left mouse button down whilst the mouse pointer is within the perimeter of the graphics window allows the whole graphics image to be dragged about.

- Pressing the middle or right buttons whilst the left button is held down causes the graphics image to be updated more or less frequently. The default is for the image to be update on every time step of the synthesis engine but each press of the middle button causes updating to occur five times less frequently, i.e. on every 5th step, then every 25th step etc.

- Pressing the right button whilst the left button is depressed reverses this process. If the graphics window is being updated on every time step, i.e. as frequently as possible and the right button is pressed whilst holding the left button down, the image is frozen until the same buttons are pressed in this combination again.

A.4 Some rules of thumb for instrument design

There are several rules of thumb for designing TAO instruments which are discussed here. When deciding upon the structure of a new instrument, Smalley's spectral typologies (introduced in section 1.4) offer a useful starting point. To recap briefly, all sounds fall into one of the following categories: *note*, *node* or *noise*. The first step in the design of a new instrument then is to answer the following questions:

- Is the instrument to produce specific *notes* and if so are they to be purely harmonic or inharmonic? Is the instrument monophonic or polyphonic?

- If clearly defined pitches are not required then is the instrument to produce certain clusters of partials or *nodes*? Once again how many distinct *nodes* are required?

- What is the intended texture of the imagined sound? Is it one smooth but continually evolving sound or a more granular texture involving large numbers of rapidly decaying sound events? If the texture of the sound is to be a single continually evolving event, then is it due to some continuous excitation or the naturally long decay time of the instrument? If it is due to some excitation then the instrument may not have to possess long decay times itself.

- What colouration of the whole sound is required and how distant are the components to seem from the listener?

- Which components are to play the role of *primary* components? (See section 6.3).

If *notes* with harmonically related partials are required then string components will fulfill this need. Depending on the kind of overall string response required the individual strings can be damped accordingly as described in section 6.5. Circular components are useful for producing inharmonic pitched sounds whereas rectangular components are much more suited to non-pitched clusters of partials, or nodes. If several distinct pitches or clusters are required then each one requires a separate component with its own characteristics.

Coupling components together

Some decision must be made as to how to couple the various components together if the sound is to be perceived as a cohesive whole. The instrument in section 6.4 shows one way of achieving this and uses strings to transmit energy from a set of circular pitched components to a common one-dimensional resonator and vice-versa, which then has microphones placed at either end. This leads to quite a pleasing stereo spatial image and because of the finite amount of time the waves take to propagate from the circular components to either microphone, the pitched sounds produced by each of the circular components seem to occupy distinct spatial locations. This effect is particularly noticeable over headphones. The stringed instrument of section

6.7.2 gives another alternative for mixing and colouring the sounds produced by the primary components in an instrument.

Microphone placement

Both the perceived size of a spatial image and the spectral balance of a sound are affected by the precise placement of microphones. Microphones placed close together lead to a very small spatial image, and whilst microphones placed further apart make the spatial image spread out more, a point is reached at which the signals on each channel become so different that the auditory perceptual system can no longer correlate them and create a single coherent spatial image. In spectral terms, all instruments require that some cells are locked, and the cells in their immediate vicinity will only be able to move with small amplitudes of vibration. In the same way that placing a pick-up near to the bridge of an electric guitar produces a brighter and more nasal sound and placing it nearer the middle of the string produces a more hollow sound, then placing TAO microphones near to or far away from locked cells (usually) has a similar effect. By far the best way to see what kind of spectral response will be obtained at a particular location on an instrument is to look at an animation of the instrument as the waves actually propagate. Visual feedback is useful for determining which parts of an instrument should be locked and damped so as to achieve the desired vibrational patterns.

Other factors to take into consideration

Performers often talk of high quality musical instruments seeming to 'sing' well without much effort being required on the part of the performer. Such subtleties have traditionally been the concern of acoustic instrument makers but with a system such as TAO, the same factors apply to the construction of virtual instruments. There are many factors to consider such as carefully choosing components which give an instrument the desired formants and which tend to vibrate in sympathy with, rather than fighting against each other. Once again the graphic animations prove invaluable in finding the right combinations of components. There is still much work to be done in understanding the relationship between an excitation model such as the bowing model provided and the instrument it is applied to. Applying a virtual bow to a stringed instrument with characteristics similar to those of a traditional stringed

instrument is one thing, but applying it to a completely different instrument will not always produce sensible results. In particular bowing an inharmonic instrument causes some difficulty because there can be no near-periodic Helmholtz behaviour as there is with a string. By far the best approach to take and the one that has been adopted throughout the development of the system is to 'try it and see/hear what happens'.

It has been found experimentally that there are two main causes of dull and lifeless sounds. For continuously excited sounds such as bowed strings the amount of damping applied to the ends of a string in particular is critical. Too much damping and the instrument becomes too stable, too little damping and the instrument becomes too unstable. The attractor of a dull sound often remains too static whereas a lively sound gives rise to an attractor which constantly shifts and changes shape. The other main factor is that instruments with uniform damping applied to every point often sound uninteresting. Natural sounds almost always show a correlation between amplitude and spectral brightness, and spectral evolution is strongly suggestive of energy dissipation. Therefore a sound whose spectral content remains the same throughout its duration is not usually very interesting. Of course we can make this spectral and amplitude decay occur over a fraction of a second or several minutes depending on the damping coefficients or decay times chosen, so we do not have to stick to realistic decays.

The overall sound texture required in a sound determines the basic decay times of the various components used and the type of score algorithm. In addition to the basic decay time of each individual component in an instrument we can alter each component's spectral evolution characteristics by damping local regions. We have seen how to simulate iterative events using TAO's score language (see section 5.5.5), and it should be possible to create sound events such as shattering sounds by simulating the effects of multiple independent bouncing objects, started off in synchrony but with individual time intervals between bounces etc. In chapter 7 a brief discussion introduced the notion of making the score's behaviour, and hence the macrostructure of a sound event, dependent upon the moment to moment behaviour of the instruments controlled by the score, thereby introducing feedback and a causal connection between micro- and macrostructure. This technique has not been explored to any great extent due to lack of time but it will almost certainly

lead to the most complex and interesting sound events.

Appendix B

TAO script language reference manual

This appendix serves as a reference manual for TAO's script language, working through the various features one by one and describing the syntax of each. Wherever instrument, microphone or cell messages can take a variety of forms each expecting a different number of arguments, each version of the message is explained together with the meanings of the various arguments.

B.1 Instrument declarations

Valid instrument declarations take one of the following forms:

(1) `String` *name*: *freq*, *decay*;
messages
...

(2) `Rectangle` *name*: *xfreq*, *yfreq*, *decay*;
messages
...

(3) `Circle` *name*: *freq*, *decay*;
messages
...

(4) `Ellipse` *name*: *xfreq*, *yfreq*, *decay*;
messages
...

```
(5)  Triangle      name:  xfreq, yfreq, decay;
                   messages
                   ...
```

Name consists of a string of alphanumeric characters and optionally underscores, as do all the identifiers used to refer to microphones and parameters within the script. `String` and `Circle` declarations require only a single frequency *freq*, determining the length of the string or the diameter of the circle respectively. For all other instruments two frequencies, *xfreq* and *yfreq*, are required, determining the size of the instrument in the x and y directions (see section 4.5 for an explanation). The *decay* argument specifies the overall decay time for the instrument and is converted into a damping value which is written to all the cells in the instrument. The *messages* contained within the body are separated by semicolons and are optional. Messages may be sent to an instrument from either inside or outside an instrument declaration. The syntax for each differs though. Within the body of an instrument declaration a message does not need to specify the name of the instrument it is being sent to. This is illustrated with the following examples:

```
(1)  Rectangle r:
          500 Hz, 75 Hz, 12.5 secs;
          lockcorners;
          setdecay(left,1/10,bottom,top,0.1 secs);
          ...

(2)  Rectangle r:
          500 Hz, 75 Hz, 12.5 secs;
          ...
     r.lockcorners.setdecay(left,1/10,bottom,top,0.1 secs);

(3)  r.lockcorners;
     r.setdecay(left,1/10,bottom,top,0.1 secs);
```

(1), (2) and (3) are all equivalent. A string of messages can be sent at once by appending them to the instrument's name separated by periods as in (2) or each can be sent to the instrument independently as in (3).

B.2 Pitch nomenclature

Pitches or frequencies may be specified in any of the following forms:

(1) <*note name*><*octave number*><*microtonal modification*>

(2) `pitch(`*octave*`.`*semitone*`)`

(3) *frequency* `Hz`

The triangular brackets in (1) indicate that there are no spaces or any other separating characters between the note name, octave and microtonal modifier. Note names are the letters `C, D, E, F, G, A, B` optionally followed by either a `#` or `b` indicating a sharp or flat. The octave number is an integer, with eight representing the octave containing middle C. The microtonal adjustment consists of a `+` or `-` followed by a fraction of the form *a*/*b* which represents the fraction of a semitone to add or subtract from the pitch specified. For example:-

`C8`	⇒	middle C
`C#7`	⇒	C♯ below middle C
`Ab8`	⇒	A♭ above middle C
`F#8+1/2`	⇒	F♯ plus a quarter tone above middle C
`B7`	⇒	B below middle C

The second pitch notation shown in (2) consists of the keyword `pitch` followed by a decimal number enclosed in brackets. The integer part specifies the octave whilst the decimal part is interpreted as an integer between zero and eleven and specifies the semitone within that octave. The examples below illustrate this more clearly. Note that if the decimal part contains more than two digits then the first two are interpreted as an integer whilst the remaining digits are interpreted as a fraction of a semitone.

`pitch(8.00)`	⇒	middle C
`pitch(7.01)`	⇒	C♯ below middle C
`pitch(8.08)`	⇒	A♭ above middle C

`pitch(8.065)` ⇒ F♯ plus a quarter tone above middle C

`pitch(7.11)` ⇒ B below middle C

B.3 Instrument messages

This section contains detailed descriptions of the various valid messages which can be passed to any instrument. Some of the messages are overloaded [1] and the appropriate version of a message is invoked automatically according to the number and type of the arguments given in the TAO script.

B.3.1 Setting an instrument's decay time

`Setdecay` and `resetdecay` enable an instrument's damping coefficient to be set in terms of a decay time. There are four overloaded versions of the message and each one provides a different way of specifying the region of the instrument affected:

(1) `setdecay(`*left*, *right*, *bottom*, *top*, *decaytime*`)`

(2) `setdecay(`*left*, *right*, *decaytime*`)`

(3) `setdecay(`*x*, *decaytime*`)`

(4) `setdecay(`*decaytime*`)`

(1) is intended for use with two-dimensional instruments and specifies a rectangular region whose decay time is altered. (2) and (3) are designed for use with strings and allow the region to be specified either as two *x* coordinates *left* and *right*, representing the left and right endpoints of the region, or a single point. Finally (4) works for any instrument and enables the decay time to be changed across the whole surface of the instrument. All coordinates should lie between zero and one.

Setting the decay time doesn't always have the effect which the user may intend. The precise effect is dependent on the size of the region chosen in relation to the size of the instrument. If the whole instrument receives a new decay time then the decay time specified has the correct effect. If however only a small region of the

[1] A function is overloaded if there exist several versions with the same name, each expecting different numbers of arguments and possibly different argument types

instrument has its decay time modified, then even if those cells affected would decay over the correct time interval if vibrating in isolation, when connected to the mass of cells which have a different decay time the effect of this local damping may be completely swamped. The decay time given should not be taken too literally.

The next set of messages mirror the ones just described but enable an instrument's decay time to be reset to the default value specified when it was created. The arguments expected by each version of the message are the same as for the above messages except for the omission of the original decay time, which the instrument itself keeps a record of.

(1) `resetdecay(`*left*`,` *right*`,` *bottom*`,` *top*`)`

(2) `resetdecay(`*left*`,` *right*`)`

(3) `resetdecay(`*x*`)`

(4) `resetdecay()`

B.3.2 Setting an instrument's damping coefficient

The next four message are identical to the `setdecay` messages except for the fact that the damping coefficient is given as a percentage, where 0% means that there is no frictional force to slow the cells down and 100% means that the cells will not move if a force is applied:

(1) `setdamping(`*left*`,` *right*`,` *bottom*`,` *top*`,` *coefficient%*`)`

(2) `setdamping(`*left*`,` *right*`,` *coefficient%*`)`

(3) `setdamping(`*x*`,` *coefficient%*`)`

(4) `setdamping(`*coefficient%*`)`

The comment above about the effect of damping various sized regions also applies to setting the damping as a percentage. If a large region is damped then it will have a big effect on the instruments vibrational patterns, whereas a small region will have less effect. The most reliable way of determining the damping coefficient is though experimentation. As with the `setdecay` messages there are four equivalent messages

to reset the damping coefficient to its original value. These messages have identical functionality to the four `resetdecay` messages but are included for purposes of consistency.

(1) `resetdamping(`*left*`,` *right*`,` *bottom*`,` *top*`)`

(2) `resetdamping(`*left*`,` *right*`)`

(3) `resetdamping(`*x*`)`

(4) `resetdamping()`

B.3.3 Locking parts of an instrument

The next set of messages enable regions of an instrument to be locked and require no arguments:

(1) `lockleft`

(2) `lockright`

(3) `locktop`

(4) `lockbottom`

(5) `lockperimeter`

(6) `lockcorners`

(7) `lockends`

The behaviour of `lockleft`, `lockright`, `locktop` and `lockbottom` is straightforward for rectangular instruments, simply locking whole sides of the instrument. For other instruments only the cells located at the extremities of the instrument are locked. The `lockcorners` message is intended for use with rectangular and triangular instruments and `lockends` is for use with strings. The `lockperimeter` message is only useful for instruments other than strings.

There are three overloaded versions of the `lock` message which allows the user to specify the region of an instrument to be locked. (1) allows a rectangular region to be specified in the same manner as for `setdecay` and `setdamping`; (2) allows a single

point to be specified with a pair of x and y coordinates; and (3) expects a single x coordinate specifying a point on a string.

(1) `lock(`*left*`,` *right*`,` *bottom*`,` *top*`)`

(2) `lock(`x`,` y`)`

(3) `lock(`x`)`

B.3.4 Accessing points on an instrument

The notation for accessing points on an instrument consists of the instrument name followed by a set of coordinates in brackets. For strings only a single x coordinate is required. For two dimensional instruments both an x and y coordinate are necessary:

(1) *instr1*`(`x`)`

(2) *instr1*`(`x`,` y`)`

The coordinates are always normalised to be between zero and one. For x, zero and one represent the left and right hand sides of the instrument respectively regardless of the instruments shape. For y, zero and one represent the bottom and top of the instrument respectively. Selecting a point on an instrument using this notation returns a reference to an individual cell whose various physical attributes can either be read or altered.

B.4 Nomenclature for accessing parts of instruments

The keywords `left`, `right`, `bottom` and `top` are provided and when used within instrument messages, or anywhere else within a script, are replaced by the appropriate numerical value of either zero or one. They are provided for script legibility.

B.5 Cell attributes of interest to the user

Each cell has a number of attributes which are of direct relevance in developing interesting new excitation models and can be accessed by the user. These include:

(1) `position`

(2) `velocity`

(3) `force`

(4) `mass`

All are measured in arbitrary numerical units. The `force` variable is usually only altered via the cell messages described in the next section although the user can set the force acting upon a cell directly within a score if required. The `mass` of each cell is set by default to a value of 3.5 which gives the material optimum frequency response. Under no circumstances should the mass of a cell be set to less than this value.

The mass can be set to any value greater than 3.5 though and the best way to use this technique is by experimentation since there is no simple one to one relationship between the mass of a cell and the sonic effect it will have on an instrument. The only rule of thumb is that a cell with a high mass will have more inertia and will therefore move more slowly than a cell with a low mass. In practice this means the cell with the higher mass will have a preference for lower frequency vibrations. See sections 6.7.3 and C.4 for practical examples of the use of this technique.

All of these attributes can be used either as input or output parameters, so for example we can set the force acting upon a cell or we can simply read it off and use the value elsewhere in another expression. When assigning new values to these cell attributes some care has to be taken since if we suddenly move a cell to a completely new position without updating its velocity and force accordingly we can expect some strange transient behaviour to appear in the material. Assuming the existence of two instruments `string1` and `rect1` then all the following are valid script fragments:

(1) `string1(`x`).position=0;`

(2) `If rect1(`x`, `y`).velocity > 2.5:` *body* ...

(3) `string1(`x`).force*=rect1(`x`, `y`).velocity/10.0;`

B.6 Cell messages

Once selected using the notation described in section 5.4.8, a cell can be sent various messages, which in the current version of TAO include `applyforce` and `bow`:

(1) `instr1(`*x*`, `*y*`).applyforce(`*force*`);`

(2) `instr1(`*x*`, `*y*`).bow(`*downward force*`, `*velocity*`);`

The arguments to both of these messages are measured in arbitrary numerical units. For *applyforce* the force applied can be of any magnitude and within a single score it is the relative magnitudes of the forces used which are of greater importance. Applying a force of one or one million to an instrument only affects the magnitude and *not* the character of vibrations. In other words unlike most physical materials the cellular elastic material does not sound brighter if we hit it harder. It only sounds brighter if we hit it more sharply i.e. a higher force is applied over a shorter time interval. For the `bow` message the parameters are more critical and the user should refer to sections 6.7.1, 6.7.2 and appendix C for examples of sensible values to use as starting points.

B.7 Microphone declarations

A microphone declaration takes one of the following forms:

(1) `Microphone` *name*`:` *outfilename*`, mono;`

(2) `Microphone` *name*`:` *outfilename*`, stereo;`

(3) `Microphone` *name*`:` *outfilename*`,` *source*`;`

(4) `Microphone` *name*`:` *outfilename*`,` *leftsource* `,` *rightsource*`;`

Name specifies the name of the microphone, i.e. the identifier with which it is referred to throughout the script and *outfilename* specifies the name of the output file to which the sound samples are sent. The sound samples are initially written to this file in raw floating point format and must be converted to a *.aiff* or similar soundfile format using a separate post-processing program `float2aiff` which also

normalises the samples to achieve maximum dynamic range and then writes them to a standard *.aiff* file. This program is described in appendix G.

In the present implementation only mono and stereo microphones are allowed. In the case of (1) and (2) above, the sources for sound output are left to be determined within the score. The way in which this is achieved is described in 5.5 and the feature is included to enable microphones to be moved around during a performance. In order to achieve continuously variable microphone positions some kind of interpolation between cells is required. This feature is not yet implemented but it would not be a very serious task to do so.

B.8 Microphone messages

A microphone object's purpose in life is to take floating point samples generated by arbitrary mathematical expressions, buffer them, and write them to an output file. In order to do this three messages are provided:

(1) `leftout:` *expression*;

(2) `rightout:` *expression*;

(3) `output:` *expression*;

(3) is used with a mono microphone. These messages will only generate output samples if placed at a scope within the score where they are active. For example:

```
Score 10 secs:
     Before 5 secs:
          mic1.leftout:  expression;
          mic1.rightout:  expression;
               ...
          ...
```

will cause `mic1` to generate sound samples only from zero to five seconds.

Note that the notation *message: expression;* is exactly equivalent to *message(expression);* and either form can be used. In this case though the expressions used to generate output samples can often involve many different points on different

components of an instrument and the non-bracketed syntax is more legible. Also in terms of visual style this syntax is more in keeping with the use of colons elsewhere in a script.

B.9 Glueing and joining

There are several different forms of the `Glue` command to allow for various combinations of one and two-dimensional instruments:

(1) `Glue` *instr1*(x_1,y_1) `to` *instr2*(x_2,y_2);

(2) `Glue` *instr1*(x_1) `to` *instr2*(x_2,y_2);

(3) `Glue` *instr1*(x_1,y_1) `to` *instr2*(x_2);

(4) `Glue` *instr1*(x_1) `to` *instr2*(x_2);

`Join` on the other hand only appears in one form with four arguments paired into coordinates for each instrument. In combination, these coordinates specify where the join is to occur:

`Join` *instr1*(x_1,y_1) `to` *instr2*(x_2,y_2)

The arguments x_1, y_1, x_2, y_2 are interpreted the following way (remember that the keywords `left`, `right`, `bottom` and `top` are provided also):

If $x_1 = 0$	and $x_2 = 1$	then join left of *instr1* to right of *instr2*
If $x_1 = 1$	and $x_2 = 0$	then join right of *instr1* to left of *instr2*
If $x_1 = 0$	and $x_2 = 0$	then join left of *instr1* to left of *instr2*
If $x_1 = 1$	and $x_2 = 1$	then join right of *instr1* to right of *instr2*

If $y_1 = 0$	and $y_2 = 1$	then join bottom of *instr1* to top of *instr2*
If $y_1 = 1$	and $y_2 = 0$	then join top of *instr1* to bottom of *instr2*
If $y_1 = 0$	and $y_2 = 0$	then join bottom of *instr1* to bottom of *instr2*
If $y_1 = 1$	and $y_2 = 1$	then join top of *instr1* to top of *instr2*

If x_1 and x_2 are both equal to either zero or one then the join runs vertically and the left or right sides of the two instruments are joined. In this mode y_1 and y_2 are used to specify a horizontal centre line running through both instruments in order to line them up for the join. If however y_1 and y_2 are both equal to either zero or one, then the join runs horizontally and x_1 and x_2 are used to specify a vertical centre line in order to line the instruments up. For a join to take place either x_1 and x_2 or y_1 and y_2 have to be equal to either zero or one simultaneously.

In the following examples only (1), (2) and (3) lead to *instr1* and *instr2* being joined together. (1) is straightforward and joins the left hand side of *instr1* to the right hand side of *instr2* lining up two points one halfway up *instr*, the other 0.7 of the way up *instr2*. (2) is similarly straightforward but shows that a join may occur between two left sides or right sides etc. In (3) x_1, x_2, y_1 and y_2 are all equal to either one or zero but in this case x_1 and x_2 are tested first and are thus interpreted as specifying the sides to be joined, leaving y_1 and y_2 to specify the centre line.

```
(1)   Join instr1(left,0.5)     to   instr2(right,0.7)

(2)   Join instr1(right,1/10)   to   instr2(right,7/10)

(3)   Join instr1(right,top)    to   instr2(left,top)

(4)   Join instr1(0.2,0.5)      to   instr2(0.7,0.3)
```

For further examples and a graphical explanation of the arguments to `Join` see section 5.4.7.

B.10 Time nomenclature

There are three units of time supported by the script language *seconds*, *milliseconds* and *minutes*:

time `secs`

time `msecs`

time `min`

The time *t* can be a constant, a parameter or a whole expression in which case it is safest to enclose the whole expression in parentheses and then put the units of

measurement after the whole parenthesised expression.

The system variable `Time` keeps a track of real time measured in seconds and can be used anywhere within the score. Another system variable `Sample` keeps track of the number of time steps elapsed since the beginning of the performance, although the user should never have cause to access this variable and should certainly never assign it a new value.

B.11 Performance parameters

Parameters are floating point variables any number of which the user can declare. A parameter declaration comes in the following form:

`Parameter` p_1=a, p_2=b,..,p_n=n;

The initial values a, b .. n may be constants or expressions possibly involving other parameters already declared and initialised, and are optional.

B.12 Score control structures

The score represents a hierarchical structure dividing the total time allotted for a performance into separate time intervals. Each time interval or instant in time specified represents an *event* of some kind. Simple events such as setting a parameter's value or locking a point on an instrument need only occur at an instant in time whilst others such as excitations occur over intervals of time.

The building blocks from which a score is constructed are referred to as *control structures*. A score starts off with the `Score` control structure which sits at the top of the hierarchy and specifies the duration of the performance:

`Score` *duration*:

Other control structures include:

(1) `At` *start time* `for` *duration*: *body* ...

(2) `From` *start time* `to` *end time*: *body* ...

(3) `Before` *end time*: *body* ...

(4) `After` *start time*: *body* ...

(5) `At` *time*: *body* ...

(6) `Every` *interval*: *body* ...

(7) `ControlRate` *interval in samples*: *body* ...

(8) `If` *condition*: *body* ...

(9) `If` *condition*: *body 1* ...
`Else`: *body 2* ...

(10) `If` *cond 1*: *body 1* ...
`ElseIf` *cond 2* : *body 2* ...
`ElseIf` *cond 3* : *body 3* ...
.
.
`Else`: *body n*

The special variables **start** and **end** are used within the score for two purposes, firstly to allow code within the body of a control structure to refer to the start and end times specified by the head. This is useful for executing instructions just once at the beginning or end of a time interval using the forms:

```
At start:      body ...
At end:        body ...
```

The second use is for the special time varying functions **linear** and **expon** described in the next section.

The next example explicitly shows how the values **start** and **end** change throughout a score consisting of nothing but nested control structures:

```
start   end

0       10        Score 10 secs:
0       2              At 0 secs for 2 secs:
0       2                   ...
0       3              Before 3 secs:
0       3                   ...
2       7              From 2 secs to 7 secs:
2       4                   Before 4 secs:
```

```
2       4                   ...
5       7               After 5 secs:
5       7                   ...
2       7               ...
5       10          After 5 secs:
5       10              ControlRate 100:
5       10                  ...
5       10              Every 0.1 secs:
5       10                  ...
5       10              ...
0       10          ...
```

When `start` and end are accessed within the body of an `At..for`, `From..to`, `Before` or `After` control structure, their values change to reflect the more local start and end time whereas when they are accessed within a `ControlRate` or `Every` structure their values are left unaltered.

B.13 Mathematical expressions

All the standard mathematical operators one would expect such as `+`, `-`, `*` and `/` are available. In addition to the standard assignment operator `=` there are four other assignment operators inherited from C++ `+=`, `-=`, `*=` and `/=` which are used in the following way:

(1) *parameter* `+=` *expression*

(2) *parameter* `-=` *expression*

(3) *parameter* `*=` *expression*

(4) *parameter* `/=` *expression*

For example `+=` adds the value of *expression* to the value held in *parameter* and then stores the value back in *parameter*. The other operators work in a similar fashion.

B.14 Mathematical functions

A variety of standard mathematical functions are available for use within a script. These are inherited from C++ and include the following, taken straight from the IRIX 5.3 manual page for the standard maths library:

acos(x)	inverse trig func
acosh(x)	inverse hyperbolic func
asin(x)	inverse trig func
asinh(x)	inverse hyperbolic func
atan(x)	inverse trig func
atanh(x)	inverse hyperbolic func
atan2(x,y)	inverse trig func
cbrt(x)	cube root
cos(x)	trig func
cosh(x)	hyperbolic func
drem(x)	remainder
exp(x)	exponential
expm1(x)	exp(x)-1
fabs(x)	absolute value
fceil(x)	integer no less than
floor(x)	integer no greater than
cos(x)	trig func
cosh(x)	hyperbolic function
exp(x)	exponential
expm1(x)	exp(x)-1
hypot(x,y)	Euclidean distance
log(x)	natural logarithm
log10(x)	logarithm to base 10
log1p(x)	log(1+x)
pow(x)	exponential x**y
rint(x)	round to nearest integer
sin(x)	trig func
sinh(x)	hyperbolic func
sqrt(x)	square root
trunc(x)	truncate to integer
tan(x)	trig func
tanh(x)	hyperbolic func

Manual pages for each individual function are available with IRIX 5.3.

In addition to the mathematical functions described, two special time varying functions `linear` and `expon` are provided. These functions come in the following form:

linear(*initial*,*final*)	changes linearly from *initial* to *final*
expon(*initial*,*final*)	changes exponentially from *initial* to *final*

Ordinarily with such functions we would have to specify the time interval over which they were supposed to change but in a TAO score these two values are implicitly specified by the values of `start` and `end` described in the previous section. Since the values of these two variables depend on their context within the score, `linear`

and **expon** return a value which changes from *initial value* to *final value* over the appropriate time interval. See section 5.5.2 for an explanation of how the values of **start** and **end** are affected by their scope within the score.

B.15 Text screen output

During a performance text can be sent to the output text window (via C++'s standard output stream) with the use of the **Display** command which is followed by a list of items to be displayed, separated by commas. Displayable items include character string constants such as **"a string of characters"**, parameter values, mathematical expressions and two special items **newline** and **sameline** which cause a carriage return and linefeed or just a carriage return respectively.

Display *item1*, *item2*, .., *itemn*;

Appendix C

Sound examples

This appendix contains the scripts which were used to generate the TAO sound examples. The TAO system has been set up such that the cellular material is capable of producing vibrations covering (almost) the whole audible spectrum. For example, supposing we want to create a TAO string with a fundamental frequency of 100 Hz. In order to achieve this frequency we can either use a large number of cells with small masses or a smaller number of cells with larger masses. The former string will have the same fundamental frequency as the latter but will exhibit a much better frequency response, being capable of higher frequency modes of vibration. TAO is 'hard-wired' at the moment to always opt for maximum frequency response. The disadvantage of this is that in order to create instruments with very low frequency modes of vibration, we have to use very large numbers of cells, and this means more computation and thus a longer wait for sounds to be produced.

For this reason, whilst all of the sound examples were produced directly by the TAO scripts listed, and were recorded without the aid of any external audio effects, some of them were composed 'in miniature' with higher pitches and shorter time intervals in the score, and were only later transposed down to lower pitches. This technique has been applied to instruments containing large, two-dimensional pieces of cellular material, for which the computational problem is more pronounced. These transposed sounds, whilst illustrating quite nicely the coherence inherent in TAO's output, do lack some definition in the higher frequencies. This problem will be solved in the future either through the use of more powerful technology, or by modifying

TAO so that the user can decide upon the audio bandwidth required of the material.

C.1 Sounds produced by a single string damped at one end

```
////////////////////////////////////////////////////////////////////////////////
// Script name: singlestring.script                                tracks 1-4
//
// This script explores the effects of damping one end of a single TAO string
// to varying degrees. The commented lines show the damping coefficients
// chosen. Note that the overall decay time of the string is infinite so it
// does not behave like a real string. This makes the effect of the local
// damping more pronounced, allowing us to concentrate on how it affects the
// string's spectral decay.
////////////////////////////////////////////////////////////////////////////////

String s:
    110 Hz, 0 secs;
    lockends;
    setdamping(left, right, 0%);
    ...

//s.setdamping(left, 1/20, 0.01%);    // long decay              track 1
//s.setdamping(left, 1/20, 0.1%);     // medium decay            track 2
//s.setdamping(left, 1/20, 1%);       // short decay             track 3
//s.setdamping(left, 1/20, 10%);      // v. short decay          track 4

Microphone mic1: dampedstring, stereo;

Score 20 secs:
    At start for 0.5 msecs:
        s(0.1).applyforce(10.0);                    // pluck the string
        ...

    At 19 secs: s.setdamping(left, 1/5, 0.5%); ...  // damp it

    mic1.leftout:  s(0.05);
    mic1.rightout:  s(0.95);

    Every 0.01 secs: Display Time, newline; ...
    ...
```

C.2 String harmonics

```
////////////////////////////////////////////////////////////////////////////////
// Script name: stringharms.script                                 tracks 5-8
//
// This script simulates string harmonics. The overall decay time of the
// string is set to be quite long and then the end of the string is
// damped in order to alter the spectral decay response. This gives the
// instrument a more realistic string-like response. Finally, the string is
// plucked and then after 2 seconds is damped at one of the nodes (one of
// the commented lines must be uncommented).
////////////////////////////////////////////////////////////////////////////////

String s:
    110 Hz, 0 secs;
    lockends;
    setdamping(left, right, 0%);
    ...

s.setdecay(60 secs);                    // overall amplitude decay of
                                        // string quite long


s.setdamping(left, 1/20, 0.05%);        // spectral decay quite


Microphone mic1: harm, stereo;

Score 20 secs:
    At start for 0.5 msecs:
        s(1/17).applyforce(10.0);
        ...

//  At 2 secs: s.setdamping(1/2, 0.5%); ...   // 2nd harmonic    track 5
//  At 2 secs: s.setdamping(1/3, 0.5%); ...   // 3rd harmonic    track 6
//  At 2 secs: s.setdamping(1/4, 0.5%); ...   // 4th harmonic    track 7
```

```
//  At 2 secs: s.setdamping(1/5, 0.5%); ...   // 5th harmonic    track 8

    At 19 secs: s.setdamping(left, 1/5, 0.5%); ...

    mic1.leftout:  s(0.05);
    mic1.rightout:  s(0.95);

    Every 0.01 secs: Display Time, newline; ...
    ...
```

C.3 Rectangular sheets joined together

This sound has been transposed down by a factor of 0.7256, equivalent to playing the samples back at 32 Khz instead of 44.1 Khz.

```
////////////////////////////////////////////////////////////////////////////////
// Script name: joinsound.script                                    track 9
//
// The instrument used in this sound example comprises six rectangular sheets
// joined together in the same kind of format as the example given in chapter
// six. Each rectangular sheet is given a long decay time but local regions
// of three of the rectangles are damped locally to change the overall spectral
// content as the sound evolves. The instrument is struck once and is then
// left to follow its own behaviour.
//
// Output is taken from four pairs of points on the instrument represented by
// the parameters l1, r1, l2, r2, l3, r3, l4 and r4. The final stereo signal
// is generated by a continual process of crossfading between these individual
// stereo signals. This is achieved by amplitude modulating the pairs of
// signals with sinusoidal signals of phase 0, pi/2, pi and 3*pi/2. The four
// sinusoidal modulation signals are phase locked and change from 10 Hz
// to 0.5 Hz over the duration of the performance.
//
// Since the local regions of damping affect the spectral content of the
// signals and since the speed of crossfading slows down throughout, an
// impression of energy dissipation is created.
//
// NOTE: This sound example has shown up a bug in TAO. The rate at which the
// crossfading occurs should change smoothly throughout the performance but
// it seems to change in discrete steps instead, staying at one rate for
// about 10 or 11 complete crossfade cycles and then changing to a slower rate.
// At the time of writing this bug has not been traced.
////////////////////////////////////////////////////////////////////////////////

Rectangle one: 100 Hz, 4000 Hz, 60 secs; ...
Rectangle two: 4000 Hz, 100 Hz, 60 secs; ...
Rectangle three: 100 Hz, 4000 Hz, 60 secs; ...
Rectangle four: 4000 Hz, 100 Hz, 60 secs; ...
Rectangle five: 4000 Hz, 100 Hz, 60 secs; ...
Rectangle six: 100 Hz, 4000 Hz, 60 secs; ...

Join one(left,top) to two(right,top);
Join two(right,bottom) to three(left,bottom);
Join three(right,bottom) to four(left,bottom);
Join four(left,top) to one(right,top);
Join four(right,1/3) to six(left,centre);
Join one(4.5/6,top) to five(centre,bottom);

one.lock(4/6,5/6,0,0);
three.lock(2/5,3/5,1,1);
five.setdecay(left,right,9/10,top, 1 secs);
six.setdecay(9/10,right,bottom,top, 1 secs);
four.setdecay(left, right, 9/10, top, 0.1 secs);
four.setdecay(left, right, bottom, 1/10, 0.1 secs);

Microphone m: joinsound, stereo;

Parameter pi2=3.141592653*2.0, crossfaderate;
Parameter pi=3.141592653;
Parameter phase2=pi/2.0, phase3=pi, phase4=pi*3/2;

Parameter angle=0.0;

Parameter l1, l2, l3, l4;
Parameter r1, r2, r3, r4;

Score 20 secs:
    At start for 10/44100:
        five(1/2, 9/10).applyforce(10.0);
```

```
    five.label(1/2, 9/10, -20, 20, "excitation", BLUE);
    ...

crossfaderate=expon(10.0, 1/2) Hz;

Every 0.1 secs:
    Display
    Display newline, "Time=",Time;
    Display " crossfaderate=",crossfaderate;
    Display " 1=",11;
    Display " 2=",12;
    Display " 3=",13;
    Display " 4=",14, newline;
    ...

angle+=pi2*crossfaderate/audiorate;

    l1=two(1/2, 0.7) * sin(angle);
    r1=two(1/2, 0.3) * sin(angle);

    l2=one(1/4, 1/2) * sin(angle + phase2);
    r2=three(1/4, 1/2) * sin(angle + phase2);

    l3=three(3/4, 1/2) * sin(angle + phase3);
    r3=one(right, 1/2) * sin(angle + phase3);

    l4=four(1/2, 1/2) * sin(angle + phase4);
    r4=six(0.8, 1/2) * sin(angle + phase4);

    m.leftout:  l1 + l2 + l3 + l4;
    m.rightout: r1 + r2 + r3 + r4;

// This bit is only for the graphics

    one.label(1/4, top, -10, 15, "one", BLACK);
    three.label(1/4, bottom, -10, -15, "three", BLACK);
    two.label(left, 1/2, -30, -5, "two", BLACK);
    four.label(right, 9/10, 10, -5, "four", BLACK);
    six.label(3/4, bottom, -10, -15, "six", BLACK);
    five.label(left, 4/5, -40, -5, "five", BLACK);
    ...
```

C.4 A prepared string buzzing against an obstacle

```
////////////////////////////////////////////////////////////////////////////
// Script name: goodbuzz2.script                                  track 10
//
// This sound uses a single string which is 'prepared' by increasing the masses
// of two of the cells to a value of fifty. The default mass of all cells when
// created is 3.5. This value is not arbitrary but makes the material
// propagate waves as quickly as possible so as to achieve a good frequency
// response, Under no circumstances should a cell's mass be made less than 3.5
// or the model becomes unstable.
//
// The string is excited with a simple impulse and then vibrates freely, except
// that an obstacle is placed in its way, one third of the way along its
// length. The obstacle's vertical position changes exponentially, over the
// twenty second duration of the score. Whenever the string's amplitude becomes
// greater than the obstacle's vertical position it is limited to this value.
// Changing the obstacle's position exponentially ensures that the string
// keeps hitting the obstacle, but only just making contact. Left and right
//  channels of output are taken from either end of the string.
////////////////////////////////////////////////////////////////////////////


String s: E6-1/2, 2 min; ...
s.setdamping(left, 1/40, 0.02%).lockleft;
s.setdamping(39/40, right, 0.02%).lockright;
s(3/4).mass=50.0;
s(1/4).mass=50.0;

Microphone m: test, stereo;

Parameter obstacle_position;

Score 20 secs:
    Every 0.1 secs:
        Display "Time=", Time, newline;
        ...

    At start for 1 msecs:
        s(0.1).applyforce(1.0);
        ...
```

```
obstacle_position = expon(9.0,0.5);

After 1 secs:
    If s(3/10).position > obstacle_position:
        s(3/10).position = obstacle_position;
        s(3/10).velocity *= -0.5;
        ...

    If s(7/10).position > obstacle_position:
        s(7/10).position = obstacle_position;
        s(7/10).velocity *= -0.5;
        ...
    ...

m.leftout: s(0.05);
m.rightout: s(0.95);
...
```

Track 11 provides another sound example which is a slight variation on the above.

C.5 A dynamically prepared string buzzing against an obstacle

```
////////////////////////////////////////////////////////////////////////////
// Script name: strangebuzz1.script                                track 12
//
// This sound is similar to the sound described in the script goodbuzz2.script
// in that it uses a 'prepared' string in which the masses of several
// individual cells are altered from the default value. In this sound though
// the masses are altered dynamically throughout the performance. In addition
// the string buzzes against an obstacle whose distance from the string
// gradually decreases. This is a good example of quite an abstract TAO
// sound still having qualities suggestive of gesture and texture.
////////////////////////////////////////////////////////////////////////////

String s: E6-1/2, 2 min; ...

s.setdamping(left, 1/40, 0.02%).lockleft;
s.setdamping(39/40, right, 0.02%).lockright;

Microphone mic1: strangebuzz1, stereo;

Parameter obstacle_position;

Score 20 secs:
    Every 0.1 secs:
        Display "Time=", Time, newline;
        ...

    At start for 1 msecs:
        s(0.1).applyforce(10.0);
        ...

    s(1/2).mass=linear(50, 1500);
    s(3/4).mass=linear(3, 1400);
    s(1/4).mass=linear(3, 1200);

    ControlRate 1000:
        Display "m1=", s(3/4).mass;
        Display "   m2=", s(1/4).mass, newline;
        ...

obstacle_position = linear(7.0,0.5);

After 1 secs:
    If s(3/10).position > obstacle_position:
        s(3/10).position = obstacle_position;
        s(3/10).velocity *= -0.5;
        ...

    If s(7/10).position > obstacle_position:
        s(7/10).position = obstacle_position;
        s(7/10).velocity *= -0.5;
        ...
    ...

mic1.leftout: s(0.05);
mic1.rightout: s(0.95);
...
```

C.6 The effects of damping on a single rectangular sheet

These sounds have all been transposed down by a factor of 0.5, equivalent to playing the samples back at 22.05 Khz instead of 44.1 Khz.

```
////////////////////////////////////////////////////////////////////////////////
// Script name: rectangle.script                                  tracks 13-16
//
// A number of sound examples were generated with this script by either leaving
// the rectangle uniformly damped or damping local regions. Note that because
// of the 60 minute decay time, this instrument does not behave like a normal
// percussion instrument. If struck it will ring on ad infinitum. The sounds
// generated by damping local regions force certain partials to die away,
// but the ones which are free to continue vibrating will do so indefinitely.
//
// Sound descriptions:
// track 13: undamped rectangular sheet. Sound continues indefinitely with
//           unchanging spectrum.
// track 14: rectangle with some damping applied in top left corner. Most
//           of the partials are affected. Only a few of the higher partials
//           involving modes of vibration which do not touch the damped
//           region are allowed to continue.
// track 15: rectangle damped at centre. A different set of partials are
//           affected this time.
// track 16: rectangle damped along left hand edge. Once again affects virtually
//           all partials, leaving only a few higher ones ringing.
//
// Note:     It is necessary in this case to apply two equal and opposite
//           impulses to the instrument as no points on the instrument are
//           locked and it would drift away from the zero position otherwise.
////////////////////////////////////////////////////////////////////////////////


Rectangle r:
    156 Hz, 200 Hz, 60 min;
    ...

//                                              track 13
//r.setdamping(left, 1/6, 5/6, top, 0.07%);     track 14
//r.setdamping(5/12, 7/12, 5/12, 7/12, 0.07%);  track 15
//r.setdamping(left, 1/6, bottom, top, 0.01%);  track 16

Microphone m: undamped_rect, stereo;

Score 15 secs:
    At start for 0.1 msecs:     r(0.25, 0.25).applyforce(5.0);
    At 0.2 msecs for 0.1 msecs: r(0.25, 0.25).applyforce(-5.0);

    Every 0.01 secs:
        Display Time, newline;
        ...

    m.leftout:  r(0.05, 0.05);
    m.rightout:  r(0.95, 0.95);
    ...
```

C.7 An illustration of implied motion, acceleration, impact and decay

Track 17 is a composite of two sounds which were created directly from TAO scripts. The first was produced by the 'obstacle' technique described in section 6.7.3 and was subsequently reversed. Another percussive sound produced with a large rectangular sheet of material damped in one corner was then appended onto the end of the first. The sound builds up from nothing, gradually accelerated and becoming more agitated until a crescendo is reached, at which point the impact of the second sound is heard followed by a gradual decay. This sound illustrates the ability of TAO to

create sounds which strongly suggest motion, acceleration and a physical cause, at the same time as being abstract in nature.

C.8 A single bowed string

```
////////////////////////////////////////////////////////////////////////////////
// Script name: highbow.script                                        track 18
//
// This sound is fairly synthetic in nature as the instrument only comprises
// a single string but nevertheless illustrates the use of a virtual bow. The
// way in which the bow's velocity and downward force change in the first few
// tenths of a second is quite critical for the kind of transients produced.
// Things to experiment with include the maximum velocity of the bow, the
// force, and the amounts of damping at the ends of the string. Too little
// damping produces very noisy sounds whereas too much produces sounds which
// are too periodic and thus not very interesting. Things to try include
// glueing one end of the string to some other instrument and taking output
// from this resonator instead of from the string.
////////////////////////////////////////////////////////////////////////////////

String s1: C8, 5 secs; ...

s1.lockends;
s1.setdamping(left,1/20,0.7%);
s1.setdamping(19/20,right,0.7%);

Microphone m: highbow, stereo;

Parameter bowforce=1.0, bowvelocity;
Parameter vibratodepth;

Score 9 secs:
    From 0 secs to 2 secs: vibratodepth=linear(0, 1/100); ...
    After 2 secs: vibratodepth=linear(1/100, 0); ...

    s1.vibrato(5 Hz, vibratodepth);

    At 0 secs for 0.2 secs:
        bowvelocity=expon(0.01, 1.0);
        ...

    From 0.2 secs to 4 secs:
        bowvelocity=linear(1.0, 5.0);
        ...

    From 4 secs to 8 secs:
        bowvelocity=linear(5.0, 1.0);
        ...

    At 0 secs for 8 secs:
        s1(0.3).bow(bowforce, bowvelocity);
        ...

m.leftout: s1(0.05);
m.rightout: s1(0.95);

Every 0.1 secs:
    Display "Time=", Time, newline;
    ...
...
```

C.9 A stringed instrument with pairs of strings bowed together

This sound has been transposed down by a factor of 0.7256, equivalent to playing the samples back at 32 Khz instead of 44.1 Khz.

```
////////////////////////////////////////////////////////////////////////////////
// Script name: bowreson.script                                       track 19
//
// This sound uses a stringed instrument with four strings and a rectangular
// resonator to which they are glued. The resonator's dimensions were chosen
```

```
// partly through experimentation and partly because a long thin rectangular
// strip of material is not too computationally expensive. The strings are
// bowed in pairs with varying bow velocity but fixed bow force. The most
// critical parameters in this script are the amounts of damping applied to
// the ends of the strings, the overall decay time of the resonator, the
// maximum bow velocity, and the initial attack of the bow's velocity.
////////////////////////////////////////////////////////////////////////////

String s1: 100 Hz, 10 secs; lockleft; ...
String s2: 150 Hz, 10 secs; lockleft; ...
String s3: 225 Hz, 10 secs; lockleft; ...
String s4: 337.5 Hz, 10 secs; lockleft; ...

Rectangle resonator: 1800 Hz, 160 Hz, 0.5 secs; ...

Glue s1(right) to resonator(1/10, 1/5);
Glue s2(right) to resonator(1/10, 2/5);
Glue s3(right) to resonator(1/10, 3/5);
Glue s4(right) to resonator(1/10, 4/5);

resonator.locktop.lockbottom;
resonator(1/2,1/2).mass=50.0;

s1.setdecay(left, 1/30, 0.03 secs).display_at(0, 50);
s2.setdecay(left, 1/30, 0.03 secs).display_at(0, 100);
s3.setdecay(left, 1/30, 0.03 secs).display_at(0, 150);
s4.setdecay(left, 1/30, 0.03 secs).display_at(0, 200);
s1.setdecay(29/30, right, 0.03 secs);
s2.setdecay(29/30, right, 0.03 secs);
s3.setdecay(29/30, right, 0.03 secs);
s4.setdecay(29/30, right, 0.03 secs);

resonator.display_at(0, 320);
resonator.amplification=50.0;
resonator.setdecay(left, 1/10, bottom, top, 0.03 secs);

Parameter bowforce, bowveloc, maxbowveloc, bowposition;
Parameter p;

Microphone mic: bowtestb, stereo;


Score 15 secs:
    Every 0.1 secs:
        Display "Time=", Time, newline;
        ...

    At start:
        bowforce=1;
        ...

    At 0 secs: maxbowveloc=1; bowposition=0.1; bowforce=1.0; ...

// bow first pair of strings together

    At 0 secs for 3 secs:
        At start for 0.1 secs: bowveloc=expon(0.01, maxbowveloc); ...
        After start + 0.1 secs:  bowveloc=expon(maxbowveloc, 0.1); ...
        s3(bowposition).bow(bowforce, bowveloc);
        s4(bowposition).bow(bowforce, bowveloc);
        ...

// bow second pair of strings together

    At 4 secs for 3 secs:
        At start for 0.1 secs: bowveloc=expon(0.01, maxbowveloc); ...
        After start + 0.1 secs:  bowveloc=expon(maxbowveloc, 0.1); ...
        s2(bowposition).bow(bowforce, bowveloc);
        s3(bowposition).bow(bowforce, bowveloc);
        ...

// bow second pair of strings together

    At 8 secs for 3 secs:
        At start for 0.1 secs: bowveloc=expon(0.01, maxbowveloc); ...
        After start + 0.1 secs: bowveloc=expon(maxbowveloc, 0.1); ...
        s1(bowposition).bow(bowforce, bowveloc);
        s2(bowposition).bow(bowforce, bowveloc);
        ...


    mic.leftout: resonator(1/3, 19/20);
    mic.rightout: resonator(1/3, 1/20);
    ...
```

C.10 Sounds based on instruments with tuned circular components

These sounds have all been transposed down by a factor of 0.3628, equivalent to playing the samples back at 16 KHz instead of 44.1 KHz.

```
////////////////////////////////////////////////////////////////////////////////
// Script name: circles.script                                    tracks 20-25
//
// This set of sounds were all produced with the same basic instrument
// described in section 6.4  consisting of six tuned circular components
// 'one', 'two', 'three', 'four', 'five' and 'six', linked together by
// one-dimensional resonators 'link1-12' which are in turn glued to two
// common one-dimensional resonators called 'resonator1' and 'resonator2'.
// Components 'link1-6' each have one end glued to the centre of each circle
// and the other glued to evenly spaced points on 'resonator1', whilst
// 'link7-12' each have one end glued near the top of each circle and the
// other glued to similarly spaced points on 'resonator2'.
//
// The sounds are created by striking each circular component but the precise
// score algorithm used requires some explanation. A continuous stream of
// impacts are generated as if two objects were bouncing with ever decreasing
// height on the circular components. As soon as their height drops below a
// certain threshold, they are taken up to a greater height and dropped again,
// repeating the whole process. A number of parameters are involved:
//
// x, y:      the x and y position at which the impact will occur on the
//            chosen circular component. These are set at the beginning of
//            each impact.
// force:     the force exerted on the circular component chosen by the
//            bouncing object. This is set at the beginning of each stream
//            of bounces and is multiplied by 'factor' thereafter until
//            the bouncing object has run out of energy at which point it
//            is set to a new larger value again.
// now:       the time at which the next impact is due or the current impact
//            started. Incremented by 'interval' at the end of each impact.
// interval: the time interval to the next impact.
// factor:    the factor by which both 'force' and 'interval' are multiplied
//            after each impact in order to simulate the impacts getting
//            progressively closer together and weaker.
// which:     takes a value 1-6 and specifies which circular component the
//            impact will occur with.
//
// Each of these parameters is actually an array of size 2 since there are
// two identical objects performing according to the same algorithm.
////////////////////////////////////////////////////////////////////////////////


Circle one: C9, 0.5 secs; lockperimeter; ...
Circle two: D#9, 0.5 secs; lockperimeter; ...
Circle three: E9, 0.5 secs; lockperimeter; ...
Circle four: F#9, 0.5 secs; lockperimeter; ...
Circle five: Ab9, 0.5 secs; lockperimeter; ...
Circle six: A9, 0.5 secs; lockperimeter; ...

String link1: 3000 Hz, 0.1 secs; display_at(50,300); ...
String link2: 3000 Hz, 0.1 secs; display_at(100,300); ...
String link3: 3000 Hz, 0.1 secs; display_at(150,300); ...
String link4: 3000 Hz, 0.1 secs; display_at(200,300); ...
String link5: 3000 Hz, 0.1 secs; display_at(250,300); ...
String link6: 3000 Hz, 0.1 secs; display_at(300,300); ...
String link7: 3000 Hz, 0.1 secs; display_at(50,300); ...
String link8: 3000 Hz, 0.1 secs; display_at(100,300); ...
String link9: 3000 Hz, 0.1 secs; display_at(150,300); ...
String link10: 3000 Hz, 0.1 secs; display_at(200,300); ...
String link11: 3000 Hz, 0.1 secs; display_at(250,300); ...
String link12: 3000 Hz, 0.1 secs; display_at(300,300); ...

String resonator1: 200 Hz, 0.5 secs; display_at(50, 400); ...
String resonator2: 200 Hz, 0.5 secs; display_at(50, 400); ...

Glue one(1/2,1/2) to link1(left);
Glue two(1/2,1/2) to link2(left);
Glue three(1/2,1/2) to link3(left);
Glue four(1/2,1/2) to link4(left);
Glue five(1/2,1/2) to link5(left);
Glue six(1/2,1/2) to link6(left);

Glue one(1/2,1/10) to link7(left);
Glue two(1/2,1/10) to link8(left);
```

```
Glue three(1/2,1/10) to link9(left);
Glue four(1/2,1/10) to link10(left);
Glue five(1/2,1/10) to link11(left);
Glue six(1/2,1/10) to link12(left);

Glue resonator1(1/7) to link1(right);
Glue resonator1(2/7) to link2(right);
Glue resonator1(3/7) to link3(right);
Glue resonator1(4/7) to link4(right);
Glue resonator1(5/7) to link5(right);
Glue resonator1(6/7) to link6(right);

Glue resonator2(1/7) to link7(right);
Glue resonator2(2/7) to link8(right);
Glue resonator2(3/7) to link9(right);
Glue resonator2(4/7) to link10(right);
Glue resonator2(5/7) to link11(right);
Glue resonator2(6/7) to link12(right);
```

```
resonator1.setdamping(left, 0.05, 5%);
resonator1.setdamping(0.95, right, 5%);
resonator2.setdamping(left, 0.05, 5%);
resonator2.setdamping(0.95, right, 5%);

Microphone mic1: woodencirclesa, stereo;
Microphone mic2: woodencirclesb, stereo;

Parameter x[2], y[2], force[2], now[2], interval[2], factor[2];
int i, which[2];

for (i=0;i<2;i++)
    {
    now[i]=0 secs;
    force[i]=random(50, 100);
    interval[i]=random(100, 300) msecs;
    factor[i]=1.0-1.0/random(5, 20);
    }

Score 20 secs:
    ControlRate 100:
        Display Time, newline;
        ...
```

```
Before 17 secs:

    At now[0] for 0.2 msecs:
        At start:
            x[0]=random(0.1, 0.9);
            y[0]=random(0.1, 0.9);
            which[0]=randomi(1, 6);
            ...

        At end:
            now[0] += interval[0];
            interval[0] *= factor[0];
            force[0] *= factor[0];
            If interval[0] < 10 msecs:
                interval[0]=random(100, 300) msecs;
                factor[0]=1.0-1.0/random(5, 20);
                force[0]=random(50, 100);
                ...
            ...

        If which[0]==1: one(x[0], y[0]).applyforce(force[0]); ...
        If which[0]==2: two(x[0], y[0]).applyforce(force[0]); ...
        If which[0]==3: three(x[0], y[0]).applyforce(force[0]); ...
        If which[0]==4: four(x[0], y[0]).applyforce(force[0]); ...
        If which[0]==5: five(x[0], y[0]).applyforce(force[0]); ...
        If which[0]==6: six(x[0], y[0]).applyforce(force[0]); ...
        ...
```

```
    At now[1] for 0.2 msecs:
        At start:
            x[1]=random(0.1, 0.9);
            y[1]=random(0.1, 0.9);
            which[1]=randomi(1, 6);
            ...

        At end:
            now[1] += interval[1];
            interval[1] *= factor[1];
            force[1] *= factor[1];
            If interval[1] < 10 msecs:
                interval[1]=random(100, 300) msecs;
                factor[1]=1.0-1.0/random(5, 20);
                force[1]=random(50, 100);
                ...
            ...
```

```
            If which[1]==1: one(x[1], y[1]).applyforce(force[1]); ...
            If which[1]==2: two(x[1], y[1]).applyforce(force[1]); ...
            If which[1]==3: three(x[1], y[1]).applyforce(force[1]); ...
            If which[1]==4: four(x[1], y[1]).applyforce(force[1]); ...
            If which[1]==5: five(x[1], y[1]).applyforce(force[1]); ...
            If which[1]==6: six(x[1], y[1]).applyforce(force[1]); ...
            ...
        ...

mic1.leftout: resonator1(0.1);
mic1.rightout: resonator1(0.9);
mic2.leftout: resonator2(0.1);
mic2.rightout: resonator2(0.9);
...
```

Appendix D

Synthesis model implementation

D.1 Introduction

This appendix describes how the synthesis engine is implemented. Instruments, microphones and cells are all implemented as object classes in C++, and whilst the implementation code itself can be found in appendix G, we concentrate here on the data structures used and on the member functions available within each class. This appendix serves, then, as a specification for the C++ library `libtao.a` which supports the TAO program.

Many of the member functions described in the following sections will already be familiar to the reader since they have a one to one correspondence with TAO script features, whilst others are hidden from the user and are only used by the system itself.

D.2 Internal representation of cells, instruments and microphones

D.2.1 The `Cell` object class

Figure D.1 shows the data structure used to represent a single cell. The structure contains the cell's mass, position, velocity, force and damping coefficient and a set of pointers to its eight neighbours. The `companion` pointer is used when two cells are glued together, in which case each cell's companion pointer is used to point to

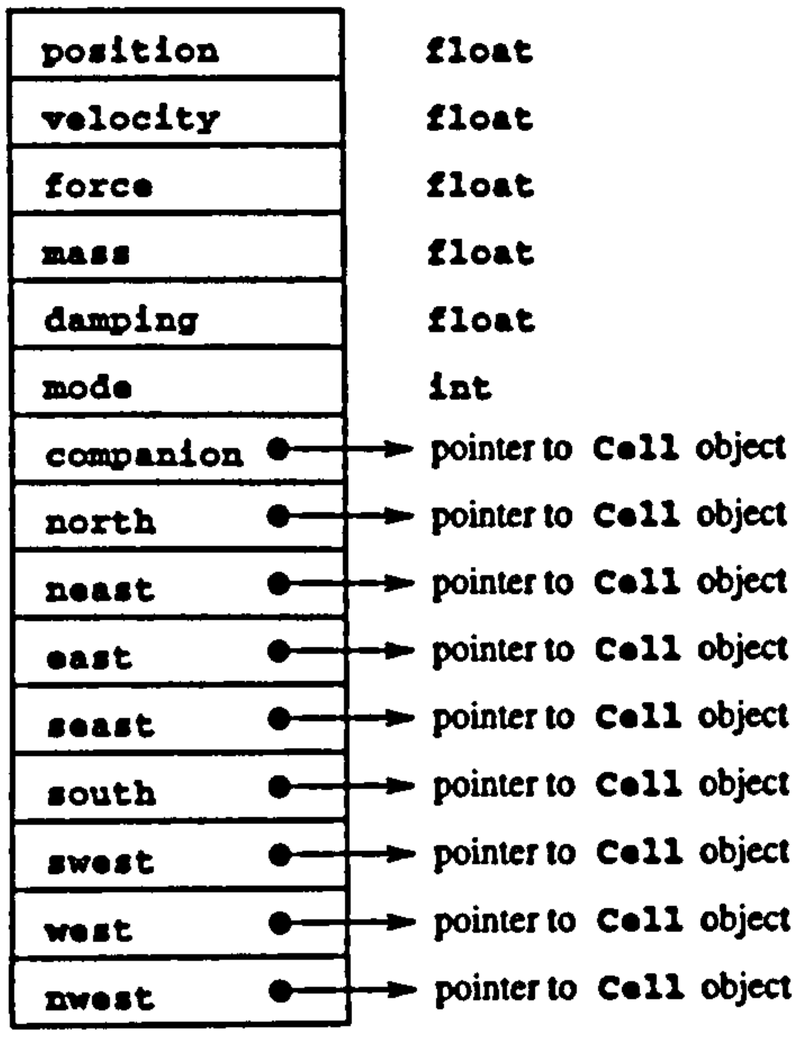


Figure D.1: `Cell` class data structure

the other cell. The `mode` variable holds information about the cell's general status such as whether it is locked, or glued to another cell. It is also used in the bowing model described in appendix F to determine whether the cell is currently sticking to the bow or slipping. Since each cell has its own `mode` variable, any number of bows can act simultaneously on different cells.

D.2.2 Internal representation of the cellular material

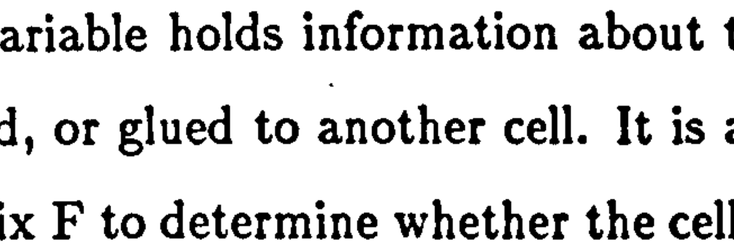


Figure D.2: `Row` data structure

In order to describe the internal representation of a piece of cellular elastic material, another data structure, the `Row` structure, is introduced in figure D.2. This structure is used to represent a single row of cells within a sheet of material or a string. It is

not an object class in itself and as such, has no member functions.

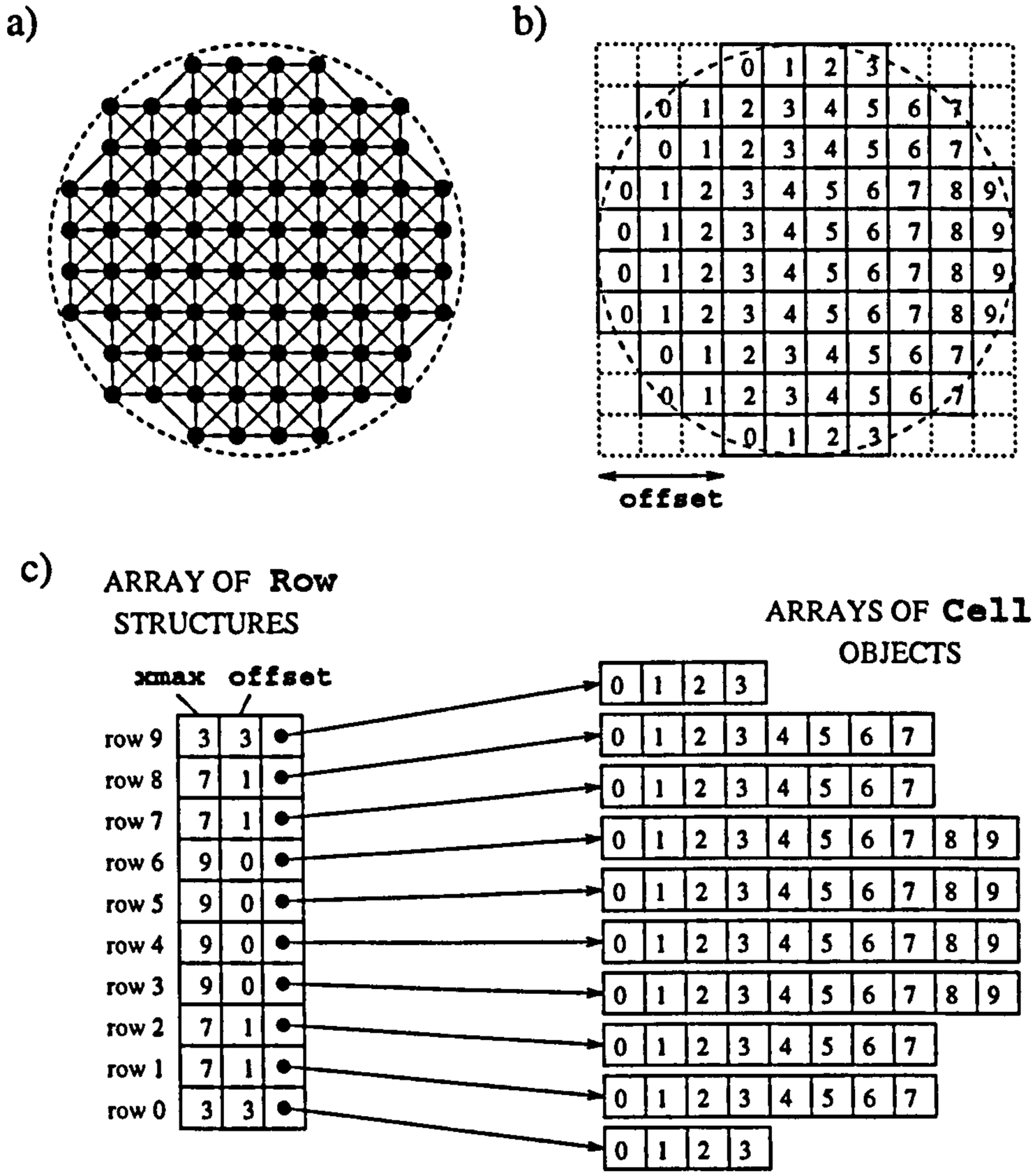


Figure D.3: Internal representation of a piece of material

Figure D.3 uses the example of a circular sheet of material to illustrate the internal representation used. There are various levels of abstraction involved. From the user's point of view the material is continuous in nature, but the discrete nature of the model dictates that the circular shape can only be approximated as in (a).

Moving down to the lowest level of abstraction shown in (c), we see that the material is actually represented as an array of `Row` structures, each representing an individual row of cells. Each `Row` structure contains a pointer to the array of cells and two other pieces of information, `offset` and `xmax`. `xmax` represents the index of the furthest cell to the right in any row and `offset` specifies how many cell positions each row

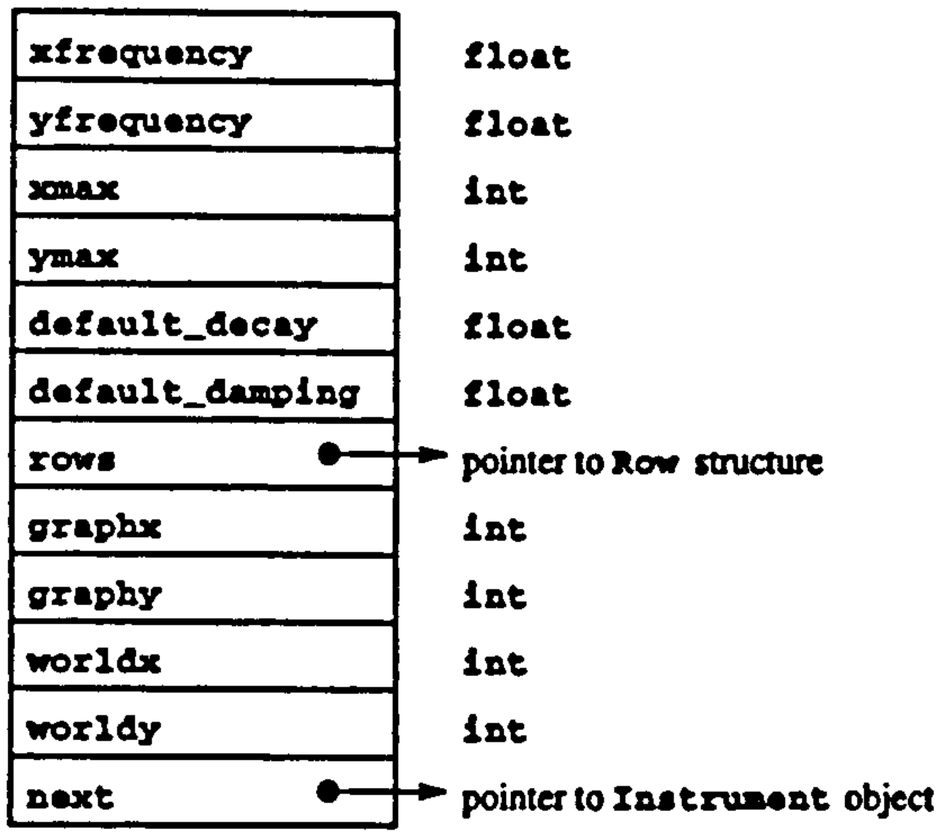


Figure D.4: Instrument class data structure

has to be shifted to the right in order to place it in the correct position relative to all the other rows. (b) serves to clarify the relationship between (a) and (c).

D.2.3 The Instrument object class

Instruments are represented by the data structure shown in figure D.4. The `Cell` and `Row` data structures are used to represent the elastic material itself but the `Instrument` object contains further information, giving a general description of the instrument. The purpose of each variable is described below:

`xfrequency` and `yfrequency` represent the instrument's x and y frequencies as specified in the orchestra declaration. They are measured in Hertz and determine the dimensions of the piece of material created.

`xmax` and `ymax` represent the dimensions of the material, measured in cells. For example, for a piece of material 100 cells wide, and 50 cells high, regardless of shape, `xmax=99` representing the index needed to access the right-most cell in the longest row, and `ymax=49` indicating the index needed to access the top-most row of cells.

`default_decay` represents the initial decay time given to the instrument, as specified in the orchestra declaration, and `default_damping` is the equivalent damping coefficient given to every cell initially in order to achieve this decay time.

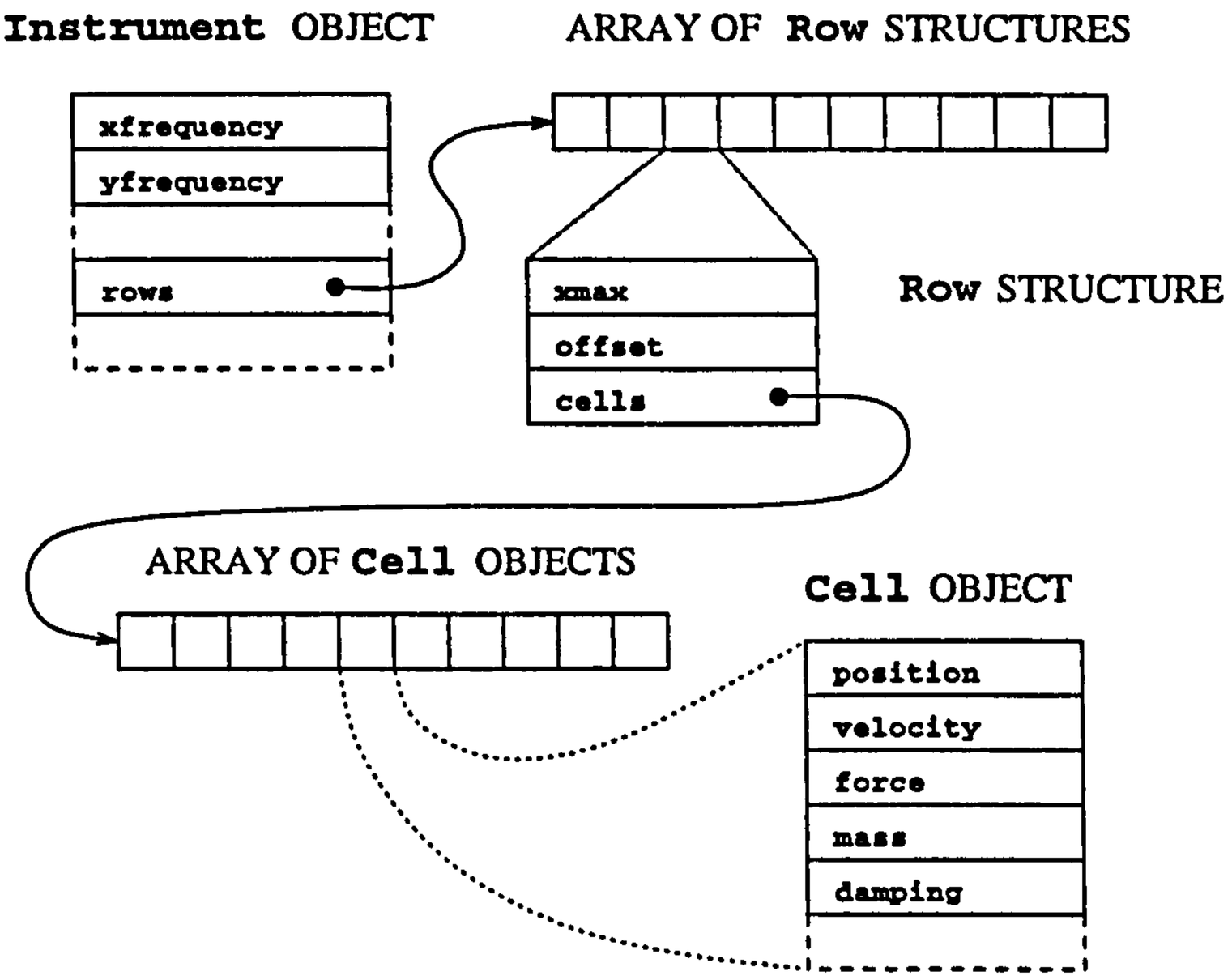


Figure D.5: The hierarchy of data structures used to represent an instrument

`graphx` and `graphy` are used in the present implementation to place the graphical representation of the instrument at a specified location in the graphics window, if graphics mode is on.

`worldx` and `worldy` have a similar function but allow instruments to be placed graphically relative to one another using cells as the coordinate system. In this coordinate system the x axis runs horizontally but the y axis is slightly off the vertical, as if it were running back into the computer screen. This gives a rudimentary sense of depth to the graphical instrument animations as can be seen from most of the instrument examples presented in this thesis. The amount of 'skewing' is determined by the variable `skewfactor`, declared in the file *main.cc*. Whenever two instruments are joined together the second instrument has its `worldx` and `worldy` variables altered so as to place it in the correct position relative to the first. This only works for straightforward joins though e.g. left to right, bottom to top etc. Joining the left side of one instrument to the left side of another might leave them overlapping in the graphics window.

The `next` pointer points at the next instrument in a globally maintained linked list

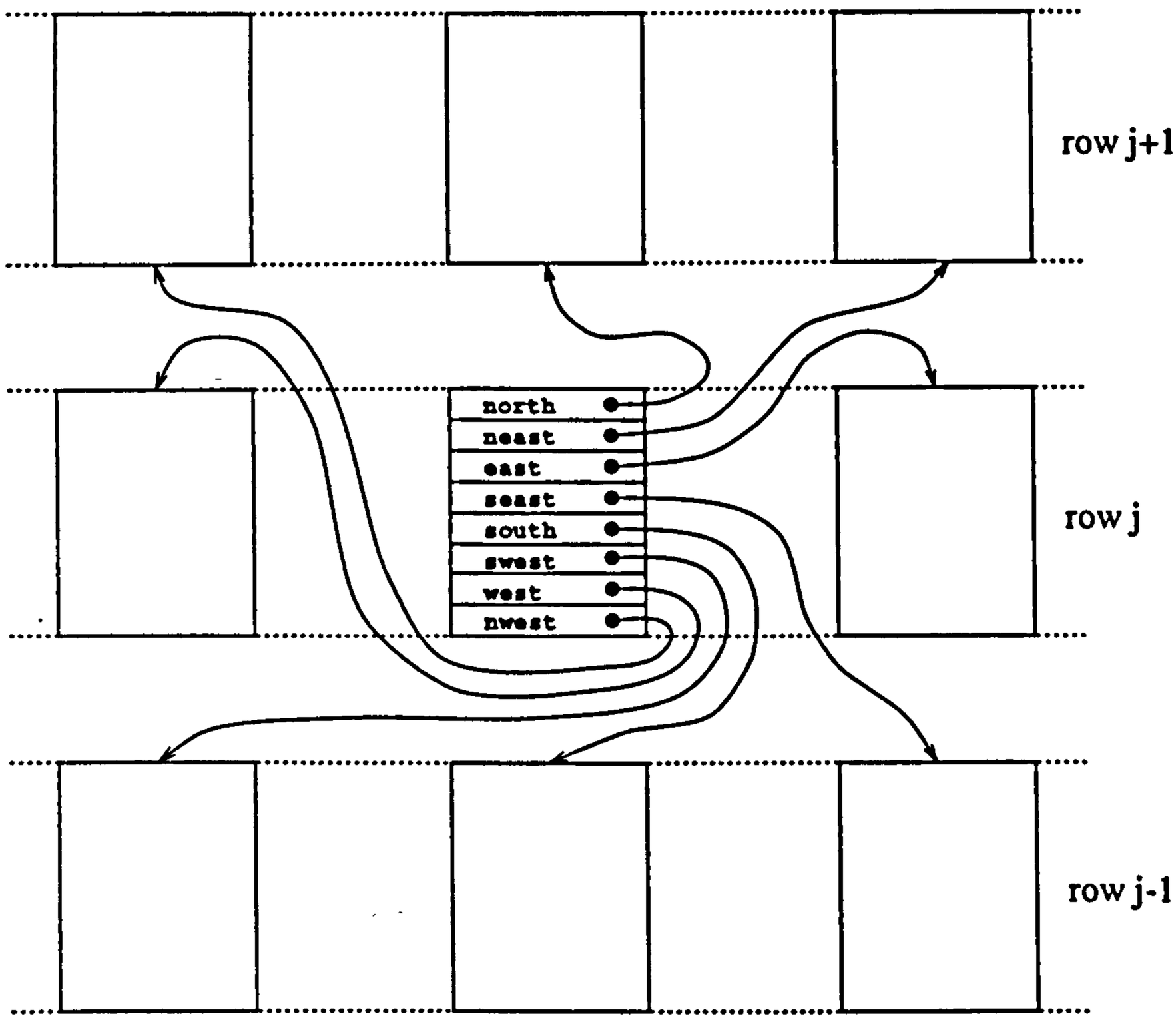


Figure D.6: A cell's pointers to its neighbours

of all the instruments created within a script. This linked list is traversed from head to tail on each time step of the synthesis model, updating each instrument encountered along the way. Note that the order in which instruments are stored in this list, and hence updated, does not matter, even when several instruments are coupled together by glueing and joining.

Figure D.5 shows how all the objects and structures introduced so far are combined in practice, following the hierarchy of structures from the `Instrument` object down through the array of `Row`'s to a single `Row` structure, and then via its associated array of `Cell` objects down to a single `Cell`. This internal representation makes random access to a single cell anywhere within an instrument a simple matter.

In addition to this random access capability, since each cell maintains a set of pointers to its neighbours it is possible to move about the surface of the material in a relative

fashion. This also works when a join between two instruments is encountered, since all that the joining algorithm does is to install new springs between the instruments by redirecting the pointers along the edges of the two pieces of material. Figure D.6 illustrates the arrangement of pointers for a single cell.

D.2.4 The `Microphone` object class

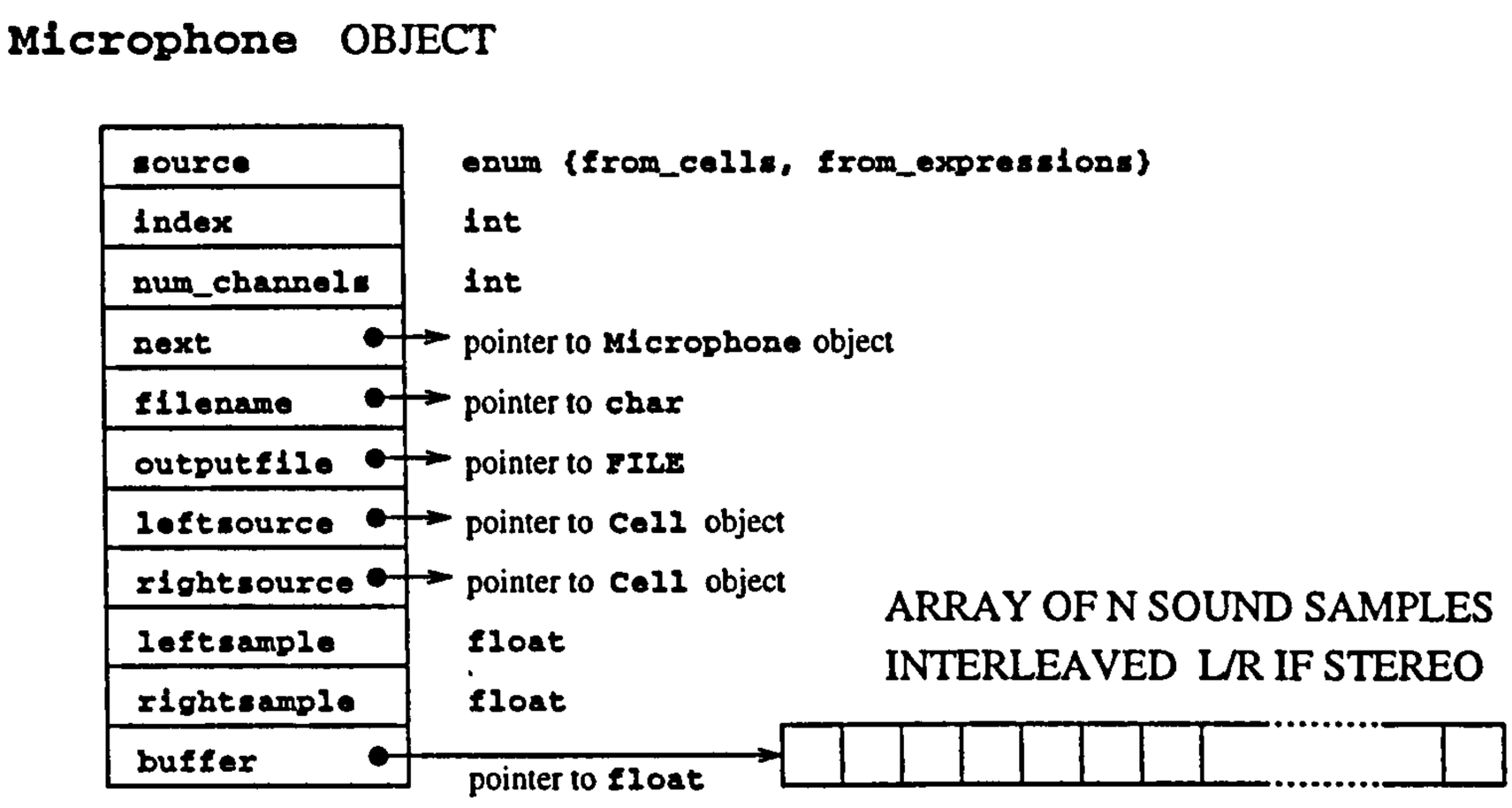


Figure D.7: `Microphone` class data structure

Figure D.7 shows the internal representation of a microphone. In the present implementation only mono and stereo microphones are supported but this situation could be easily changed to produce soundfiles with arbitrary numbers of channels. The data structure would have to be modified in order to provide arbitrary storage space for interleaved sound samples depending on the number of channels.

The `Microphone` object contains a pointer, `filename`, to the name of the file to which the sound samples are to be sent and a pointer, `outfile`, to the file itself. Since there are two different types of microphones, those which have their sound sources specified when the microphone is declared, and those which take their sound samples from arbitrary mathematical expressions in the score, there are four other variables `leftsource`, `rightsource`, `leftsample` and `rightsample`. `leftsource` and `rightsource` are used for microphones with static sound sources and point to the cells from which the output will be taken. `leftsample` and `rightsample` are used to

temporarily store samples generated by the `leftout` and `rightout` messages, ready for writing to the microphone's output buffer pointed to by the variable `buffer`. For monophonic microphones, only `leftsource` or `leftsample` are used. The samples are written to the output buffer until it fills up, at which point the entire contents of the buffer are written to the file pointed to by `outputfile`. Sound samples are interleaved *left, right* if output is in stereo. Once the buffer is emptied, `index` is set to zero again.

D.2.5 Implementation of the Glue facility

Figure D.8 illustrates how `Glue` facility first described in section 4.3 is implemented. In (a) the two cells chosen for glueing are highlighted. Each cell has its `companion` pointer redirected to point at the other cell. In addition, the first cell's `mode` variable is given the `#define`'d value `CELL_MASTER_MODE`, to indicate that it is to act as the *master* cell whilst the second cell's `mode` variable is given the value `CELL_SLAVE_MODE`, making it the *slave* cell.

The forces acting upon each cell in an instrument are calculated by the `Instrument` member function `calculate_my_forces()`. When this function encounters the master cell it treats the slave cell's neighbours as if they belonged to the master cell and calculates the total force acting on the master cell due to both cell's neighbours. When the `Instrument` member function `update_my_position()` subsequently updates all the cell positions and velocities in an instrument and encounters the master cell, the newly calculated position and velocity are simply copied to the slave cell. The slave cell has no part to play in the actual cellular update rules and simply follows the master cell's movements.

D.2.6 Implementation of the Join facility

Joining causes two nominated sides of two instruments to be 'sewn' together with newly created springs. The information required by the join algorithm is explained in detail in sections 5.4.7 and B.9 but to recap briefly, four coordinates x_1, x_2, y_1 and y_2 are given. Either the two x coordinates or the two y coordinates are used to specify the sides of the instruments to be joined in which case the remaining coordinates are used to specify a centre line which has the effect of lining up two points on the respective edges of the two instruments.

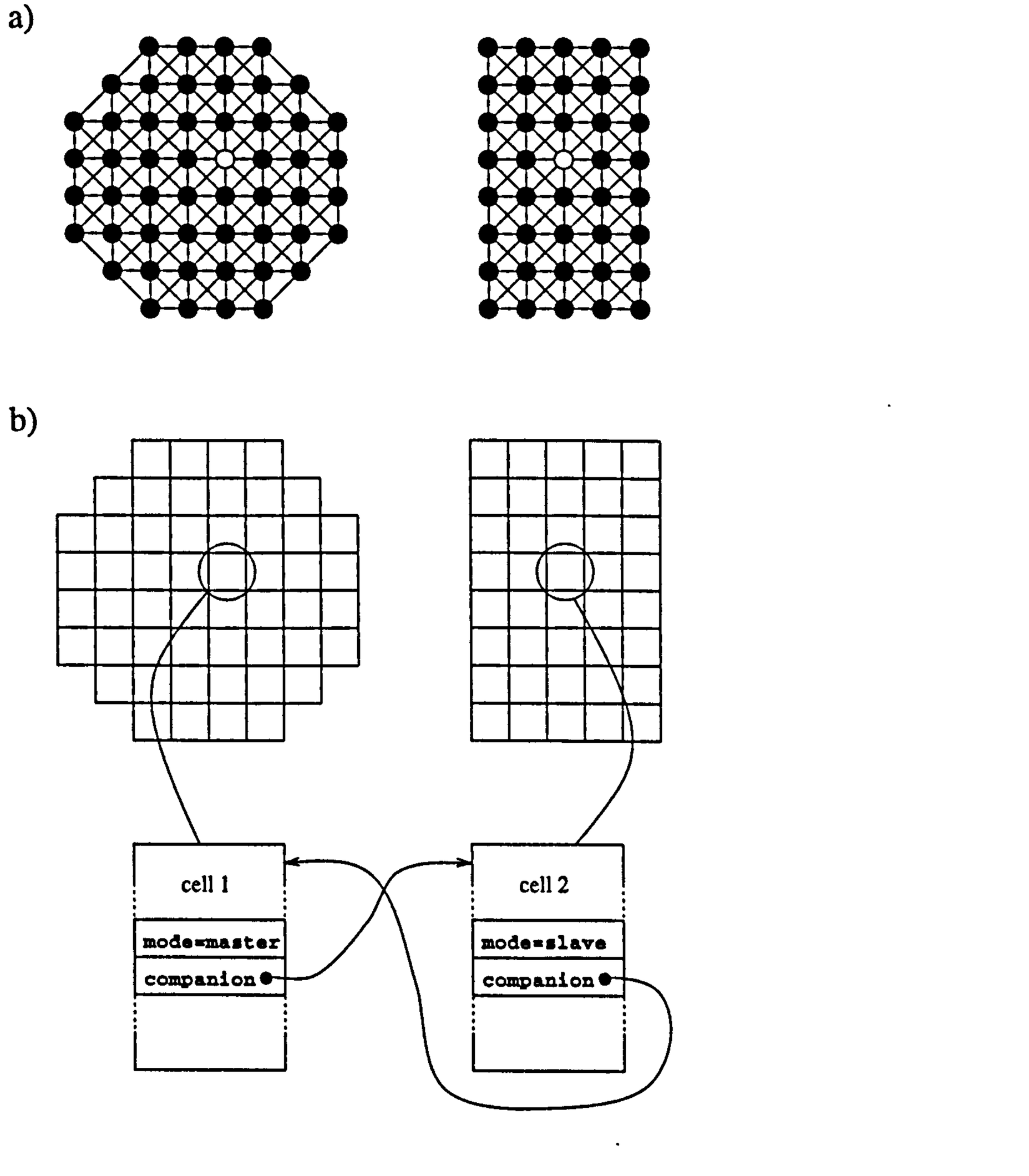


Figure D.8: Implementation of glueing

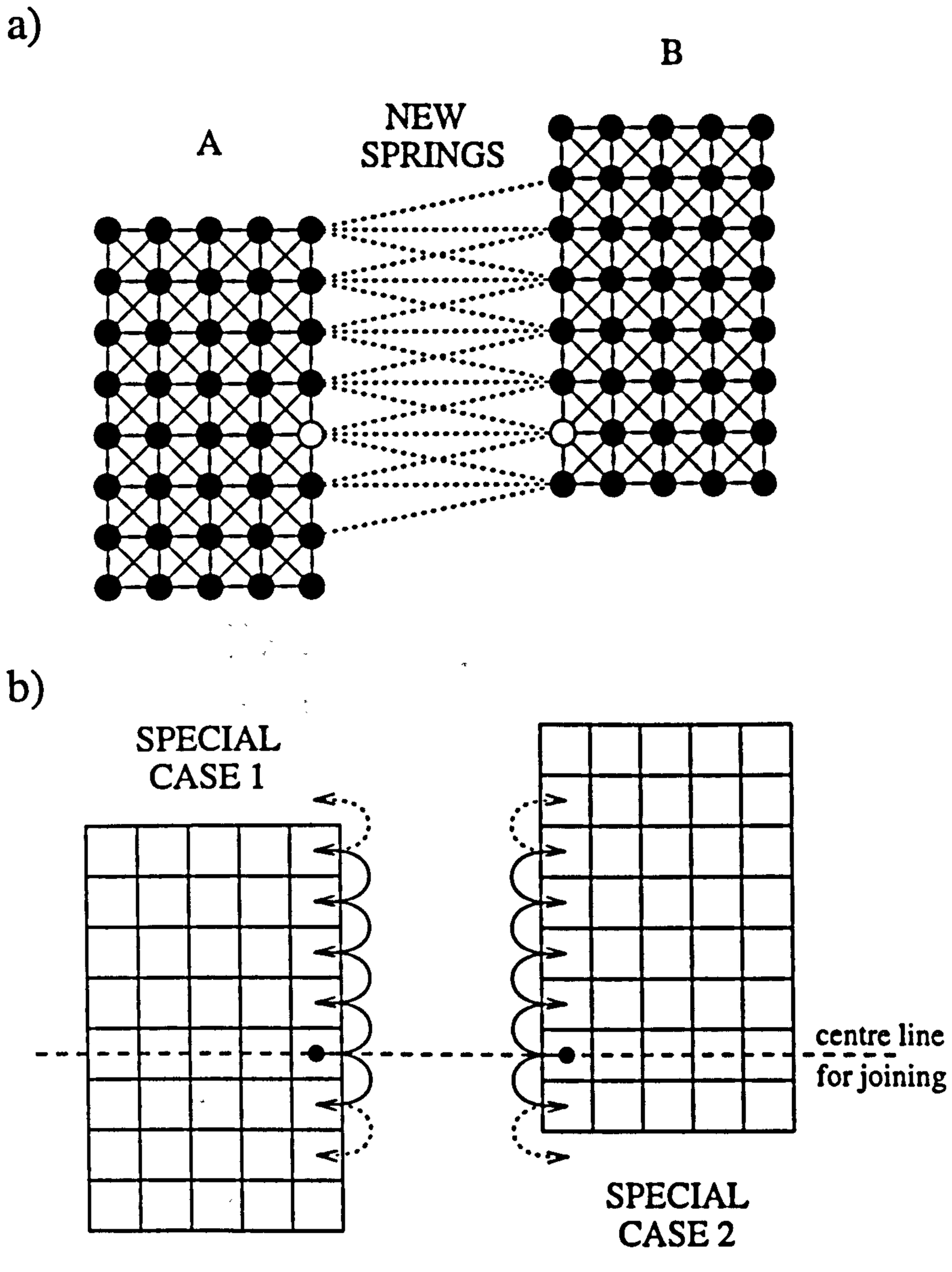


Figure D.9: Joining two pieces of material together

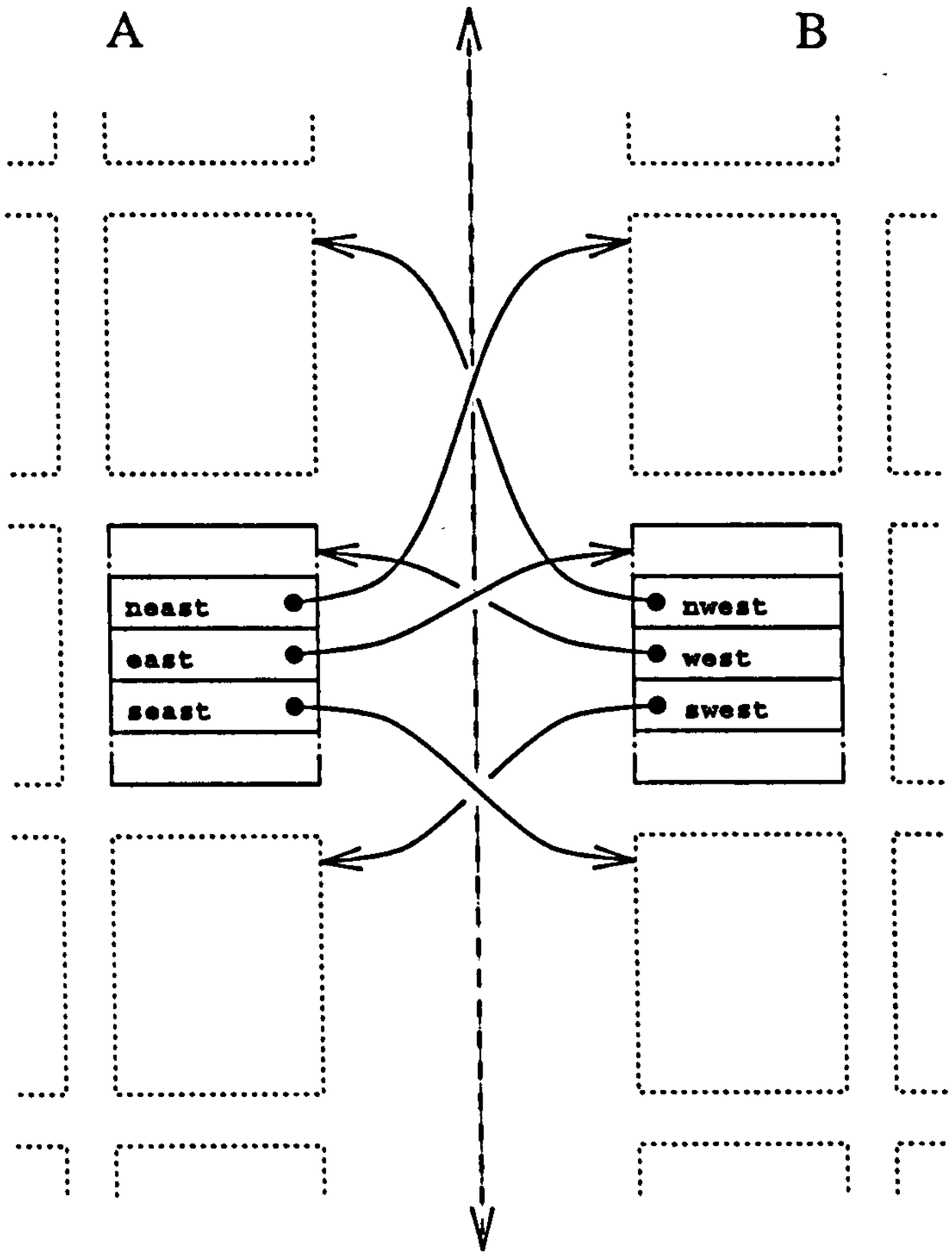


Figure D.10: The general case of joining two facing cells anywhere along the seam

Figure D.9 illustrates how two instruments, A and B, are joined in practice. In (a) we see the individual cells and existing springs, and the two cells specified by the centre line are highlighted. The process of joining starts at these cells and migrates in one direction first, sewing the cells together until a boundary is reached on one of the instruments. This process then starts from the centre line again and migrates in the opposite direction until another boundary is encountered as in (b).

Figure D.10 shows what happens at the microscopic scale. As the process of joining migrates up or down the material, each pair of facing cells have their (previously null) neighbour pointers redirected. This process continues until the migration can

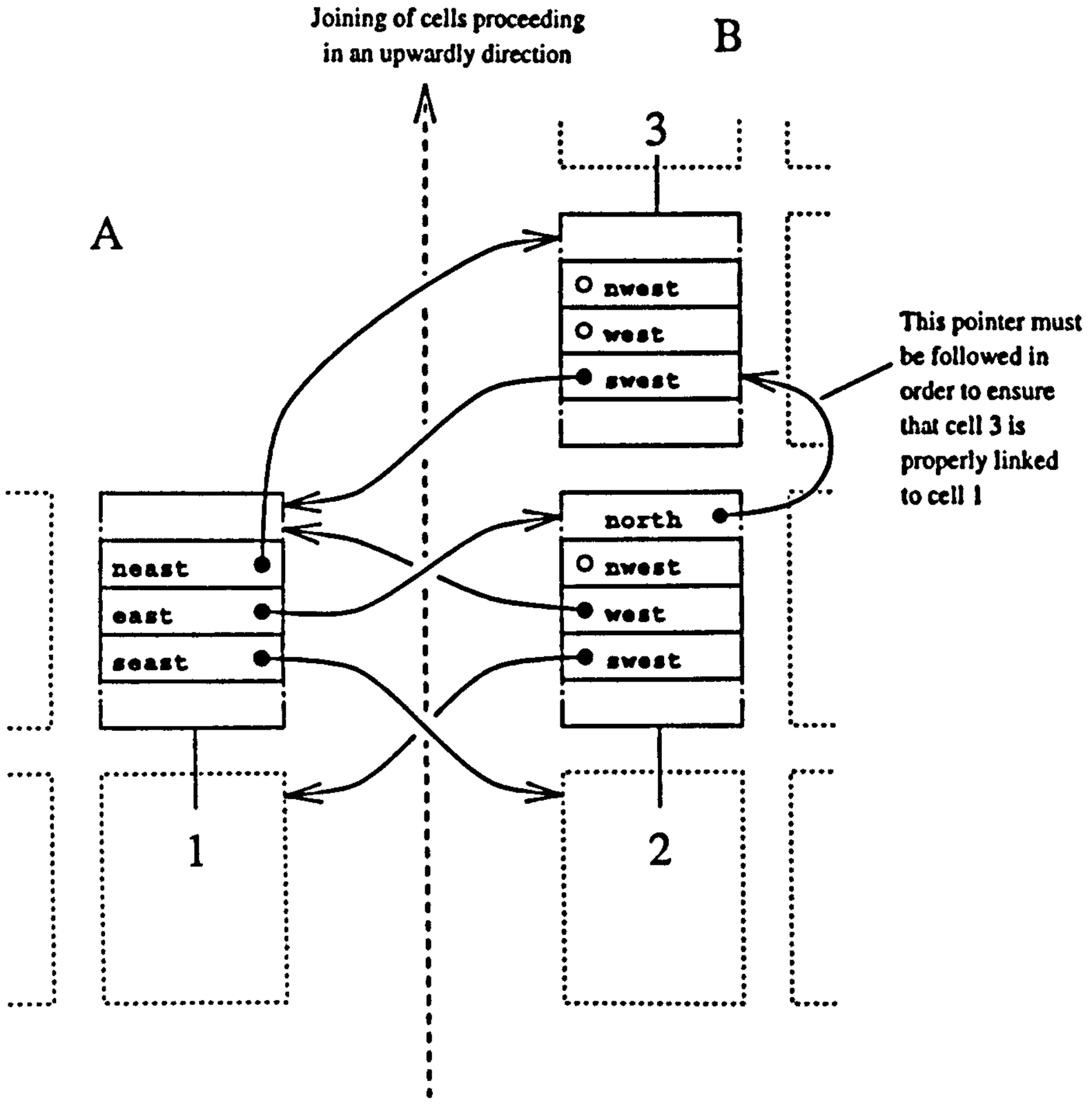


Figure D.11: Special case 1 - joining cells at the northern boundary

proceed no further. At either end of the join where a boundary is encountered on one of the instruments a special case occurs. Returning for a moment to figure D.9a we can see why. At either end of the join there is one extra spring which extends slightly beyond the boundary which caused the joining process to stop. Since a spring is implemented as two reciprocal pointers between two cells, it is important to ensure that both pointers are properly redirected. The two special cases, occurring at the northern boundary of instrument A and at the southern boundary of instrument B respectively, are described below.

Figure D.11 shows, in detail, what happens at the top of the join depicted in figure D.9. Cell 1 lies at the northern edge of instrument A. Cell 3 lies just beyond this

boundary on instrument B and this cell must have its **swest** pointer redirected to cell 1 to match up with the corresponding pointer coming from that cell. This is achieved indirectly via cell 2's **north** pointer.

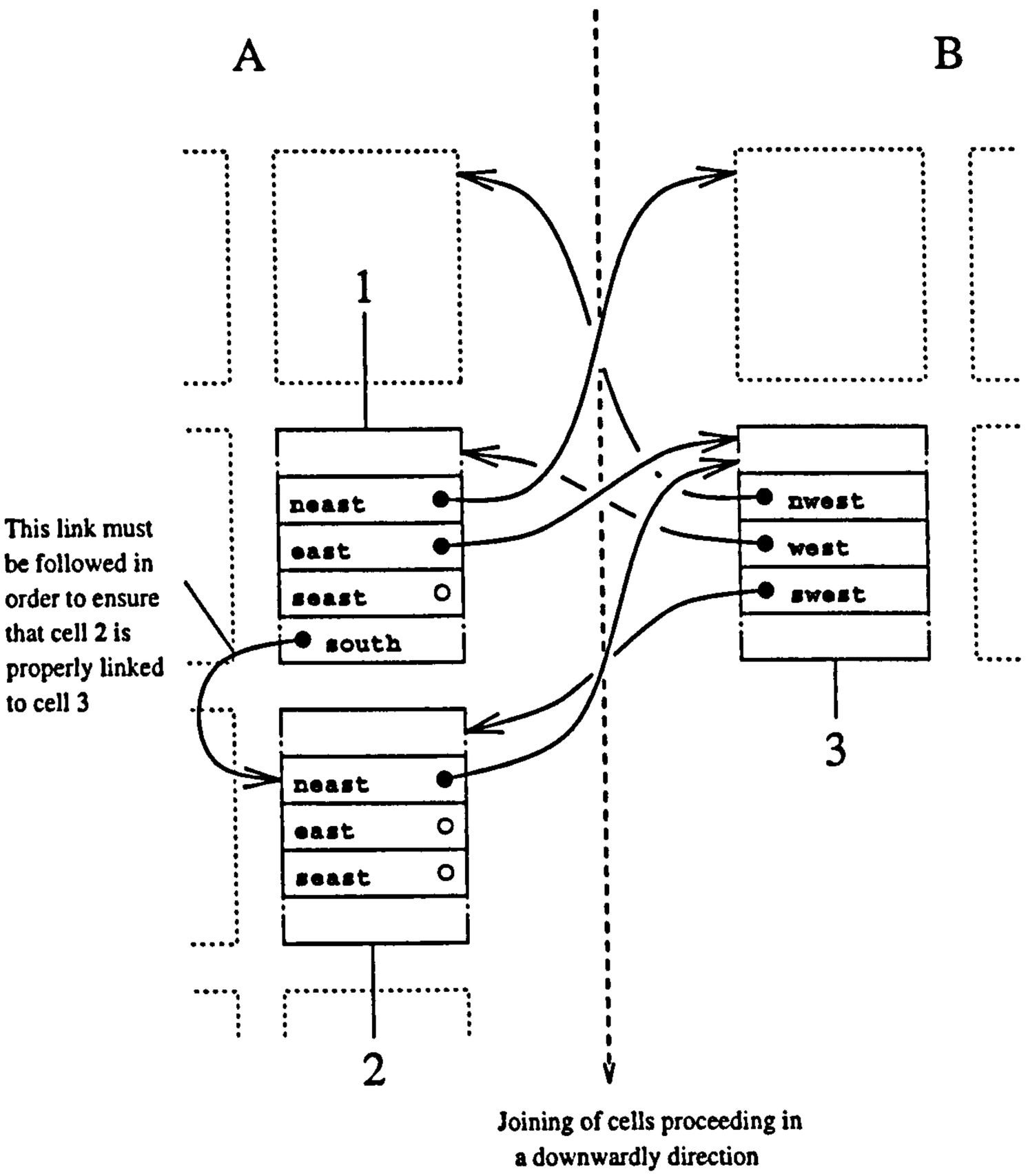


Figure D.12: Special case 2 - joining cells at the southern boundary

Similarly, for the second special case shown in figure D.12, cell 2 lies just beyond the southern boundary of instrument B and this cell must have its **neast** pointer redirected to cell 3. Once again this is achieved indirectly via cell 1's **south** pointer.

It is vital to ensure that every pointer which a cell has to a neighbouring cell is matched by a reciprocal pointer coming from that neighbouring cell. Failing to do this leads to a kind of one-way spring which continually introduces energy into the instrument, leading to exponential growth in the amplitude of vibrations.

D.3 List of functions

We now turn to a comprehensive list of the functions and operators provided by the library `libtao.a` which the TAO computer music program is based upon. Each function synopsis specifies what part the function has to play and the file in which it can be found. If a more detailed account of any function is required, the reader can refer to appendix G which contains a complete listing of the implementation code. It should be possible after reading this appendix to write a C++ program utilising the various TAO objects and functions by compiling and linking it with the library `libtao.a`.

The functions and operators are divided into the following categories:

- Functions and operators for interaction with cells.
- Functions used in the creation of instruments.
- Functions and operators for accessing points on an instrument.
- Functions used in locking and damping parts of an instrument.
- Functions for glueing and joining pieces of material.
- Graphics related functions.
- Functions used in the creation of microphones.
- Functions used to send sound samples to a microphone.
- System functions for animating instruments.
- System functions for updating microphones.
- System functions which drive the whole synthesis engine and the graphics.
- Other global functions.

D.3.1 Functions and operators for interaction with cells

`applyforce(float F)`
Cell.cc
Applies a force `F` to a cell and also applies smaller sympathetic forces to the cells immediate neighbours. This prevents an irritating mode of vibration which sometimes occurs when a sharp impulse is applied to a single cell resulting in alternate cells vibrating up and down 180 degrees out of phase. This can be heard as a distinctive high pitched whistle in the sounds produced and occurs because the material has no stiffness.

`bow(float f_bow, float v_bow)`
Cell.cc
Simulates the interaction of a virtual bow with a cell, based on frictional sticking and slipping. The algorithm used is explained in appendix F. `f_bow` is the normal force exerted by the bow on the instrument and `v_bow` is its velocity.

`lock()`
Cell.h
Forces a cell to remain fixed in the position it is in when the function is called. It does so by changing the cell's `mode` variable.

operator `float()`
Cell.h
When an object of class `Cell` appears in an expression expecting a numerical value, this operator automatically returns the value of the cell's `position` variable. This is most often used in microphone output statements where we can just specify the points on the various instruments we wish the output samples to be taken from rather than having to type `instr1(x,y).position` each time.

D.3.2 Functions used in the creation of instruments

`String(float freq, float decay)`
`Circle(float freq, float decay)`
`Rectangle(float xfreq, float yfreq, float decay)`
`Triangle(float xfreq, float yfreq, float decay)`
`Ellipse(float xfreq, float yfreq, float decay)`

String.cc, Circle.cc, Rectangle.cc, Triangle.cc, Ellipse.cc
Classes `String`, `Circle`, `Rectangle`, `Triangle` and `Ellipse` are derived classes of base class `Instrument`. Each class has its own constructor function which, because of the inheritance mechanism provided by C++, invokes the `Instrument` constructor function first. This function creates a basic skeleton of a data structure which contains information common to all instruments. The more specific constructor functions provided by each derived class know how to create pieces of material of the appropriate shapes. Once the rows of cells have been created they are linked together with springs by the `Instrument` member function `link_cells()`. The cells are all initialised by the `Instrument` member function `initialise_cells()`. Finally each newly created instrument is placed at the end of a global linked list using the member function `add_to_global_list()`. These functions are listed below.

`Instrument(float xfreq, float yfreq, float decay)`
Instrument.cc
Creates an `Instrument` object which holds all the information common to all instrument shapes. This function has to rely on the `String`, `Circle`, `Rectangle`, `Triangle` and `Ellipse` constructor functions to actually create a piece of material of the correct shape and size. `xfreq`, `yfreq` and `decay` represent the frequency, in Hertz, of the instrument in the x and y directions and an initial uniform decay time measured in seconds. For a string `yfreq=0`.

`hertz2cells(float freq)`
Instrument.h
Static member function used to convert the argument `freq` measured in hertz into a numerical value representing the number of cells required to achieve that frequency.

`decay2damping(float decay)`
Instrument.h
Static member function used to convert the argument `decay` measured in seconds into a numerical value which, when written to a cell's `damping` variable, causes the cell's vibrations to die away with the correct decay time. Note that if the surrounding cells have totally different damping values, then the decay time will not be as expected.

`initialise_cells()`
Instrument.cc
Sets the velocities, positions and forces of all cells to zero, sets all neighbour pointers to NULL and sets the mode of each cell to a default value. Since the frequency of an instrument is given as a real number measured in Hertz, but the material is discrete in nature, once the width and height of the instrument, measured in numbers of cells, have been determined, the masses of all the cells have to be slightly adjusted from the default mass in order to adjust the frequency to the originally specified value. The compensation is calculated such that the instrument ends up with the correct x frequency since a string which is out of tune is more of a problem than an inharmonic two-dimensional instrument with a slight error in the y frequency.

`link_cells()`
Instrument.cc
Works its way through the rows of cells in a newly created piece of material linking neighbouring cells together with springs. Works for any shape of material.

`add_to_global_list()`
Instrument.h
Adds a newly created instrument to the global linked list maintained by TAO.

D.3.3 Functions and operators for accessing points on an instrument

operator: `(float x, float y)`
Instrument.cc
When placed immediately after an identifier of class `Instrument` this operator selects and returns a reference to the cell at position `(x,y)`. For an explanation of the coordinate system used see section 5.4.8. Also has the side effect of placing a blue marker on the graphics screen, if the graphics are turned on, to mark the cell accessed.

operator: `(float x)`
Instrument.cc
Exactly the same as operator `(float x, float y)` but intended for one dimensional instruments where only the `x` coordinate need be specified.

`at(float x, float y)`
Instrument.cc
Exactly the same as for operator `(float x, float y)` in that it selects and returns a reference to the cell specified by the instrument coordinates `x` and `y` except it doesn't affect the graphics display.

D.3.4 Functions used in locking and damping parts of an instrument

`setdamping(float x1, float x2, float y1, float y2, float damping)`
Instrument.cc
Sets the damping value of each cell to the value `damping` over the region specified. See section 5.4.4 for an explanation of the coordinate system used. Returns a reference to the cell for whom the function was invoked via the C++ special variable `this`. This mechanism allows messages to be strung together seperated by periods.

`setdamping(float left, float right, float damping)`
Instrument.cc
Version for one dimensional instruments where only the left and right endpoints of the damped region need be specified. Coordinates are still normalised to be between zero and one. Returns a reference to the cell for whom the function was invoked.

`setdamping(float position, float damping)`
Instrument.cc
Version for one dimensional instruments which only allows the damping to be set at a single point, not over a region. Returns a reference to the cell for whom the function was invoked.

`setdamping(float damping)`
Instrument.cc
Sets the damping value of every cell within an instrument to `damping` Returns a reference to the cell for whom the function was invoked.

`resetdamping(float x1, float x2, float y1, float y2)`
`resetdamping(float left, float right)`
`resetdamping(float position)`
`resetdamping()`

Instrument.cc
Equivalent to the setdamping family of functions above, but reset the damping value back to the value `default_damping` which was set when the instrument was created. All four functions return a reference to the cell for whom they were invoked.

`setdamping(float x1, float x2, float y1, float y2, float decay)`
`setdamping(float left, float right, float decay)`
`setdamping(float decay)`

Instrument.cc
Equivalent to the setdamping family of functions above, but set the damping value in terms of a decay time measured in seconds. The instrument will have the correct decay time if all cells are damped with these functions, but if a smaller, local region is damped, the effect is not as predictable. All four functions return a reference to the cell for whom the function was invoked.

`resetdecay(float x1, float x2, float y1, float y2)`
`resetdecay(float left, float right)`
`resetdecay()`

Instrument.cc
Equivalent to the resetdamping family of functions above, included only for consistency and compatibility. All return a reference to the cell for whom the function was invoked.

`lock(float x, float y)`
Instrument.cc
Locks a single cell at location `(x,y)` on an instrument and returns a reference to the cell for whom the function was invoked.

`lock(float x1, float x2, float y1, float y2)`
Instrument.cc
Locks a rectangular region. Similar to `setdamping(x1, x2, y1, y2, ...)` in the coordinate system used to specify the region. See section 5.4.4 for an explanation.

`lockleft()`, `lockright()`, `locktop()`, `lockbottom()` `lockcorners()`, `lockperimeter()`, `lockends()`
Instrument.cc
`lockleft`, `lockright`, `locktop` and `lockbottom` are all straightforward for a rectangular sheet, locking whole sides at a time. For other shapes of material only the furthest cells west, east, north or south are locked. `lockcorners()` only makes sense for rectangular and triangular instruments and `lockends()` is designed for use with strings. All return a reference to the cell for whom they were invoked.

D.3.5 Functions for glueing and joining pieces of material

`glue(Instrument &i1,float x1,float y1,Instrument &i2,float x2,float y2)`
`glue(Instrument &i1,float x1,float y1,Instrument &i2,float x2)`
`glue(Instrument &i1,float x1,Instrument &i2,float x2,float y2)`
`glue(Instrument &i1,float x1,Instrument &i2,float x2)`

Instrument.cc
Given two instruments and sets of coordinates specifying two cells, glues those two cells *and* their corresponding neighbours together. Glueing single cells together sometimes leads to unstable properties since the material has no stiffness as such. All of these functions return a reference to the cell for whom they are invoked.

`glue_cells(Cell *c1, Cell *c2)`
Instrument.cc
Given pointers to two cells, glues the cells together.

join(Instrument &i1, float x1, float y1, Instrument &i2, float x2, float y2)

Instrument.cc

Joins two pieces of material with straight edges by effectively installing a new set of springs to sew the two instruments together so that they act as one. There are eight different cases:

if `x1=0`: if `x2=0`: join the left of `i1` to the left of `i2`
if `x2=1`: join the left of `i1` to the right of `i2`
if `x1=1`: if `x2=0`: join the right of `i1` to the left of `i2`
if `x2=1`: join the right of `i1` to the right of `i2`
if `y1=0`: if `y2=0`: join the bottom of `i1` to the bottom of `i2`
if `y2=1`: join the bottom of `i1` to the top of `i2`
if `y1=1`: if `y2=0`: join the top of `i1` to the bottom of `i2`
if `y2=1`: join the top of `i1` to the top of `i2`

If the join runs north to south then `y1` and `y2` are used to specify two points on the respective edges to be joined which will be lined up. In effect they define a centre line which specifies where the joining is to begin. If the join runs east to west then `x1` and `x2` are used to specify the centre line for joining. `x1`, `x2`, `y1` and `y2` are all specified as instrument coordinates. For a more detailed explanation of the join parameters see sections 5.4.7 and B.9 Note that `i1` and `i2` can refer to the same instrument making it possible to construct cylindrical and toroidal instruments from a rectangular sheet. This function relies on the functions described below.

`join_left_to_left(Cell &cell1, Cell &cell2)`
`join_left_to_right(Cell &cell1, Cell &cell2)`
`join_right_to_left(Cell &cell1, Cell &cell2)`
`join_right_to_right(Cell &cell1, Cell &cell2)`
`join_bottom_to_bottom(Cell &cell1, Cell &cell2)`
`join_bottom_to_top(Cell &cell1, Cell &cell2)`
`join_top_to_bottom(Cell &cell1, Cell &cell2)`
`join_top_to_top(Cell &cell1, Cell &cell2)`

Instrument.cc
All of these functions join two pieces of material with straight edges by installing a new set of springs, effectively 'sewing' them together so that they act as one. Joining starts at the two cells specified and migrates along the edges of the two pieces of material in one direction until a boundary is reached. The joining process then recommences from the starting cells, and migrates in the opposite direction. For a more detailed explanation of the algorithm used, see section D.2.6.

D.3.6 Graphics related functions

`display()`
Instrument.cc
Displays the instrument in the graphics window at a position determined by the `Instrument` member variables `worldx`, `worldy`, `graphx` & `graphy`, and the global variables `winoriginx` & `winoriginy`. Uses external functions from SGI graphics library `gl_s`: `bgnline()`, `endline()`, `v2s()`: begin line and end line and vertex functions. see `<gl.h>` `color()`: sets graphics colour. `BLACK`, `WHITE`, `CYAN`, `MAGENTA`, `BLUE`, `GREEN` `YELLOW` and `RED` allowed.

`display_at()`
Instrument.h
Sets the screen x and y coordinates at which the bottom left of an instrument will be displayed in the graphics window. Note that the exact position of the instrument is also affected by the global variables `winoriginx`, `winoriginy` and the `Instrument` member variables `worldx` and `worldy`.

`place_at()`
Instrument.h
Sets the world x and y coordinates of the instrument. World coordinates are measured in cells and this function is used by the system when two instruments are joined in order to place the second in the correct world position relative to the first. Note that the two instruments joined must have the same values for `graphx` and `graphy` or they won't appear in the correct relative positions in the graphics window.

`screenx(float x, float y)`
Instrument.cc
Returns the current screen x coordinate of the cell specified by the instrument coordinates `x` and `y`.

`screeny(float x, float y)`
Instrument.cc
Returns the current screen y coordinate of the cell specified by the instrument coordinates `x` and `y`.

`label(float x, float y, int xoffset, int yoffset, char *caption, int colour)`
Instrument.cc
Displays the text string `caption` in the graphics window at a position determined by the instrument coordinates `x` and `y`. If the cell specified is displayed at screen coordinates (*x*, *y*) then the caption will be placed at (*x*+`xoffset`, *y*+`yoffset`). The caption is displayed in the specified colour where `colour` is one of `RED`, `GREEN`, `BLUE`, `YELLOW`, `MAGENTA`, `CYAN` or `BLACK`, which are `#define`'d constants from header file `<gl.h>`.

`label(float x, int xoffset, int yoffset, char *caption, int colour)`
Instrument.cc
Version of the `label` function given above for use with strings.

`graphics_init()`
main.cc
Initialises graphics system, opens a window entitled 'TAO graphical output'. Sets doublebuffer mode for animation and clears the screen to white.

update_graphics()
main.cc
This function deals with everything associated with the graphics window, save actually drawing the instruments. There are a number of mouse functions provided. Holding the **left mouse** button down and moving the mouse in the graphics window causes the whole graphics image be **dragged** about. Holding the **left mouse** button down and pressing the **middle mouse** button causes the global variable `graphics_update_step` to be multiplied by a factor of five. The graphics window is updated on every `graphics_update_step`'th time step of the synthesis engine. If `graphics_update_step=500` then it becomes 1 again. Holding the **left mouse** button down and pressing the **right mouse** button causes `graphics_update_step` to be divided by a factor of 5. If it is already 1 then the animation is frozen until the **left mouse** button is held and the **right mouse** button is pressed again. Elapsed time in seconds since beginning of performance is displayed at bottom left of the graphics window. External functions used include `getsize()`, `origin()`, `cmov2i()`, `color()`, `charstr()`, `getbutton()` all of which are provided in the SGI graphics library and declared in `<gl.h>`. For more information see the appropriate IRIX 5.3 manual pages.

D.3.7 Functions used in the creation of microphones

Microphone(const char *soundfilename, int channels)
Microphone.cc
Creates a microphone object whose sound samples will be sent to a file called `/var/tmp/<name>.tao`. The microphone writes `channels` channels of output (1 or 2 in the present implementation). No decision is made at declaration time about the actual sources for the sound samples. This is left to be determined by the member functions `leftout()` & `rightout()` described in file `Microphone.h`, and `update()`, described below. In practice `leftout()` and `rightout()` are usually invoked within the score part of a TAO script.

`Microphone(const char *sfname, Cell &l, Cell &r)`
Microphone.cc
Similar to the above function except that sound sources are given at declaration time in the form of two references to cells `l` amd `r`. This automatically determines that `num_channels=2`.

`Microphone(const char *sfname, Cell &c)`
Microphone.cc
Mono version of constructor function described above. `num_channels=1`

`add_to_global_list()`
Microphone.cc
Adds a newly created microphone to the global list maintained by TAO. Note that microphones and instruments are stored in separate linked lists.

`setleft(Cell &l)` `setright(Cell &l)`
Microphone.h
These functions are used for microphones with static sound sources and set the left and right sources respectively to the cells referenced by `l` and `r`.

D.3.8 Functions used to send sound samples to a microphone

`leftout(float value)`
`rightout(float value)`
Microphone.h
These functions simply write the numerical values specified into the `Microphone` member variables `leftsample` and `rightsample` respectively, ready for writing to the microphone's output buffer.

`output(float value)`
Microphone.h
This function writes the numerical value specified into the `Microphone` member variable `leftsample`, ready for writing to the microphone's output buffer.

D.3.9 System functions for animating instruments

`calculate_my_forces()`
Instrument.cc
Starts at the bottom left of an instrument and works its way from left to right along each row of cells and then up each row until the top right of the instrument is reached. Calculates the total force acting on each cell due to the springs connecting it to its neighbours. If the cell is a *master* cell it treats the *slave* cell's neighbours as its own in order to calculate the combined force acting on both cells. If it is a slave cell then no calculations are made.

`update_my_position()`
Instrument.cc
Starts at the bottom left of an instrument and works its way from left to right along each row of cells and then up each row until the top right of the instrument is reached. The force acting upon each cell is used to calculate the cell's acceleration, new velocity and new position. Also multiplies the new velocity by the cell's damping value (between 0 and 1). This value is converted from the percentage value given in a TAO script. 100% gives a damping value of zero and 0% gives a value of one. If the cell is a *master* cell, then the newly calculated force, velocity and position are copied to the *slave* cell.

D.3.10 System functions for updating microphones

`update()`
Microphone.cc
Causes sound samples to be written to the microphone's sample buffer. If the buffer is full then its entire contents are written to the output file stream `outputfile`, and `index` is reset to zero. Otherwise `index` is incremented by `num_channels`. If the `Microphone` member variable `source` has the value `from_cells`, then the samples are taken directly from the cells pointed to by `leftsource` and `rightsource` or just `leftsource` for a mono microphone. However, if `source` has the value `from_expressions`, then the samples are taken from the values of `leftsample` and `rightsample` or just `leftsample` for a mono microphone.

`update_all()`
Microphone.cc
Starts at the head of the linked list of microphones and updates each one in turn by invoking the member function `update()`.

D.3.11 System functions which drive the whole synthesis engine and the graphics

`calculate_forces()`
`update_positions()`
`display_all()`

Instrument.cc
Each one of these functions scans the linked list of instruments and invokes the appropriate member function for each instrument in the list. For example `calculate_forces` causes `calculate_my_forces` to be invoked for each instrument etc.

`main()`
main.cc
The user compiles a TAO script called `example.script` by typing: `tao example` which causes the script to be translated into an intermediate form stored in the file `tao_scriptfile`. This is `#include`'d into the `main` function, and once further processed by the C++ preprocessor, becomes a fragment of compilable C++ code. The user's instrument, microphone and parameter declarations translate directly into C++ variable declarations, and other TAO language features such as the score control structures, screen output, mathematical expressions etc. translate into equivalent C++ language features. Once the C++ preprocessor has finished its translation, the file `main.cc` is compiled, producing an executable with the same name as the script but with a `.exe` suffix. Following the example above, the executable produced would be called `example.exe`.

D.3.12 Other global functions

`randomi(int low, int high)`
main.cc
Returns a random integer between low and high inclusive.

`random(float low, float high)`
main.cc
Returns a random floating point number between low and high inclusive.

`pitch(float value)`
main.cc
Takes a decimal value of the form <octave>.<semitone> and returns a frequency in Hertz. For example: `pitch(8.00)` ⇒ 261.6 Hz or middle C. `pitch(8.01)` ⇒ the frequency of C sharp above middle C. `pitch(8.09)` ⇒ 440 Hz or A above middle C. `pitch(6.03)` ⇒ the frequency of E♭ in the second octave below middle C. `pitch(8.06333)` ⇒ the frequency of F♯ + 1/3 of a semitone in middle C octave.

`pitch(const char *note)`
main.cc
Takes a string of characters representing a note name and returns a frequency in Hertz. For example: `pitch("C8")` ⇒ 261.6 Hz or middle C. `pitch("C#8")` ⇒ the frequency of C sharp above middle C. `pitch("A8")` ⇒ 440 Hz or A above middle C. `pitch("Eb6")` ⇒ the frequency of E♭ in the second octave below middle C. `pitch("F#8+1/3")` ⇒ the frequency of F♯ + 1/3 of a semitone in middle C octave.

Appendix E

Script language implementation

The objects and functions described in appendix D are built into the library `libtao.a`. In theory, any synthesis scenario which can be described in a TAO script could also be described as a C++ program, making use of this library. In fact, compiling a TAO script leads to the automatic generation of such a program. All the details of the compilation and linking of this program are hidden from the user.

In order for this process to occur, the script must first be translated into an appropriate fragment of C++ code dealing with `Instrument`, `Microphone` and `Cell` objects. The details of this translation process are described in this appendix.

In practice, the translation process is carried out partly by the Unix `sed` command (stream editor) which matches quite complicated patterns of characters in an input stream and allows them to be replaced with other patterns of characters. The command is fairly low level in its nature and would not be used in a proper distribution version of TAO, but it serves its purpose for the current prototype. The translation of a TAO script requires several `sed` scripts which are listed in section G.3.

The output from these `sed` scripts only partially translates the TAO script. This partially translated version is stored in a file called `tao_scriptfile` which is `#include`'d into the `main` function, where it is further processed by the C++ preprocessor via a set of `#define`'d macros in the file `main.cc`. Apart from a straight translation of TAO script features into C++ language features, extra C++ code has to be added to the automatically generated source code in order to drive the synthesis engine and

provide some of the more subtle score features such as the `start` and `end` variables and their associated scoping facility, described in section 5.5.2.

E.1 Translating Instrument, Microphone and Parameter declarations

The orchestra part of a TAO script bears a simple one to one relationship with its C++ equivalent. The `Instrument`, `Microphone` and `Parameter` declarations correspond to declarations of C++ variables of type `Instrument`, `Microphone` and `float`. The following examples illustrate the precise translation which occurs:

TAO: `Circle circle1:` *f* `Hz,` *t* `secs; ...`

⇓

C++: `Circle circle1(`*f*`,` *t*`);`

Similarly:

TAO: `Microphone mic1:` *filename*`, stereo;`

⇓

C++: `Microphone mic1("`*filename*`",2);`

Parameter declarations are even simpler to translate and only involve replacing the keyword `Parameter` with the C++ keyword `float`.

E.2 Translating instrument messages

Many of the instrument messages are only superficially different in syntax from the actual C++ member functions used to implement them. Messages such as `lockleft`, `lockright`, `lockbottom`, `locktop`, `lockperimeter`, `lockends` and `lockcorners` require no arguments in a TAO script but the corresponding C++ member function do require an empty set of brackets after the function name:

TAO:		C++:
`lockleft`	⇒	`lockleft()`

```
lockright      ⇒    lockright()
lockbottom     ⇒    lockbottom()
locktop        ⇒    locktop()
etc.
```

The `setdamping` family of messages which TAO provides specify the damping coefficient as a percentage. In practice though, the damping value stored in each cell, D, is related to this percentage, d, by the formula $D = 1 - d/100$, leading to the following translation:

TAO: `setdamping(`*x1*`,` *x2*`,` *y1*`,` *y2*`,` *d%*`)`

⇓

C++: `setdamping(`*x1*`,` *x2*`,` *y1*`,` *y2*`, 1.0-`*d*`/100.0)`

E.3 Translating microphone messages

The microphone messages `output`, `leftout` and `rightout` are translated as follows:

```
TAO:   output:      sample;
       leftout:     lsample;
       rightout:    rsample;

                    ⇓

C++:   output       (sample);
       leftout      (lsample);
       rightout     (rsample);
```

E.4 Translating positional and time nomenclature

The keywords `left`, `right`, `bottom` and `top` are simply translated into numerical constants:

```
TAO:                    C++:

left           ⇒        0.0
right          ⇒        1.0
bottom         ⇒        0.0
top            ⇒        1.0
centre         ⇒        0.5
```

The keywords `secs`, `msecs` and `min` are translated into multiplications or divisions by the appropriate factors:

n `secs`	⇒	`n*1.0`
n `msecs`	⇒	`n/1000.0`
n `min`	⇒	`n*60.0`

E.5 Translating the score

The task of translating the score is slightly more complicated, but at a basic level consists of translating TAO control structures into appropriate C++ `if` statements. The `Score` control structure itself is translated into a `for` loop which iteratively updates the synthesis engine an appropriate number of times in order to generate the correct number of samples specified by the score duration. The statements contained in the body of the score are executed from top to bottom on each iteration and the set of `if` statements serve to enable the body of each control structure only at the correct times. A variable called `Sample` keeps track of the number of time steps elapsed.

The C++ program generated by the compilation of a TAO script must also include all the appropriate code to drive the synthesis engine. More specifically this includes code for:

- traversing the linked list of instruments and calculating the internal forces acting upon the cells of each;
- traversing the linked list again in order to update the positions and velocities of all the cells;
- traversing the linked list of microphones and updating each, writing the contents of the microphone's sample buffer to the designated output file if the buffer is full;
- generating the graphics images;
- detecting mouse movement and button presses and acting accordingly;
- updating the value of the variable `Time`.

All of this extra house-keeping code is packed into the `for` statement itself as we shall see in the complete script translation example given at the end of this appendix.

E.5.1 Translating the score control structures

The score control structures are translated in a number of intermediate stages. The following examples show the first stage of translation for the various control structures:

TAO:
```
From x secs to y secs:
    body
    ...

    ⇓
```

C++:
```
If (Sample >= (long)(x * modelrate)
    && Sample <= (long)(y * modelrate))
    {
    body
    }
```

This `if` statement enables the instructions to be executed on every time step from x seconds up to and including y seconds. Similarly for an `At .. for` block the translation is:-

TAO:
```
At x secs for y secs:
    body
    ...

    ⇓
```

C++:
```
If (Sample >= (long)(x * modelrate)
    && Sample <= (long)((x+y) * modelrate))
    {
    body
    }
```

For the `Every` and `ControlRate` structures the translation proceeds as follows:-

TAO:
```
Every x secs:
    body
    ...
```

⇓

```
C++:   If (Sample % (x * modelrate)==0)
           {
           body
           }
```

```
TAO:   ControlRate x:
           body
           ...
```

⇓

```
C++:   If (Sample % x == 0)
           {
           body
           }
```

E.5.2 Adding code to update the values of start and end

The special variables `start` and `end` described in section 5.5.2 are actually ordinary C++ floating point variables, but in order that their values are updated throughout the score, when entering and leaving the scope of control structures, two stacks `startstack[]` and `endstack[]` are used. The extra code needed to push and pop start and end times on and off these two stacks is added by the system at the beginning and end of every control structure's body.

Since `if` statements form the basis of all the TAO control structures, there must also be some mechanism for transmitting the start and end times tested for in the *head* of each `if` statement to the instructions contained in the *body*. This is achieved with two further variables `START` and `END`. For example:

```
TAO:   At x secs for y secs:
           body
           ...
```

translates first to:

```
C++:   if (Sample >= (long)(x * modelrate)
           && Sample <= (long)((x+y) * modelrate))
           {
```

body
}

and then to the following which includes all the code necessary to keep the values of `start` and `end` up to date as a new level of scope is entered:

C++:
```
if ( Sample >= (long)(START=x*modelrate) &&
Sample <= (long)(END=(x + y)*modelrate)
{
n++;startstack[n]=start;endstack[n]=end;
start=START;end=END;
{body}
start=startstack[n];end=endstack[n];n--;
}
```

When the `if` statement compares the value of the variable `Sample` against the start and end times given to see if the statements contained in the body should be executed, it also stores these times in the variables `START` and `END` so that once the old values of `start` and `end` have been pushed onto the stack they can take up their new values.

E.6 An example of a complete script translation

To clarify the translation process and put together all the elements described so far, the following example shows the translation of a whole TAO script into its equivalent C++ program. This program contains all the code needed to create the instruments described by the user, and bring them to life, whilst carrying out the user's specified score algorithm.

```
String string1:
    C#7+1/2, 17 secs;
    lockright;
    setdamping(left, 1/20, 0.1%);
    ...


Rectangle rect1:
    100 Hz, 340 Hz, 25 secs;
    lock(left, bottom);
    lockright;
    setdecay(2/5, 3/5, 2/5, 3/5, 1 secs);
    ...


Glue string1(left) to rect1(right, 1/3);

Parameter damping_coefficient;

Score 30 secs:
```

```
At 0 secs for 1 msecs:
    string1(0.1).applyforce(10.0);
    ...

From 5 secs to 5.001 secs:
    string1(0.9).applyforce(5.0);
    ...

ControlRate 100:
    damping_coefficient=expon(1, 0.001);
    ...

Every 0.1 secs:
    Display "Time=", Time, newline;
    ...
...
```

This script translates into the following purely C++ program which now has all the extra code required to drive the synthesis engine. It also contains code to keep the values of `start` and `end` up to date. Comments have been added to this code and the visual layout has been improved by hand for the sake of clarity but essentially this piece of code when placed inside the `main` function is ready for compilation and linking with the TAO library and will carry out the synthesis described in the script when executed.

```
////////////////////////////////////////////////////////////////////////////////
// TAO:   String string1:
//            C#7+1/2, 17 secs;
//            lockright;
//            setdamping(left, 1/20, 0.1%);
//            ...
//

String string1(pitch("C#7+1/2.0"),17*1.0);
string1.lockright();
string1.setdamping(0.0, 1/20.0, 1.0-0.1/100.0);
////////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////////
// TAO:   Rectangle rect1:
//            100 Hz, 340 Hz, 25 secs;
//            lock(left, bottom);
//            lockright;
//            setdecay(2/5, 3/5, 2/5, 3/5, 1 secs);
//            ...
//

   Rectangle rect1(100 ,340 ,25*1.0);
   rect1.lock( 0.0 ,  0.0 );
   rect1.lockright();
   rect1.setdecay(2/5.0, 3/5.0, 2/5.0, 3/5.0, 1*1.0);
////////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////////
// TAO:   Glue string1(left) to rect1(right, 1/3);
//

   Instrument::glue(string1, 0.0, rect1, 1.0, 1/3.0);
////////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////////
// TAO:   Parameter damping_coefficient;
//

   float damping_coefficient;
////////////////////////////////////////////////////////////////////////////////


////////////////////////////////////////////////////////////////////////////////
// TAO:   Score 30 secs:
//
```

```
// Start of the score head

   NumSamples=(long)(30*1.0*modelrate);
   cout << "Calculating " << NumSamples << " samples\n";

   startstack[1]=start=0.0; endstack[1]=end=30*1.0;
   startstack[0]=start; endstack[0]=end;
   START=start;END=end;

   for(Sample=0, Time=0.0;

       graphics_on?(color(7),1):0,        // if in graphics mode set color to WHITE.
       graphics_on?(clear(), 1):0,        // and clear the graphics screen.
       Instrument::calculate_forces(),    // calculate the forces for each instrument.
       update_graphics(),                 // mouse functions for graphics window.
       Sample<=NumSamples;                // enough samples generated yet?
           (graphics_on&&(Sample%graphics_update_step==0))? // display instruments.
               (Instrument::display_all(), 1):0,
           (graphics_on&&(Sample%graphics_update_step==0))? // swap front and back buffers.
               (swapbuffers(),1):0,
           Instrument::update_positions(),  // update the positions of each instrument.
           Microphone::update_all(),        // update all the microphones.
           Sample++,
           Time=Sample/modelrate)           // calculate elapsed time since performance
                                            // began.
           {
           n++;startstack[n]=start;endstack[n]=end;

// End of the score head ////////////////////////////////////////////////////////////////

// Start of the score body //////////////////////////////////////////////////////////////

                {
                ////////////////////////////////////////////////////////////////////////
                // TAO:    At 0 secs for 1 msecs: <body> ...
                //

                   if(Sample<=(long)((END=(0*1.0+1/1000.0))*modelrate) &&
                      Sample>=(long)((START=(0*1.0))*modelrate))
                       {
                       n++;startstack[n]=start;endstack[n]=end;start=START;end=END;
                           {
                           string1(0.1).applyforce(10.0);
                           }
                       start=startstack[n];end=endstack[n];n--;
                       }
                ////////////////////////////////////////////////////////////////////////

                ////////////////////////////////////////////////////////////////////////
                // TAO:    From 5 secs to 5.001 secs: <body> ...
                //

                   if(Sample<=(long)((END=(5.001*1.0))*modelrate) &&
                      Sample>=(long)((START=(5*1.0))*modelrate))
                       {
                       n++;startstack[n]=start;endstack[n]=end;start=START;end=END;
                           {
                           string1(0.9).applyforce(5.0);
                           }
                       start=startstack[n];end=endstack[n];n--;
                       }
                ////////////////////////////////////////////////////////////////////////

                ////////////////////////////////////////////////////////////////////////
                // TAO:    ControlRate 100: <body> ...
                //

                   if(Sample%(long)100==0)
                       {
                       n++;startstack[n]=start;endstack[n]=end;
                           {
                           damping_coefficient=((float)(1)*expf(1.0/(end-start)*
                                        logf((float)(0.001)/(float)(1))*(Time-start)));
                           }
                       start=startstack[n];end=endstack[n];n--;
                       }
                ////////////////////////////////////////////////////////////////////////

                ////////////////////////////////////////////////////////////////////////
                // TAO:    Every 0.1 secs: <body> ...
                //
```

```
                if(Sample%(long)(0.1*1.0*modelrate)==0)
                    {
                    n++;startstack[n]=start;endstack[n]=end;
                        {
                        // Display "Time=", Time, newline;

                        cout << "    " << setw(0) << setprecision(4)
                     << setiosflags(ios::fixed) << "Time=" << " " << Time
                << " " << '\n' << flush;
                        }
                    start=startstack[n];end=endstack[n];n--;
                    }
               ////////////////////////////////////////////////////////////////////////
               }

// End of the score body ////////////////////////////////////////////////////////////
start=startstack[n];end=endstack[n];n--;
}
```

Appendix F

Details of the bowing model used

F.1 Classical description of the behaviour of a bowed string

This appendix describes the bowing model provided by TAO. Figure F.1 shows the idealised motion of a bowed string when a clean note is obtained (Rossing, 1990). Although the overall amplitude envelope of the string is round, the motion actually consists of a fairly sharply defined corner dividing the string into two straight line segments. The corner traverses the string and is negatively reflected each time it reaches one of the terminated ends. The point at which the string is bowed can either be sticking or slipping at any instant in time. When it is sticking to the bow it moves slowly upwards at the same velocity as the bow, as shown in (c) to (h) and (a). As it does so the corner dividing the string into two travels towards the right hand end of the string, where it is reflected.

As the corner travels back from the right hand end of the string and finally reaches the bowed point again, the 'kick' caused is enough to make the string slip. This point in the cycle of motion is depicted in (a). Once slipping, the string rapidly moves downwards as the corner travels towards the left hand end where it is reflected once again. As it passes the bowed point, as in (c), the bow picks the string up again and the whole process is repeated. It should be emphasised that this precise motion

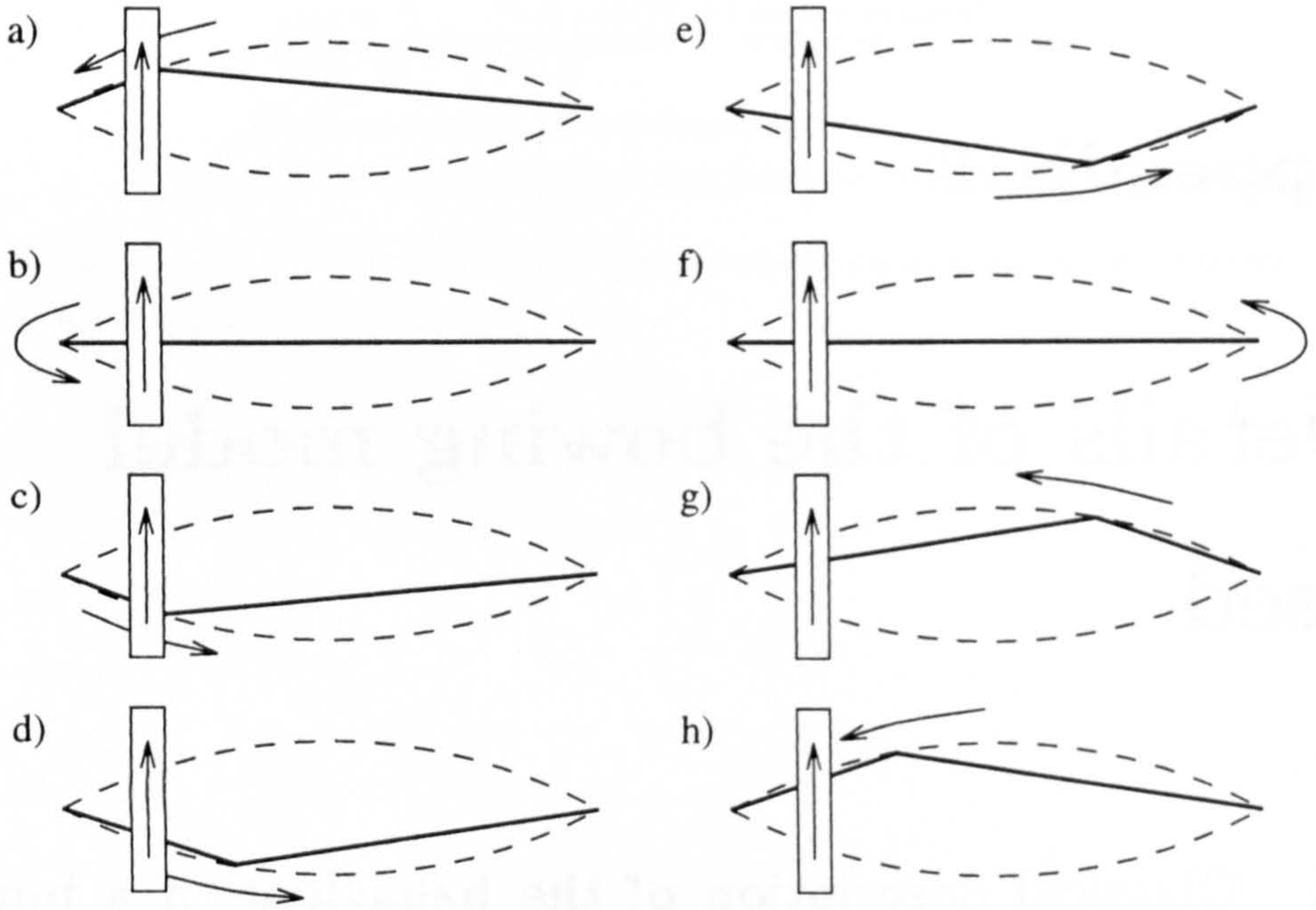


Figure F.1: Classic Helmholtz motion of a bowed string

dividing the string into two perfectly straight line segments is an idealisation of the actual shape of a real string. Also the model tells us nothing about the transient motions which the string must pass through in order to reach this state of dynamic equilibrium. This mode of motion is referred to as *Helmholtz* motion, after Hermann von Helmholtz who first observed it experimentally.

F.2 Description of an established bowed string model

The bowing model provided by TAO is loosely based on a model described by Woodhouse (1992) which is reproduced here, in brief form, for the purposes of comparison. Woodhouse's model comprises two elements, a linear element representing the string, and a non-linear element representing the interaction of the bow with the string. The behaviour of the non-linear element is derived from the graph shown in figure F.2, which represents the relationship between the frictional force exerted by the bow and velocity of the string at the bowed point. The vertical portion of the slope represents the sticking state. The velocity of the string is constant in this portion

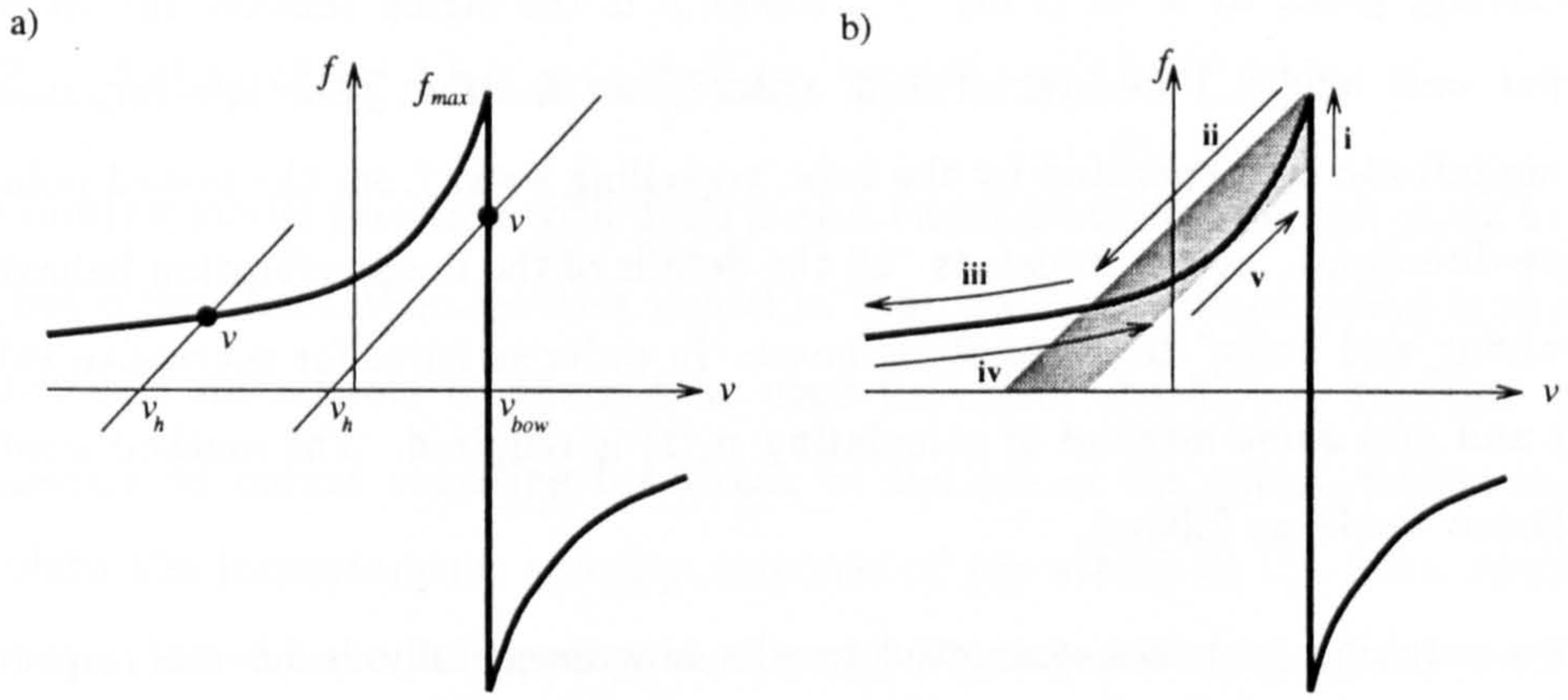


Figure F.2: Relationship between frictional bow force and relative velocity between bow and string

of the graph because the string travels at the same velocity as the bow, v_b. The frictional force can take on a range of values, however, as the bow drags the string further and further away from its rest position. The force increases until a threshold value is reached, at which point the bow can now longer hold the string and it begins to slip. As this occurs the static frictional force is replaced by a dynamic frictional force.

The curved portion to the left of the graph represents the way in which the dynamic frictional force changes with the string's velocity. As we descend down the slope the string's velocity decreases and eventually it ends up travelling in the opposite direction to the bow. The magnitude of the relative velocity between bow and string therefore increases and the associated dynamic frictional force decreases. Toward the very left of the graph the frictional force is usually of the order of $0.2 f_b$, where f_b is the normal force exerted by the bow on the string (Mcintyre, 1983).

In Woodhouse's model only the bowed point of the string is modelled. At a given instant in time the frictional force, $f(t)$, exerted by the bow and the velocity of the string at the bowed point are given by the equation:

$$v(t) = (Y/2) f(t) + v_h(t)$$

where $v(t)$ is the instantaneous velocity response to the force, $v_h(t)$ is the component

of the velocity due to the past history of the string, and Y is the wave admittance of the string, given by $Y = (Tm)^{-1/2}$, where T is the string tension and m is its mass per unit length. The factor $Y/2$ arises from the fact that two impulses of equal magnitude are generated by the bow, travelling away from the bowed point in opposite directions. $v_h(t)$ represents "all the details of the linear vibration behaviour of the string and body" to quote Woodhouse. In order to solve for successive values of $f(t)$ and $v(t)$ some method of calculating $v_h(t)$ is required. The method used by Woodhouse works as follows.

The two outgoing impulses generated by the bow eventually reach the respective ends of the string and are negatively reflected, but they also change in shape due to the effects of the bridge and body at one end and the players finger and fingerboard at the other. This 'smearing' is simulated with the use of two 'corner rounding functions'. By convolving the impulse generated by the bow with these functions, the value of $v_h(t)$ can be calculated. Successive reflections of the initial impulse become increasingly smoothed out and eventually die away due to the repeated convolution with the corner rounding functions.

Once the value of $v_h(t)$ is known it is possible to calculate new values for $f(t)$ and $v(t)$ by finding the intersection of the straight line $v = (Y/2)f + v_h$ with the curve given in figure F.2. By repeating this process iteratively the dynamic behaviour of the string can be simulated.

Note that outside the shaded portion shown in figure F.2(b) the straight line intersects the graph unambiguously. To the left of this shaded region the string is slipping and to the right it is sticking. What happens in the shaded region depends on the current state of the string. If the string is sticking, then the straight line intersects the vertical portion of the graph, but as the static frictional force grows, the line moves into the shaded region from the right. The static frictional force continues to grow, but at the point where it can no longer be sustained, the frictional forces jumps suddenly down to the value on the curved portion of the graph, following arrow (ii), indicating that the string has begun to slip. If the string is slipping, then the shaded region is entered from the left, and the frictional force jumps suddenly up to the value on the straight line portion of the graph, following arrow (v), indicating that the string has begun to stick again. This introduces an element of hysteresis

into the model and shows that its state changes in a discontinuous manner.

F.3 Adapting the model to work with TAO

The bowing model provided with TAO is also based around the graph given in figure F.2 but differs from Woodhouse's model in that the resonator or string is explicitly modelled in its entirety so there is no need for either the history function v_h and its associated corner rounding functions, or the use of the wave admittance Y to calculate the instantaneous velocity response of the string to the force applied by the bow. Instead the string may be directly interrogated at any point in time to find its position, velocity or acceleration, and if we want to find out the effect which applying a force will have on the position and velocity of the string, we can simply apply the force at the appropriate point and let the model do the rest.

Before describing in detail the way in which the model works, the reader is reminded of the steps involved in animating a TAO instrument, first given in section 4.6:

1. the internal forces acting upon each cell are calculated;
2. any external forces are applied;
3. the cell positions are updated.

All interaction with TAO instruments is via the physical parameters of individual cells or groups of cells and in the case of the bowing model a single cell provides the interface between the bow and the instrument and acts both as an input and output. The interaction between the bow and the chosen cell takes place on step 2 above, at which point in time the internal force acting on the cell, due to its spring connections, has just been calculated, but its position and velocity have not yet been updated. There are two things we need to do in order to simulate the sticking and slipping of the bow:

1. If the string is sticking to the bow all we need to do is: calculate the static frictional force needed to keep the string travelling at the same velocity as the bow; check that this force does not exceed the maximum threshold force which can be sustained by the static friction between the bow and the string; and apply this force to the cell.

2. If the string and bow are slipping past each other all we need to do is: calculate the relative velocity between them, and hence the dynamic frictional force; check that the condition for the string and bow to start sticking again is not met; and apply this force to the cell.

This gives us the basis for a discrete time domain simulation with a TAO instrument. At any instant in time if f_b is the downward force exerted by the bow, v_c is the velocity of the cell being bowed, and v_b is the velocity of the bow, then the relative velocity between the bow and cell v_r is given by $v_r = v_b - v_c$. Acceleration, in the context of this discrete time based simulation is given by $a = \frac{\delta v}{\delta t}$, but from section 4.6.5, $\delta t = 1$, so effectively $a = \delta v = v_b - v_c$.

Therefore, in order to keep the cell travelling at a constant velocity, we simply calculate the difference between the current velocity, v_c, and the required velocity, v_b, and this value becomes the required acceleration. Since by Newton's 2nd law of motion $f = ma$, the total force required in order to cancel out the internal force f_c acting on the cell and ensure that the cell has the correct acceleration to maintain velocity v_b is given by $f_{stick} = m(v_b - v_c) - f_c$. This is the case when the string is sticking to the bow.

When the string is slipping, the dynamic frictional force f_{slip} is given by:

$$f_{slip} = f_b \left(0.2 + 0.8 * \frac{1}{1 + |\, v_r \,|}\right)$$

Remember that as the relative velocity $|\, v_r \,|$ between string and bow increases, the dynamic frictional force f_{slip} tends towards $0.2 f_b$, and that when $v_r = 0$, according to the graph in figure F.2, $f_{slip} \approx f_b$ (assuming that the coefficient of friction relating the downward force of the bow to the maximum frictional force possible is unity, which it almost is for rosined surfaces (Mcintyre, 1983)). For values in between these two extremes the curve appears to drop off at a rate $\propto \frac{1}{|v_r|}$. Now we know how to calculate the static and dynamic frictional forces, all we need to decide is under what conditions the model flips from one state to the other. The change from *stick* mode to *slip* mode occurs when the static frictional force required is greater than the downward force of the bow can sustain, in other words when $f_{stick} \geq f_b$. The change from *slip* to *stick* occurs when the corner traversing the string reaches

the bow and the string starts to travel in the same direction as the bow, i.e. when $v_c \geq 0$. We can now put all of this information together to form the algorithm given below. This algorithm is executed on every discrete time step.

1. Calculate the relative velocity between bow and string $v_r = v_b - v_c$.

2. Calculate the acceleration needed to keep the string moving with the same velocity as the bow $a_s = (v_b - v_c)\delta t = v_r$ (since $\delta t = 1$)

3. Decide whether the cell is sticking or slipping and apply the appropriate static or dynamic frictional force. Also check to see whether the conditions for a change of state are met:

```
if in stick mode:
        f_stick = m a_c − f_c
        if f_stick > f_b:
                change to slip mode
        else:
                applyforce (f_stick)

else if in slip mode:
f_slip = f_b (0.2 + 0.8 * 1/(1+|v_r|))
if v_c ≥ 0:
        change to stick mode
else:
        applyforce (f_slip)
```

By iteratively executing this algorithm we can simulate the continuous interaction of the bow with the string or indeed with any other instrument. The control parameters v_b and f_b may be varied within the score in the same way as any other performance parameters, and by using the `mode` field of a cell (see appendix D for an explanation) to indicate whether the cell is in *stick* or *slip* mode, any number of bows with independent control parameters may be applied to any number of locations on one or more instruments, simultaneously. Examples of the bowing model in action can be found in sections 6.7.1 and 6.7.2.

Appendix G

Implementation code

G.1 C++ implementation of the TAO library `libtao.a`

G.1.1 File `Cell.h`

```
////////////////////////////////////////////////////////////////////////////
// File name: Cell.h                          (c) 1996 Mark Pearson
//
// Content: Definition of Cell object class.
//
// Member variables:
//      mode:       used to hold a variety of information such as whether
//                  the cell is glued to another and if so, whether it is
//                  the master or slave cell. Also whether the cell is locked,
//                  and whether it is sticking or slipping when bowed.
//      north, south, east, west, neast, seast, nwest, swest:
//                  pointers to this cell's neighbouring cells.
//      companion: if this cell is glued to another then the companion
//                  pointer points to the other cell.
//      mass:       the cell's mass measured in arbitrary numerical units.
//      damping:    a value between 0 and 1 which the velocity of the cell
//                  is multiplied by every time step leading to energy
//                  dissipation.
//      position, velocity, force:
//                  once again all measured in arbitrary numerical units.
//                  Each is a scalar value measuring the magnitude of a
//                  vector in the y direction, i.e. vertical displacement,
//                  vertical velocity and vertical force acting upon the
//                  cell due to its springs.
////////////////////////////////////////////////////////////////////////////

#ifndef CELL_H
#define CELL_H

#define CELL_LOCK_MODE  0x01
#define CELL_SLAVE_MODE  0x02
#define CELL_MASTER_MODE  0x04
#define CELL_BOW_STICK_MODE  0x08

struct Cell
    {
    int mode;
    Cell *north, *south, *east, *west;
    Cell *neast, *nwest, *seast, *swest;
    Cell *companion;
    float mass, damping;
    float position, velocity, force;
    void applyforce(float F);
    void bow(float f_bow, float v_bow);
    void lock() {mode |= CELL_LOCK_MODE;}
    operator float() {return position;}
    };

#endif
```

G.1.2 File Cell.cc

```
////////////////////////////////////////////////////////////////////////////
// File name: Cell.cc                          (c) 1996 Mark Pearson
//
// Content: Definitions of Cell object class member functions
////////////////////////////////////////////////////////////////////////////

#include "Cell.h"
#include <math.h>
#include <iostream.h>

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

////////////////////////////////////////////////////////////////////////////
// Function name:   Cell::applyforce(float F)
//
// Functionality:
//      Apart from applying the given force to the cell specified, also
//      applies smaller sympathetic forces to the neighbouring cells to
//      ensure that the force is spread over a small region. This is to
//      compensate for the material's lack of stiffness.
//
// Variables:
//      All Cell class member variables except the argument F which represents
//      the force being applied to the cell.
////////////////////////////////////////////////////////////////////////////

void Cell::applyforce(float F)
    {
    force+=F;
    if (north) north->force+=F/2.0;
    if (south) south->force+=F/2.0;
    if (east) east->force+=F/2.0;
    if (west) west->force+=F/2.0;
    if (neast) neast->force+=F/2.82;
    if (seast) seast->force+=F/2.82;
    if (nwest) nwest->force+=F/2.82;
    if (swest) swest->force+=F/2.82;
    }


////////////////////////////////////////////////////////////////////////////
// Function name:   Cell::bow(float f_bow, v_bow)
//
// Functionality:
//     Simulates the interaction of a virtual bow with a cell, based on
//     frictional sticking and slipping. The algorithm is explained in
//     section F.3. The function given below is an exact
//     implementation of this algorithm.
//
// Arguments:
//     f_bow:     downward force of bow.
//     v_bow:     velocity of bow.
//
// Local variables:
//     f_stick:     static frictional force exerted by the bow on the cell.
//     f_slip:      dynamic frictional force exerted by the bow.
//     force_exerted:
//                  whether sticking or slipping one of the above forces is
//                  applied. This variable stores the chosen frictional force.
//     v_relative:  relative velocity between bow and cell.
//     a_cell:      acceleration needed to keep cell's velocity equal to v_bow.
//
// Cell class member variables:
//     velocity, force, mass.
////////////////////////////////////////////////////////////////////////////

void Cell::bow(float f_bow, float v_bow)
    {
    static float f_stick, f_slip, force_exerted;
    static float v_relative, a_cell;

    v_relative=a_cell=v_bow-velocity;                   // a=dv/dt but dt=1 so a=dv.

    if (mode & CELL_BOW_STICK_MODE)                     // if in 'stick' mode.
```

```
    {
    f_stick=mass*a_cell-force;
    if (f_stick>f_bow) mode&=!CELL_BOW_STICK_MODE; // if static frictional
    else force_exerted=f_stick;                    // force required is too
    }                                              // great, change to
                                                   // 'slip' mode.

else                                               // if in 'slip' mode.
    {
    f_slip=f_bow/(1.0+fabs(v_relative));
    if (velocity>=0.0) mode|=CELL_BOW_STICK_MODE;  // if the cell starts
    else force_exerted=f_slip;                     // travelling in the same
    }                                              // direction as the bow,
                                                   // change to 'stick' mode.

applyforce(force_exerted);                         // apply the appropriate
}                                                  // frictional force.
```

G.1.3 File Instrument.h

```
////////////////////////////////////////////////////////////////////////////////
// File name:   Instrument.h                    (c) 1996 Mark Pearson
//
// Content:     Definition of Instrument object class and Row structure.
//
//      In a TAO script the user deals with objects of class String,
//      Rectangle, Circle, Ellipse and Triangle but all of these are
//      derived classes of base class Instrument. The Instrument object
//      class contains the following member variables:
//
// Member variables:
//      xfrequency:     frequency in hertz in the horizontal direction.
//      yfrequency:     frequency in hertz in the vertical direction.
//      default_decay:  decay time given uniformly to the instrument when it
//                      is first created.
//      default_damping: the equivalent damping coefficient.
//      rows:           array of Row structures, each representing a
//                      single row of cells.
//      graphx, graphy: determine where an instrument will be displayed
//                      in the graphics window. Measured in screen
//                      coordinates.
//      worldx, worldy: determine where the instrument lies in terms of
//                      the world coordinate system measured in cells.
//                      Joining two pieces of material causes the second
//                      to be placed in the correct position relative to
//                      the first. This ultimately affects only where they
//                      are displayed graphically relative to each other.
//      next:           pointer to next instrument created. Used to maintain
//                      a linked list of all instruments created within one
//                      script.
//      xmax, ymax:     size of the bounding box which just fits around
//                      the instrument in cells. Xmax is the width - 1 of
//                      the instrument measured in cells and ymax is the
//                      height - 1 measured in cells.
//      amplification:  the factor by which the amplitude of vibrations is
//                      emphasised when displayed graphically. Has no effect
//                      on sound output.
//
// Static member variables:
//      list, current:  head of linked list of instruments, and current
//                      instrument during updating.
//      default_mass:   the default mass which all cells are given initially.
//                      Should not be altered as it has been chosen for
//                      optimum performance and would upset conversion from
//                      hertz to cells.
//      global_amplification:
//                      global amplification factor for all instruments
//                      when displayed graphically. Has no effect on sound
//                      output.
//
////////////////////////////////////////////////////////////////////////////////

#ifndef INSTRUMENT_H
#define INSTRUMENT_H

#include <stdlib.h>
#include <math.h>
#include "Cell.h"
#include <iostream.h>

#define Hz2CellConst 24000.0  // Used to convert a frequency in Hz into the
                              // appropriate number of cells needed to achieve
                              // this frequency.

#define Decay2DampingConst 0.000375
                              // Used to convert a decay time into a damping
```

```
                                // value suitable for the 'damping' field of
                                // a cell. When the velocity of a cell is
                                // repeatedly multiplied by this damping value
                                // on every time step of the synthesis engine
                                // its vibrations will decay over the given
                                // decay time.

// NOTE: Hz2CellConst & Decay2DampingConst and audiorate, modelrate &
//       bandwidthlevel (all from main.cc) are all interrelated and must
//       not be changed.

struct Row
    {
    int xmax;
    int offset;
    Cell *cells;
    };

class Instrument
    {
protected:
    float xfrequency, yfrequency;
    float default_decay, default_damping;
    Row *rows;
    int graphx, graphy;
    int worldx, worldy;
    Instrument *next;

    void initialise_cells();
    void link_cells();
    void calculate_my_forces();
    void update_my_position();

    static Instrument *list;
    static Instrument *current;
    static float default_mass;
    static void glue_cells(Cell *c1, Cell *c2);
    static void join_left_to_left(Cell &cell1, Cell &cell2);
    static void join_left_to_right(Cell &cell1, Cell &cell2);
    static void join_right_to_left(Cell &cell1, Cell &cell2);
    static void join_right_to_right(Cell &cell1, Cell &cell2);
    static void join_bottom_to_bottom(Cell &cell1, Cell &cell2);
    static void join_bottom_to_top(Cell &cell1, Cell &cell2);
    static void join_top_to_bottom(Cell &cell1, Cell &cell2);
    static void join_top_to_top(Cell &cell1, Cell &cell2);

public:
    int xmax, ymax;
    float amplification;

    Instrument(float xfreq, float yfreq, float decay);
    Instrument &setdecay(float x1, float x2, float y1, float y2, float decay);
    Instrument &setdecay(float left, float right, float decay);
    Instrument &setdecay(float decay);
    Instrument &resetdecay(float x1, float x2, float y1, float y2);
    Instrument &resetdecay(float left, float right);
    Instrument &resetdecay();
    Instrument &setdamping(float x1, float x2, float y1, float y2, float damping);
    Instrument &setdamping(float left, float right, float damping);
    Instrument &setdamping(float position, float damping);
    Instrument &setdamping(float damping);
    Instrument &resetdamping(float x1, float x2, float y1, float y2);
    Instrument &resetdamping(float left, float right);
    Instrument &resetdamping(float position);
    Instrument &resetdamping();
    Instrument &vibrato(float rate, float depth);
    Instrument &lock(float x1, float x2, float yi, float y2);
    Instrument &lock(float x, float y);
    Instrument &lockleft();
    Instrument &lockright();
    Instrument &locktop();
    Instrument &lockbottom();
    Instrument &lockperimeter();
    Instrument &lockcorners();
    Instrument &lockends();
    Cell &at(float x, float y);
    Cell &operator()(float x, float y);
    Cell &operator()(float x);

    float screenx(float x, float y);
    float screeny(float x, float y);
    void label(float x, float y, int xoffset, int yoffset,
               char *caption, int colour);
    void label(float x, int xoffset, int yoffset,
               char *caption, int colour);
    void display();
    void display_at(int x, int y) {graphx=x;graphy=y;}
```

```
    void place_at(int x, int y) {worldx=x;worldy=y;}

    void add_to_global_list()
        {
        if (list==NULL) list=this; else current->next=this;
        current=this;
        }

    static float global_amplification;
    static void calculate_forces();
    static void update_positions();
    static void display_all();
    static float decay2damping(float decay) {return (1.0-(Decay2DampingConst/decay));}
    static int hertz2cells(float freq) {return (int)(Hz2CellConst/freq);}
    static void glue(Instrument &i1, float x1, float y1,
                     Instrument &i2, float x2, float y2);
    static void glue(Instrument &i1, float x1, float y1,
     Instrument &i2, float x2);
    static void glue(Instrument &i1, float x1,
     Instrument &i2, float x2, float y2);
    static void join(Instrument &i1, float x1, float y1,
     Instrument &i2, float x2, float y2);
    };


#endif
```

G.1.4 File Instrument.cc

```
////////////////////////////////////////////////////////////////////////////////
// File name:   Instrument.cc                    (c) 1996 Mark Pearson
//
// Content:     Definition of Instrument class member functions
//
// Notes:
//      Throughout this file many functions have to access individual cells
//      within an instrument. Whenever this occurs certain conventions are
//      observed. The variables x and y are always coordinates in the
//      instrument coordinate system, normalised between 0 and 1, where 0
//      & 1 mean left & right respectively for x, and bottom & top for y.
//      The local variables i and j are always integer coordinates and refer
//      to the cell in a row and the row number respectively. Whenever a
//      function is called with x and y as arguments, a conversion to i and
//      j occurs.
//
//      There are two coordinate systems used in the graphical animations:
//      world and screen. World coordinates are measured in units of cells,
//      so for example if a cell is at world coordinates (x,y) then its north
//      west neighbour is at (x-1,y+1). This coordinate system makes it a
//      simple matter to place instruments in the correct position,
//      graphically speaking, relative to each other. Screen coordinates are
//      measured in pixels.
////////////////////////////////////////////////////////////////////////////////

#include "Instrument.h"
#include <iostream.h>
#include <math.h>
#include <gl.h>
#include <sys/types.h>
#include <sys/times.h>

Instrument *Instrument::list=NULL;                  // No instruments to start with.
Instrument *Instrument::current=NULL;
float Instrument::default_mass=3.5;                 // Set to optimum value for
                                                    // frequency response of
                                                    // material. Leave well alone!!
float Instrument::global_amplification=0.0;
extern int graphics_on;                             // main.cc
extern long Sample;                                 // main.cc
extern int graphics_update_step;                    // main.cc
extern short interframedelay;                       // main.cc
extern int winoriginx, winoriginy;                  // main.cc
extern float skewfactor, xscale, yscale;            // main.cc


////////////////////////////////////////////////////////////////////////////////
// Constructor name:
//      Instrument(float xfreq, float yfreq, float decay)
//
// Functionality:
//      Since classes String, Rectangle, Circle etc. are derived from class
//      Instrument, when an object of any of these classes is created, an
//      instrument object is created first, and serves as the basic skeleton
```

```
//      for all the specific instrument shapes. This constructor function
//      creates that skeleton. The details are filled in by the derived
//      class' constructor function.
//
// Arguments:
//      Frequency in hertz of the instrument in the x and y directions and
//      an initial uniform decay time measured in seconds.
//
// Instrument class member variables:
//      xfrequency, yfrequency, default_decay, default_damping,
//      amplification, graphx, graphy, worldx, worldy.
////////////////////////////////////////////////////////////////////////////

Instrument::Instrument(float xfreq, float yfreq, float decay)
    {
    xfrequency=xfreq;
    yfrequency=yfreq;
    default_decay=decay;
    default_damping=decay2damping(decay);
    amplification=1.0; // Only for graphics display, not sound synthesis;
    graphx=0;          // ditto
    graphy=0;          // ditto
    worldx=0;          // ditto
    worldy=0;          // ditto
    }

////////////////////////////////////////////////////////////////////////////
// Operator name:
//      (float x, float y)
//
// Functionality:
//      When placed immediately after an identifier signifying an object of
//      class instrument, selects and returns a reference to the cell at
//      position (x,y). For an explanation of the coordinate system used see
//      section 5.4.8. Also has the side effect of placing
//      a blue marker on the graphics screen to mark the cell accessed, if
//      the graphics are turned on.
//
// Arguments:
//      x, y:       instrument coordinates.
//
// Instrument class member variables:
//      rows, graphx, graphy, worldx, worldy, amplification,
//      global_amplification.
//
// Row structure member variable:
//      cells.
//
// Local variables:
//      i, j:       cell number and row number coordinates (see note at head
//                  of this file).
//      c:          pointer to cell being accessed.
//      left, bottom: origin of bottom left hand corner of bounding box
//                  surrounding instrument measured in screen coordinates.
//      scrnx, scrny: screen coordinate position of blue marker.
//
// External variables: (all from file main.cc)
//      winoriginx, winoriginy, xscale, yscale, skewfactor,
//      graphics_on, graphics_update_step, Sample.
////////////////////////////////////////////////////////////////////////////

Cell &Instrument::operator()(float x, float y)
    {
    register int j=(int)(ymax*y), i=(int)(rows[j].xmax*x);
    register Cell *c=&rows[j].cells[i];
    int left, bottom;

    left=winoriginx+graphx;
    bottom=winoriginy+graphy;
    if(graphics_on && Sample%graphics_update_step==0)
                                    // the graphics are refreshed on every
        {                           // 'graphics_update_step'th time step of
                                    // the synthesis engine.
        int scrnx, scrny;

        scrnx=(left+(worldx+i+rows[j].offset)*xscale + (worldy+j)*skewfactor);
        scrny=(bottom+(worldy+j)*yscale+c->position*amplification*
                                                    global_amplification);

        color(BLUE);
        circf(scrnx, scrny, 3);
        }
    return *c;
    }

////////////////////////////////////////////////////////////////////////////
```

```
// Operator name:
//      (float x)
//
// Functionality:
//      Exactly the same as operator (float x, float y) but for one
//      dimensional instruments where only the x coordinate is needed.
////////////////////////////////////////////////////////////////////////////////

Cell &Instrument::operator()(float x)
    {
    register int i=(int)(rows[0].xmax*x);
    register Cell *c=&rows[0].cells[i];
    int left, bottom;

    left=winoriginx+graphx;
    bottom=winoriginy+graphy;

    if(graphics_on && Sample%graphics_update_step==0)
        {
        int scrnx, scrny;
        scrnx=(left+(worldx+i+rows[0].offset)*xscale + worldy*skewfactor);
        scrny=(bottom+worldy*yscale+c->position*amplification*
                                                global_amplification);

        color(BLUE);
        circf(scrnx, scrny, 3);
        }
    return *c;
    }


////////////////////////////////////////////////////////////////////////////////
// Member function name:
//      at(float x, float y)
//
// Functionality:
//      Exactly the same as for operator '(float x, float y)' in that it
//      selects and returns a reference to the cell specified by the
//      instrument coordinates x and y except it doesn't affect the graphics
//      at all.
////////////////////////////////////////////////////////////////////////////////

Cell &Instrument::at(float x, float y)
    {
    register int j=(int)(ymax*y), i=(int)(rows[j].xmax*x);
    register Cell *c=&rows[j].cells[i];
    return *c;
    }


////////////////////////////////////////////////////////////////////////////////
// Member function name:
//      screenx(float x, float y)
//
// Functionality:
//      Returns the screen x coordinate of the cell specified by the
//      instrument coordinates x and y.
//
// Local variables:
//      i, j:       cell number and row number coordinates (see note at head
//                  of this file).
//      left, bottom: origin of bottom left hand corner of bounding box
//                  surrounding instrument measured in screen coordinates.
//
// Instrument class member variables:
//      xmax, ymax, rows, graphx, graphy, worldx, worldy.
//
// External variables: (all from file main.cc)
//      winoriginx, winoriginy, xscale, yscale, skewfactor,
////////////////////////////////////////////////////////////////////////////////

float Instrument::screenx(float x, float y)
    {
    int i, j;
    int left, bottom;

    left=winoriginx+graphx;
    bottom=winoriginy+graphy;

    j=(int)(ymax*y);
    i=(int)(rows[j].xmax*x);
    return left+(worldx+i+rows[j].offset)*xscale+(worldy+j)*skewfactor;
    }


////////////////////////////////////////////////////////////////////////////////
// Member function name:
```

```
//      screeny(float x, float y)
//
// Functionality:
//      Returns the screen y coordinate of the cell specified by the
//      instrument coordinates x and y.
//
// Local variables:
//      j:          row number containing the cell specified.
//      left, bottom:
//                  origin of bottom left hand corner of bounding box
//                  surrounding instrument, measured in screen coordinates.
//
// Instrument class member variables:
//      xmax, ymax, rows, graphx, graphy, worldx, worldy, amplification,
//      global_amplification.
//
// Instrument class member function:
//      at(x, y).
//
// External variables: (all from file main.cc)
//      winoriginx, winoriginy, yscale.
////////////////////////////////////////////////////////////////////////////////

float Instrument::screeny(float x, float y)
    {
    int left, bottom;

    left=winoriginx+graphx;
    bottom=winoriginy+graphy;

    int j=(int)(ymax*y);
    return bottom+(worldy+j)*yscale+at(x, y).position*amplification*
                                                    global_amplification;
    }


////////////////////////////////////////////////////////////////////////////////
// Member function name:
//      label(float x, float y, int xoffset, int yoffset,
//            char *caption, int colour)
//
// Functionality:
//      Places a text caption on the graphics screen at a position
//      determined by the instrument coordinates x and y. If the cell
//      specified is displayed at screen coordinates (scrnx, scrny) then
//      the caption will be placed at (scrnx+xoffset, scrny+yoffset). The
//      is displayed in the specified colour where 'colour' is one of RED,
//      GREEN, BLUE, YELLOW, MAGENTA, CYAN or BLACK, which are #defined
//      constants from header file <gl.h>.
//
// Instrument class member functions:
//      screenx(x, y), screeny(x, y).
////////////////////////////////////////////////////////////////////////////////

void Instrument::label(float x, float y, int xoffset, int yoffset,
                       char *caption, int colour)
    {
    cmov2(screenx(x, y)+xoffset, screeny(x, y)+yoffset);
    color(colour);
    charstr(caption);
    }


////////////////////////////////////////////////////////////////////////////////
// Member function name:
//      label(float x, int xoffset, int yoffset,
//            char *caption, int colour)
//
// Functionality:
//      Version of label function given above for one dimensional instruments.
////////////////////////////////////////////////////////////////////////////////

void Instrument::label(float x, int xoffset, int yoffset,
                       char *caption, int colour)
    {
    cmov2(screenx(x, 0)+xoffset, screeny(x, 0)+yoffset);
    color(colour);
    charstr(caption);
    }


////////////////////////////////////////////////////////////////////////////////
// Member function name:
//      link_cells()
//
// Functionality:
//      When an instrument is first created the data structures representing
```

```
//      the cells, rows and the instrument object are set up. The shape of
//      the instrument and hence the number of rows and number of cells in
//      each row are determined by the particular constructor function of
//      the class derived from the instrument base class. This function
//      sets up the neighbour pointers of all the cells in the instrument,
//      regardless of its shape. In other words it installs the springs,
//      automatically detecting boundaries and making sure that there are no
//      ragged edges.
//
// Local variables:
//      i, j:           cell number and row number coordinates (see note at head
//                      of this file).
//      thisrow:        pointer to the current row.
//      northoffset:    offset in cells of row above relative to 'thisrow'.
//      southoffset:    offset in cells of row below relative to 'thisrow'.
//      northi:         'i' always specifies the cell number in a particular row.
//                      Since different rows have different offsets, cell 'i' in
//                      thisrow corresponds to cell 'northi' in the row above.
//      southi:         similar to northi.
//      thisxmax:       number of cells - 1 in thisrow.
//      northxmax:      number of cells - 1 in row above thisrow.
//      southxmax:      number of cells - 1 in row below thisrow.
//      c; north, south, east, west:
//                      pointers to current cell and four of its neighbours.
//
// Instrument class member variables:
//      rows, ymax.
////////////////////////////////////////////////////////////////////////////////

void Instrument::link_cells()
    {
    register i, j;
    Cell *thisrow;
    int northoffset, southoffset, northi, southi;
    int thisxmax, northxmax, southxmax;

    for(j=0;j<=ymax;j++)
        {
        if(j<ymax)
            {
            northoffset=rows[j].offset-rows[j+1].offset;
            northxmax=rows[j+1].xmax;
            }

        if(j>0)
            {
            southoffset=rows[j].offset-rows[j-1].offset;
            southxmax=rows[j-1].xmax;
            }

        thisxmax=rows[j].xmax;
        thisrow=rows[j].cells;

        for(i=0;i<=thisxmax;i++)
            {
            if(i==0) thisrow[i].west=NULL;
            else thisrow[i].west=&(thisrow[i-1]);
            if(i==thisxmax) thisrow[i].east=NULL;
            else thisrow[i].east=&(thisrow[i+1]);

            northi=i+northoffset;
            southi=i+southoffset;

            if(j==0 || southi<0 || southi>southxmax)
                thisrow[i].south=NULL;
            else
                thisrow[i].south=&rows[j-1].cells[southi];

            if(j==ymax || northi<0 || northi>northxmax)
                thisrow[i].north=NULL;
            else
                thisrow[i].north=&rows[j+1].cells[northi];
            }
        }

    Cell *c, *north, *south, *east, *west;

    for(j=0;j<=ymax;j++)
        {
        for(i=0, c=rows[j].cells;i<=rows[j].xmax;i++, c++)
            {
            if(north=c->north)      c->neast=north->east;
            else if(east=c->east)   c->neast=east->north;
            else c->neast=NULL;

            if(north)               c->nwest=north->west;
            else if(west=c->west)   c->nwest=west->north;
            else c->nwest=NULL;
```

```
            if(south=c->south)      c->seast=south->east;
            else if(east=c->east)   c->seast=east->south;
            else c->seast=NULL;

            if(south)               c->swest=south->west;
            else if(west=c->west)   c->swest=west->south;
            else c->swest=NULL;
            }
        }
    }

////////////////////////////////////////////////////////////////////////////
// Member function name:
//      initialise_cells()
//
// Functionality:
//      Since the material is discrete in nature but a continuous range
//      of frequencies is needed, once the width and height of an instrument
//      in cells have been determined, the masses of the cells have to be
//      adjusted slightly away from the default mass in order to adjust the
//      frequency to the originally specified value. This compensation is
//      calculated from the given x frequency since the compensation must
//      work for strings and most 2D instruments are inharmonic in nature, so
//      the error in yfrequency will not be noticable. Also sets the
//      velocities, positions and forces of all cells to zero, and
//      initialises a few other variables.
//
// Local variables:
//      i, j:       usual use, j=row number and i=cell number in chosen row.
//      intended_freq: xfrequency specified in the instrument declaration.
//      actual_freq: xfrequency which would result if the cells were given
//                  the default_mass, having decided how many cells wide
//                  and high the instrument is.
//      c:          pointer to current cell.
//
// Instrument class member variables:
//      rows, ymax.
////////////////////////////////////////////////////////////////////////////

void Instrument::initialise_cells()
    {
    Cell *c;
    register i, j;
    float intended_freq, actual_freq, compensation_factor;

    intended_freq=xfrequency;
    actual_freq=Hz2CellConst/(xmax+1);
    compensation_factor=powf(4.0, log10f(actual_freq/intended_freq)/log10f(2.0));

    for (j=0;j<=ymax;j++)
        {
        for (i=0, c=rows[j].cells;i<=rows[j].xmax;i++, c++)
            {
            c->mode=CELL_BOW_STICK_MODE;
            c->companion=NULL;
            c->mass=Instrument::default_mass*compensation_factor;
            c->position=0.0;
            c->velocity=0.0;
            c->force=0.0;
            c->damping=default_damping;
            }
        }
    }


////////////////////////////////////////////////////////////////////////////
// Member function name:
//      calculate_my_forces()
//
// Functionality:
//      Starts at bottom left of instrument and works its way across each
//      row and then up to the next row until it reaches the top right.
//      For each cell the total force due to the springs connecting it to
//      its neighbours is calculated. If the cell is a master cell it
//      treats the slave cell's neighbours as its own in order to calculate
//      the combined force acting on both cells. If it is a slave cell then
//      no calculations are made as these will either already have been
//      performed for the master cell or will be due to be performed for the
//      master cell.
//
// Local variables:
//      i, j:       j=row number and i=cell number in chosen row.
//      c, north, south, east, west, neast, seast, nwest, swest:
//                  pointers to current cell and its neighbouring cells and
//                  also pointers to slave companion cell's neighbours if this
//                  cell has one.
```

```
//      slave:      if this cell is glued to another and is acting as the
//                  master cell then 'slave' points to the companion slave
//                  cell.
//      myposition: position of the current cell.
//      dp, count:  the force exerted on cell c by a neighbouring cell is
//                  given simply by the neighbouring cell's position minus
//                  c's position, since the coefficient of elasticity is
//                  set to unity. Therefore the total force acting on c
//                  due to all the neighbouring cells is given by the
//                  sum of the positions of the neighbouring cells minus
//                  (number of neighbours * c's position). The variable dp
//                  keeps track of this sum and count keeps track of the
//                  number of neighbours.
//
// Instrument class member variables:
//      rows, ymax.
//////////////////////////////////////////////////////////////////////////////
void Instrument::calculate_my_forces()
    {
    register i, j, count;
    register Cell *c, *slave, *north, *south, *east, *west;
    register Cell *neast, *nwest, *seast, *swest;
    static float myposition,  dp;

    for (j=0;j<=ymax;j++)
        for (i=0, c=rows[j].cells;i<=rows[j].xmax;i++, c++)
            {
            dp=0.0;myposition=c->position;count=0;

            if (! (c->mode & CELL_SLAVE_MODE))
                {
                dp+=
                    ((north=c->north)?(count++,north->position):0.0) +
                    ((south=c->south)?(count++,south->position):0.0) +
                    ((east=c->east)?(count++,east->position):0.0) +
                    ((west=c->west)?(count++,west->position):0.0) +
                    ((neast=c->neast)?(count++,neast->position):0.0) +
                    ((seast=c->seast)?(count++,seast->position):0.0) +
                    ((nwest=c->nwest)?(count++,nwest->position):0.0) +
                    ((swest=c->swest)?(count++,swest->position):0.0);

                if (c->mode & CELL_MASTER_MODE)
                    {
                    slave=c->companion;
                    dp+=
                    ((north=slave->north)?(count++,north->position):0.0) +
                    ((south=slave->south)?(count++,south->position):0.0) +
                    ((east=slave->east)?(count++,east->position):0.0) +
                    ((west=slave->west)?(count++,west->position):0.0) +
                    ((neast=slave->neast)?(count++,neast->position):0.0) +
                    ((seast=slave->seast)?(count++,seast->position):0.0) +
                    ((nwest=slave->nwest)?(count++,nwest->position):0.0) +
                    ((swest=slave->swest)?(count++,swest->position):0.0);
                    }
                }
            c->force=dp - count * myposition;
            }
    }


//////////////////////////////////////////////////////////////////////////////
// Member function name:
//      update_my_position()
//
// Functionality:
//      Starts at bottom left of instrument and works its way across each
//      row and then up to the next row until it reaches the top right.
//      The force acting upon each cell is used to calculate the cells
//      acceleration, new velocity and new position. Also multiplies the
//      new velocity by the damping value (between 0 and 1). This value is
//      converted from the percentage value given in a TAO script. 100% -> 0
//      and 0% -> 1.
//
// Local variables:
//      i, j:       j=row number and i=cell number in chosen row.
//      c:          pointer to current cell.
//
// Instrument class member variable:
//      rows.
//////////////////////////////////////////////////////////////////////////////

void Instrument::update_my_position()
    {
    static int i, j;
    static Cell *c;
```

```
for (j=0;j<=ymax;j++)
    for (i=0, c=rows[j].cells;i<=rows[j].xmax; i++, c++)
        {
        if(!(c->mode & CELL_LOCK_MODE || c->mode & CELL_SLAVE_MODE))
            {
            c->velocity+=c->force/c->mass;
            c->velocity*=c->damping;
            c->position+=c->velocity;
            }
        if(c->mode & CELL_MASTER_MODE)
            {
            c->companion->force=c->force;
            c->companion->velocity=c->velocity;
            c->companion->position=c->position;
            }
        }
}

////////////////////////////////////////////////////////////////////////////
// Member function name:
//      setdamping(float x1,float x2,float y1,float y2,float damping)
//
// Functionality:
//      Sets the damping value of each cell to the value 'damping' over the
//      region specified. Note that the coordinate system is relative to a
//      bounding box surrounding the instrument and, although x and y are
//      normalised to be between 0 and 1, this coordinate system differs
//      from the one used to access a point within an instrument and
//      is described in section 5.4.4.
//
// Returns:
//      A reference to the cell for whom the function was invoked via the
//      C++ special variable 'this'.
//
// Local variables:
//      i1, i2, j1, j2:
//                  j1=bottom row number, j2=top row number, i1=left cell number
//                  and i2=right cell number.
//      imin, imax: column numbers where damped region begins and ends
//                  respectively. If an instrument is some shape other than
//                  rectangular then these are measured, in cells, relative
//                  to a bounding box surrounding the instrument where column
//                  0 is the left hand extremity of the instrument and
//                  column xmax is the right hand extremity.
//
// Instrument class member variables:
//      xmax, ymax, rows.
////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::setdamping(float x1,float x2,float y1,float y2,float damping)
    {
    int i1, i2, j1, j2, imin, imax;
    register i, j;

    i1=(int)(x1*xmax);
    i2=(int)(x2*xmax);
    j1=(int)(y1*ymax);
    j2=(int)(y2*ymax);

    for (j=j1;j<=j2;j++)
        {
        imin=rows[j].offset;
        imax=rows[j].offset+rows[j].xmax;

        for (i=i1;i<=i2;i++)
            {
            if (i>=imin && i<=imax)
                {
                rows[j].cells[i-imin].damping=damping;
                }
            }
        }
    return *this;
    }


////////////////////////////////////////////////////////////////////////////
// Member function name:
//      setdamping(float left, float right, float damping)
//
// Functionality:
//      Version for one dimensional instruments where only the left and
//      right ends of the damped region need to be specified still in
//      normalised coordinates between 0 and 1.
//
// Returns:
//      A reference to the cell for whom the function was invoked.
```

```
//
// Local variables:
//      i1, i2:     i1=left cell number and i2=right cell number.
//      imin, imax: column numbers where damped region begins and ends
//                  respectively. If an instrument is some shape other than
//                  rectangular then these are measured, in cells, relative
//                  to a bounding box surrounding the instrument where column
//                  0 is the left hand extremity of the instrument and
//                  column xmax is the right hand extremity.
//
// Instrument class member variables:
//      xmax, rows.
////////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::setdamping(float left, float right, float damping)
    {
    int i1, i2, imin, imax;
    register i;

    i1=(int)(left*xmax);
    i2=(int)(right*xmax);

    imin=rows[0].offset;
    imax=rows[0].offset+rows[0].xmax;

    for (i=i1;i<=i2;i++)
        {
        if (i>=imin && i<=imax)
            {
            rows[0].cells[i-imin].damping=damping;
            }
        }
    return *this;
    }


////////////////////////////////////////////////////////////////////////////////
// Member function name:
//      setdamping(float position, float damping)
//
// Functionality:
//      Version for one dimensional instruments which only allows the damping
//      to be set at a single point, not over a region.
//
// Returns:
//      A reference to the cell for whom the function was invoked.
//
// Local variables:
//      i:          i=cell number affected.
//
// Instrument class member variable:
//      rows.
////////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::setdamping(float position, float damping)
    {
    int i;

    i=(int)(position*rows[0].xmax);
    rows[0].cells[i].damping=damping;

    return *this;
    }


////////////////////////////////////////////////////////////////////////////////
// Member function name:
//      setdamping(float damping)
//
// Functionality:
//      Sets the damping value of every cell within an instrument to 'damping'
//
// Returns:
//      A reference to the cell for whom the function was invoked.
//
// Instrument class member function:
//      setdamping(float x1, float x2, float y1, float y2, damping).
////////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::setdamping(float damping)
    {
    setdamping(0.0, 1.0, 0.0, 1.0, damping);
    return *this;
    }


////////////////////////////////////////////////////////////////////////////////
```

```
// Member function names:
//      resetdamping(float x1, float x2, float y1, float y2)
//      resetdamping(float left, float right)
//      resetdamping(float position)
//      resetdamping()
//
// Functionality:
//      Equivalent to the setdamping family of functions above, but reset the
//      damping value back to Instrument::default_damping.
//
// Returns:
//      A reference to the cell for whom the function was invoked.
//
// Instrument class member variable:
//      default_damping.
//
// Instrument class member function:
//      setdamping(float x1, float x2, float y1, float y2, damping).
////////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::resetdamping(float x1,float x2,float y1,float y2)
    {
    setdamping(x1, x2, y1, y2, default_damping);
    return *this;
    }

Instrument &Instrument::resetdamping(float left, float right)
    {
    setdamping(left, right, 0.0, 0.0, default_damping);
    return *this;
    }

Instrument &Instrument::resetdamping(float position)
    {
    setdamping(position, position, 0.0, 0.0, default_damping);
    return *this;
    }

Instrument &Instrument::resetdamping()
    {
    setdamping(0.0, 1.0, 0.0, 1.0, default_damping);
    return *this;
    }

////////////////////////////////////////////////////////////////////////////////
// Member function names:
//      setdamping(float x1, float x2, float y1, float y2, float decay)
//      setdamping(float left, float right, float decay)
//      setdamping(float decay)
//
// Functionality:
//      Equivalent to the setdamping family of functions above, but set the
//      damping value in terms of a decay time measured in seconds. At the
//      moment the results of setting a particular decay time for a particular
//      region are not fully understood and these functions need to be updated
//      to provide more predictable results for any shape of size of
//      instrument and any region of damping.
//
// Returns:
//      A reference to the cell for whom the function was invoked.
//
// Instrument class member functions:
//      setdamping(float x1, float x2, float y1, float y2, damping),
//      decay2damping(float decay).
////////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::setdecay(float x1,float x2,float y1,float y2,float decay)
    {
    setdamping(x1, x2, y1, y2, decay2damping(decay));
    return *this;
    }

Instrument &Instrument::setdecay(float left, float right, float decay)
    {
    setdamping(left, right, 0.0, 0.0, decay2damping(decay));
    return *this;
    }

Instrument &Instrument::setdecay(float decay)
    {
    setdamping(0.0, 1.0, 0.0, 1.0, decay2damping(decay));
    return *this;
    }

////////////////////////////////////////////////////////////////////////////////
```

```
// Member function names:
//      resetdecay(float x1, float x2, float y1, float y2)
//      resetdecay(float left, float right)
//      resetdecay()
//
// Functionality:
//      Equivalent to the resetdamping family of functions above, included
//      only for consistency and compatibility.
//
// Returns:
//      A reference to the cell for whom the function was invoked.
//
// Instrument class member variable:
//      default_damping.
//
// Instrument class member function:
//      setdamping(float x1, float x2, float y1, float y2, damping).
////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::resetdecay(float x1,float x2,float y1,float y2)
    {
    setdamping(x1, x2, y1, y2, default_damping);
    return *this;
    }


Instrument &Instrument::resetdecay(float left, float right)
    {
    setdamping(left, right, 0.0, 0.0, default_damping);
    return *this;
    }

Instrument &Instrument::resetdecay()
    {
    setdamping(0.0, 1.0, 0.0, 1.0, default_damping);
    return *this;
    }


////////////////////////////////////////////////////////////////////////////
// Member function name:
//      vibrato(float rate, float depth)
//
// Functionality:
//      Applies a sinusoidal vibrato of frequency 'rate' hertz to an
//      instrument. The depth is given as a proportion of the fundamental
//      frequency of the instrument and the vibrato is achieved by modulating
//      the masses of all the cells. Only works for slight modulations as
//      making the masses too small makes the model become unstable.
//
// Returns:
//      A reference to the cell for whom the function was invoked.
//
// Local variables:
//      c:              pointer to current cell.
//      i, j:           j=row number, i=cell number in that row.
//      base_freq:      base frequency of instrument.
//      new_freq:       frequency required due to vibrato modulation.
//      compensation_factor:
//                      factor to multiply the cell masses by.
//
// Instrument class member variables:
//      rows, ymax, xmax, default_mass
//
// External functions:
//      sin, powf, log10f (from <math.h>)
////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::vibrato(float rate, float depth)
    {
    Cell *c;
    register i, j;
    float base_freq, new_freq, compensation_factor;
    extern float Time; // from main.cc

actual_freq=Hz2CellConst/(xmax+1);
new_freq=actual_freq*(1.0+(depth*sin(rate*Time*6.2831853)));
compensation_factor=powf(4.0, log10f(actual_freq/new_freq)/log10f(2.0));

for (j=0;j<=ymax;j++)
    {
    for (i=0, c=rows[j].cells;i<=rows[j].xmax;i++, c++)
        {
        c->mass=Instrument::default_mass*compensation_factor;
        }
    }

return *this;
}
```

```
////////////////////////////////////////////////////////////////////////////
// Member function name:
//      lock(float x, float y)
//
// Functionality:
//      Locks a single point at (x,y) on an instrument.
//
// Returns:
//      A reference to the cell for whom the function was invoked.
//
// Local variables:
//      i, j:       j=row number, i=cell number in that row.
//
// Instrument class member variables:
//      rows.
////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::lock(float x, float y)
    {
    int i, j;

    j=(int)(y*ymax);
    i=(int)(x*rows[j].xmax);

    rows[j].cells[i].mode |= CELL_LOCK_MODE;

    return *this;
    }


////////////////////////////////////////////////////////////////////////////
// Member function name:
//      lock(float x1, float x2, float y1, float y2)
//
// Functionality:
//      Locks a rectangular region. Similar to setdamping(x1, x2, y1, y2, ...)
//      in the coordinate system used to specify the region.
//
// Returns:
//      A reference to the cell for whom the function was invoked.
//
// Local variables:
//      i1, i2, j1, j2:     j1=bottom row number, j2=top row number,
//                          i1=left cell number, i2=right cell number.
//      imin, imax:         minimum and maximum values of i respectively
//                          for the instrument in question.
//      i, j:               j=row number, i=cell number.
//
// Instrument class member variables:
//      rows.
////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::lock(float x1, float x2, float y1, float y2)
    {
    int i1, i2, j1, j2, imin, imax;
    register i, j;

    i1=(int)(left*xmax);
    i2=(int)(right*xmax);
    j1=(int)(bottom*ymax);
    j2=(int)(top*ymax);

    for (j=j1;j<=j2;j++)
        {
        imin=rows[j].offset;
        imax=rows[j].offset+rows[j].xmax;

        for (i=i1;i<=i2;i++)
            {
            if (i>=imin && i<=imax)
                {
                rows[j].cells[i-imin].mode |= CELL_LOCK_MODE;
                }
            }
        }
    return *this;
    }


////////////////////////////////////////////////////////////////////////////
// Member function names:
//      lockleft():         locks the leftmost cells in an instrument
//      lockright():        locks the rightmost cells in an instrument
//      locktop():          locks the topmost cells in an instrument
//      lockbottom():       locks the bottommost cells in an instrument
//      lockcorners():      only meaningful for rectangular and triangular
//                          instruments.
//      lockperimeter():    self-explanatory
```

```
//      lockends():         designed for 1D instruments.
//
// Functionality:
//      Lock various specific regions of an instrument.
//
// Return:
//      A reference to the cell for whom the function was invoked.
//
// Local variables:
//      i, j:               j=row number, i=cell number.
//
// Instrument class member variables:
//      rows.
////////////////////////////////////////////////////////////////////////////////

Instrument &Instrument::lockleft()
    {
    register j;

    for(j=0;j<=ymax;j++)
        if(rows[j].offset==0)
            rows[j].cells[0].mode |= CELL_LOCK_MODE;

    return *this;
    }

Instrument &Instrument::lockright()
    {
    register j;

    for(j=0;j<=ymax;j++)
        if(rows[j].offset+rows[j].xmax==xmax)
            rows[j].cells[rows[j].xmax].mode |= CELL_LOCK_MODE;

    return *this;
    }

Instrument &Instrument::locktop()
    {
    register i;

    for(i=0;i<=rows[ymax].xmax;i++)
        rows[ymax].cells[i].mode |= CELL_LOCK_MODE;

    return *this;
    }

Instrument &Instrument::lockbottom()
    {
    register i;

    for(i=0;i<=rows[0].xmax;i++)
        rows[0].cells[i].mode |= CELL_LOCK_MODE;

    return *this;
    }


Instrument &Instrument::lockperimeter()
    {
    register j;

    locktop();
    lockbottom();

    for(j=0;j<=ymax;j++)
        rows[j].cells[0].mode |= CELL_LOCK_MODE;

    for(j=0;j<=ymax;j++)
        rows[j].cells[rows[j].xmax].mode |= CELL_LOCK_MODE;

    return *this;
    }

Instrument &Instrument::lockcorners()
    {
    lock(0.0, 0.0);
    lock(1.0, 0.0);
    lock(0.0, 1.0);
    lock(1.0, 1.0);
    return *this;
    }

Instrument &Instrument::lockends()
    {
    lockleft();
    lockright();
    return *this;
```

```
    }

////////////////////////////////////////////////////////////////////////////
// Member function names:
//      glue(Instrument &i1, float x1, float y1,        // glue 2D to 2D
//              Instrument &i2, float x2, float y2)
//      glue(Instrument &i1, float x1, float y1,        // glue 2D to 1D
//              Instrument &i2, float x2)
//      glue(Instrument &i1, float x1,                  // glue 1D to 2D
//              Instrument &i2, float x2, float y2)
//      glue(Instrument &i1, float x1,                  // glue 1D to 1D
//              Instrument &i2, float x2)
//
// Functionality:
//      Given two instruments and sets of coordinates for selecting two
//      cells, glues them and their corresponding neighbours together.
//
// Return:
//      A reference to the cell for whom the function was invoked.
//
// Instrument class member function:
//      glue_cells(Cell *c1, Cell *c2).
////////////////////////////////////////////////////////////////////////////

void Instrument::glue(Instrument &i1, float x1, float y1,
                      Instrument &i2, float x2, float y2)
    {
    Instrument::glue_cells(&i1(x1, y1), &i2(x2, y2));
    Instrument::glue_cells(i1(x1, y1).east, i2(x2, y2).east);
    Instrument::glue_cells(i1(x1, y1).west, i2(x2, y2).west);
    Instrument::glue_cells(i1(x1, y1).north, i2(x2, y2).north);
    Instrument::glue_cells(i1(x1, y1).south, i2(x2, y2).south);
    Instrument::glue_cells(i1(x1, y1).neast, i2(x2, y2).neast);
    Instrument::glue_cells(i1(x1, y1).nwest, i2(x2, y2).nwest);
    Instrument::glue_cells(i1(x1, y1).seast, i2(x2, y2).seast);
    Instrument::glue_cells(i1(x1, y1).swest, i2(x2, y2).swest);
    }

void Instrument::glue(Instrument &i1, float x1, float y1,
                      Instrument &i2, float x2)
    {
    Instrument::glue_cells(&i1(x1, y1), &i2(x2));
    Instrument::glue_cells(i1(x1, y1).east, i2(x2).east);
    Instrument::glue_cells(i1(x1, y1).west, i2(x2).west);
    }

void Instrument::glue(Instrument &i1, float x1,
                      Instrument &i2, float x2, float y2)
    {
    Instrument::glue_cells(&i1(x1), &i2(x2, y2));
    Instrument::glue_cells(i1(x1).east, i2(x2, y2).east);
    Instrument::glue_cells(i1(x1).west, i2(x2, y2).west);
    }

void Instrument::glue(Instrument &i1, float x1,
                      Instrument &i2, float x2)
    {
    Instrument::glue_cells(&i1(x1), &i2(x2 ));
    Instrument::glue_cells(i1(x1).east, i2(x2).east);
    Instrument::glue_cells(i1(x1).west, i2(x2).west);
    }


////////////////////////////////////////////////////////////////////////////
// Member function name:
//      glue_cells(Cell *c1, Cell *c2)
//
// Functionality:
//      Given pointers to two cells, glues the cells together.
////////////////////////////////////////////////////////////////////////////

void Instrument::glue_cells(Cell *c1, Cell *c2)
    {
    if (!c1 || !c2) return;

    c1->companion=c2;
    c2->companion=c1;
    c1->mode |= CELL_MASTER_MODE;
    c2->mode |= CELL_SLAVE_MODE;
    }


////////////////////////////////////////////////////////////////////////////
// Member function name:
//      join(Instrument &i1, float x1, float y1,
//           Instrument &i2, float x2, float y2)
```

```
//
// Functionality:
//      Joins two pieces of material with straight edges by effectively
//      installing a new set of springs to sew the two instruments together
//      so that they act as one. There are eight different cases:-
//
//      (1) if x1=0: if x2=0: join the left of i1 to the left of i2
//                   if x2=1: join the left of i1 to the right of i2
//      (2) if x1=1: if x2=0: join the right of i1 to the left of i2
//                   if x2=1: join the right of i1 to the right of i2
//      (3) if y1=0: if y2=0: join the bottom of i1 to the bottom of i2
//                   if y2=1: join the bottom of i1 to the top of i2
//      (2) if y1=1: if y2=0: join the top of i1 to the bottom of i2
//                   if y2=1: join the top of i1 to the top of i2
//
//      If we are joining horizontally then y1 and y2 serve to specify a
//      centre line at which the joining should begin and conversely if we
//      are joining vertically x1 and x2 specify a centre line. Once again
//      x1, x2, y1 and y2 are all specified as instrument coordinates.
//      For an explanation of the join parameters see section 5.4.7.
//      Note that i1 and i2 can refer to the same instrument making it
//      possible to construct cylindrical and toroidal instruments from a
//      rectangular sheet.
//
// Instrument class member function:
//      join_left_to_left(Cell &cell1, Cell &cell2),
//      join_left_to_right(Cell &cell1, Cell &cell2),
//      join_right_to_left(Cell &cell1, Cell &cell2),
//      join_right_to_right(Cell &cell1, Cell &cell2),
//      join_bottom_to_bottom(Cell &cell1, Cell &cell2),
//      join_bottom_to_top(Cell &cell1, Cell &cell2),
//      join_top_to_bottom(Cell &cell1, Cell &cell2),
//      join_top_to_top(Cell &cell1, Cell &cell2),
////////////////////////////////////////////////////////////////////////////

void Instrument::join(Instrument &i1, float x1, float y1,
                      Instrument &i2, float x2, float y2)
    {
    if (x1==0.0)
        {
        if (x2==0.0)
            {
            Instrument::join_left_to_left(i1(x1, y1), i2(x2, y2));
            }
        else if (x2==1.0)
            {
            Instrument::join_left_to_right(i1(x1, y1), i2(x2, y2));
            i2.worldx=i1.worldx-(i2.xmax+1);
            i2.worldy=(int)(i1.worldy+i1.ymax*y1-i2.ymax*y2);
            }
        }
    else if (x1==1.0)
        {
        if (x2==0.0)
            {
            Instrument::join_right_to_left(i1(x1, y1), i2(x2, y2));
            i2.worldx=i1.worldx+(i1.xmax+1);
            i2.worldy=(int)(i1.worldy+i1.ymax*y1-i2.ymax*y2);
            }
        else if (x2==1.0)
            {
            Instrument::join_right_to_right(i1(x1, y1), i2(x2, y2));
            }
        }
    else if (y1==0.0)
        {
        if (y2==0.0)
            {
            Instrument::join_bottom_to_bottom(i1(x1, y1), i2(x2, y2));
            }
        else if (y2==1.0)
            {
            Instrument::join_bottom_to_top(i1(x1, y1), i2(x2, y2));
            i2.worldx=(int)(i1.worldx+i1.xmax*x1-i2.xmax*x2);
            i2.worldy=i1.worldy-(i2.ymax+1);
            }
        }
    else if (y1==1.0)
        {
        if (y2==0.0)
            {
            Instrument::join_top_to_bottom(i1(x1, y1), i2(x2, y2));
            i2.worldx=(int)(i1.worldx+i1.xmax*x1-i2.xmax*x2);
            i2.worldy=i1.worldy+(i1.ymax+1);
            }
        else if (y2==1.0)
            {
            Instrument::join_top_to_top(i1(x1, y1), i2(x2, y2));
```

```
        }
    }
}

////////////////////////////////////////////////////////////////////////////////
// Member function names:
//      join_left_to_left(Cell &cell1, Cell &cell2)
//      join_left_to_right(Cell &cell1, Cell &cell2)
//      join_right_to_left(Cell &cell1, Cell &cell2)
//      join_right_to_right(Cell &cell1, Cell &cell2)
//      join_bottom_to_bottom(Cell &cell1, Cell &cell2)
//      join_bottom_to_top(Cell &cell1, Cell &cell2)
//      join_top_to_bottom(Cell &cell1, Cell &cell2)
//      join_top_to_top(Cell &cell1, Cell &cell2)
//
// Functionality:
//      Join two pieces of material with straight edges by effectively
//      installing a new set of springs to sew the two instruments together
//      so that they act as one. Joining starts at the two cells specified
//      and migrates along the edges of the two pieces of material in one
//      direction until a boundary is reached. Then back to the starting
//      cells to migrate in the opposite direction. For a more detailed
//      explanation see section 5.4.7.
////////////////////////////////////////////////////////////////////////////////

void Instrument::join_left_to_left(Cell &cell1, Cell &cell2)
    {
    Cell *c1=&cell1, *c2=&cell2;

// migrate northwards until a boundary is reached.

    while (c1 && c2)
        {
        c1->west=c2;
        c1->nwest=c2->north;
        c1->swest=c2->south;
        c2->west=c1;
        c2->nwest=c1->north;
        c2->swest=c1->south;

        c1=c1->north; if (c1==&cell1) break;
        c2=c2->north; if (c2==&cell2) break;
        }

    if (c1) c1->swest=c1->south->west;
    if (c2) c2->swest=c2->south->west;

    c1=&cell1; c2=&cell2;

// back to starting position and migrate southwards

    while (c1 && c2)
        {
        c1->west=c2;
        c1->nwest=c2->north;
        c1->swest=c2->south;
        c2->west=c1;
        c2->nwest=c1->north;
        c2->swest=c1->south;

        c1=c1->south; if (c1==&cell1) break;
        c2=c2->south; if (c2==&cell2) break;
}

    if (c1) c1->nwest=c1->north->west;
    if (c2) c2->nwest=c2->north->west;
    }

void Instrument::join_left_to_right(Cell &cell1, Cell &cell2)
    {
    Cell *c1=&cell1, *c2=&cell2;

// migrate northwards until a boundary is reached.

    while (c1 && c2)
        {
        c1->west=c2;
        c1->nwest=c2->north;
        c1->swest=c2->south;
        c2->east=c1;
        c2->neast=c1->north;
        c2->seast=c1->south;

        c1=c1->north; if (c1==&cell1) break;
        c2=c2->north; if (c2==&cell2) break;
        }
```

```
    if (c1) c1->swest=c1->south->west;
    if (c2) c2->seast=c2->south->east;

    c1=&cell1; c2=&cell2;

// back to starting position and migrate southwards

    while (c1 && c2)
        {
        c1->west=c2;
        c1->nwest=c2->north;
        c1->swest=c2->south;
        c2->east=c1;
        c2->neast=c1->north;
        c2->seast=c1->south;

        c1=c1->south; if (c1==&cell1) break;
        c2=c2->south; if (c2==&cell2) break;
        }

    if (c1) c1->nwest=c1->north->west;
    if (c2) c2->neast=c2->north->east;
    }


void Instrument::join_right_to_left(Cell &cell1, Cell &cell2)
    {
    Cell *c1=&cell1, *c2=&cell2;

// migrate northwards until a boundary is reached.

    while (c1 && c2)
        {
        c1->east=c2;
        c1->neast=c2->north;
        c1->seast=c2->south;
        c2->west=c1;
        c2->nwest=c1->north;
        c2->swest=c1->south;

        c1=c1->north; if (c1==&cell1) break;
        c2=c2->north; if (c2==&cell2) break;
        }

    if (c1) c1->seast=c1->south->east;
    if (c2) c2->swest=c2->south->west;

    c1=&cell1; c2=&cell2;

// back to starting position and migrate southwards

    while (c1 && c2)
        {
        c1->east=c2;
        c1->neast=c2->north;
        c1->seast=c2->south;
        c2->west=c1;
        c2->nwest=c1->north;
        c2->swest=c1->south;

        c1=c1->south; if (c1==&cell1) break;
        c2=c2->south; if (c2==&cell2) break;
        }

    if (c1) c1->neast=c1->north->east;
    if (c2) c2->nwest=c2->north->west;
    }


void Instrument::join_right_to_right(Cell &cell1, Cell &cell2)
    {
    Cell *c1=&cell1, *c2=&cell2;

// migrate northwards until a boundary is reached.

    while (c1 && c2)
        {
        c1->east=c2;
        c1->neast=c2->north;
        c1->seast=c2->south;
        c2->east=c1;
        c2->neast=c1->north;
        c2->seast=c1->south;

        c1=c1->north; if (c1==&cell1) break;
        c2=c2->north; if (c2==&cell2) break;
        }
```

```
    if (c1) c1->seast=c1->south->east;
    if (c2) c2->seast=c2->south->east;

    c1=&cell1; c2=&cell2;

// back to starting position and migrate southwards

    while (c1 && c2)
        {
        c1->east=c2;
        c1->neast=c2->north;
        c1->seast=c2->south;
        c2->east=c1;
        c2->neast=c1->north;
        c2->seast=c1->south;

        c1=c1->south; if (c1==&cell1) break;
        c2=c2->south; if (c2==&cell2) break;
        }

    if (c1) c1->neast=c1->north->east;
    if (c2) c2->neast=c2->north->east;
    }


void Instrument::join_bottom_to_bottom(Cell &cell1, Cell &cell2)
    {
    Cell *c1=&cell1, *c2=&cell2;

// migrate eastwards until a boundary is reached

    while (c1 && c2)
        {
        c1->south=c2;
        c1->seast=c2->east;
        c1->swest=c2->west;
        c2->south=c1;
        c2->seast=c1->east;
        c2->swest=c1->west;

        c1=c1->east; if (c1==&cell1) break;
        c2=c2->east; if (c2==&cell2) break;
        }

    if (c1) c1->swest=c1->west->south;
    if (c2) c2->swest=c2->west->south;

// back to starting position and migrate westwards

    c1=&cell1; c2=&cell2;

    while (c1 && c2)
        {
        c1->south=c2;
        c1->seast=c2->east;
        c1->swest=c2->west;
        c2->south=c1;
        c2->seast=c1->east;
        c2->swest=c1->west;

        c1=c1->west; if (c1==&cell1) break;
        c2=c2->west; if (c2==&cell2) break;
        }

    if (c1) c1->seast=c1->east->south;
    if (c2) c2->seast=c2->east->south;
    }


void Instrument::join_bottom_to_top(Cell &cell1, Cell &cell2)
    {
    Cell *c1=&cell1, *c2=&cell2;

// migrate eastwards until a boundary is reached

    while (c1 && c2)
        {
        c1->south=c2;
        c1->seast=c2->east;
        c1->swest=c2->west;
        c2->north=c1;
        c2->neast=c1->east;
        c2->nwest=c1->west;

        c1=c1->east; if (c1==&cell1) break;
        c2=c2->east; if (c2==&cell2) break;
        }
```

```
    if (c1) c1->swest=c1->west->south;
    if (c2) c2->nwest=c2->west->north;

    c1=&cell1; c2=&cell2;

// back to starting position and migrate westwards

    while (c1 && c2)
        {
        c1->south=c2;
        c1->seast=c2->east;
        c1->swest=c2->west;
        c2->north=c1;
        c2->neast=c1->east;
        c2->nwest=c1->west;

        c1=c1->west; if (c1==&cell1) break;
        c2=c2->west; if (c2==&cell2) break;
        }

    if (c1) c1->seast=c1->east->south;
    if (c2) c2->neast=c2->east->north;
    }


void Instrument::join_top_to_bottom(Cell &cell1, Cell &cell2)
    {
    Cell *c1=&cell1, *c2=&cell2;

// migrate eastwards until a boundary is reached

    while (c1 && c2)
        {
        c1->north=c2;
        c1->neast=c2->east;
        c1->nwest=c2->west;
        c2->south=c1;
        c2->seast=c1->east;
        c2->swest=c1->west;

        c1=c1->east; if (c1==&cell1) break;
        c2=c2->east; if (c2==&cell2) break;
        }

    if (c1) c1->nwest=c1->west->north;
    if (c2) c2->swest=c2->west->south;

    c1=&cell1; c2=&cell2;

// back to starting position and migrate westwards

    while (c1 && c2)
        {
        c1->north=c2;
        c1->neast=c2->east;
        c1->nwest=c2->west;
        c2->south=c1;
        c2->seast=c1->east;
        c2->swest=c1->west;

        c1=c1->west; if (c1==&cell1) break;
        c2=c2->west; if (c2==&cell2) break;
        }

    if (c1) c1->neast=c1->east->north;
    if (c2) c2->seast=c2->east->south;
    }


void Instrument::join_top_to_top(Cell &cell1, Cell &cell2)
    {
    Cell *c1=&cell1, *c2=&cell2;

// migrate eastwards until a boundary is reached

    while (c1 && c2)
        {
        c1->north=c2;
        c1->neast=c2->east;
        c1->nwest=c2->west;
        c2->north=c1;
        c2->neast=c1->east;
        c2->nwest=c1->west;

        c1=c1->east; if (c1==&cell1) break;
        c2=c2->east; if (c2==&cell2) break;
        }
```

```
    if (c1) c1->neast=c1->east->north;
    if (c2) c2->neast=c2->east->north;

    c1=&cell1; c2=&cell2;

// back to starting position and migrate westwards

    while (c1 && c2)
        {
        c1->north=c2;
        c1->neast=c2->east;
        c1->nwest=c2->west;
        c2->north=c1;
        c2->neast=c1->east;
        c2->nwest=c1->west;

        c1=c1->west; if (c1==&cell1) break;
        c2=c2->west; if (c2==&cell2) break;
        }

    if (c1) c1->nwest=c1->west->north;
    if (c2) c2->nwest=c2->west->north;
    }


////////////////////////////////////////////////////////////////////////////
// Member function name:
//      display()
//
// Functionality:
//      Displays the instrument in the graphics window at a position
//      determined by worldx, worldy, graphx, graphy, winoriginx & winoriginy.
//
// Instrument class member variables:
//      rows, graphx, graphy, worldx, worldy, amplification,
//      global_amplification.
//
// External variables: (all from file main.cc)
//      winoriginx, winoriginy, xscale, yscale, skewfactor,
//      graphics_on, graphics_update_step, Sample.
//
// External functions:
//      bgnline(), endline(), v2s():
//                  begin line and end line and vertex functions. see <gl.h>
//      color():    sets graphics colour. BLACK, WHITE, CYAN, MAGENTA, BLUE,
//                  GREEN, YELLOW, RED allowed.
////////////////////////////////////////////////////////////////////////////

void Instrument::display()
    {
    register short i, j;
    Cell *c;
    float ss;
    short v[2];
    int left,bottom;

    left=winoriginx+graphx;
    bottom=winoriginy+graphy;

    color(BLACK);
    linewidth(1);

    for(j=ymax;j>=0;j--)    // draw horizontal lines through rows of cells
        {
        bgnline();
        for(i=0, c=rows[j].cells;i<=rows[j].xmax;i++, c++)
            {
            ss=c->position;
            if (c->damping < default_damping) color(BLUE);
            else color(BLACK);
            v[0]=(short)(left+(worldx+i+rows[j].offset)*xscale+(worldy+j)*skewfactor);
            v[1]=(short)(bottom+(worldy+j)*yscale+ss*amplification*global_amplification);
            v2s(v);
            }
        endline();
        }

color(BLACK);

if (ymax>0)     // if instrument is 2D, draw line round perimeter
    {
    linewidth(1);

    bgnline();

    for(i=0, c=rows[0].cells;i<=rows[0].xmax;i++, c++)  // across bottom
        {
```

```
        ss=c->position;
        v[0]=(short)(left+(worldx+i+rows[0].offset)*xscale+worldy*skewfactor);
        v[1]=(short)(bottom+worldy*yscale+ss*amplification*global_amplification);
        v2s(v);
        }

    for(j=0;j<=ymax;j++)     // up right
        {
        c=&rows[j].cells[rows[j].xmax];
        ss=c->position;
        v[0]=(short)(left+(worldx+rows[j].xmax+rows[j].offset)*xscale+(worldy+j)*skewfactor);
        v[1]=(short)(bottom+(worldy+j)*yscale+ss*amplification*global_amplification);
        v2s(v);
        }

    for(i=rows[ymax].xmax;i>=0;i--)     // across top
        {
        c=&rows[ymax].cells[i];
        ss=c->position;
        v[0]=(short)(left+(worldx+i+rows[ymax].offset)*xscale+(worldy+ymax)*skewfactor);
        v[1]=(short)(bottom+(worldy+ymax)*yscale+ss*amplification*global_amplification);
        v2s(v);
        }

    for(j=ymax;j>=0;j--)     // down left
        {
        c=&rows[j].cells[0];
        ss=c->position;
        v[0]=(short)(left+(worldx+rows[j].offset)*xscale+(worldy+j)*skewfactor);
        v[1]=(short)(bottom+(worldy+j)*yscale+ss*amplification*global_amplification);
        v2s(v);
        }

    endline();
    }

for(j=0;j<=ymax;j++)     // scan cells again to mark any
    {                            // locked or glued ones

    for(i=0, c=rows[j].cells;i<=rows[j].xmax;i++, c++)
        {
        ss=c->position;
        v[0]=(short)(left+(worldx+i+rows[j].offset)*xscale+(worldy+j)*skewfactor);
        v[1]=(short)(bottom+(worldy+j)*yscale+ss*amplification*global_amplification);
        if(c->mode & CELL_LOCK_MODE)
            {
            color(BLACK);       // mark locked cells in black
            circfs(v[0], v[1], 2);
            }

        if(c->companion)
            {
            color(RED);         // mark glued cells in red
            circfs(v[0], v[1], 3);
            }
        }
    }
}


////////////////////////////////////////////////////////////////////////////
// Member function names:
//      calculate_forces(),
//      update_positions(),
//      display_all().
//
// Functionality:
//      Cause all instruments to be updated by scanning the linked list
//      and calling the appropriate member functions for each instrument.
//
// Local variable:
//      i: current instrument.
////////////////////////////////////////////////////////////////////////////

void Instrument::calculate_forces()
    {
    for (Instrument *i=Instrument::list;i;i=i->next)
    i->calculate_my_forces();
    }


void Instrument::update_positions()
    {
    for (Instrument *i=Instrument::list;i;i=i->next)
    i->update_my_position();
    }
```

```
void Instrument::display_all()
    {
    for (Instrument *i=Instrument::list;i;i=i->next)
    i->display();
    }
```

G.1.5 File String.h

```
////////////////////////////////////////////////////////////////////////////
// File name: String.h                          (c) 1996 Mark Pearson
//
// Content: Definition of String object class.
//
// Member variables:
//              none.
//
// Class String is derived from class Instrument. It has no member variables
// or member functions, only a constructor which knows how to create a one
// dimensional piece of material.
////////////////////////////////////////////////////////////////////////////

#ifndef STRING_H
#define STRING_H

#include "Cell.h"

#ifndef String
#define String not_gl_String
#endif

class String : public Instrument
    {
public:
    String(float freq, float decay);
    };

#endif
```

G.1.6 File String.cc

```
////////////////////////////////////////////////////////////////////////////
// File name: String.cc                         (c) 1996 Mark Pearson
//
// Content: Definition of String constructor function.
//
// Constructor name:  String(float freq, float decay)
//
// Arguments:
//      Fundamental frequency of string in hertz which determines the
//      length, and decay time in seconds.
//
// Local variables:
//      xsize:      size of instrument in x direction, measured in cells.
//
// Instrument class member variables:
//      xmax, ymax, rows, next.
//
// Instrument class member functions:
//      add_to_global_list()
//      initialise_cells()
//      link_cells()
//      hertz2cells(frequency)
////////////////////////////////////////////////////////////////////////////

#include "Instrument.h"
#include "String.h"


// This is necessary because there is a 'String' type in the graphics
// library.

#ifndef String
#define String not_gl_String
#endif


String::String(float freq, float decay)
    : Instrument(freq, 0.0, decay)
    {
    int xsize=hertz2cells(freq), ysize=1;
    xmax=xsize-1;
    ymax=0;
```

```
    rows=new Row[ysize];
    next=NULL;

    rows[0].xmax=xsize-1;
    rows[0].offset=0;
    rows[0].cells=new Cell[xsize];

    add_to_global_list();
    initialise_cells();
    link_cells();
    }
```

G.1.7 File Circle.h

```
////////////////////////////////////////////////////////////////////////////////
// File name: Circle.h                              (c) 1996 Mark Pearson
//
// Content: Definition of Circle object class.
//
// Member variables:
//              none.
//
// Class Circle is derived from class Instrument. It has no member
// variables or member functions, only a constructor which knows how to
// create a circular sheet of material.
////////////////////////////////////////////////////////////////////////////////

#ifndef CIRCLE_H
#define CIRCLE_H

#include "Cell.h"

class Circle : public Instrument
    {
public:
    Circle(float freq, float decay);
    };

#endif
```

G.1.8 File Circle.cc

```
////////////////////////////////////////////////////////////////////////////////
// File name: Circle.cc                             (c) 1996 Mark Pearson
//
// Content: Definition of Circle constructor function.
//
// Constructor name:  Circle(float diameter_freq, float decay)
//
// Arguments:
//      frequency of circular sheet in hertz. This determines the diameter.
//      Also decay time in seconds.
//
// Local variables:
//      j:      current row number
//      x, y:       j (0..ymax) is translated into y (-yradius..+yradius)
//                  which is used to calculate x from the circle equation
//                  x^2 + y^2 = r^2.
//      radius:     measured in cells.
//      local_xmax: xmax for current row. (see file Instrument.h for xmax)
//      offset:     offset of current row needed to place it in the correct
//                  position relative to all the other rows.
//      xsize:      overall size of instrument in x direction, measured in
//                  cells.
//      ysize:      overall size of instrument in y direction, measured in
//                  rows.
//
// Instrument class member variables:
//      xmax, ymax, rows, next.
//
// Instrument class member functions:
//      add_to_global_list()
//      initialise_cells()
//      link_cells()
//      hertz2cells(frequency)
////////////////////////////////////////////////////////////////////////////////

#include <math.h>
#include "Instrument.h"
#include "Circle.h"
#include <iostream.h>

Circle::Circle(float diameter_freq, float decay)
```

```
: Instrument(diameter_freq, diameter_freq, decay)
{
register j;
float x, y, radius;
int local_xmax, offset;

int xsize, ysize=hertz2cells(diameter_freq);    // diameter measured in cells

xmax=0;
ymax=ysize-1;                    // the cirle is 'diameter' cells high.

rows=new Row[ysize];             // create the right number of rows
next=NULL;

radius=ysize/2.0;

    for (j=0;j<ysize;j++)                    // create one row at a time starting
{                                // from bottom.
        y=j-(ysize-1.0)/2.0;                 // as j goes from 0 to ymax calculate
        x=sqrt(radius*radius-y*y);           // y for circle equation. Then x
        xsize=((int)(x+0.5))*2;              // round x up/down to nearest integer
        local_xmax=xsize-1;
        offset=(ysize-xsize)/2;              // the row is the correct length but
        rows[j].xmax=local_xmax;             // must be offset relative to the
        rows[j].offset=offset;               // bounding box.
        if(xmax < local_xmax+offset) xmax=local_xmax+offset;
                                 // keep track of longest row
        rows[j].cells=new Cell[xsize];  // create 'xsize' new cells
        }

    add_to_global_list();,       // These functions are from the Instrument
    initialise_cells();          // base class.
    link_cells();                // install springs between all the cells.
    }
```

G.1.9 File Rectangle.h

```
////////////////////////////////////////////////////////////////////////////
// File name: Rectangle.h                    (c) 1996 Mark Pearson
//
// Content: Definition of Rectangle object class.
//
// Member variables:
//              none.
//
// Class Rectangle is derived from class Instrument. It has no member
// variables or member functions, only a constructor which knows how to create
// a rectangular sheet of material.
////////////////////////////////////////////////////////////////////////////

#ifndef RECTANGLE_H
#define RECTANGLE_H

#include "Cell.h"

class Rectangle : public Instrument
    {
public:
    Rectangle(float xfreq, float yfreq, float decay);
    Cell &at(float x, float y) {
        return rows[(int)(ymax*y)].cells[(int)(xmax*x)];
        }
    };

#endif
```

G.1.10 File Rectangle.cc

```
////////////////////////////////////////////////////////////////////////////
// File name: Rectangle.cc                   (c) 1996 Mark Pearson
//
// Content: Definition of Rectangle constructor function.
//
// Constructor name:  Rectangle(float xfreq, float yfreq, float decay)
//
// Arguments:
//      x and y frequencies of sheet, which determine the width and height
//      measured in cells. Also decay time measured in seconds.
//
// Local variables:
//      j:      current row number
//      xsize:      size of instrument in x direction, measured in cells.
//      ysize:      size of instrument in y direction, measured in rows.
```

```
//
// Instrument class member variables:
//      xmax, ymax, rows, next.
//
// Instrument class member functions:
//      add_to_global_list()
//      initialise_cells()
//      link_cells()
//      hertz2cells(frequency)
////////////////////////////////////////////////////////////////////////////////

#include "Instrument.h"
#include "Rectangle.h"

Rectangle::Rectangle(float xfreq, float yfreq, float decay)
    : Instrument(xfreq, yfreq, decay)
    {
    register j;

    int xsize=hertz2cells(xfreq), ysize=hertz2cells(yfreq);
    xmax=xsize-1;
    ymax=ysize-1;

    rows=new Row[ysize];
    next=NULL;

    for (j=0;j<ysize;j++)
        {
        rows[j].xmax=xmax;
        rows[j].offset=0;
        rows[j].cells=new Cell[xsize];
        }

    add_to_global_list();
    initialise_cells();
    link_cells();
    }
```

G.1.11 File Triangle.h

```
////////////////////////////////////////////////////////////////////////////////
// File name: Triangle.h                    (c) 1996 Mark Pearson
//
// Content: Definition of Triangle object class.
//
// Member variables:
//              none.
//
// Class Triangle is derived from class Instrument. It has no member
// variables or member functions, only a constructor which knows how to
// create a triangular sheet of material which a vertically straight edge
// on the right hand side and a vertex opposite it on the left hand side.
////////////////////////////////////////////////////////////////////////////////

#ifndef TRIANGLE_H
#define TRIANGLE_H

#include "Cell.h"

class Triangle : public Instrument
    {
public:
    Triangle(float xfreq, float yfreq, float decay);
    };

#endif
```

G.1.12 File Triangle.cc

```
////////////////////////////////////////////////////////////////////////////////
// File name: Triangle.cc                   (c) 1996 Mark Pearson
//
// Content: Definition of Triangle constructor function.
//
// Constructor name:  Triangle(float xfreq, float yfreq, float decay)
//
// Arguments:
//      x and y frequencies of instrument and decay time.
//
// Local variables:
//      j:      current row number
//      x:          used in calculating how long each row should be.
//      local_xmax: xmax for current row (length - 1).
```

```
//      local_xsize: length of current row in cells.
//      offset:      offset of current row needed to place it in the correct
//                   position relative to all the other rows.
//      xsize:       overall size of instrument in x direction, measured in
//                   cells.
//      ysize:       overall size of instrument in y direction, measured in
//                   rows.
//
// Instrument class member variables:
//      xmax, ymax, rows, next.
//
// Instrument class member functions:
//      add_to_global_list()
//      initialise_cells()
//      link_cells()
//      hertz2cells(frequency)
////////////////////////////////////////////////////////////////////////////
#include "Instrument.h"
#include "Triangle.h"
#include "iostream.h"

Triangle::Triangle(float xfreq, float yfreq, float decay)
    : Instrument(xfreq, yfreq, decay)
    {
    register j;
    float x;
    int local_xsize, local_xmax, offset;

    int xsize=hertz2cells(xfreq), ysize=hertz2cells(yfreq);
    xmax=xsize-1;
    ymax=ysize-1;

    rows=new Row[ysize];
    next=NULL;

    for (j=0;j<ysize;j++)
        {
        if(j<ysize/2) x=xsize*2.0*(j+1)/ysize;
        if(j>=ysize/2) x=xsize*2.0*(ysize/2.0-(j-ysize/2.0))/ysize;
        local_xsize=(int)(x+0.5);
        local_xmax=local_xsize-1;
        offset=xsize-local_xmax;
        rows[j].xmax=local_xmax;
        rows[j].offset=offset;
        if(xmax < local_xmax+offset) xmax=local_xmax+offset;
        rows[j].cells=new Cell[local_xsize];
        }

    add_to_global_list();
    initialise_cells();
    link_cells();
    }
```

G.1.13 File Ellipse.h

```
////////////////////////////////////////////////////////////////////////////
// File name: Ellipse.h                    (c) 1996 Mark Pearson
//
// Content: Definition of Ellipse object class.
//
// Member variables:
//              none.
//
// Class Ellipse is derived from class Instrument. It has no member
// variables or member functions, only a constructor which knows how to
// create a elliptical sheet of material.
////////////////////////////////////////////////////////////////////////////

#ifndef ELLIPSE_H
#define ELLIPSE_H

#include "Cell.h"

class Ellipse : public Instrument
    {
public:
    Ellipse(float xfreq, float yfreq, float decay);
    };

#endif
```

G.1.14 File `Ellipse.cc`

```
////////////////////////////////////////////////////////////////////////////////
// File name: Ellipse.cc                          (c) 1996 Mark Pearson
//
// Content: Definition of Ellipse constructor function.
//
// Constructor name:  Ellipse(float xfreq, float yfreq, float decay)
//
// Arguments:
//      X and y frequencies in hertz, which determine the size of the
//      sheet and decay time measured in seconds.
//
// Local variables:
//      j:      current row number
//      x, y:         j (0..ymax) is translated into y (-yradius..+yradius)
//                    which is used to calculate x from the equation of an
//                    ellipse x^2/a + y^2/b = c^2.
//      xradius, yradius.
//      a, b:         a=xradius^2, b=yradius^2.
//      local_xmax: xmax for current row. (see file Instrument.h for xmax)
//      local_xsize: number of cells in current row.
//      offset:       offset of current row needed to place it in the correct
//                    position relative to all the other rows.
//      xsize:        overall size of instrument in x direction, measured in
//                    cells.
//      ysize:        overall size of instrument in y direction, measured in
//                    rows.
//
// Instrument class member variables:
//      xmax, ymax, rows, next.
//
// Instrument class member functions:
//      add_to_global_list()
//      initialise_cells()
//      link_cells()
////////////////////////////////////////////////////////////////////////////////

#include <math.h>
#include "Instrument.h"
#include "Ellipse.h"
#include <iostream.h>

Ellipse::Ellipse(float xfreq, float yfreq, float decay)
    : Instrument(xfreq, yfreq, decay)
    {
    register j;
    float x, y, xradius, yradius;
    float a, b;
    int local_xmax, local_xsize, offset;
    int xsize=hertz2cells(xfreq), ysize=hertz2cells(yfreq);

    xmax=0;
    ymax=ysize-1;

    rows=new Row[ysize];
    next=NULL;

    xradius=xsize/2.0;a=xradius*xradius;
    yradius=ysize/2.0;b=yradius*yradius;

    for (j=0;j<ysize;j++)
        {
        y=j-(ysize-1.0)/2.0;
        x=sqrt(a*(1.0-y*y/b));
        local_xsize=((int)(x+0.5))*2;
        local_xmax=local_xsize-1;
        if (local_xmax>xmax) xmax=local_xmax;
        offset=(xsize-local_xsize)/2;
        rows[j].xmax=local_xmax;
        rows[j].offset=offset;
        if(xmax < local_xmax+offset) xmax=local_xmax+offset;
        rows[j].cells=new Cell[xsize];
        }

    add_to_global_list();
    initialise_cells();
    link_cells();
    }
```

G.1.15 File `Microphone.h`

```
////////////////////////////////////////////////////////////////////////////////
// File name:   Microphone.h                      (c) 1996 Mark Pearson
//
```

```
// Content:     Definition of Microphone object class.
//
// Member variables:
//      source:         microphones come in two types, those which have their
//                      source or sources defined at the time of declaration,
//                      and those which, given floating point expressions
//                      in the score, write the results of these expressions
//                      to a file. The variable 'source' determines whether the
//                      microphone takes its signal from pre-defined cells or
//                      from expressions. It takes one of the two values:
//                      'from_cells' or 'from_expressions'.
//      index:          index into sample buffer. When sample buffer is full,
//                      it is written to 'outputfile'.
//      num_channels:   number of channels. Limited to 1 or 2 in current
//                      implementation but relatively trivial to change.
//      buffer:         pointer to floating point sample buffer.
//      filename:       pointer to name of file to which raw floating point
//                      sound samples are written.
//      outputfile:     output file stream to which samples are written.
//      leftsource:     pointer to cell which is source for samples when
//                      microphone is mono, or left samples when microphone
//                      is stereo.
//      rightsource:    pointer to cell which is source for right samples
//                      when microphone is stereo.
//      leftsample:     floating point value which is either mono sample or
//                      left sample in a stereo microphone.
//      rightsample:    floating point value which is right sample in a
//                      stereo microphone.
//      next:           pointer to next microphone in linked list.
//
// Static member variables:
// list, current: pointer to head of linked list of microphones, and
//                      pointer to current microphone during updating.
//
////////////////////////////////////////////////////////////////////////////////

#ifndef MICROPHONE_H
#define MICROPHONE_H

#include <fstream.h>
#include "Cell.h"

#define stereo 2
#define mono 1

class Microphone
    {
public:
    static const buffersize;                    // defined in file Microphone.cc
    Microphone(const char *soundfilename, int channels);
    Microphone(const char *file, Cell &l, Cell &r);
    Microphone(const char *file, Cell &c);
    Microphone &setleft(Cell &l) {leftsource=&l;return *this;}
    Microphone &setright(Cell &r) {rightsource=&r;return *this;}
    Microphone &leftout(float value) {leftsample=value;return *this;}
    Microphone &rightout(float value) {rightsample=value;return *this;}
    Microphone &output(float value) {leftsample=value;return *this;}
    void update();
    static void update_all();

private:
    void add_to_global_list()
        {
        if (list==NULL) list=this;
        else current->next=this;

        current=this;
        }

    enum
        {
        from_cells, from_expressions
        };

    int source, index, num_channels;
    float *buffer;
    char *filename;
    ofstream outputfile;
    Cell *leftsource;
    Cell *rightsource;
    float leftsample;
    float rightsample;
    Microphone *next;
    static Microphone *list, *current;
    };

#endif
```

G.1.16 File Microphone.cc

```
////////////////////////////////////////////////////////////////////////////
// File name: Microphone.cc                    (c) 1996 Mark Pearson
//
// Content: Definition of Microphone class member functions
////////////////////////////////////////////////////////////////////////////

#include <strstream.h>
#include <fstream.h>
#include <string.h>
#include "Microphone.h"
#include "Instrument.h"
#include "Cell.h"

Microphone *Microphone::list=NULL, *Microphone::current=NULL;
const Microphone::buffersize=5000;

////////////////////////////////////////////////////////////////////////////
// Constructor name:
//      Microphone(const char *soundfilename, int channels)
//
// Functionality:
//      Creates a microphone object whose sound samples will be sent to
//      a file called '/var/tmp/<name>.tao'. The microphone writes 'channels'
//      channels of output (1 or 2 in the present implementation). No
//      decision is made at declaration time about the actual sources for the
//      sound samples. This is left to be determined by the member functions
//      leftout() & rightout() described in file Microphone.h, and update()
//      described below. In practice leftout() and rightout() are called
//      within the score.
//
// Arguments:
//      Pointer to a string of characters representing <name> and number of
//      channels (only 1 or 2 in present implementation).
//
// Instrument class member variables:
//      source, index, num_channels, filename, buffer,
//      next, leftsource, rightsource.
//
// Local variables:
//      tempname:       if filename points to the string 'file1' then
//                      tempname will point to the string '/var/tmp/file1.tao'
////////////////////////////////////////////////////////////////////////////

Microphone::Microphone(const char *soundfilename, int channels)
    {
    source=from_expressions;
    index=0;
    num_channels=channels;
    filename=new char[50];
    buffer=new float[buffersize];
    next=NULL;
    leftsource=NULL;
    rightsource=NULL;

    ostrstream tempname(filename, 50);
    tempname << "/var/tmp/" << soundfilename << ".tao" << ends;

    outputfile.open(filename, ios::trunc);
    outputfile.close();

    add_to_global_list();
    }


////////////////////////////////////////////////////////////////////////////
// Constructor name:
//      Microphone(const char *sfname, Cell &l, Cell &r)
//
// Functionality:
//      Similar to above function except that sound sources in the form
//      of references to two cells are given. This automatically determines
//      that num_channels=2.
//
// Arguments:
//      Pointer to a string of characters representing <name> and
//      references to two cells which will provide samples for the left
//      and right channels of output. The cells' positions are used to
//      generate the samples.
//
// Instrument class member variables:
//      source, index, num_channels, filename, buffer,
//      next, leftsource, rightsource.
//
// Local variables:
```

```
//      tempname:    '/var/tmp/<name>.tao'
////////////////////////////////////////////////////////////////////////////

Microphone::Microphone(const char *sfname, Cell &l, Cell &r)
    {
    source=from_cells;
    index=0;
    num_channels=2;
    filename=new char[50];
    buffer=new float[buffersize];
    next=NULL;
    leftsource=&l;
    rightsource=&r;

    ostrstream tempname(filename, 50);
    tempname << "/var/tmp/" << sfname << ".tao" << ends;

    outputfile.open(filename, ios::trunc);
    outputfile.close();

    add_to_global_list();
    }


////////////////////////////////////////////////////////////////////////////
// Constructor name:
//      Microphone(const char *sfname, Cell &c)
//
// Functionality:
//      Mono version of constructor function described above.
////////////////////////////////////////////////////////////////////////////

Microphone::Microphone(const char *sfname, Cell &c)
    {
    source=from_cells;
    index=0;
    num_channels=1;
    filename=new char[50];
    buffer=new float[buffersize];
    next=NULL;
    leftsource=&c;
    rightsource=NULL;

    ostrstream tempname(filename, 50);
    tempname << "/var/tmp/" << sfname << ".tao" << ends;

    outputfile.open(filename, ios::trunc);
    outputfile.close();

    add_to_global_list();
    }


////////////////////////////////////////////////////////////////////////////
// Member function name:
//      update()
//
// Functionality:
//      causes sound samples to be written to the sample buffer. If the
//      buffer is full then it is written to the output file stream
//      outputfile and index is set to 0. Otherwise index is incremented by
//      num_channels.
//
// Instrument class member variables:
//      source, buffer, index, num_channels, leftsource, rightsource,
//      leftsample, rightsample.
//
// External variables:
//      Sample:      counter which keeps track of how many time steps have
//                   elapsed since the beginning of a performance.
//      bandwidthlevel:
//                   an integer which specifies how often samples are
//                   generated, i.e. on every time step, on every other
//                   time step, on every third time step etc. The higher
//                   the value the better the frequency response of the
//                   synthesis model, but the worse the computational
//                   burden. This should not be altered from its default
//                   value of 2.
////////////////////////////////////////////////////////////////////////////

void Microphone::update()
    {
    extern long Sample; // from 'main.cc'
    extern bandwidthlevel;

    if (Sample%bandwidthlevel!=0) return; // Throw away samples

    if (index<buffersize)
```

```
{
if (source==from_cells)
    {
    if (num_channels==2)
            {
            buffer[index++]=leftsource->position;
            buffer[index++]=rightsource->position;
            }
        if (num_channels==1)
            {
            buffer[index++]=leftsource->position;
            }
        }

    if (source==from_expressions)
        {
        if (num_channels==2)
            {
            buffer[index++]=leftsample;
            buffer[index++]=rightsample;
            }
        if (num_channels==1)
            {
            buffer[index++]=leftsample;
            }
        }

    if (index==buffersize)
        {
        outputfile.open(filename, ios::app);
        outputfile.write((unsigned char *)buffer, (int)(buffersize*sizeof(float)));
        outputfile.close();
        index=0;
        }
    }


////////////////////////////////////////////////////////////////////////////
// Member function name:
//      update_all()
//
// Functionality:
//      starts at the head of the linked list of microphones and updates
//      them all, one by one.
////////////////////////////////////////////////////////////////////////////

void Microphone::update_all()
    {
    for(Microphone *m=list;m;m=m->next) m->update();
    }
```

G.1.17 File main.cc

```
////////////////////////////////////////////////////////////////////////////
// File name: main.cc                          (c) 1996 Mark Pearson
//
// Content:
//      Global variables and functions, macros for translating TAO script
//      into a C++ code fragment and main function.
//
// Purpose:
//      A TAO script is actually a fragment of C++ code in clever disguise.
//      Instrument, microphone and parameter declarations in a TAO script
//      are translated directly into C++ variable declarations of the
//      appropriate type. The score control structures are translated into
//      C++ control structures and most of the other code such as mathematical
//      expressions are left exactly as they appear in the script. Part of
//      this translation process is carried out by the UNIX sed command which,
//      given a set of scripts, pattern matches and replaces strings of
//      characters in an input stream. The sed scripts are contained
//      in files c4, c5, c6, c7, string_sed, rectangle_sed, ellipse_sed
//      circle_sed, triangle_sed. The file 'tao' contains a short UNIX script
//      which causes the TAO script to be translated, #included into the
//      main function defined later in this file, and compiled. For a full
//      explanation of the implementation of both the synthesis model and
//      script language at a less code specific level, see appendices
//      D and E.
//
// Global variables:
//      audiorate:  sample rate of output, currently fixed at 44.1 KHz.
//      modelrate, bandwidthlevel:
//              if audio samples are generated on every time step of
//              the synthesis engine, the material's frequency response
//              is not very good. To improve the situation (at the
```

```
//              expense of more computational power) we can sample
//              instruments on every other time step. We have to make
//              the instruments twice as big though to achieve the same
//              fundamental frequency, but in effect the spatial and
//              temporal resolution of the model is increased leading
//              to sounds with more high frequency clarity.
//              modelrate = audiorate * bandwidthlevel. These variables
//              should not be changed as other constants are affected
//              such as Hertz2CellConst and Decay2DampingConst (both from
//              Instrument.h).
//
// Macros used in implementation of score language:
//      a, b, INTERVAL, TIME all measured in seconds.
//
//      FromTo(a,b):            intermediate translation of 'From .. to' and
//                              'At .. for' TAO control structures.
//      Before(a), After(a):    translations of 'Before' and 'After' control
//                              structures.
//      Every(INTERVAL):        translation of 'Every' control structure
//      ControlRate(DIVIDEBY): translation of 'ControlRate' control structure.
//      At(TIME):               translation of 'At' control structure
//      If(CONDITION):          translation of 'If' control structure
//      Elseif(CONDITION):      translation of 'Elseif'
//      linear(y1,y2):          returns a time varying value which changes
//                              linearly from y1 to y2 over the time interval
//                              specified by the variables 'start' and 'end'.
//      expon(y1,y2):           returns a time varying value which changes
//                              exponentially from y1 to y2 over the time
//                              interval specified by the variables 'start'
//                              and 'end'.
//      Parameter:              Parameter x,y simply translates to float x,y
//
//      Score(duration):        A much more ugly macro not intended for human
//                              consumption! Basically sets up a C++ for loop
//                              with the number of iterations determined by
//                              duration. TAO makes use of the fact that a
//                              C++ for loop of the form:
//
//                                for ( init ; cond ; step )
//
//                              can have multiple items seperated by commas
//                              in each part. This is a sneaky way of inserting
//                              all the code to update instruments and mic's,
//                              update the graphics etc. into the 'head' of the
//                              for loop leaving the body of the loop free to
//                              take the body of the TAO score.
////////////////////////////////////////////////////////////////////////////////


#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <strstream.h>
#include <gl.h>
#include <gl/device.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>

#ifndef String
#define String not_gl_String
#endif

#include "Instrument.h"
#include "String.h"
#include "Rectangle.h"
#include "Circle.h"
#include "Triangle.h"
#include "Ellipse.h"
#include "Microphone.h"

// macros for units of measurement

#define Hz *1.0
#define secs *1.0
#define min *60.0
#define msecs /1000.0
#define samples /44100.0

// these variables are described at the head of this file and should be
// left well alone.

float audiorate=44100.0;
int bandwidthlevel=2;
float modelrate=audiorate*bandwidthlevel;
```

```
////////////////////////////////////////////////////////////////////////////
// Macros used in implementation of score language:
//      Once the half-translated TAO script file contained in the file
//      'tao_scriptfile' has been #included into the main function, these
//      macros perform any further translation needed to make the script
//      into a fully compilable fragment of C++ code.
////////////////////////////////////////////////////////////////////////////

#define FromTo(a, b) if(Sample<=(long)((END=(b))*modelrate) && \
                                        Sample>=(long)((START=(a))*modelrate))
#define Before(a) if(Sample<(long)(START=start, (END=(a))*modelrate))
#define After(a) if(Sample>=(long)(END=end, (START=(a))*modelrate))
#define Every(INTERVAL) if(Sample%(long)(INTERVAL*modelrate)==0)
#define ControlRate(DIVIDEBY) if(Sample%(long)DIVIDEBY==0)
#define At(TIME) if(Sample==(long)(TIME*modelrate))
#define AtSample(TIME) if(Sample==TIME)
#define If(CONDITION) if(CONDITION)
#define Elseif(CONDITION) else if(CONDITION)
#define Else else
#define When(RHYTHM) if(RHYTHM)

#define linear(y1, y2) ((Time-start)/(end-start)*(y2-y1)+y1)
#define expon(y1, y2) ((float)(y1)*fexp(1.0/(end-start)*flog((float)(y2)/ \
                                                (float)(y1))*(Time-start)))

#define Parameter float

#define Display cout <<"   "<<setw(0)<<setprecision(4)<<setiosflags(ios::fixed)<<
#define newline '\n'<<flush
#define sameline '\r'<<flush
#define Flag int
#define Position Cell &


////////////////////////////////////////////////////////////////////////////
// The 'Score' macro is a mess to look at but was only ever intended to be
// a temporary measure for the prototype TAO system. The first thing on the
// agenda if any further work is done to the system is to replace the UNIX
// sed scripts and all of these macros with a proper TAO language parser.
// This would cleanly and robustly translate a TAO script with error checking.
// At the moment error checking is only provided at the level of C++ compiler
// errors and thus only detects syntax errors in the translated version of the
// TAO script, which the user should not be concerned with.
////////////////////////////////////////////////////////////////////////////

#define Score(duration) \
NumSamples=(long)(duration*modelrate); \
cout << "Calculating " << NumSamples << " samples\n"; \
startstack[1]=start=0.0; endstack[1]=end=duration; \
startstack[0]=start; endstack[0]=end; \
START=start;END=end; \
for(Sample=0, Time=0.0; \
graphics_on?(color(WHITE),1):0, \
graphics_on?(clear(), 1):0, \
Instrument::calculate_forces(), \
update_graphics(), \
Sample<=NumSamples; \
(graphics_on&&(Sample%graphics_update_step==0))?(Instrument::display_all(), 1):0, \
(graphics_on&&(Sample%graphics_update_step==0))?(swapbuffers(),1):0, \
Instrument::update_positions(), \
Microphone::update_all(), \
Sample++,Time=Sample/modelrate)


////////////////////////////////////////////////////////////////////////////
// Global function names:
// randomi(int low, int high)
// random(float low, float high)
// pitch(float value)
// pitch(const char *note)
//
// Described below.
////////////////////////////////////////////////////////////////////////////

int randomi(int low, int high);
float random(float low, float high);
float pitch(const char *note);
float pitch(float value);


////////////////////////////////////////////////////////////////////////////
// Global function name:
// randomi(int low, int high)
//
// Functionality:
// Returns a random integer between low and high inclusive.
////////////////////////////////////////////////////////////////////////////
```

```
int randomi(int low, int high)
    {
    return (low+(random()%(high-low+1)));
    }


////////////////////////////////////////////////////////////////////////////
// Global function name:
// random(float low, float high)
//
// Functionality:
// Returns a random floating point number between low and high inclusive.
////////////////////////////////////////////////////////////////////////////
float random(float low, float high)
    {
    return (float)randomi((int)(low*100000),(int)(high*100000))/100000.0;
    }


////////////////////////////////////////////////////////////////////////////
// Global function name:
// pitch(float value)
//
// Functionality:
// Takes a decimal value of the form <octave>.<semitone> and returns
//      a frequency in hertz. For example pitch(8.00) -> 261.6 Hz or middle C,
//      pitch(8.09) -> 440 Hz or A above middle C, pitch(7.09) -> 220 Hz.
////////////////////////////////////////////////////////////////////////////

float pitch(float value)
    {
    float octave, semitone, frequency;

    octave = ftrunc(value);
    semitone = value - octave;
    value = octave + semitone * 100.0 / 12.0;

    frequency = pow(2, value - 8.0) * 261.6;
    return frequency;
    }


////////////////////////////////////////////////////////////////////////////
// Global function name:
// pitch(const char *note)
//
// Functionality:
// Takes a string of characters representing a note name and
//      returns a frequency in hertz. For example:-
//
//          pitch("C8")  -> 261.6 Hz or middle C.
//          pitch("C#8") -> frequency of C sharp above middle C.
//          pitch("A8")  -> 440 Hz or A above middle C.
//          pitch("Eb6") -> frequency of E flat in second octave below
//                          middle C.
//          pitch("F#8+1/3") -> frequency of F sharp + 1/3 of a semitone
//                          in middle C octave.
////////////////////////////////////////////////////////////////////////////

float pitch(const char *note)
    {
    int l=strlen(note);
    float octave, semitone, frequency, value;
    int charno=0;

    switch(note[charno++])
        {
    case 'C' : semitone=0.00;break;
    case 'D' : semitone=0.02;break;
    case 'E' : semitone=0.04;break;
    case 'F' : semitone=0.05;break;
    case 'G' : semitone=0.07;break;
    case 'A' : semitone=0.09;break;
    case 'B' : semitone=0.11;break;
    }

if(note[charno]=='b')
    {
    semitone-=0.01;
    charno++;
    }
else if(note[charno]=='#')
    {
    semitone+=0.01;
    charno++;
```

```
        }

    octave=1.0*(note[charno++]-'0');
    if(note[charno]>='0' && note[charno]<='9')
octave=octave*10.0+1.0*(note[charno++]-'0');

    int plus=FALSE, minus=FALSE;

    if(note[charno]=='+' || note[charno]=='-')
        {
        float dividend=0.0;

        if(note[charno]=='+') plus=TRUE;
        if(note[charno]=='-') minus=TRUE;
        charno++;

        while(note[charno]>='0' && note[charno]<='9')
            {
            dividend*=10.0;
            dividend+=(float)(note[charno]-'0');
            charno++;
            }

        if(note[charno++]!='/')
            cerr << "Pitch error: / expected" << endl;

        float divisor=0.0;

        while(note[charno]>='0' && note[charno]<='9')
            {
            divisor*=10.0;
            divisor+=(float)(note[charno]-'0');
            charno++;
            }

        if(plus) semitone+=dividend/(divisor*100.0);
        if(minus) semitone-=dividend/(divisor*100.0);
        }

value = octave + semitone * 100.0 / 12.0;

frequency = pow(2, value - 8.0) * 261.6;
return frequency;
}

////////////////////////////////////////////////////////////////////////////////
// Global variables used by graphics:
//      mousex, mousey:
//          current mouse position in screen coords with origin at bottom left
//          of graphics window.
//      mdev[2]:
//          mdev[0] is device MOUSEX, x position of mouse.
//          mdev[1] is device MOUSEY, y position of mouse.
//      mval[2]:
//          mval[0] is value of device MOUSEX
//          mval[1] is value of device MOUSEY
//      lastval[2]:
//          keeps track of previous position of mouse when polled.
//      org[2]:
//          x and y coordinates of bottom left of graphics window relative to
//          screen origin.
//      size[2]:
//          x and y dimensions of graphics window in pixels.
//      middleflag, rightflag:
//          used to keep track of whether middle or right mouse buttons
//          were pressed when previously polled, i.e. to see if there has
//          been any change of state.
//      winoriginx, winoriginy:
//          origin of TAO's window coordinate system relative to the graphics
//          window origin. For example, increasing winoriginx by 100 moves the
//          whole TAO graphical animation 100 pixels to the right.
//      skewfactor:
//          instruments are displayed in oblique projection. This variable
//          determines how skewed the instruments appear. A value of 0
//          displays a rectangle as a rectangle etc.
//      xscale, yscale:
//          number of pixels between successive cells in the x and y
//          directions. Current values of skewfactor, xscale and yscale seem
//          to produce a clear visual representation.
//      drag:
//          flag determining whether the the graphical image was being dragged
//          with the left mouse button last time it was polled.
////////////////////////////////////////////////////////////////////////////////

int graphics_on = FALSE;

#define X 0
```

```
#define Y 1
#define XY 2
short mousex, mousey;
short mval[XY], lastval[XY];
Device mdev[XY];
long org[XY], size[XY];
int middleflag=FALSE, rightflag=FALSE;
int winoriginx=200, winoriginy=200;
float skewfactor=0.5, xscale=4.0, yscale=3.0;
int drag=FALSE;


//////////////////////////////////////////////////////////////////////////////
// Global function name:
// graphics_init()
//
// Functionality:
// Initialises graphics system, opens a window entitled 'TAO graphical
//      output'. Sets doublebuffer mode for animation and clears the screen
//      to white.
//////////////////////////////////////////////////////////////////////////////

void graphics_init()
    {
    char win_name[]="TAO graphical output";

    prefsize(1000,700);
    winopen(win_name);
    doublebuffer();
    gconfig();
    color(WHITE);
    clear();swapbuffers();
    color(WHITE);
    clear();
    }

int graphics_update_step=1;

float Time=0.0;          // time elapsed since beginning of performance
                         // in seconds.
ostrstream timestream;   // used to create a string of characters
                         // representing the time elapsed.


//////////////////////////////////////////////////////////////////////////////
// Global function name:
// update_graphics()
//
// Functionality:
// Everything associated with the graphics window save actually drawing
//      the instruments. There are a number of mouse functions provided:
//
//      Holding the left mouse button down and moving the mouse in the
//      graphics window causes the whole graphics image be dragged about.
//      Useful for instruments which are too big to fit on the screen.
//
//      Holding left mouse button down and pressing middle mouse button
//      causes 'graphics_update_step' to be multiplied by a factor of 5. The
//      graphics window is updated on every 'graphics_update_step'th time step
//      of the synthesis engine. If graphics_update_step=500 then it becomes
//      1 again.
//
//      Holding left mouse button down and pressing right mouse button
//      causes graphics_update_step to be divided by a factor of 5. If it
//      is already 1 then the animation is frozen until left mouse + right
//      mouse are pressed again.
//
//      Elapsed time in seconds since beginning of performance is displayed
//      at bottom left of graphics window.
//
// Local variables:
//      lastmousex, lastmousey:
//          the coordinates of the previous mouse position when dragging
//          the image with the left mouse button.
//
// External functions:
//      getsize(), origin(), cmov2i(), color(), charstr(), getbutton()
//      all provided in SGI graphics library and declared in <gl.h>.
//      See IRIX 5.3 man pages for explanations of their functionality.
//////////////////////////////////////////////////////////////////////////////

void update_graphics()
    {
    static int lastmousex=0, lastmousey=0;

    if (graphics_on)
        {
        getsize(&size[X], &size[Y]);
```

```
    getorigin(&org[X], &org[Y]);
    }

if (graphics_on)
    {
    cmov2i(20, 20);
    color(BLACK);
    timestream <<setw(0)<<setprecision(4)<<setiosflags(ios::fixed);
    timestream << "Elapsed time=" << Time << " seconds";
    charstr(timestream.str());
    timestream.seekp(0, ostream::beg);
    }

if (graphics_on && getbutton(LEFTMOUSE) && getbutton(MIDDLEMOUSE))
    {
    if (!middleflag)
        if (graphics_update_step==500)
            graphics_update_step=1;
        else graphics_update_step*=5;
    middleflag=TRUE;
    }
else if (middleflag) middleflag=FALSE;

if (graphics_on && getbutton(LEFTMOUSE) && getbutton(RIGHTMOUSE))
    {
    if (!rightflag)
        if (graphics_update_step==1)
            {
            while(getbutton(RIGHTMOUSE));
            while(!(getbutton(LEFTMOUSE) && getbutton(RIGHTMOUSE)));
            while(getbutton(LEFTMOUSE));
            }
        else graphics_update_step/=5;
    rightflag=TRUE;
    }
else if (rightflag) rightflag=FALSE;

if (graphics_on && !drag && getbutton(LEFTMOUSE))
    {
    drag=TRUE;
    getdev(XY, mdev, mval);
    mousex=mval[X]-org[X];
    mousey=mval[Y]-org[Y];
    lastmousex=mousex;
    lastmousey=mousey;
    }

    if (graphics_on && drag && getbutton(LEFTMOUSE))
        {
        getdev(XY, mdev, mval);
        mousex=mval[X]-org[X];
        mousey=mval[Y]-org[Y];
        winoriginx+=mousex-lastmousex;
        winoriginy+=mousey-lastmousey;
        lastmousex=mousex;
        lastmousey=mousey;
        }

    if (graphics_on && drag && !getbutton(LEFTMOUSE)) drag=FALSE;
    }


////////////////////////////////////////////////////////////////////////////////
// Global function name:
//      main()
//
// Functionality:
//      The user compiles a TAO script 'example.script' by typing:
//
//          'tao example'
//
//      which causes the script to be translated into an intermediate form
//      stored in the file 'tao_scriptfile'.
//
//      This is #included into the main function below and once further
//      processed by the C++ macros #defined at the beginning of this file,
//      it is now a fragment of executable C++ code. The user's instrument,
//      microphone and parameter declarations translate directly into
//      C++ variable declarations and other TAO language features such as
//      the score control structures, screen output, mathematical expressions
//      etc. translate into equivalent C++ language features. Once the C++
//      preprocessor has done its stuff this file is compiled leading to
//      and executable called 'example.exe' following the example given
//      above.
//
// Local variables:
//      start, end, startstack[], endstack[]:
//          Two special variables 'start' and 'end' are available throughout
```

```
//          the score. Their values depend on context, specifically the
//          times specified in the head of the control structure whose
//          body they appear in. For example 'At 0 secs for 5 secs' leads to
//          start=0 and end=5. The way in which the system keeps track of
//          the different values of start and end is by means of two stacks
//          startstack[] and endstack[]. Every time control passes into the
//          body of a nested control structure the current values of start
//          and end are pushed onto the stacks and new values are calculated.
//          On leaving the body, the old values are popped from the stacks.
//      START, END:
//          Used to store the times specified in the head of a control
//          structure ready to be transferred to start and end once their
//          values have been pushed onto the stacks.
//      n:  start and end stack pointer.
//
// Global variables:
//      Sample, NumSamples:
//          Current sample number (time steps of synthesis engine not
//          sound samples) and total number of samples to synthesise.
//      graphics_on:
//          flag indicating whether to display instrument animations or
//          just proceed with synthesis.
////////////////////////////////////////////////////////////////////////////////

long Sample=0, NumSamples=0;

void main(int argc, char *argv[])
    {
    if (argc==3 && strcmp(argv[1], "-g")==0)
        {
        graphics_on=TRUE;
        Instrument::global_amplification=atof(argv[2]);
        }
    else graphics_on=FALSE;

    if (graphics_on)
        {
        graphics_init();
        getorigin(&org[X], &org[Y]);
        getsize(&size[X], &size[Y]);
        mdev[X]=MOUSEX;
        mdev[Y]=MOUSEY;
        }

    register float start=0.0, end=0.0, START=0.0, END=0.0;
    float startstack[20], endstack[20];
    register int n=0;

    srandom((int)time(0));


#include "tao_scriptfile"

    if (graphics_on) gexit();
    }
```

G.2 C implementation of the `float2aiff` program

G.2.1 File `float2aiff.c`

```
/*****************************************************************************/
/* File name: float2aiff.c                          (c) 1996 Mark Pearson  */
/*                                                                           */
/* Content:     The functions used to convert a raw floating point '.tao'    */
/*              soundfiles into a '.aiff' file.                              */
/*****************************************************************************

#include <math.h>
#include <stdio.h>
#include "audio.h"

#include "Convert.c"

struct form_chunk
    {
    char    ckID[4];
    long    ckSize;
    char    formType[4];
    };

struct comm_chunk
    {
    char    ckID[4];
```

```
    long    ckDataSize;
    short   numChannels;
    long    numSampleFrames;
    short   sampleSize;
    char    sampleRate[10];
    };

struct ssnd_chunk
    {
    char    ckID[4];
    long    ckDataSize;
    long    offset;
    long    blockSize;
    };

#define MONO    1
#define STEREO  2

#define FLOAT sizeof(float)
#define LONG sizeof(long)
#define SHORT sizeof(short)
#define CHAR sizeof(char)

/*************************************************************************/
/* Function name: float_to_stereo_aiff(char *floatfilename,              */
/*                       char *AIFFfilename, float samplerate)           */
/*                                                                       */
/* Functionality:                                                        */
/*      Takes two pointers to char representing the full path names of both */
/*      the '.tao' raw floating point soundfile and the '.aiff' soundfile */
/*      to which the samples will be written, and a numerical value      */
/*      representing the sample rate at which the samples will be played */
/*      back, and performs the conversion.                               */
/*************************************************************************/

float_to_stereo_aiff(char *floatfilename, char *AIFFfilename, float samplerate)
    {
    FILE *floatfile, *AIFFfile;
    float maxsample=0.0;
    float scaleby;
    long number_of_samples=0;
    float floatsample;
    short shortsample;
    char buffer[10];

    struct form_chunk *FORM;
    struct comm_chunk *COMM;
    struct ssnd_chunk *SSND;

    FORM=(struct form_chunk *)malloc(sizeof (struct form_chunk));
    COMM=(struct comm_chunk *)malloc(sizeof (struct comm_chunk));
    SSND=(struct ssnd_chunk *)malloc(sizeof (struct ssnd_chunk));

    printf("Converting floatfile %s into AIFFfile %s\n",
        floatfilename, AIFFfilename);

    floatfile=fopen(floatfilename, "r");

    printf("Checking for maximum sample value...\n");

    while(1)
        {
        fread(&floatsample, FLOAT, 1, floatfile);
        if (feof(floatfile)) break;

        number_of_samples++;
        if (fabsf(floatsample)>maxsample) maxsample=fabsf(floatsample);
        }

        printf("Maximum sample value is %f\nNumber of samples is %ld\n\n",
                                            maxsample, number_of_samples);


        rewind(floatfile);


        AIFFfile=fopen(AIFFfilename, "wb");

    /******************************* FORM stuff ***********************************/

        strncpy(FORM->ckID, "FORM", 4);
        FORM->ckSize =
        sizeof(long) +
        sizeof(struct comm_chunk) +
        sizeof(struct ssnd_chunk) +
        number_of_samples * 2;

        strncpy(FORM->formType, "AIFF", 4);
```

```
    fwrite(FORM->ckID, sizeof(char), 4, AIFFfile);
    fwrite(&FORM->ckSize, sizeof(long), 1, AIFFfile);
    fwrite(FORM->formType, sizeof(char), 4, AIFFfile);

/****************************** COMM stuff ***********************************/

    strncpy(COMM->ckID, "COMM", 4);
    COMM->ckDataSize =
    sizeof(short) + /* numChannels */
    sizeof(long) + /* numSampleFrames */
    sizeof(short) + /* sampleSize */
    10u * sizeof(char); /* IEEE extended sampleRate */

    COMM->numChannels=2;
    COMM->numSampleFrames=number_of_samples/COMM->numChannels;
    COMM->sampleSize=16;

    ConvertToIeeeExtended ( (double)samplerate, COMM->sampleRate );

    fwrite(COMM->ckID, sizeof(char), 4, AIFFfile);
    fwrite(&COMM->ckDataSize, sizeof(long), 1, AIFFfile);
    fwrite(&COMM->numChannels, sizeof(short), 1, AIFFfile);
    fwrite(&COMM->numSampleFrames, sizeof(long), 1, AIFFfile);
    fwrite(&COMM->sampleSize, sizeof(short), 1, AIFFfile);
    fwrite(COMM->sampleRate, sizeof(char), 10, AIFFfile);


/****************************** SSND stuff ***********************************/

    strncpy(SSND->ckID, "SSND", 4);
    SSND->ckDataSize =
    sizeof(long) + /* offset */
    sizeof(long) + /* blockSize */
    COMM->numSampleFrames *
    COMM->numChannels *
    sizeof(short);

    SSND->offset=0;
    SSND->blockSize=0;

    fwrite(SSND, sizeof(struct ssnd_chunk), 1, AIFFfile);

/******   All done, File pointer now points to area for first sample   ******/

    number_of_samples=0;

    printf("Writing samples, please wait\n");

    while(1)
        {
        fread(&floatsample, FLOAT, 1, floatfile);
        if (feof(floatfile)) break;

        shortsample=(short)(floatsample*32000.0/maxsample);
        fwrite(&shortsample, SHORT, 1, AIFFfile);
        fread(&floatsample, FLOAT, 1, floatfile);
        shortsample=(short)(floatsample*32000.0/maxsample);
        fwrite(&shortsample, SHORT, 1, AIFFfile);

        number_of_samples+=2;
        if (number_of_samples%10000==0)
            {printf(".");fflush(stdout);}
        }

    printf("\nDone\n");
    fclose(floatfile);
    fclose(AIFFfile);
    }

main(int argc, char **argv)
    {
    if (argc==4)
        {
        float_to_stereo_aiff(argv[1], argv[2], atof(argv[3]));
        }

    else (fprintf(stderr, "Usage: float2aiff <floatfilename>
                                        <AIFFfilename> <samplerate>\n"));
    }
```

G.2.2 File Convert.c

```
/*
 * C O N V E R T   T O   I E E E   E X T E N D E D
 */
```

```
/* Copyright (C) 1988-1991 Apple Computer, Inc.
 * All rights reserved.
 *
 * Warranty Information
 *  Even though Apple has reviewed this software, Apple makes no warranty
 *  or representation, either express or implied, with respect to this
 *  software, its quality, accuracy, merchantability, or fitness for a
 *  particular purpose.  As a result, this software is provided "as is,"
 *  and you, its user, are assuming the entire risk as to its quality
 *  and accuracy.
 *
 * This code may be used and freely distributed as long as it includes
 * this copyright notice and the above warranty information.
 *
 * Machine-independent I/O routines for IEEE floating-point numbers.
 *
 * NaN's and infinities are converted to HUGE_VAL or HUGE, which
 * happens to be infinity on IEEE machines.  Unfortunately, it is
 * impossible to preserve NaN's in a machine-independent way.
 * Infinities are, however, preserved on IEEE machines.
 *
 * These routines have been tested on the following machines:
 *    Apple Macintosh, MPW 3.1 C compiler
 *    Apple Macintosh, THINK C compiler
 *    Silicon Graphics IRIS, MIPS compiler
 *    Cray X/MP and Y/MP
 *    Digital Equipment VAX
 *
 *
 * Implemented by Malcolm Slaney and Ken Turkowski.
 *
 * Malcolm Slaney contributions during 1988-1990 include big- and little-
 * endian file I/O, conversion to and from Motorola's extended 80-bit
 * floating-point format, and conversions to and from IEEE single-
 * precision floating-point format.
 *
 * In 1991, Ken Turkowski implemented the conversions to and from
 * IEEE double-precision format, added more precision to the extended
 * conversions, and accommodated conversions involving +/- infinity,
 * NaN's, and denormalized numbers.
 */

#ifndef HUGE_VAL
# define HUGE_VAL HUGE
#endif /* HUGE_VAL */

# define FloatToUnsigned(f)  \
    ((unsigned long)(((long)(f - 2147483648.0)) + 2147483647L + 1))


static void
ConvertToIeeeExtended(double num, char *bytes)
{
    int    sign;
    int expon;
    double fMant, fsMant;
    unsigned long hiMant, loMant;

    if (num < 0) {
        sign = 0x8000;
        num *= -1;
    } else {
        sign = 0;
    }

    if (num == 0) {
        expon = 0; hiMant = 0; loMant = 0;
    }
    else {
        fMant = frexp(num, &expon);
        if ((expon > 16384) || !(fMant < 1)) {    /* Infinity or NaN */
            expon = sign|0x7FFF; hiMant = 0; loMant = 0; /* infinity */
        }
        else {    /* Finite */
            expon += 16382;
            if (expon < 0) {    /* denormalized */
                fMant = ldexp(fMant, expon);
                expon = 0;
            }
            expon |= sign;
            fMant = ldexp(fMant, 32);
            fsMant = floor(fMant);
            hiMant = FloatToUnsigned(fsMant);
            fMant = ldexp(fMant - fsMant, 32);
            fsMant = floor(fMant);
            loMant = FloatToUnsigned(fsMant);
        }
```

```
    }

    bytes[0] = expon >> 8;
    bytes[1] = expon;
    bytes[2] = hiMant >> 24;
    bytes[3] = hiMant >> 16;
    bytes[4] = hiMant >> 8;
    bytes[5] = hiMant;
    bytes[6] = loMant >> 24;
    bytes[7] = loMant >> 16;
    bytes[8] = loMant >> 8;
    bytes[9] = loMant;
}
```

G.3 Unix sed scripts used in the translation of a TAO script

G.3.1 File tao_sed_script1

```
##############################################################################
## File: tao_sed_script1                (c) 1996 Mark Pearson
##
## Content:
##     UNIX 'sed' script which removes any linebreaks from microphone output
##     statement 'output'.
##############################################################################

: start
/output:/,/;/ !p
/output:/,/;/ !d

/output:.*;/ b next
/output:/,/;/ H
/output:/ s/\n//g
: next
t start
```

G.3.2 File tao_sed_script2

```
##############################################################################
## File: tao_sed_script2                (c) 1996 Mark Pearson
##
## Content:
##     UNIX 'sed' script which removes any linebreaks from microphone output
##     statements 'leftout' and 'rightout'.
##############################################################################

: start
/tout:/,/;/ !p
/tout:/,/;/ !d

/tout:.*;/ b next
/tout:/,/;/ H
/tout:/ s/\n//g
: next
t start
```

G.3.3 File tao_sed_script3

```
##############################################################################
## File: tao_sed_script3                (c) 1996 Mark Pearson
##
## Content:
##     UNIX 'sed' script which takes any lines contained within the tokens
##     'Display' and ';' and puts them onto a single line. This is necessary
##     since the other sed scripts which process Display statements further,
##     contained in file 'c4' only work on single lines, not across many
##     lines.
##
##     Display "Time=", Time,      -->   Display "Time=", Time, a, b, newline;
##          a, b,
##          newline;
##############################################################################

: start
```

```
/Display/,/;/ !p
/Display/,/;/ !d

/Display.*;/ b next
/Display/,/;/ N
/Display/ s/\n//g
: next
t start
```

G.3.4 File tao_sed_script4

```
#############################################################################
## File: tao_sed_script4                    (c) 1996 Mark Pearson
##
## Content:
##      A TAO script is actually a fragment of C++ code in clever disguise.
##      Instrument, microphone and parameter declarations in a TAO script are
##      translated directly into C++ variable declarations of the appropriate
##      types. The score control structures are translated into C++ control
##      structures and most of the other code such as mathematical expressions
##      are already valid C++ code and are left as they are.
##
##      Part of this translation is performed by the UNIX sed scripts contained
##      in this file and the rest is performed by macros #defined in the file
##      'main.cc'. 'sed' is a standard UNIX command which matches patterns of
##      characters in a stream and then replaces them with others. The current
##      translation mechanisms were only intended as a temporary measure to get
##      a prototype system up and running, and would need to be replaced with
##      a properly designed parser if the system was further developed.
#############################################################################


#############################################################################
## Temporarily replace special characters in string literals with substitutes
## so that they are not mistakenly translated by the rest of the scripts in this
## file. Once all the translation is done with, replace these tokens with the
## original characters.
#############################################################################
s/\(\".*\):\(.*\"..\)/\1COLON\2/g
s/\(\".*\),\(.*\"..\)/\1COMMA\2/g
s/\(\".*\) \(.*\"..\)/\1TAB\2/g
s/\(\".*\)%\(.*\"..\)/\1PERCENT\2/g

#############################################################################
## Replace any tab characters with spaces to make pattern matching more simple.
#############################################################################

s/ / /g

#############################################################################
## Take the score control structure parameters and put brackets round them for
## C++. Also put { } brackets around the whole block and add the necessary
## C++ code to allow each block to access its own start and end times.
#############################################################################

s/Score \(.*\):/Score(\1) {/g

#############################################################################
## Score control structure translations:
##      A TAO score control structure is actually a C++ if statement in
##      disguise. The times specified in the head of a From..to, At..for,
##      Before, After, At, Every and ControlRate control structure are compared
##      against the value of the variable 'Sample' which counts the number of
##      time steps which have elapsed since the beginning of a performance,
##      to see if the body of the control structure should be executed or not.
##
##      For control structures which specify a time interval over which the
##      body should be activated the two special variables 'start' and 'end'
##      allow code within the body to refer to the start and end times
##      specified in the head. This feature is explained more comprehensively
##      in sections 5.5.2 and E.5.2. In order to implement it, it is necessary
##      to keep track of the values of 'start' and 'end' when nested control
##      structures are entered and exited. This is achieved with the use of
##      two stacks 'startstack[]' and 'endstack[]'. The variables 'START' and
##      'END' are used in the process of passing the start and end times into
##      the body. Their values are determined by code which is added in during
##      the second phase of translation effected by the C++ macros in file
##      'main.cc'.
##
##      From <t1> to <t2>:              FromTo(<t1>, <t2>) {
##        <body>                  -->       n++;startstack[n]=start;endstack[n]=end;
##        ...                               START=start; END=end; {<body>}
##                                          start=startstack[n];end=endstack[n];n--;
##                                          }
##
```

```
##      At <t> for <dur>:                FromTo(<t>, <t>+<dur>) {
##        <body>              -->           n++;startstack[n]=start;endstack[n]=end;
##        ...                               START=start; END=end; {<body>}
##                                          start=startstack[n];end=endstack[n];n--;
##                                          }
##
##      Before <t>:                      Before(<t>) {
##        <body>          -->               n++;startstack[n]=start;endstack[n]=end;
##        ...                               {<body>}
##                                          start=startstack[n];end=endstack[n];n--;
##                                          }
##
##      After <t>:                       After(<t>) {
##        <body>          -->               n++;startstack[n]=start;endstack[n]=end;
##        ...                               {<body>}
##                                          start=startstack[n];end=endstack[n];n--;
##                                          }
##
##      Every <t>:                       Every(<t>) {
##        <body>          -->               n++;startstack[n]=start;endstack[n]=end;
##        ...                               {<body>}
##                                          start=startstack[n];end=endstack[n];n--;
##                                          }
##
##      At <t>:                          At(<t>) {
##        <body>          -->               n++;startstack[n]=start;endstack[n]=end;
##        ...                               {<body>}
##                                          start=startstack[n];end=endstack[n];n--;
##                                          }
#############################################################################

s/From \(.*\) to \(.*\):/FromTo (\1,\2) {/g
s/At \(.*\) for \(.*\):/FromTo(\1,\1+\2) {/g
s/Before \(.*\):/Before (\1) {/g
s/After \(.*\):/After (\1) {/g
s/Every \(.*\):/Every (\1) {/g
s/At \(.*\):/At (\1) {/g
s/AtSample \(.*\):/AtSample (\1) {/g
s/When \(.*\):/When (\1) {/g
s/ElseIf \(.*\):/Elseif (\1) {/g
s/If \(.*\):/If (\1) {/g
s/Else:/Else{/g
s/ControlRate \(.*\):/ControlRate (\1) {/g
s/\.\.\./start=startstack[n];end=endstack[n];n--;}}/g

s/FromTo *([^\{]*)/&{n++;startstack[n]=start;endstack[n]=end;start=START;end=END;/g
s/After *([^\{]*)/&{n++;startstack[n]=start;endstack[n]=end;start=START;end=END;/g
s/Before *([^\{]*)/&{n++;startstack[n]=start;endstack[n]=end;start=START;end=END;/g
s/At *([^\{]*)/&{n++;startstack[n]=start;endstack[n]=end;/g
s/Every *([^\{]*)/&{n++;startstack[n]=start;endstack[n]=end;/g
s/Score *([^\{]*)/&{n++;startstack[n]=start;endstack[n]=end;/g
s/ControlRate *([^\{]*)/&{n++;startstack[n]=start;endstack[n]=end;/g
s/If *([^\{]*)/&{n++;startstack[n]=start;endstack[n]=end;/g
s/Elseif *([^\{]*)/&{n++;startstack[n]=start;endstack[n]=end;/g
s/Else{/&{n++;startstack[n]=start;endstack[n]=end;/g
s/ and / \&\& /g

#############################################################################
## Put brackets around any list of parameters in the form:
##           <name> : <p1>, <p2>, <p3> ... ;
##
##      -->  <name> (<p1>, <p2>, <p3>) ;
#############################################################################

s/:\([^;]*\)/(\1)/g


#############################################################################
## Translate:
##           Glue instr1(<position1>) to instr2(<position2>);
## into:
##           Instrument::glue(instr1,<position1>,instr2,<position2>);
## and:
##           Join instr1(a,b) to instr2(a,b);
## into:
##           Instrument::join(instr1,a,b,instr2,a,b);
#############################################################################

s/Glue *\([A-Za-z_0-9]*\)(\(.*\)) *to *\([A-Za-z_0-9]*\)(*\(.*\)).*;
                                        /Instrument____glue(\1,\2,\3,\4);/g
s/Join *\([A-Za-z_0-9]*\)(\(.*\)) *to *\([A-Za-z_0-9]*\)(*\(.*\)).*;
                                        /Instrument____join(\1,\2,\3,\4);/g


#############################################################################
## A 'Display' statement in a TAO script is translated into a set of data items
## to be sent to the C++ standard output stream 'cout'. In a Display statement
## items are seperated by commas but in a 'cout' statement they are seperated
```

```
## by the token '<<'. Therefore replace commas with << in any Display statements.
## Also replace spaces between items with actual (," ",) spaces.
##
## So:       Display a,  b,  c   -->   cout << a << "  " << b << "  " << c ;
## wheras:   Display a,b,c       -->   cout << a << b << c ;
################################################################################

/Display[^;]*/  s/,\([ ]*\)/,\"\1\",/g
/Display[^;]*/  s/,/ << /g


################################################################################
## Put quotes around the filename in a microphone declaration.
##
##          Microphone m: file1, stereo   -->   Microphone m: "file1", stereo
################################################################################

/Microphone/  s/([ ]*\([A-Za-z_0-9]*\) *,/(\"\1\",/g


##s/START/START+2.0\/44100/g
##s/END/END-2.0\/44100/g

################################################################################
## Translate pitches (e.g. C4, Eb5, C#6) into C++ form. Function 'pitch()' is
## defined in main.cc
##
##          C4       -->     pitch("C4")
##          Eb5      -->     pitch("Eb5")
##          C#6      -->     pitch("C#6")
################################################################################

s/\([A-G][\#b]\{0,1\}[0-9]\{1,2\}[+-]\{0,1\}[0-9]*\/*[0-9]*\)/pitch(\"\1\")/g

################################################################################
## Nomenclature for accessing boundaries of instruments.
##
##          left     -->     0.0
##          right    -->     1.0
##          bottom   -->     0.0
##          top      -->     1.0
################################################################################

s/\([^a-z_A-Z]\)left\([^A-Z_a-z]\)/\1 0.0 \2/g
s/\([^a-z_A-Z]\)right\([^A-Z_a-z]\)/\1 1.0 \2/g
s/\([^a-z_A-Z]\)bottom\([^A-Z_a-z]\)/\1 0.0 \2/g
s/\([^a-z_A-Z]\)top\([^A-Z_a-z]\)/\1 1.0 \2/g
s/\([^a-z_A-Z]\)centre\([^A-Z_a-z]\)/\1 0.5 \2/g

################################################################################
## Add brackets to member function calls with no arguments.
##
##          s.lockleft.lockright   -->   s.lockleft().lockright()
################################################################################

s/lockleft/lockleft()/g
s/lockright/lockright()/g
s/locktop/locktop()/g
s/lockbottom/lockbottom()/g
s/lockends/lockends()/g
s/lockperimeter/lockperimeter()/g
s/lockcorners/lockcorners()/g


################################################################################
## Force floating point arithmetic even if a fraction is expressed as one
## integer divided by another.
##
##          <integer1> / <integer2>   -->   <integer1> / <integer2>.0
##          1/2                       -->   1/2.0
################################################################################

s/\([0-9][0-9]*\) *\/ *\([0-9][0-9]*\)/\1\/\2\.0/g

################################################################################
## Replace units of measurement with the appropriate conversion
##
##          10 secs         -->      10*1.0        ## force conversion to float
##          5 min           -->      5*60.0
##          25 msecs        -->      25/1000.0
##          300 Hz          -->      300
################################################################################

s/msecs/\/1000.0/g
s/secs/*1.0/g
s/min/*60.0/g
s/Hz//g
```

```
##############################################################################
## Convert percentage damping to a value between 0 and 1.
##
##   setdamping( .. , <damping>%)   -->  setdamping( .. , (1.0-<damping>/100.0))
##############################################################################

/damping/ s/,\([^,]*\)%/, 1\.0-\1\/100\.0/g

##############################################################################
## Replace tokens within string constants with the original special characters
## which they represent to restore the strings to their original state.
##############################################################################

s/\(\".*\)COLON\(.*\"\)/\1:\2/g
s/\(\".*\)COMMA\(.*\"\)/\1,\2/g
s/\(\".*\)TAB\(.*\"\)/\1 \2/g
s/\(\".*\)PERCENT\(.*\"\)/\1%\2/g
s/____/::/g
```

G.3.5 File string_sed_script

```
##############################################################################
## File: string_sed_script              (c) 1996 Mark Pearson
##
## Content:
##     UNIX sed script which translates a string declaration in a TAO script
##     in the following ways:
##
##     First it puts the whole declaration onto a single line to make searching
##     and replacing possible. In a TAO script messages can be passed to an
##     instrument within a declaration without having to explicitly give the
##     name of the instrument. However C++ must always have the name of an
##     object in order to pass messages to it. This sed script looks for
##     any messages and appends the name of the instrument together with a
##     period onto the front of each such message found.
##
##     String s1:                          String s1: 100 Hz, 10 secs;
##        lockends;                        s1.lockends;
##        setdamping(left, 1/10, 1%);      s1.setdamping(left, 1/10, 1%);
##        ...
##############################################################################


: start
/String/,/\.\.\./ !p
/String/,/\.\.\./ !d

/String *[A-Za-z0-9_]\{1,\}.*\.\.\./ b next
/String/,/;/ !
/String/ s/\n//g
: next
t start
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockleft/String \1\2\1\.lockleft/g
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockright/String \1\2\1\.lockright/g
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)locktop/String \1\2\1\.locktop/g
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockbottom/String \1\2\1\.lockbottom/g
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockperimeter/String \1\2\1\.lockperimeter/g
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockends/String \1\2\1\.lockends/g
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockcorners/String \1\2\1\.lockcorners/g
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)lock *(/String \1\2\1\.lock(/g
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)set/String \1\2\1\.set/g
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)display/String \1\2\1\.display/g
s/String *\([A-Za-z0-9_]\{1,\}\)\(.*\)amplif/String \1\2\1\.amplif/g
/String/ s/\.\.\.//1
```

G.3.6 File rectangle_sed_script

```
##############################################################################
## File: rectangle_sed_script           (c) 1996 Mark Pearson
##
## Content:
##     UNIX sed script which translates a rectangle declaration in a TAO
##     script in the following ways:
##
##     First it puts the whole declaration onto a single line to make searching
##     and replacing possible. In a TAO script messages can be passed to an
##     instrument within a declaration without having to explicitly give the
##     name of the instrument. However C++ must always have the name of an
##     object in order to pass messages to it. This sed script looks for
##     any messages and appends the name of the instrument together with a
##     period onto the front of each such message found.
##
```

```
##     Rectangle r1:                          Rectangle r1: 100 Hz, 200 Hz, 10 secs;
##        100 Hz, 200 Hz, 10 secs;  =>  r1.lockcorners;
##        lockcorners;
##        ...
##############################################################################


: start
/Rectangle/,/\.\.\./ !p
/Rectangle/,/\.\.\./ !d

/Rectangle *[A-Za-z0-9_]\{1,\}.*\.\.\./ b next
/Rectangle/,/;/ H
/Rectangle/ s/\n//g
: next
t start
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockleft/Rectangle \1\2\1\.lockleft/g
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockright/Rectangle \1\2\1\.lockright/g
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)locktop/Rectangle \1\2\1\.locktop/g
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockbottom/Rectangle \1\2\1\.lockbottom/g
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockperimeter/Rectangle \1\2\1\.lockperimeter/g
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockends/Rectangle \1\2\1\.lockends/g
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockcorners/Rectangle \1\2\1\.lockcorners/g
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lock *(/Rectangle \1\2\1\.lock(/g
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)set/Rectangle \1\2\1\.set/g
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)display/Rectangle \1\2\1\.display/g
s/Rectangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)amplif/Rectangle \1\2\1\.amplif/g
/Rectangle/ s/\.\.\.//1
```

G.3.7 File circle_sed_script

```
##############################################################################
## File: circle_sed_script                (c) 1996 Mark Pearson
##
## Content:
##     UNIX sed script which translates a circle declaration in a TAO
##     script in the following ways:
##
##     First it puts the whole declaration onto a single line to make searching
##     and replacing possible. In a TAO script messages can be passed to an
##     instrument within a declaration without having to explicitly give the
##     name of the instrument. However C++ must always have the name of an
##     object in order to pass messages to it. This sed script looks for
##     any messages and appends the name of the instrument together with a
##     period onto the front of each such message found. See files 'string_sed'
##     and 'rectangle_sed' for examples.
##############################################################################

: start
/Circle/,/\.\.\./ !p
/Circle/,/\.\.\./ !d

/Circle *[A-Za-z0-9_]\{1,\}.*\.\.\./ b next
/Circle/,/;/ H
/Circle/ s/\n//g
: next
t start
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockleft/Circle \1\2\1\.lockleft/g
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockright/Circle \1\2\1\.lockright/g
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)locktop/Circle \1\2\1\.locktop/g
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockbottom/Circle \1\2\1\.lockbottom/g
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockperimeter/Circle \1\2\1\.lockperimeter/g
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockends/Circle \1\2\1\.lockends/g
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockcorners/Circle \1\2\1\.lockcorners/g
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lock(/Circle \1\2\1\.lock(/1
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)set/Circle \1\2\1\.set/g
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)display/Circle \1\2\1\.display/g
s/Circle *\([A-Za-z0-9_]\{1,\}\)\(.*\)amplif/Circle \1\2\1\.amplif/g
/Circle/ s/\.\.\.//1
```

G.3.8 File triangle_sed_script

```
##############################################################################
## File: triangle_sed_script              (c) 1996 Mark Pearson
##
## Content:
##     UNIX sed script which translates a triangle declaration in a TAO
##     script in the following ways:
##
##     First it puts the whole declaration onto a single line to make searching
```

```
##      and replacing possible. In a TAO script messages can be passed to an
##      instrument within a declaration without having to explicitly give the
##      name of the instrument. However C++ must always have the name of an
##      object in order to pass messages to it. This sed script looks for
##      any messages and appends the name of the instrument together with a
##      period onto the front of each such message found. See files 'string_sed'
##      and 'rectangle_sed' for examples.
###############################################################################

: start
/Triangle/,/\.\.\./ !p
/Triangle/,/\.\.\./ !d

/Triangle *[A-Za-z0-9_]\{1,\}.*\.\.\./ b next
/Triangle/,/;/ H
/Triangle/ s/\n//g
: next
t start
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockleft/Triangle \1\2\1\.lockleft/g
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockright/Triangle \1\2\1\.lockright/g
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)locktop/Triangle \1\2\1\.locktop/g
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockbottom/Triangle \1\2\1\.lockbottom/g
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockperimeter/Triangle \1\2\1\.lockperimeter/g
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockends/Triangle \1\2\1\.lockends/g
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockcorners/Triangle \1\2\1\.lockcorners/g
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)lock *(/Triangle \1\2\1\.lock(/g
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)set/Triangle \1\2\1\.set/g
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)display/Triangle \1\2\1\.display/g
s/Triangle *\([A-Za-z0-9_]\{1,\}\)\(.*\)amplif/Triangle \1\2\1\.amplif/g
/Triangle/ s/\.\.\.//1
```

G.3.9 File ellipse_sed_script

```
###############################################################################
## File: ellipse_sed_script              (c) 1996 Mark Pearson
##
## Content:
##      UNIX sed script which translates a ellipse declaration in a TAO
##      script in the following ways:
##
##      First it puts the whole declaration onto a single line to make searching
##      and replacing possible. In a TAO script messages can be passed to an
##      instrument within a declaration without having to explicitly give the
##      name of the instrument. However C++ must always have the name of an
##      object in order to pass messages to it. This sed script looks for
##      any messages and appends the name of the instrument together with a
##      period onto the front of each such message found. See files 'string_sed'
##      and 'rectangle_sed' for examples.
###############################################################################

: start
/Ellipse/,/\.\.\./ !p
/Ellipse/,/\.\.\./ !d

/Ellipse *[A-Za-z0-9_]\{1,\}.*\.\.\./ b next
/Ellipse/,/;/ H
/Ellipse/ s/\n//g
: next
t start
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockleft/Ellipse \1\2\1\.lockleft/g
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockright/Ellipse \1\2\1\.lockright/g
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)locktop/Ellipse \1\2\1\.locktop/g
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockbottom/Ellipse \1\2\1\.lockbottom/g
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockperimeter/Ellipse \1\2\1\.lockperimeter/g
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockends/Ellipse \1\2\1\.lockends/g
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)lockcorners/Ellipse \1\2\1\.lockcorners/g
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)lock(/Ellipse \1\2\1\.lock(/1
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)set/Ellipse \1\2\1\.set/g
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)display/Ellipse \1\2\1\.display/g
s/Ellipse *\([A-Za-z0-9_]\{1,\}\)\(.*\)amplif/Ellipse \1\2\1\.amplif/g
/Ellipse/ s/\.\.\.//1
```

G.4 The tao shell script

```
###############################################################################
## File: tao                             (c) 1996 Mark Pearson
##
## Content:
##      Shell script which causes a TAO script to be translated into a C++
##      code fragment which is then #included into the main function in
##      file main.cc. This file is then compiled to produce an executable
```

```
##      which implements the synthesis described in the TAO script.
##
##      e.g. for a TAO script called 'example.script':
##
##      typing 'tao example' produces an executable called 'example.exe'
##
##      File 'example.script' is processed by the various sed scripts, and
##      redirected into file 'tao_scriptfile'. This is the file which is
##      #included into the main() function, further processed by C++ macros
##      #defined in 'main.cc' and then compiled.
#############################################################################

echo 'Processing script file:' $1.script

sed < $1.script -f $TAOPATH/translation/string_sed_script |
sed -f $TAOPATH/translation/circle_sed_script |
sed -f $TAOPATH/translation/rectangle_sed_script |
sed -f $TAOPATH/translation/triangle_sed_script |
sed -f $TAOPATH/translation/ellipse_sed_script |
sed -f $TAOPATH/translation/tao_sed_script1 |
sed -f $TAOPATH/translation/tao_sed_script2 |
sed -f $TAOPATH/translation/tao_sed_script3 |
sed -f $TAOPATH/translation/tao_sed_script4 > $TAOPATH/src/tao_scriptfile

echo 'Making synthesis program:' $1.exe

CC -L$TAOPATH/lib/ -o $1.exe $TAOPATH/src/main.cc -lm -lgl -ltao

echo 'type' $1.exe 'for synthesis of sound only, or'
echo $1.exe '-g <amplification> for visualisation, where'
echo '<amplification> determines exaggeration of waves graphically'
```

Bibliography

Abelson, H., Sussman, G. J. and Sussman, J. (1985). *Structure and interpretation of computer programs*, Cambridge, Massachussetts, MIT press.

Adrien, J.-M. (1991). The missing link: Modal synthesis, *Representations of Musical Signals*, Cambridge, Massachussetts. MIT press.

Adrien, J.-M., Causse, R. and Rodet, X. (1987). Sound synthesis by physical models, application to strings, *Proceedings of international computer music conference*, pp. 264–269.

Bak, P. and Kan, C. (1991). Self-organised criticality, *Scientific American* pp. 26–33.

Belkin, A. (1991). Whos playing? the computers role in musical performance, *Proceedings of international computer music conference*, pp. 131–134.

Beyls, P. (1989). The musical universe of cellular automata, *Proceedings of international computer music conference*, pp. 34–41.

Beyls, P. (1990). Subsymbolic approaches to musical composition, a behavioural model, *Proceedings of international computer music conference*, pp. 280–283.

Beyls, P. (1992). Dynamic models for musical interaction in virtual reality, *Proceedings of international computer music conference*, pp. 358–359.

Borin, G., De Poli, G. and Sarti, A. (1992). Algorithms and structure for synthesis using physical models, *Computer Music Journal* **16**(4): 30–42.

Bregman, A. S. (1990). *Auditory scene analysis: the perceptual organisation of sound*, Cambridge, MA. MIT press.

Cadoz, C. (1988). Instrumental gesture and musical composition, *Proceedings of international computer music conference*, pp. 1–12.

Cadoz, C. and Ramstein, C. (1990). Capture, representation and composition of the instrumental gesture, *Proceedings of international computer music conference*, pp. 53–56.

Cadoz, C., Florens, J.-L. and Luciani, A. (1995). Musical sounds, animated images with CORDIS-ANIMA and its multimodal interfaces, Demonstration at international computer music conference.

Cadoz, C., Luciani, A. and Florens, J. L. (1993). CORDIS-ANIMA: A modeling system for sound and image synthesis, the general formalism, *Computer Music Journal* **17**(1): 19–29.

Capra, F. (1992). *The Tao of physics*, London, Flamingo.

Chafe, C. (1995). Adding vortex noise to wind instrument physical models, *Proceedings of international computer music conference*, pp. 57–60.

Composer's Desktop Project manual (1994).

Connor, J. J. and Brebbia, C. A. (1978). *Finite element techniques for fluid flow*, London, Butterworth.

Cook, P. R. (1993). SPASM, a real-time vocal tract physical model controller; and singer, the companion software synthesis system, *Computer Music Journal* **17**(1): 30–43.

di Scipio, A. (1991). Further experiments with non-linear dynamic systems, composition and digital synthesis, *Proceedings of international computer music conference*, pp. 352–355.

Djoharian, P. (1993). Generating models for modal synthesis, *Computer Music Journal* **17**(1): 57–65.

Dodge, C. and Jerse, T. A. (1985). *Computer music, synthesis, composition and performance*, New York, Schirmer books.

Emmerson, S. (1990). The relation of language to materials., *in* S. Emmerson (ed.), *The language of electroacoustic music*, London, Macmillan press, chapter 2.

Florens, J.-L., Razafindrakoto, A., Luciani, A. and Cadoz, C. (1986). Optimized real time simulation of objects for musical synthesis and animated image synthesis, *Proceedings of international computer music conference.*, pp. 65–70.

Garnett, G. E. (1987). Modeling piano sound using waveguide digital filtering techniques, *Proceedings of international computer music conference*, pp. 89–95.

Gell-Mann, M. (1995). *The Quark and the Jaguar, adventures in the simple and complex*, London, Abacus.

Gibson, J. J. (1979). *The ecological approach to visual perception*, New Jersey, Lawrence Erlbaum.

Gleick, J. (1991a). *Chaos - making a new science*, London, Cardinal.

Gleick, J. (1991b). *Nature's chaos*, London, Cardinal.

Green, D. G. (1990). Cellular automata models in biology, *Journal of mathematical and computer modeling* **13**(6): 69–74.

Hearn, D. and Baker, M. P. (1986). *Computer graphics*, London, Prentice-Hall.

Hofstadter, D. R. (1986). Mathematical chaos and strange attractors, *Metamagical Themas*, Oxford, Clarendon press.

Hunt, A., Kirk, R. and Orton, R. (1991). Musical applications of the cellular automata workstation, *Proceedings of international computer music conference*, pp. 165–168.

Incerti, E. and Cadoz, C. (1995). Topology, geometry, matter of vibrating structures simulated with CORDIS-ANIMA. sound synthesis methods, *Proceedings of international computer music conference*, pp. 96–103.

Jaffe, D. A. (1995). Ten criteria for evaluating synthesis techniques, *Computer Music Journal* **19**(1): 76–87.

Keefe, D. H. (1992). Physical modeling of wind instruments, *Computer Music Journal* **16**(4): 57–73.

Lakshmi, M. R. (1989). Cellular automaton fluids - a rewiew, *Sadhana.*

Lewis, P. H. (1990). Computer vision, BSc computer science course notes.

Mackenzie, J. P. (1995). Chaotic predictive modelling of sound, *Proceedings of international computer music conference*, pp. 49–56.

Mcintyre, M. E. (1983). On the oscillations of musical instruments, *Journal of the Acoustical Society of America* **75**(5): 1325–45.

Miranda, E. R. (1993). Cellular automata music - an interdisciplinary project, *Interface* **22**(1): 3–21.

Morrison, J. D. and Adrien, J.-M. (1993). MOSAIC: a framework for modal synthesis, *Computer Music Journal* **17**(1): 45–56.

Pearson, M. (1995). TAO: a physical modelling system and related issues, *Organised Sound* **1**(1): 43–50.

Pearson, M. and Howard, D. M. (1995). A musician's approach to physical modelling, *Proceedings of international computer music conference*, pp. 578–80.

Pearson, M. and Howard, D. M. (1996). Recent developments with the TAO physical modelling system, *Proceedings of international computer music conference*, pp. 97–9.

Reynolds, C. (1987). Flocks, herds and schools - a distributed behavioural model, *ACM Computer Graphics* **21**(4): 25–34.

Roads, C. (1987). Granular synthesis, *Foundations of computer music*, Cambridge, Mass. MIT press, pp. 145–159.

Rossing, T. D. (1990). *The science of sound*, Reading, Mass. Addison Wesley.

Serra, X. (1986). A computer model for bar percussion instruments, *Proceedings of international computer music conference*, pp. 257–262.

Sheldrake, R. P. (1988). *The presence of the past*, London, Harper Collins.

Smalley, D. (1990). Spectro-morphology and structuring processes, *in* S. Emmerson (ed.), *The language of electroacoustic music*, London, Macmillan press, chapter 4.

Smith, J. O. (1987). Waveguide filter tutorial, *Proceedings of international computer music conference*, pp. 9–16.

Smith, J. O. (1992). Physical modeling using digital waveguides, *Computer Music Journal* **16**(4): 74–91.

Strogatz, S. H. and Stewart, I. (1993). Coupled oscillators and biological synchronisation, *Scientific American* pp. 68–75.

Toffoli, T. and Margolis, N. (1987). *Cellular automata machines - a new environment for modeling*, MIT press, Cambridge, Massachusetts.

Truax, B. (1986). Real time granular synthesis with the dmx 100, *Proceedings of international computer music conference*, pp. 231–235.

Truax, B. (1987). Real-time granulation of sampled sound with the dmx 100, *Proceedings of international computer music conference*, pp. 138–145.

Truax, B. (1990a). Chaotic non-linear systems and digital synthesis - an exploratory study, *Proceedings of international computer music conference*, pp. 100–103.

Truax, B. (1990b). Time-shifting of sampled sound with a real-time granulation technique, *Proceedings of international computer music conference*, pp. 104–107.

Vercoe, B. (1992). *CSOUND: a manual for the audio processing system.*

Waldrop, M. M. (1994). *Complexity: the emerging science at the edge of order and chaos*, England. Penguin.

Warren, W. and Verbrugge, R. (1984). Auditory perception of breaking and bouncing events: a cases study in ecological acoustics, *Journal of Experimental Psychology: Human Perception and Performance* **10**(5): 704–712.

Warren, W., Kim, E. and Husney, R. (1987). The way the ball bounces: visual and auditory perception of elasticity and the bounce pass, *Perception* **16**: 309–336.

Webb, P. (1993). *Self-modifying waveform synthesis using cellular automata*, Master's thesis, Dept. of Electronics, University of York, England.

Windsor, W. L. (1995). *A perceptual approach to the description and analysis of acousmatic music*, PhD thesis, City University, London. also available at http://www.shef.ac.uk/uni/academic/I-M/mus/staff/wlw/wlwthesis/-wlwthesis_ToC.html.

Wishart, T. (1990). Sound symbols and landscapes, *in* S. Emmerson (ed.), *The language of electroacoustic music*, London, Macmillan press, chapter 3.

Wishart, T. (1994). *Spectral transformations.* in Composer's Desktop Project manual.

Wolfram, S. (1984). Universality and complexity in cellular automata, *Physica* **10D**: 1–35.

Wolfram, S. (1986). *Theory and applications of cellular automata*, Singapore, World Scientific Publishing Co. Pte. Ltd.

Woodhouse, J. (1992). Physical modeling of bowed strings, *Computer Music Journal* **16**(4): 43–56.

Woolley, B. (1992). *Virtual worlds*, London, Penguin.