University of Edinburgh
Department of Computer Science

Tools for the Visualisation of Scottish Country Dances

4th Year Project Report

Ian Brockbank

November 3, 1994

Abstract

The aim of the project was to provide tools to make it easier for beginners to learn Scottish Country Dancing. The approach taken was to provide a tool to allow dances to be animated. Scottish Country Dances were analysed for structure, and from this a file format and an internal data structure were derived. A parser was written to build the data structure from files in the specified format. A module was then developed to produce animations of these dances.

Acknowledgments

Thanks are due to my supervisor, Frank Stacey, for many chats throughout the year, some of them even relevant to the project. His advice on matters related to both computing and dancing has been most useful, not least for gettting me thinking about other avenues of exploration and alternate approaches to problems.

Thanks are also due to Rob Procter for referring me to Stéphane Chatty's paper ([17]).

Contents

List of Figures

List of Tables

1. Introduction

There are three major forms of dancing in Scotland. First there are the Highland dances, such as the Sword Dance and the Highland Fling. These are mainly solo dances, with the emphasis on footwork. Then there are the ceilidh dances. These are couple dances in the main. They are known to many and done with much vigour and laughter, but seldom with any great finesse. Finally there are the Scottish Country dances. It is with these that this project is concerned.

These involve groups, or "sets", of six, eight, ten or sometimes more dancers in couples with a "man" partnering a "woman", although sometimes due to a shortage of one sex or the other some men are female, or (less commonly) women male. The dancers in the group co-operate to describe patterns, or "figures", of varying complexity. A dance generally consists of a sequence of these formations which leaves the dancers in a permuted order; the dance is repeated until all dancers are back in their original places.

Since there are around fifty basic figures, and more are being created all the time, there is clearly the potential for a large number of different dances. Indeed one count puts the number at around 10 000 [16]. This makes it difficult to remember which dance is which. Also the number of figures is a great barrier to beginners learning to dance.

There are a couple of representations in common use nowadays. One is a textual form as used by the Royal Scottish Country Dance Society (RSCDS) [3]. This is descriptive, but can lead to confusion through sheer weight of words and the need to learn the terminology. Accordingly, in 1955 F. L. Pilling brought out a book [1] which describes the dances in a more diagrammatic mnemonic form. These diagrams can also get involved and confusing, and both forms require the reader to be familiar with the formations. The two representations are described in more detail in chapter 2.

The aim of this project is to provide tools to aid the user in visualising how figures work, and how they combine to make dances. It does this primarily through animation.

A file format has been developed with the aim of denoting dances in a compact yet human-readable form. This draws from both the structure of dances as determined during the development of the data structures, and from the standard text crib format. An interface has also been developed using the MS Windows operating environment. This allows the easy editing of files as well as viewing of the animations using video recorder style controls.

Note on the use of the personal pronoun

To avoid awkward constructions the feminine forms of the personal pronouns, *she* and *her*, will also be used to mean *he*, *him* and *his* throughout this report.

2. Existing methods of description

The methods of description which currently exist are discussed here. A brief description is given of each, and their strengths and weaknesses detailed.

The descriptions covered are:

1. Text Cribs, as published by the RSCDS.
2. The diagrammatic form developed by F.L.Pilling.
3. An MSc by Richard Goss at the University of St. Andrews dated 1984.

2.1 Text cribs

These are typified by those published by the Royal Scottish Coutry Dance Society [3]. Figure 2.1 gives an example, taken from RSCDS book 1.

Cribs in this format are written in English prose, with set guidelines for how concepts should be expressed. As can be seen from the example, such a crib has a header giving the name of the dance and its type. Its source is also often quoted, as well as the initial configuration of the participating dancers, and requested tunes.

There then follows a list of bars, together with instructions. These instructions are given using standard terms and following approximately the same structure every time, namely *participants actions* [*extra information*] [*finishing positions*] ["*while*" *participants* . . .].

By using standard English this notation can be as expressive as the devisor of the dance or the writer of the crib can make it. However the prose lacks a definitive formality which can lead to differing interpretations of the same dance when it is danced by different groups. It can also become extremely verbose, which drastically slows down the process of reading and understanding the instructions.

The standard terms and layout mean that someone familiar with the format can find information quickly and efficiently. The down side of this is that the jargon makes the description all but impenetrable to the non-expert reader.

2.2 Pilling-style diagrams

These were first published in 1955 by F. L. Pilling [1]. They have been taken over by a committee, revised several times to add new (or missing) dances and to make the notation more consistent, both with itself and between dances. The book is currently in its sixth edition [2].

Figure 2.2 shows a sample dance using this notation. It is of a dance which was devised to look impressive in this notation and so is more complex than the average diagram. It should still give an idea of the format. For comparison, the textual version spans several pages.

The approach taken by Mr Pilling was to denote figures using diagrams, with set mnemonic symbols for common figures. For instance a circle is represented by a circle with the number of people involved given in the centre. A reel, which is like a figure of eight for three people, or a figure of eight with an extra loop for four people, is shown as two (or three, or $n-1$ for n people) small circles side by side and touching in a line.

Duke of Perth

RSCDS I–3

(Reel)

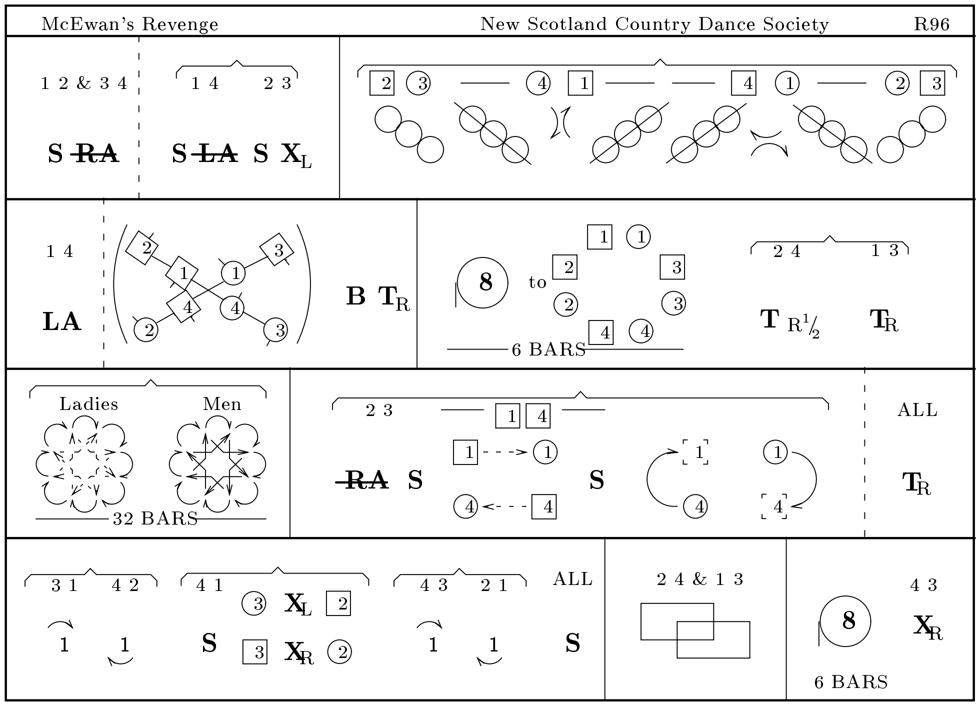
Four couple longwise set

Tune: Duke of Perth's Reel.

Music	**Description**
Bars	
1–4	1st couple turn with right hands and cast off one place on own sides. Second couple step up on bars 3–4.
5–8	1st couple turn with left hands to face first corners.
9–10	1st couple turn corners with right hands.
11–12	1st couple turn with left hands to face second corners
13–14	1st couple turn second corners with right hands.
15–16	1st couple turn with left hands to face first corners.
17–20	1st couple set to and turn first corners.
21–24	1st couple set to and turn second corners.
25–30	Reels of three on the sides, 1st woman with 2nd and 3rd men, 1st man with 3rd and 2nd women.
31–32	1st couple cross over to own sides, in second place.

Repeat, having passed a couple.

Figure 2.1: "Duke of Perth"—a sample dance in the text crib format. From RSCDS book 1.



Notation courtesy of F. L. Pilling. Used with permission from F. L. Pilling's successors. Dance copyright EUNSCDS 1994

Figure 2.2: "McEwan's Revenge"—A sample dance in the format used by Pilling

Many of the common figures have representations which take up much less space than written forms. This is the main strength of this format, in that it allows a dance to be expressed extremely compactly and hence it is possible to remind oneself of how a dance runs in a matter of moments.

That is all very well, and obviously the diagrammatic mnemonics, where used, have international application. Other mnemonics are less intuitive. For instance a bold-face S is used to indicate a setting step, and a T for a turn. This is all right for the English-speakers, once they have learned the notation, but not necessarily so for people in other countries. A German would not find "T" an intuitive shorthand for *drehen*. Furthermore, figures for which no symbol has been defined are shown by name: "TOURNÉE" in *From Scotia's Shores we're Noo Awa'* [2], for example. It is no more obvious what is meant here than when it occurs in the RSCDS-style cribs if the movement is unfamiliar.

Another failing of this format is that it does not give the qualifying information about a figure which may be given in an RSCDS-style crib, and so is even more open to differing interpretations. To be fair, the notation was never designed for this. In the preface to the sixth edition they say:

> We should like to emphasise that the book is designed as an aide-memoire and, although we have tried to include as much detail as possible in diagrammatic form, anybody who wishes to learn a dance should refer to the original printed instructions.

Where a movement is non-standard Pilling uses arrows—solid for men, dashed for ladies—to trace the path followed on a representation of the set. This is fine for simple movements, indeed it is often a lot clearer than the corresponding RSCDS-style description, but it can get messy when there are several people moving at once.

Finally, some movements, such as two couples changing places on the sideline, are much easier to express in text than in the Pilling format. The corresponding representation would explicitly show the women and men simultaneously "crossing", as near the bottom left-hand corner of figure 2.2.

2.3 MSc thesis by Richard Goss

Richard Goss' thesis submitted as part of his MSc in Computational Science at the University of St. Andrews in 1984 [16] was entitled *Computer Plotting of Country Dance Figures.* It was borrowed and studied in the hopes that it might contain some useful information.

He was concerned with methods of expressing Country Dances which were totally unambiguous, culturally independent and standard. His approach was to build a database of figures which could be referred to by another database of dances. This way figures would be described in the same way every time. The figures database could be used to produce textual descriptions of the figures, or to generate FORTRAN code to operate a plotter. The dance database was organised to allow searches by name, devisor, source, tunes or component figures.

This approach required the user to learn the various contractions used to refer to the figures. It resulted in extremely compact representations which were totally unintelligible.

For instance

```
927R1(21R)CORN␣RIGS/A6M1␣␣ASI1␣␣AGG1␣␣C8P2␣:969PI
```

produced, after being run through his program:

```
    CORN RIGS is a 32 bar (duple minor) Contra in 2/4 (Reel or Hornpipe)
  time.
  RSCDS(1927):  progression=1.  Secondary source(s) - 1969PI

  1:cast;cast up;
  1:figure of eight across; (in down);;
  1:down and back;
  12:poussette;
```

The figure definitions, sources, progressions, text and graphics plots were all hand-coded. The programs he developed were purely administrative.

2.4 Shortcomings

A major problem facing a beginnner is learning the basic movements, and how these movements fit together in dances. The methods of notation described above all fall down here. They all assume that the dancer is familiar with the figures and the jargon, and that she can visualise how the movements follow on one from the other.

The main aim of this project was to provide a suite of tools to tackle just this problem: to aid dancers in learning dancing and dances. These were intended to allow the user to extend the functionality with relative ease, something not provided by Richard Goss' system. This would mean the collection of dances could be as up-to-date as required.

Various possible avenues of exploration were considered, the following being some of the more prominent:

1. A representation of dances which allowed them to be stored efficiently. This also needed to be possible to be read by humans, to allow easy verification.
2. A parser which would read in files in a format very close to the text crib format described in section 2.1, and convert them to an internal representation for further use.
3. A module which would take dances in the internal representation and use them to generate the corresponding cribs, in either RSCDS-style or Pilling-style format.
4. A module which would take dances in the internal representation and from them generate an animation, which would allow people to see the paths traced by the various dancers as the formations develop.
5. A module which would allow the easy generation of dances, much in the manner of a painting program. This would be simplified by having an engine for drawing cribs, such as the Pilling generator in 3 above.
6. A module to allow dances to be found according to various criteria: name (complete or partial match), devisor, figures occuring, etc. This would require the dances to be held in a central database of some form.
7. A module to aid the writing of ball programmes. There are guidelines for putting together such programmes, most of which are common sense. Also a mix of dances is recommended, with as wide a spread of figures as is possible. A database such as described in 6 above would help ensure this.
8. The incorporation of music into the database, and/or musical accompaniment to the animations.

Many of these require an internal representation, and so the development of a data structure which could be expanded to cater for most, if not all of the above options was made a priority. Also some method for getting dances into the computer was needed. 1 and 2 were the only two options which provided this. It was decided to take the route of 1, while ensuring the file format was comprehensible to humans and so could be written by them. This had the major advantage of speed: if little attempt was made to model English prose and the file format matched the data structures the development of the compiler would be made much easier.

4, the animation, was chosen as the next path to follow, as this seemed to pose the most interesting and novel challenges. It also had the benefit of using the computer to provide functionality which paper-based media could never hope for. It was felt that it would also be the most useful avenue for teaching people dancing.

The crib generators and devising module, 3 and 5, were planned as the next stages, but unfortunately had to be abandoned due to pressure of time.

3. The structure of Scottish Country Dances

This chapter discusses the structure inherent in Scottish Country Dances. After defining the dancing terminology used throughout the report, dances are analysed to determine their basic make-up. This analysis is used in later chapters when developing a file format and data structure.

3.1 Terminology

3.1.1 Orientation terms

A dance starts with the dancers arranged in groups, known as *sets*. Normally the members of a set are paired in *couples*, of which one person is nominally a *woman*, and the other is a *man*. Due to shortages of one sex or the other, "men" may be female or, less commonly, "women" may be male. From now on I will use "men" to refer to dancers dancing as men, and "women" or "ladies" to refer to dancers dancing as women.

Where one couple has significantly more work than the others they are referred to as the *leading* or *dancing couple*. If there are two or more couples in this situation they are referred to as the *dancing couples*.

Sets are oriented with the *top* being the section of the perimeter nearest to the music.

There are two common styles of set: *longwise* sets and sets based on a circle. A third style sometimes used is *round the room*. These are described below, together with terminology peculiar to each.

Longwise sets

Longwise sets are arranged with a row of men facing a row of women. Each person has their partner directly opposite. People in a row are evenly spaced. The men have their left shoulders to the music, and hence the top of the set, and the women have their right shoulders to the music. The couples are numbered consecutively from the top of the set.

The two *sidelines* are imaginary lines drawn through the correct starting positions of the women and of the men. These are straight and parallel. The *women's side* is the sideline on which the women start, and the *men's side* is similarly defined for the men. A location in the *centre* or *middle* of the set lies on a line parallel to the sidelines, half way between the two. A dancer is said to be on her *own side of the dance* if she is on the same sideline as she started the dance. Correspondingly, she is said to be on the *opposite side of the dance* if she is on the other sideline. These are usually shortened to *own side* and *opposite side* respectively. The term *wrong side* is often incorrectly used where *opposite side* is meant.

Positions are described according to who occupied that location at the start of the dance. So for instance "first man's position" is the topmost position on the right hand sideline as viewed from the top of the set. Also "third woman's position" is the third nearest starting location on the left hand side, again viewed from the top of the set. If it is clear from the context where the intended location lies along the line drawn between the couple the specifier is usually dropped, giving (eg) "third position". *Place* is used interchangeably for *position*.

In a longwise set *up* means towards the top of the set, and *down* or *off* means away from the top. The end of the set furthest from the top is naturally enough called the *bottom*. *Above* and *below* follow as would be expected.

The orientation *up and down the set* means parallel with the sidelines, and *across the set* means at right angles to the sidelines. These are often shortened to *up and down* and *across* respectively.

A common arrangement has the dancing couple back to back in the middle of the set, with the woman facing the men's side, and the man facing the women's side. The positions above and below them are occupied by dancers on the sidelines. In this situation the dancing couple are said to be facing their *corners*. The person on their right is their *first corner*, with the person on their left their *second corner*. Continuing anticlockwise, some people use the terms *third corner* and *fourth corner* to describe their partner's first and second corners respectively. Note that the dancing couple and their first corners are on a diagonal line across the set, and similarly for the dancing couple and their second corners. This has given rise to the descriptions *first corner diagonal* and *second corner diagonal*.

This setup usually occurs with first couple as the dancing couple. They are in the middle of the set in second place. Second couple is above them in first place, on their own sidelines, and third couple are in third place, likewise on their own sidelines. The first and second corners are then people of the opposite sex.

Round the room sets

Such sets have the dancers arranged in two lines facing each other, with each line also having another set behind them. The sets make a large circle around the room, hence the name. The progression in such dances consists of the two lines passing to meet the next line of dancers.

Square, triangular and other polygonal sets

The other sort of set has the dancers arranged around the perimeter of a circle. The men have their partners on their right, and couples are evenly spaced. For obvious reasons such sets are called *triangular sets* if there are three couples on the perimeter, *square sets* if there are four couples, and so on. I will generally refer to sets of this type as circular or polygonal sets.

In some cases a dance has more than four couples in a square set. For five and six couples the extra dancers are generally in the centre of the set. For multiples of four dancers the dancers are arranged with two, three or more couples on each side of the square.

The term *corner* is also used in a circular set. In this case, however, each person only has one corner. This is the nearest person on the other side from their partner. So the men have their corners on their left, and the women have their corners on their right.

3.1.2 Other terms

There are three tempos of dance commonly used: *reels*, *jigs* and *strathspeys*. Reels[1] are in duple time, either $\frac{2}{4}$ or $\frac{4}{4}$. Jigs are in $\frac{6}{8}$ time. Many dancers make no distinction between these two, as the steps used are identical for both and it takes a practised ear

[1]Care must be taken to distinguish between the tempo *reel* and the figure of the same name, which occurs in all tempos.

to distinguish the duple beat from the triple. However all dancers can tell the difference between them and the strathspey. This is altogether slower and, theoretically, more graceful, and uses a different suite of steps. *Hornpipes* are just reels. When a dance is composed of sections in different tempos it is called a *medley*. For instance, the dance *Schiehallion* [2] has 64 bars of strathspey followed by 64 bars of reel.

A dance consists of the dancers performing a series of formations, or *figures*, which leave them in a permuted order at the end. The dance is then usually repeated a number of times until everyone is back in their original positions.

Dances are composed of *phrases*, usually eight *bars* long. One bar is the amount of music needed to complete the basic travelling step of the tempo. In most cases this corresponds directly to the bars of the music, but in some cases (notably $\frac{2}{4}$ reels) one dancing bar corresponds to two musical bars.

In many cases two dancers will be instructed to *pass by the left (or right) shoulder*. In this case they dance past each other so that the other dancer is on the specified side of them. This means that if they get too close the specified shoulders will collide.

3.2 Structure

In this section the general format of Scottish Country Dances will be discussed in order to highlight their inherent structure. This information will be useful when developing a data structure which represents them naturally. The dance "The Millwheel", described in figure 3.1, will be used to illustrate this discussion.

As can be seen from the sample dance, the instructions consist of a preamble followed by a list of bar numbers with associated descriptions.

In the preamble are the details of the dance such as its name, its source, its type, the intitial set layout and the tunes set. In the example the name is "The Millwheel" and the source is "Brockbank", showing that it was written by someone called Brockbank. Its type is given as an "8x32 bar Strathspey". This means it is in Strathspey tempo, the dance is 32 bars long, and is repeated 8 times in all. The initial set layout is given as a "Four couple longwise set", and the tune is "Grist to the Mill".

The instructions for the dance then follow. These take the form of a list of instructions. Each instruction is preceded by the numbers of the bars during which the formations should be carried out, followed by a description of the formations.

Consider the first formation:

> 1–8 1st and 3rd couples dance a double figure of eight, 1st couple crossing down and 3rd couple casting up to start.

This starts on bar one and finishes on bar eight. A list of participants is given, followed by the formation performed, with some extra information finishing the description off.

It is possible to have more than one formation being performed at a given time, for instance in bars 25–28.

> 25–28 1st couple pass first corners by the right shoulder and dance round their position back into the centre, passing their partner by the right shoulder to face second corners *while* 1st corners advance into the middle of the dance, turn once round right hand, and return to place.

This has first couple performing one movement—passing first corners by the right shoulder ... second corners—and at the same time their first corners perform another movement—advance into the middle of the dance ... to place.

The Millwheel

Brockbank

8x32 bar Strathspey

Four couple longwise set

Tune: Grist to the Mill.

Music	**Description**
Bars	
1–8	1st and 3rd couples dance a double figure of eight, 1st couple crossing down and 3rd couple casting up to start.
9–16	1st and 2nd couples dance the knot: turn partner half-way round with the right hand into allemande hold, curve round and up the lady's side, release right hands and pass the ladies in front of the men into the centre, and continue turning left hand. Finish 2nd couple in first place, 1st couple in the centre of the dance facing first corners.
17–24	1st couple with 2nd and 3rd couples dance the Millwheel:
1–2	1st lady with 2nd couple and 1st man with 3rd couple dance right hands across in a wheel half way round.
3–4	2nd and 3rd couples dance left hands across in a wheel half way round *while* 1st couple chase 1/4 of the way clockwise round the set.
5–6	1st lady with 3rd and 2nd ladies, 1st man with 3rd and 2nd men dance right hands across in a wheel half way round.
7–8	1st couple turn 1 1/4 by the left hand to face first corners.
25–28	1st couple pass first corners by the right shoulder and dance round their position back into the centre, passing their partner by the right shoulder to face second corners *while* 1st corners advance into the middle of the dance, turn once round right hand, and return to place.
29–32	1st couple repeat the movement with their second corners, passing each other by the right shoulders at the end to finish in second place on their own sides.

Repeat, having passed a couple.

Figure 3.1: "The Millwheel"—A sample dance illustrating basic dance structure, concurrent motion and recursive definition.

Formations can even be built up out of simpler formations, as in bars 17–24.

17–24 1st couple with 2nd and 3rd couples dance the Millwheel:

- 1–2 1st lady with 2nd couple and 1st man with 3rd couple dance right hands across in a wheel half way round.
- 3–4 2nd and 3rd couples dance left hands across in a wheel half way round *while* 1st couple chase 1/4 of the way clockwise round the set.
- 5–6 1st lady with 3rd and 2nd ladies, 1st man with 3rd and 2nd men dance right hands across in a wheel half way round.
- 7–8 1st couple turn 1 1/4 by the left hand to face first corners.

Here the figure being danced is called the *Millwheel*. It is built up from four smaller groups of movements—a right hands across, a simultaneous left hands across and chase, another right hands across, and finally a left hand turn. Note that, with the exception of the starting details such as who wrote it, or which tunes to play, this definition is almost identical to the description of the dance itself. This suggests that dances have a heirarchical or recursive nature. In fact one could even go as far as to say a dance is a figure which is defined in terms of smaller figures, which may themselves be defined in terms of smaller figures, and so on.

This suggests that the figure is the basic unit to consider, rather than the dance, as may seem at first sight more appropriate. Having determined this the structure of a figure now needs to be ascertained. As can be seen from the sample, and from the figure on bars 17–24, a figure is composed of a list of sub-figures.

Consider bars 9–16:

9–16 1st and 2nd couples dance the knot ... . Finish 2nd couple in first place, 1st couple in the centre of the dance facing first corners.

These form one sub-figure. There is an indication of when the movments occur (bars 9–16), who is involved (1st and second couples) and what they do (dance the knot). In this case there is also an indication of where the dancers finish the figure (Finish 2nd couple in first place, ... first corners).

Who is involved, what they do and where they end up are all lists, of participants, figures and (participant,location) pairs respectively.

Since figures are recursively defined there need to be some base cases. The simplest level which could be taken is to consider these to be just a sequence of curved and straight-line movements, and express all other figures in terms of these. However some movements are so common as to deserve definition in their own right. These are the basic repertoire a dancer needs to know. Examples are *crossing*, *casting*, *reels*, *turns* and *hands across*. It was decided to stop the recursion at this level of definition. Such figures will be referred to as *atomic* figures from now on. The other figures, which are defined in terms of smaller figures, will be referred to as *composite* figures.

4. The file format—an intuitive description

This chapter gives an informal description of the language developed for entering dances for animation. It outlines the various features, and uses a sample dance file to illustrate these. The dance chosen for this description is the *Duke of Perth*, published by the RSCDS [4]. Figure 4.1 on page 13 gives a possible file for it in this format.

A dance file contains several figure definitions. These have three sections, separated by `$` symbols. The first section gives the identifying details about the figure. The second section deals with the initial configuration of the dancers, and the third provides a list of actions, together with their times. A final `$` symbol terminates the definition. The sections will now be explained in more detail.

4.1 The identifying details

The first section, the identifying details, contains the name of the dance or figure, optionally followed by a shorthand form to make the descriptions more compact. A further three items of information relevant to dances may then be added, each preceded by a `\`. These are the source or the devisor of the dance, the dance type, and the tunes set for it. Three fields must be provided if any are, but the source and tunes fields may be empty. The source and tunes are just strings as far as the program is concerned, with a `\` and a `$` respectively marking the ends. The dance type is a number indicating how many times the dance is repeated to bring everyone back to their initial positions, an `x`, and another number indicating the number of bars in each repetition of the dance. This is followed by either `R`, `J`, `S` or `H` indicating whether the dance is a reel, a jig, a strathspey or a hornpipe respectively.

The Duke of Perth has the following header:

```
$ Duke of Perth \ RSCDS I,8 \ 8x32R \
  Duke of Perth's Reel $ 4CLS $
```

This shows it is from `RSCDS I, 8` (RSCDS book 1, dance number 8), is an `8x32R`: an eight times through 32 bar reel, and has tune “Duke of Perth’s Reel”. No shortened name has been given.

On the other hand, the figure `Corner-Partner` has this header:

```
$ Corner-Partner, CP $ $
```

It is not a dance in its own right, so does not have the extra details. It has provided a shorthand name, however: its name is `Corner-Partner` and its shortened name is `CP`. Note that case is not significant; `Corner-Partner` could be written as `CORNER-PARTNER`, `corner-partner` or even `CorNeR-paRtNEr`.

4.2 The position section

The second section of a figure definition gives the initial configuration of the dancers. This section may be left empty for a subfigure, but must be specified for figures which

```
%% Sample file defining the dance Duke of Perth
  (from RSCDS book 1, number 8)
  -- this is a multi-line comment %%
$ Duke of Perth \ RSCDS I,8 \ 8x32R \
  Duke of Perth's Reel $ 4CLS $

1,2: 1C: T(RH); % First couple turn right hand
3,2: 1C: C(1)  % First couple cast off one place
3,2: 2C: SU;   % while second couple step up
5,4: 1C: T(LH);  % First couple turn left hand
9,8: 1C,2C,3C: CP; % dance corner-partner
17,8: 1L,2M,3M & 1M,3L,2L: STC;
25,6: 1L,2M,3M & 1M,3L,2L: R(3,LS,100);
% 1st couple set to and turn corners (8 bars)
% and dance reels of three with their
% corners (6 bars)
31,2: 1C: X(RH);
% Finally, first couple cross over
% a single-line comment
$

% Defining the figures.

$ Corner-Partner, CP $ $
1,2: 1L,2M & 1M,3L: T(RH);
3,2: 1C: T(LH);
5,2: 1L,3M & 1M,2L: T(RH);
7,2: 1C: T(LH);
$

$ Set to and turn corners, STC $ $
1,2: #1,#2: S;
3,2: #1,#2: T(BH); % sets to first and turns
5,2: #1,#3: S;    % repeats with second
7,2: #1,#3: T(BH);
$
```

Figure 4.1: A sample dance file—“Duke of Perth”

may be performed on their own, such as dances. The specification starts with a number giving the number of dancers. This is followed by one of two forms.

The simplest is a colon, and a position for each dancer, given as x and y coordinates, followed by an angle in degrees measured anticlockwise from the positive end of the x-axis, indicating the direction in which the dancer is facing. These coordinates are given in round brackets. `(100,100,270)`, the position of first lady in a longwise set, is an example. The brackets are used as delimiters.

The other form has a `C` following the initial number, and then a set-type marker. This can be either `LS`, meaning a longwise set, or `TS`, `SQS`, `HXS` or `CircularSet`, meaning a set with the couples arranged around the perimeter of a circle. A set with four couples arranged in a square, with one or two couples in the middle, as in *Gavin's Reel* (five couple) or *The Iona Cross* (six couple) is given as a `5cSQS` and a `6cSQS` respectively. Dances with several couples on each side of a square, such as *The Sixteensome Reel*, with two couples on each side, or eight couples in total, or *The Thirtytwosome Reel*, with four couples on each side, are given as `8cSqs` and `16cSqs` respectively.

Looking back at the Duke of Perth, we find it is given as a `4CLS`—a four couple longwise set. The two subfigures both omit the position, and so start with the dancers in the positions in which they are left by the previous figure.

4.3 The definition section

The definition section takes the form of a series of actions with start time "," duration ":" a list of participants ":" a figure, optionally with a list of parameters enclosed in round brackets, and optionally ":" a list of destinations. The subfigure is terminated with a ";".

The starting time and duration are given in bars. The list of destinations is a list of positions as in the first form of the initial configuration as described above. The participants in the list are separated by either commas or `&` characters. This allows the devisor to split them up into logical blocks. The participants are either a `#` sign followed by a number, so `#3` is the third dancer specified when the figure was called, or a number followed by `L` or `W` meaning woman, `M` meaning man or `C` meaning couple. In fact the second form is translated into the first: `1L` is translated to `#1`, `1M` to `#2`, `2L` to `#3`, and so on. `1C` is translated to `1L,1M` which itself translates to `#1,#2`, and similarly for the other couples.

Taking the Duke of Perth example again, we have:

```
25,6: 1L,2M,3M & 1M,3L,2L: R(3,LS,100);
```

This figure starts on bar 25 and lasts for 6 bars. The dancers are first woman with second and third men in one group, and first man with third and second women in another group. The figure, `R(3,LS,100)`, will be described below. There are no finishing positions given.

In set and turn corners, we have

```
3,2: #1,#2: T(BH); % sets to first and turns
```

This subfigure starts on bar three relative to the start of the figure, and last for two bars. So if the figure `STC` started on bar 9, this subfigure would occur on bars 11 and 12. The dancers are numbers one and two. Referring to the figure for the Duke of Perth, we see

```
17,8: 1L,2M,3M & 1M,3L,2L: STC;
```

This figure is therefore called with dancers `1L,2M,3M` and `1M,3L,2L`. On bars 19 and 20, corresponding to the line from the definition of `STC`, the dancers are first woman and second man, and first man and third woman. This also illustatrates that order is important when specifying dancers.

A figure is specified by either the name or the short name of either a defined figure or a built in atomic figure. This may be followed by a list of parameters separated by commas and with the whole list enclosed in round brackets.

The parameters may be either numbers or one of the following special parameters:

`LH`, `RH` and `BH`: These signify left hand, right hand and both hands respectively. These parameters are used in turns, for example.

`LS` and `RS`: These, used in reels and when two people pass, indicate left or right shoulder.

`PH` and `AH`: These signify promenade and allemande hold respectively.

`NH`: This is used to show no hands should be given when crossing, for instance.

`NR` or `NRH` This shows the dancers give nearer hands, in a lead down the middle and back, for instance.

`X` and `C` plus `U`, `D` or `O`: These signify a cross or a cast to start (eg) a figure of eight. Followed by `U` means "up" while a `D` or an `O` means "down".

In the definition of the Duke of Perth, we have

```
17,8: 1L,2M,3M & 1M,3L,2L: STC;
25,6: 1L,2M,3M & 1M,3L,2L: R(3,LS,100);
```

The first figure is `STC`, the set and turn corners figure defined later in the file. This has no parameters. The second figure is `R(3,LS,100)`. This is a reel. Looking at the specification of the reel (which is an atomic figure) the parameters show it is a reel of three, starting with the dancing person giving left shoulder, and is a full reel, 100%.

Note that a figure may be referred to before it has been defined, as in the sample file with the `CP` and `STC` figures. As a result of the mechanism for this the program will also accept references to undefined figures. When the dance is animated such figures are ignored.

4.4 Comments

Notice the comment forms used throughout. There are two types: a single line comment, started using `%` and running to the end of the line, and a comment which can span multiple lines and is started and finished by `%%`. Comments can start at any point on a line or within the file. They can even occur in the middle of tokens. Both types appear in the sample file.

The layout of the file is also unimportant. The sample file has been laid out with one subfigure to a line, but this is not necessary. Several subfigures may be given on one line, or a subfigure may be split over several lines, or whatever.

5. The file format—formal definition

This chapter gives the syntax and intended semantics of the file format in a rigorous fashion. The complete syntax is given in BNF, and then the semantics of each syntactical element is described in detail.

5.1 The syntax

The syntax is given in a slightly extended version of Backus-Naur form. Non-terminal tokens are indicated as follows: *token*, and terminal tokens as here: ‘`token`’. A definition is of the form *non-terminal* → *token-list*. Where one non-terminal token has several possible derivations they are indicated by *derivation1* | *derivation2* | . . . . All terminal tokens are case-insensitive. Where there are several mutually-exclusive single-character tokens these are indicated using [‘*token-list*’]. Here the *token-list* contains single characters and/or ranges of characters marked by *start–finish*. If it starts with ˆ all characters are valid *except* those listed. \n, \r and \t are used for the new-line, carriage-return and tab characters, following the C convention. So for instance the digits are represented by [‘`0-9`’], and everything except a new-line and a tab would be represented by [‘`^\n\t`’]. The end-of-file character is represented by ‘`<EOF>`’, and the symbol ε is used to represent the empty token. The shorthand *(tokens)*? will be used to denote zero or one occurences of the token or bracketed group of tokens. Similarly *(tokens)** will be used to denote zero or more ocurrences, and *(tokens)+* to denote one or more occurrences.

5.2 The semantics

This section takes each syntactical element in turn and outlines the intended meaning.

All angles used in this section will be expressed in degrees, as is used in the files. This particularly applies to f coordinates.

5.2.1 Terminal characters

alpha → [‘`A-Z`’] (1)

digit → [‘`0-9`’] (2)

punctuation → [‘`,.;:"?!`’] (3)

wschar → [‘` \t\n\r`’] (4)

The terminal characters from which lexical elements are built up fall into four categories: letters (1), digits (2), punctuation (3) and whitespace characters—space, new-line, carriage return and tab. Note that letters are case insensitive: “c” and “C” are equivalent at all times.

5.2.2 Comments

comment → *line-comment* | *block-comment* | ε (1)

line-comment → ‘`%`’ *(*[‘`^%\n`’] *(*[‘`^\n\r`’]*)*)*? (2)

$$\textit{block-comment} \rightarrow \text{`\texttt{\%\%}'} \ ([\text{`}\texttt{\^{}[`\%\%'\,]}\text{'}])^* \ \text{`\texttt{\%\%}'} \tag{3}$$

There are two sorts of comment: single-line comments and multi-line comments.

Single-line comments (2) start with a single '%' character and continue until the next newline or carriage return character. They may start at any point on a line. The only restriction is that the character immediately following the starting '%' may not also be a '%'.

Multi-line comments (3) start and finish with two '%' characters side by side. These '%'s may not be separated by anything, not even spaces and newlines. The comments may span as many lines as required, or take up less than a line. They may be started at any point on a line. For obvious reasons they may not contain two '%' characters side by side.

5.2.3 Whitespace

$$\textit{ws-opt} \rightarrow (\textit{wschar} \mid \textit{comment})^* \tag{1}$$

$$\textit{ws} \rightarrow \textit{wschar ws-opt} \tag{2}$$

$$\textit{wschar} \rightarrow [\text{`}\ \texttt{\textbackslash t\textbackslash n\textbackslash r}\text{'}] \tag{3}$$

Whitespace includes space, tab, newline and carriage return characters. Comments do not count as whitespace for the sake of spacing, but are treated in the same way where whitespace is optional. Whitespace may be inserted between tokens. It is used as a token delimiter in only a very small number of cases.

5.2.4 Terminal strings

$$\textit{punctuation-string} \rightarrow (\textit{space-string} \mid \textit{punctuation})^* \tag{1}$$

$$\textit{space-string} \rightarrow (\textit{string} \mid \textit{wschar})^* \tag{2}$$

$$\textit{string} \rightarrow \textit{alpha}\ (\textit{string-char})^* \tag{3}$$

$$\textit{string-char} \rightarrow \textit{alpha} \mid \textit{digit} \mid \text{`\_'} \tag{4}$$

A string starts with a letter, and can contain letters, numbers and the underscore character '_'. Some lexemes have delimiters built in and can accept strings containing whitespace. Such strings start at the first string character (4) and continue until the last string character before the delimiter. They may contain string characters and whitespace. Such strings are used for dance names, for instance. A third type of string used may contain punctuation, as well as all the characters allowed in a *space-string*. These may start with any string or punctuation character and are terminated analogously to *space-string*s.

5.2.5 Numbers

$$\begin{aligned} \textit{float} \rightarrow\ & (\textit{signed-number})?\ (\text{`.'})?\ \textit{number} \\ & \mid\ \textit{signed-number} \end{aligned} \tag{1}$$

$$\textit{signed-number} \rightarrow (\text{`-'})?\ \textit{number} \tag{2}$$

$$\textit{number} \rightarrow (\textit{digit})+ \tag{3}$$

Numbers are exactly what would be expected: a series of digits in decimal. They may be preceded by a negative sign, and can be either integral or floating point. Floating point numbers consist of an optional negative sign, followed by zero or more digits, followed by a full stop to signify the decimal point, followed by zero or more further digits giving the fractional part. If the part preceding the decimal point is missed out it is taken to be zero. Integral numbers ((2) and (3)) may be given in place of floating point numbers but not the other way round.

5.2.6 The file and figure definitions

dance-file → *(figure-definition)** '`<EOF>`' (1)

figure-definition → '`$`' *figure-names (dance-details)?* '`$`' *start-setup* '`$`' *(figure)** '`$`' (2)

dance-details → '\\' *(source)?* '\\' *type* ('`+`' *type)** '\\' *(tunes)?* (3)

source → *punctuation-string* (4)

type → *number* '`X`' *number* ['`RJSH`'] (5)

tunes → *punctuation-string* (6)

A file (1) contains a series of zero or more figure definitions (2), with the end-of-file character marking the end of the file. These definitions start with a '`$`' character, followed by the names of the figure as described in section 5.2.7. The dance details then follow as described below. Note that these are optional as they only make sense for a figure which is also a complete dance. Another '`$`' character is followed the initial configuration as described in section 5.2.8 and then yet another '`$`' character. The description of the component figures (section 5.2.11) is followed by a final '`$`'.

If the extra details are given they consist of the source, dance type and tunes, each preceded by a '\\'. The only detail which must be provided (if any are provided) is the type.

The source is either who devised the dance or where it was found. This can be a *punctuation-string* as described in section 5.2.4, or empty.

The dance type consists of a number indicating how many times the dance is repeated, an '`X`', and '`R`', '`J`', '`S`' or '`H`' signifying a reel, jig, strathspey or hornpipe respectively. For a medley this may be followed by a '`+`' and another type. This may be continued up to 10 times.

5.2.7 Naming figures

figure-names → *figure-name* ('`,`' *short-name)?* (1)

figure-name → *space-string* (2)

short-name → *space-string* (3)

Since a figure name is delimited by a '`$`' and either a '\\' or another '`$`' (5.2.6, 5.2.6) the string used for its name can contain spaces. The person writing the dance may also specify a shortened form, separating the two with a comma. The figure name may be up to 50 characters long, and the shortened form up to 10.

5.2.8 The initial configuration of the dancers

start-setup → *number-participants* ‘`:`’ *(position)**
  | *digit* ‘`C`’ *set-type* (1)

number-participants → *number* (2)

The initial configuration of the dancers is given in either of two forms:

1. A number of dancers, followed by a colon, followed by a list of starting positions (section 5.2.10). Care must be taken to ensure that the number of positions specified corresponds to the number of dancers stated.
2. A standard set type. This is given by a number of couples followed by a ‘`C`’, followed by a type of set, as described in section 5.2.9.

5.2.9 The sets

set-type → *longwise-set* | *square-set* (1)

longwise-set → ‘`LS`’ | ‘`LongwiseSet`’ (2)

square-set → ‘`TS`’ | ‘`TriangularSet`’ | ‘`SQ`’ | ‘`SQS`’ | ‘`SquareSet`’ | ‘`HX`’ | ‘`HXS`’ |
  ‘`HexagonalSet`’ | ‘`CircularSet`’ (3)

The standard sets are either longwise or circular. Longwise sets are indicated by ‘`LongwiseSet`’, or ‘`LS`’ for short. Remember that the parser makes no case distinctions, and hence ‘`lonGWiSeSEt`’ (for example) is equally valid.

Circular sets may be indicated in several ways, reflecting the different shapes such sets form. Three couples in a circle form a triangle, four form a square and six form a hexagon. These are the most commonly used such formations. Accordingly ‘`TriangularSet`’ (or ‘`TS`’), ‘`SquareSet`’ (or ‘`SQ`’ or ‘`SQS`’) or ‘`HexagonalSet`’ (or ‘`HX`’ or ‘`HXS`’) may be used. The generic term ‘`CircularSet`’ may also be used. Note that, with the exceptions outlined below, the program makes no distinction between any of these, so a ‘`7CTriangularSet`’, a ‘`7CHexagonalSet`’ and a ‘`7CCircularSet`’ all result in seven couples evenly spaced around the perimeter of a circle.

The exception is the SquareSet. There are various forms of square sets for more than four couples, and the program will try to reflect this. A ‘`5CSquareSet`’ or a ‘`6CSquareSet`’ will produce a square set with the extra couple(s) in the centre, as in, for instance, Gavin's Reel [2]. A square set with a multiple of four couples, will have the couples arranged equally along four sides. So an ‘`8CSquareSet`’ as in the *Sixteensome Reel* would have two couples on each side, a ‘`12CSquareSet`’ would have three couples on each side, and so on.

Note that these terms contain no spaces.

In actual fact only the first three letters of the set type are significant, and so any truncation to three letters or longer is accepted.

5.2.10 Positions

position → *relative-position* | *absolute-position* (1)

relative-position → ‘`#`’ *dancer absolute-position* | ‘`@`’ *dancer absolute-position* (2) *absolute-position* → '(' *signed-number* (',' *signed-number* (',' *signed-number*)?)? ')' (3)

Positions may be relative or absolute.

An absolute position is enclosed in round brackets '`(`' and '`)`'. The devisor may give only the x-coordinate, the x- and y-coordinates or all three coordinates, with the f-coordinate as well. Coordinates are signed numbers (section 5.2.5) separated by commas and, optionally, whitespace. Unspecified coordinates are taken to be 0. So '`(100,0,0)`', '`(100,0)`' and '`(100)`' are identical.

Relative positions have exactly the same syntax as absolute positions, with the addition of either a '`#`' or an '`@`' symbol and a dancer (section 5.2.14) at the front, optionally separated from the absolute component with whitespace.

The '`#`' forms are evaluated by adding the absolute coordinates to the coordinates of the location of the given dancer at the time of evaluation, while the '`@`' forms are calculated with respect to the dancer's original position. So if second man in a longwise set is at location $(250, 0, 0)$, '`#2M(-50,50)`' gives $(200, 50, 0)$. '`@2M(-50,50)`', on the other hand, gives $(150, -50, 90)$, since second man started the dance in second man's place (surprisingly enough)—$(200, -100, 90)$.

5.2.11 The component figures

figure → *start* '`,`' *duration* '`:`' *dancers* (['`,&`'] *dancers*)* '`:`' *subfigure* '`:`' (*relative-position*)* '`;`' (1)

subfigure → *figure-name* ('`(`' *parameter* ([' `,`'] *parameter*)* '`)`')?
| *repeat* (2)

The component figures which make up a dance consist of an indication of when they occur, followed by a list of dancers and a figure. A list of finishing positions may optionally be appended. The various sections are separated by colons, and may also be surrounded by whitespace.

The time indication gives a starting bar and a duration in bars, separated by a comma. The list of dancers is described in section 5.2.14.

The description of the figure consists of a figure name, either the full or the shortened version, optionally followed by parameters to the figure (section 5.2.12) in round brackets. Alternatively a repeat may be specified, as described in section 5.2.13. Note that a figure may be referred to before it has been defined, and it is even possible to include figures which are not defined. Undefined figures are treated as composite figures with no subfigures in animations, and hence have no effect.

The optional position list is a list of relative positions as described in section 5.2.10, optionally separated by whitespace, although since positions are terminated by a closing brace this is not necessary. It gives the positions of the dancers involved in the figure when the figure finishes. The first position corresponds to the first dancer, the second to the second, and so on. It is not necessary to specify a final position for all the dancers.

5.2.12 Figure parameters

parameter → *number* | '`L`' | '`R`' | '`B`' | '`LH`' | '`RH`' | '`BH`' | '`LS`' | '`RS`' | '`P`' | '`A`' | '`PH`' | '`AH`'
| '`N`' | '`NH`' | '`NHL`' | '`NHR`' | '`NR`' |NRH | '`XU`' | '`XD`' | '`CU`' | '`CD`' | '`CO`' (1)

The parameters to a figure are enclosed in round backets (section 5.2.11). They are given as a comma-separated list.

A parameter may be either a number (section 5.2.5) or one of the strings below:

'`L`', '`R`' and '`B`' These signify left, right and both respectively. They are most commonly used to represent the hands given in a turn, and may be followed by an '`H`' in recognition of this. They are also sometimes used to represent the nearer shoulders of two dancers passing, and may be followed by an `S` to show this. Note that `RH` and `RS` are not equivalent, although some figures do treat them so.

'`P`' and '`A`' These signify promenade and allemande hold respectively, and may be followed by an `H`.

'`N`' This signifies none when followed by an `H` or alone, and nearer when followed by an `R` (see below). It is used to specify that the dancers should not give hands when crossing over, for instance. It may be followed by an `H` and a `R` or an `L` to show that the dancers should pass by the right or left shoulder respectively.

`NR` This signifies nearer, as in nearer hands. It may be followed by an `H`. It is used to show that the dancers should give nearer hands in a lead down the middle and back, for instance.

'`X`' and '`C`' + '`U`', '`D`' or '`O`' These represent crossing or casting. The second letter specifies whether it is up ('`U`') or down, also referred to as *off* for a cast ('`D`', '`O`'). These are used in figures of eight to specify how the dancers should start the figure.

5.2.13 Repeats

repeat → '`RPT`' | '`REPEAT`' (1)

A repeat starts with one of the repeat markers '`REPEAT`' or, more succinctly, '`RPT`'. It indicates that the figure to be performed is the same as that given in the previous figure. Note that here previous figure means previous in the file and not in time.

5.2.14 List of dancers

dancers → *number* '`C`' | *dancer* (1)

dancer → *number* ['`MLW`'] | '`#`' *number* (2)

A list of dancers may be separated by commas and/or ampersands. This is to allow the devisor to highlight their grouping. The program makes no distinction between the two separators.

A dancer is given by either a number, followed by '`M`', signifying a man, '`W`' or '`L`', signifying a woman, or '`C`', sigifying a couple. Alternatively she may be specified using the form '`#`' followed by a number. This means the dancer with the same internal number as that given, starting at 1. In fact '`1L`' is expanded internally to '`#1`', '`1M`' to '`#2`', '`2L`' to '`#3`', and so on. '`1C`' is expanded to '`1M,1L`' which is expanded to '`#1,#2`', and similarly for the other couples. Note that the dancer specified as part of a relative position (section 5.2.10) may not be a couple.

6. Features of C++

6.1 The basics

C++ was described by its principal designer, Bjarne Stroustrup of AT&T Bell Laboratories, as "C with classes" [26]. The aim was to provide object-oriented extensions to C, without losing any of the functionality provided by C. The compiler used, Borland C++ version 4.0, fully implements AT&T's C++ version 3.0 [18].

Object-orientation is a hotly disputed term. Wegner [27, summarised in [26]] describes three classes of languages. *Object-based* languages provide for data abstraction and access to an object's "state" through its own operators. *Class-based* languages refine this class by adding abstract data types, where objects belong to classes—their abstract data types. A *class* is more a template for an object than a strict type. The class' template may be copied, and perhaps be modified when copied. This leads to the addition of the third and smallest set of languages, the *object-oriented* languages. These add inheritance to the properties of the other sets. Inheritance allows one class to be defined in terms of another, with the "child' class *inheriting* most of the properties of the "parent" class.

C++ extends the syntax of the `struct` construct in C, and adds a new `class` construct, to provide these "class templates". There is little material difference between the two, so I shall use "class" for both throughout. A `class` can be declared to have data associated with it, as with a `struct`. It can also have *member functions* defined to act on the data in the class.

An important part of C++, and one which can save a lot of work re-inventing the wheel, is that of inheritance. A class can be declared as inheriting properties from one or more *parent* classes. It then has all member functions and all data members of the parent classes, with the same functionality as before, except those which are specifically over-ridden in the definition of the new class. These properties can then be extended with further properties. This allows a family of objects to be defined which have some properties in common and some unique to the child classes. An item of the child class type is also an item of the parent class type, although the converse is not necessarily true. Parent classes may not have members with the same name. The exception to this is where all but one of the conflicting classes are marked as `virtual` in the inheritance list. In this case the non-virtual class has precedence.

All classes have at least one *constructor* and a *destructor*. These are functions called when an instance of a class is created, and when it is destroyed. They have the same name as the class, preceded by a `~` in the case of the destructor, and have no return value. The constructors may take parameters, but the destructor may not. If either is not provided the compiler provides a default one. Constructors and destructors are not inherited, but the constructor of a parent class may be called before entering the body of a child constructor.

Polymorphism allows a child to redefine a member function so that whenever that function is called of a member of the child class, the appropriate version of the function is called, even if the function is called from a pointer of type pointer-to-base-class. Such functions are marked by the keyword `virtual` preceding the function declarations of all the classes.

Member functions and data items may be marked as `public`, `protected` or `private`. `Public` declarations are available to all objects, and even to global statements. `Protected` declarations are available to all instances of that class and derived classes and to functions which have been specifically declared as `friend` functions, but not to anything else. `Private` declarations are only available to instances of that class and to friend functions, not to derived classes. This mechanism provies the data-hiding features. The standard practice is to make all data items either `protected` or `private`, and to provide access functions to set and query these data items. This means the object has complete control over alterations to its data.

The classic example is a set of graphics items, a circle and a rectangle, for instance. These all have certain properties in common, such as their position on screen, and other properties specific to each, such as the radius for the circle, or the lengths of the sides for the rectangle. They also share a virtual function `Draw` which draws the object on screen. These might share a base class `GraphObj` declared as follows.

```
class GraphObj
{
public:
  GraphObj();
  ~GraphObj();

  void SetLocation(POINT p);
  POINT GetLocation();

  virtual void Draw();
protected:
  POINT location;
};
```

The functions in `GraphObj` would have a later definition of the form

```
void GraphObj::Draw()
{
  :
  function body
  :
}
```

From this class definition we might have classes `Circle` and `Rectangle` based on it as follows:

```
class Circle : public GraphObj
{
public:
  Circle();
  Circle(int radius);
  ~Circle();

  void SetRadius(int r = 1);
  int GetRadius();

  virtual void Draw();
protected:
  int radius;
};

class Rectangle : public GraphObj
{
public:
  Rectangle();
  ~Rectangle();

  void SetSize(POINT);
  POINT GetSize();

  virtual void Draw();
protected:
  POINT size;
};
```

Note that both `Circle` and `Rectangle` also have the `public` member functions `SetLocation` and `GetLocation`, and the `protected` data item `location`, inherited from `GraphObj`. If `location` had been `private` instead of `protected` then `Circle` and `Rectangle` would not have had access to it.

Member functions are called using the usual C structure de-reference operators "`.`" and "`->`". As well as the specified parameters, a hidden "`this`" parameter, a pointer to the instantiation of the class, is passed to allow the function to know which item is being acted upon. So, if an object `C` of type `Circle` had been created, it would be drawn using the command `C.Draw();`. If `P` was of type `GraphObj*`, then the commands `P = &C; P->Draw();` would have exactly the same effect; the `Draw` function called would be that corresponding to the actual item, the `Circle`, and not to the type of the pointer, a `GraphObj`.

6.2 Overloading

In C++ functions may be *overloaded*. This means different functions with the same names may be defined, as long as the types of their parameters are different. So, for instance, in C the programmer might have written

```
int greater_than_int(int a, int b);
float greater_than_float(float a, float b);
```

In C++ she could use the same function name, and the compiler would select the correct version for the parameters passed. In particular, the standard operators, `+`, `==` and so on, may be overloaded for a new class. So instead of writing a function `int IsEqual(Class1 &C1, Class1 &C2);` the programmer could define a member function `int operator ==(Class1 &Other);` and then the much more intuitive syntax `C1 == C2;` could be used to test for equality, for example.

6.3 Templates

The template features of C++ can save writing a lot of very similar code. Effectively they allow compile-time parameters as well as run-time parameters to functions and declarations. Their power lies in the fact that these parameters can be any constant, including types. The classic example, as described by Borland [18], is a function which compares two items and returns the greater. In C this might be

```
int max(int a, int b)
{
  return (a > b) ? a : b;
}
```

This is fine if all you want to compare are integers. However if `doubles` were to be compared another function would have to be written which differed in only the types declared:

```
double max(double a, double b)
{
  return (a > b) ? a : b;
}
```

And then if a `struct` had been defined

```
struct tagPOINT
{
  int x,y;
} POINT;
```

and these were to be compared? This is where templates come in. The following definition is completely general purpose, taking the type as a parameter.

```
template<class T> T max(T a, T b)
{
  return (a > b) ? a : b;
}
```

This even covers the `struct`, assuming the programmer has overloaded the `>` operator (see section 6.2). The structure definition would then become as follows, and the function `max<POINT>` would compare two `POINT`s, and return the greater.

```
class POINT
{
  int x,y;

  int operator >(POINT &Other)
  {
    int result;
    // some code making your comparison here
    return result;
  }
};
```

Templates have been used to define a generic dynamic array of objects, accessible by their index. Object were sorted by order of insertion—ie additions were made in space at the high end of the array. If the array was full it was extended to provide more space.

These dynamic arrays were used to store lists of figure details in the various data structures (see chapter 8). Initially only two different versions of these lists were used: one of dancer indices (`int`) and one of parameters. During the course of the project other lists were found to be needed: of locations (`struct Location`), and pointers to dancers (`struct Dancer*`). Having the template list made it the work of a moment to add types to handle these lists.

6.4 Advantages and disadvantages of C++

C++ was a natural language for expressing the problem, as it involves various objects with various properties interacting, and this lends itself to an object-oriented approach. The language also allows for hierarchies of related objects, each inheriting the properties of object above them in the hierarchy, and becoming more specialised as the hierarchy is traversed.

There were a few problem encountered, however. The first was that the resolution mechanisms for polymorphic functions are not set up when a parent class constructor is called from a child constructor. This is described in more detail, together with the work-around used, in section 8.3.2.

Another problem which is a common gripe with MS-Windows and C++ is that MS-Windows cannot pass the hidden `this` parameter to class member functions, and so callbacks which are member functions must be `static` members. This is described in more detail in section 8.3.1.

A further problem occured when using `virtual` functions which called the versions of the functions defined in parent classes. Since this takes the form of a function call it is not possible for the parent function to do both setting up and cleaning up.

This problem arose when allowing figures to act upon different `Set`s. An optional `Set` parameter was added to the `Evaluate` function (see section 8.1). The simple approach would be to have the pointers inside the `Figure` class made to point to the passed-in `Set` if non-null, at the start of the function, and to point back again to the original `Set` at the end of the function. This would be fine, only it would be nice for the functions in child classes to be still able to access the relevant `Set` after calling the parent function. By that time the parent function has finished, so everything has been reset.

This problem was solved by having a data member in the parent class of type `Set*`. If a `Set` was passed in, this pointer was assigned to point to it. Otherwise the pointer was made to point to the original `Set` of the `Figure`. Since this pointer was inherited by all derived classes, they could do all their `Set` access through it and always get the right `Set`. There would be no tidying up needed, as a pointer to the original `Set` was also held, and so it could always be accessed through that.

7. The operating and programming environments

Microsoft's Windows 3.1 was chosen as the operating environment. This was because I already had experience with writing programs for it, and it was the environment installed on my computer. It is also more likely that other dancers would have a PC-compatible running under MS-Windows than a Sun workstation running under X-Windows.

Note that people argue whether MS-Windows is actually an operating system or not. Those who argue that it is not point out that it does not itself provide some essential facilities such as file-handling, but instead passes them on to the underlying MS-DOS operating system. For simplicity in the following discussion I will consider it to be combined with MS-DOS to form a windowing operating system.

7.1 MS-Windows programming

MS-Windows programming is event-driven. The program is notified of *events* by a message from the MS-Windows kernel. Examples of such events are a key-press, a mouse button click or a menu item being selected. The program then acts on these events (or not) and returns control to the operating environment and the user. The next section will expand on how this works.

This explanation is based on the books by Petzold [21] and Schulman, Maxey and Pietrek [22]. Please refer to these for further details.

7.1.1 The structure of an MS-Windows program

A windows program has at least two functions. The `WinMain` function roughly corresponds to the `main` function in a standard C program in that it is the first function called when a program is run. This function is used for initialisation. A `WNDCLASS` structure is set up with the details of the program's main window, and this is registered with the operating system. Details in this structure include *handles* (effectively pointers) to an icon, a mouse cursor and a menu structure. Also the name of the program as a `char*`, flags indicating whether repaint messages should be sent when the window is resized and, most important of all, the address of the `WndProc` function.

Having registered the window class a window of this class is then created to become the application's *main window*. Once this window has been displayed the function goes into a *message loop*. In its simplest form this looks like figure 7.1. The operating system is polled for outstanding *messages* from events. The `GetMessage` function returns 0 if the program has been terminated, thus finishing the `while` loop. Otherwise the `MSG`

```
while (GetMessage (&msg, NULL, 0, 0))
{
  TranslateMessage (&msg);
  DispatchMessage (&msg);
}
```

Figure 7.1: A simple MS-Windows message loop.

structure `msg` will contain details of the appropriate event. The `TranslateMessage` call takes this message and converts keypress messages into the corresponding character messages for the keyboard the system has been told is in use. Finally, the `DispatchMessage` function call gets the message sent to the appropriate window function.

So what is this window function? It is the function whose address was given to MS-Windows in the `WNDCLASS` structure when the window class was registered. Since it has a set format, the MS-Windows kernel can call it with `*(wndclass.lpfnWndProc)(<parameters>);`. These parameters are

`hwnd` The handle of the window, passed back to MS-Windows as a parameter to many system calls.

`message` The message, encoded as a `WORD`, or 16 bit unsigned integer. The `windows.h` header file defines symbolic constants for these. For instance `WM_VSCROLL` is defined to be `0x0115`, and `WM_SIZE` is defined to `0x0005`.

`wParam` A 16 bit parameter to the message. This usually qualifies the message. For example in `WM_VSCROLL` it tells the program whether to scroll the display up or down by either a line or a page, or to take the position from the scroll bar *thumb*, which is passed in in the next and final parameter.

`lParam` A 32 bit parameter usually used to pass data to the function. For example in `WM_VSCROLL`, when `wParam` holds `SB_THUMBPOSITION`, `lParam` holds the appropriate value in its lower 16 bits. Note that since pointers are also 32 bit this could be a pointer to a much larger data structure.

The standard form for a `WndProc` procedure is a large `switch` statement with the message values as guards. For a complex program this can become extremely unwieldy and doubtfully efficient. Several vendors have taken advantage of the modularity and the message-passing to develop C++ libraries which encapsulate this functionality and increase the efficiency. One, the Borland *ObjectWindows* library used in the development of this program, is discussed in section 7.2.

7.1.2 Callbacks

There is another way for MS-Windows to pass a message to a program. This involves the program giving the address of a function which is to be called whenever a certain event occurs. Then when the event does occur, instead of putting a message in the program's message queue the kernel calls that function with set parameters, just as happens with the window procedure functions. The `WndProc` and similar functions are in fact functions of this type. Such functions are called *callbacks*, and have to expect the documented parameters. Giving the kernel the address of a callback function is referred to as *installing a callback*.

Callbacks have a range of uses. A timer callback was used in this project to provide real-time control of the animation. Other examples are window and dialog procedures (such as `WndProc`), functions called by MS-Windows functions which enumerate through a set of items and service routines which replace some of the functions provided. However it is not possible to install callbacks for arbitrary events.

7.1.3 Multitasking

MS-Windows uses non pre-emptive or cooperative multitasking. A program relinguishes control when it polls its message queue using the `GetMessage` and related functions. If there are only `WM_PAINT` and `WM_TIMER` messages in the program's message queue when it calls `GetMessage`, MS-Windows will switch to another program which does have pending messages, and only return to the program once all other programs have been serviced. This means that `GetMessage` may take a long time to return. It also means programs must be written to relinguish control every few milliseconds even if they are in the middle of some long calculations, as otherwise the system will appear to come to a grinding halt.

7.2 The Borland ObjectWindows library

As part of their C++ compiler package, Borland International provide a set of C++ class libraries collectively named *ObjectWindows*. These originally appeared as an add-on to version 3.0 of their compiler, and have been extensively revised and expanded in version 2.0 of the libraries as currently packaged with version 4.0 of the compiler. The following discussion is based on the accompanying instruction manuals [19, 20].

The purpose of the *ObjectWindows* libraries is to encapsulate the functionality of an MS-Windows program. Classes provided include:

1. An application class, `TApplication`. This provides the program initialisation, and holds the main message loop. It also handles the dispatching of messages to the classes corresponding to the appropriate MS-Windows elements. The programmer derives a class from this which overrides the default actions where necessary. In particular the `MainWindow` is set to a class derived from the base `TFrameWindow` which has the particular functionality she wants.

2. Window classes, `TWindow` and `TFrameWindow`. In MS-Windows everything is regarded as a window [21]. *ObjectWindows* mimics this by having a class derived from `TWindow` [19]. `TFrameWindow` is used for the interface elements usually called windows. The programmer derives a class from `TWindow` to act as the drawable area and makes it a child window of either a `TFrameWindow` or a derived class.

3. Menu classes. These provide a wrapper for the various types of menu—pull-down, both user and system defined, and pop-up. These classes are usually used without modification.

4. Dialog boxes. These are windows used to inform the user and to get information from her. They commonly have child windows which act as text entry boxes, buttons, list selection boxes and so on, generically referred to as *controls*. See 5 for more on these. Dialog boxes may be application modal, in which case the program is halted until the box is closed, but the user may switch to other programs, system modal, where the user cannot even switch to a different program, or modeless, where the program may also be accessed. These last tend to be tool palettes and the like.

5. Control objects. As mentioned in 4 these are commonly used to pass information between the user and the program in dialog boxes. They may also appear on standard windows. They include:

    - Buttons. These are usually used to trigger a change in program state.
    - Text boxes, both static and for user entry.
    - Check boxes, for selecting and deselecting options, and radio buttons for choosing between a group of related options.

    - List boxes and combo boxes. These provide lists of options to the user. Combo boxes combine a text entry box with a list box, allowing the user to type the selection in.
    - Scroll bars. These allow the user to specify a point on a range.

6. Printers. These provide a wrapper for the printing interface.

7. Graphics objects. These are simply the graphics objects provided by MS-Windows, re-expressed as C++ objects. They include

    - Device Contexts. These control access to display areas, whether screen, memory, printer, or whatever. They also store the state of the given display area.
    - Pens and Brushes. Pens are used to control how lines are displayed, and brushes specify the pattern used when areas are filled.
    - Fonts.
    - Palettes. These control the mapping of colour indices to colours in the display.
    - Bitmaps and DIBs. DIBs are Device Independent Bitmaps. These are self-explanatory.
    - Regions. These are commonly used for restricting display operations.
    - Icons. These are special bitmaps mainly used to represent programs.
    - Cursors. These vary the shape used for the mouse.

The *ObjectWindows* library is more time-efficient than a switch statement, as it sets up a table of messages and handler functions. When a message is received this table can be interrogated, using the message value as a hash value, and the address of the relevant function retrieved. This reduces what is effectively a linear look-up (`if` ... `then` ... `else if` ... `then else` ...) in the number of messages handled to a constant overhead from the hashing. On the down side it does require space to store the message table.

8. The data structures

This chapter discusses the various data structures developed in the course of the project. The purpose and main features of each are outlined.

The first section looks at the central group of data structures, the `Figure` classes. These are used throughout all the modules. The supporting classes needed for the animation are then detailed in the following sections.

Throughout this discussion I will use the term "class" solely to mean the C++ datatype, and use synonyms where the meaning "set" or "group" is intended.

8.1 It figures.

The central construct of a Scottish Country Dance was argued in section 3.2 to be the figure. The main category of data structures developed was therefore intended to provide a representation of this. There was more than one such data structure because different structures were appropriate for different purposes. All the structures had some elements in common, with some having more in common than others. This suggested building a heirarchy of classes, with the common properties built into the base classes. The other classes would then inherit these properties and refine them, or add new properties. In this way an acyclic graph of classes is built up. Figure 8.1 shows the hierarchy developed.

8.1.1 The central properties

The properties central to all figures include their name, starting and finishing times and duration. Obviously only two of the three times need to be stored, since the third can be calculated from the other two: `end = start + duration`. It was decided to store the start time and the duration, and to calculate the end time when it was required.

Other central properties associate the `Figure` with lists of dancers, parameters, and starting and ending locations. These are all based on the dynamic array `template` class described in section 6.3. The dancer list gives indices into the central array of dancers, described in section 8.3.2. The parameters are `int`s, with symbolic parameters, such as the hand used in a turn, encoded as predefined constants. The starting and ending locations are not necessary; they allow these to be specified where they do not follow automatically from the figure calculations.

An equality operator is defined which compares the names of the two figures and returns `TRUE` (1) if they are the same. Another property tests whether the figure is atomic, ie whether the name matches one of the predefined figures. If it is atomic then a final property returns a unique identifier for that atomic figure. Composite figures are all assigned the `composite` identifier. This allows the animation routines to indentify the figure quickly without the need for a string matching.

The class `FigureInfo` encapsulates all of these. There are three branches of subclass derived from it. These provide a class for holding descriptions of figures in a central look-up table, a class to do the same job for figures being animated, and classes to provide the animation routines.

8.1.2 The subfigure description classes

These two classes provide a description of a subfigure of a main figure. They provide the details given above in section 8.1.1, and also hold pointers to fuller descriptions of

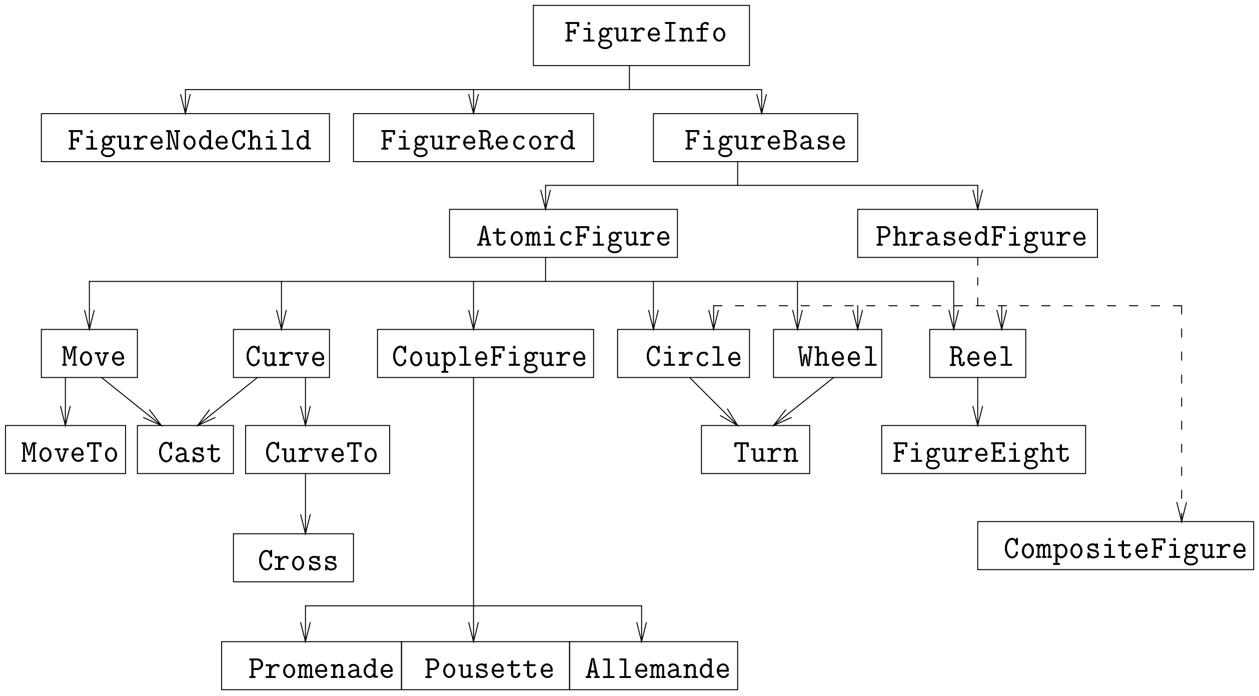


Figure 8.1: The `Figure` hierarchy.

the figure. `FigureNodeChild` provides this functionality for the entries in the central table of definitions, and `FigureRecord` does its job for the animated composite figures. They both provide a `<=` operator. The main difference between the two classes is that `FigureNodeChild` is sorted by its name, whereas `FigureRecord` is sorted by starting time. It is possible that the two classes could have been combined, but by the time this was realised it was felt more effort would be expended than the gains justified.

8.1.3 The animation `Figure` classes

These are the classes which do the work during the animation. They all inherit from `FigureBase`, which provides the housekeeping functions associated with the animation: adding the figure to and removing it from the list of active figures, updating pointers to the `Set`(s) being acted upon, and indicating whether the figure is actually active and running. `FigureBase` also provides prototypes for the functions `Start`, `Finish`, `StartLocation`, `EndLocation` and `Evaluate`. These are called at appropriate stages in the animation. They are all defined as `virtual` functions, as described in section 6.1. This means new classes can be added without affecting the code which uses them.

The functions are described in more detail below.

`Start` The `Start` function does any setting up required. This may be as little as adding the figure to the list of active figures, or may include some quite substantial calculations, as in the `Curve` or `Circular` figures (sections 9.2.2 and 9.2.5). Note that even when this function is redefined in a derived class, the base function is also called using the *BaseClass*::*VirtualFunction* syntax to ensure that the necessary initial setting up is performed.

`Finish` The `Finish` function removes the figure from the active list. It is not usually redefined, but may be if a figure needs to tidy up.

`StartLocation` **and** `EndLocation` These functions provide the locations of the dancers at the start and end of the figure. This is useful to allow animations to run both forwards and backwards, and for jumping in at an arbitrary point.

`Evaluate` The `Evaluate` function does the main work. It is called at each timestep for each figure in the active list. This function is almost always redefined to provide altered functionality. The only exceptions are where figures are special cases of other figures, as `Cross`es are special cases of `Curve`s, for instance.

The different figures fall into two groups: atomic figures, described in section 9.2, and composite figures, encapsulated in the one class `CompositeFigure`. This is described below.

8.1.4 The `CompositeFigure` class

A composite figure is built up out of smaller figures, which may be either atomic or again composite. The class `CompositeFigure` holds a list of subfigures represented as `FigureRecord`s. These subfigures are obtained from the description given in the file.

When a `CompositeFigure` is called at time t it runs through its list of subfigures to see if there are any which should be active at that time and are not. If it finds any such subfigures it calls its `Start` function to makes it active. The figure itself is responsible for calling its `Finish` function when the time is past its finishing time.

An optimisation which is performed is to sort the subfigures in order of starting time, and to keep a pointer to the first subfigure not yet activated. This reduces the number of figures which need to be checked at each call, and hence the overhead of a fruitless call.

The possibility of suspending the figure until its children had finished was considered. This would remove the overhead when the figure had nothing to do. Naïvely this might seem quite simple—have the figure suspended when it starts up another figure, and have the child figure reactivate the parent figure when it finishes. This will not work, however, because it is possible for one figure to have two child figures, one of which starts while another is still progressing. If the parent figure was suspended, the first child figure would be started on time, but the second would not be started until the first had finished, if at all.

8.2 The central look-up table of figure definitions

The figures are read in from a file and stored in a central look-up table. The class `FigureDefs` encapsulates this.

It contains a dynamic array of `FigureNode`s used for storing the definitions of the figures. These contain the names (full and abbreviated) of the figures, together with a list of `FigureRecord`s as described in section 8.1.2 and the extra details of a dance, such as devisor and tunes, if this is relevant.

The `FigureDefs` class provides functions to add a new figure to the definitions, to add another subfigure to an existing defined figure and to access the details of a given record. It also provides the names of all the defined figures in an array of `char`. This is used during the animation to offer the user a list of figures for animation.

8.3 The supporting animation data structures

Stéphane Chatty puts forward a model for animated interfaces [17] which has four types of entity: *tempos*, *instruments*, *rhythms* and *dancers*. *Dancers* are the visible end of the structure, which move or change shape according to the *notes* which they are sent. These *notes*, really messages in disguise, are sent by *instruments* every time they receive a message from a *tempo* or *rhythm*. *Tempos* emit steady beats, and *rhythms* filter these beats to give patterned beats. *Notes* may be positions, colours, numbers, or even real musical notes. Note that only the *tempos* have to worry about real-time issues—all the others act on messages they receive.

So what application does this have to this project? The idea of separating out the various functions into separate data types seemed useful, as did the hiding of the real-time issues inside one module.

8.3.1 The real-time module

The `Tempo` class handles all the real-time issues. It holds a doubly-linked list of `FigureBase` or derived classes as described above. These are the currently active figures. Note that the polymorphism of C++ allows the `Tempo` class to treat all derived classes as if they were `FigureBase`s; in fact the `Tempo` class need not know if there are a hundred derived classes or none at all.

When the `Start` function is called, the `Tempo` class starts off a timer and installs a callback to receive the timer messages. This callback function runs down the list of active figures held by the `Tempo` class, calling the `Evaluate` function for each figure. Once all figures have been dealt with the `Tempo` calls the `Display` function for all the `Sets` (see the next section). This causes the `Sets` to call the `Display` functions of their component `Dancers` which in turn causes the dancers to redraw themselves in their new positions on screen.

There is a slight subtlety here in that the callback is a member function of a C++ class. All member functions expect a hidden parameter `this` which is a pointer to the specific instantiation of the class. MS-Windows does not know which class installed the callback, and so cannot pass this hidden parameter when calling it. The way round this problem is to make the callback function a `static` member of the class. This means it does not expect the parameter `this`. However it also means the function has no way of knowing which instantiation of the class has the data it needs. In fact, it has no access to any of the instantiation-specific data of any of the instantiations. This was handled by having a `static` data member of the class, which is common to all instantiations of the class, and also to `static` member functions. This was a pointer to an instantiation of the `Tempo` class. One drawback with this approach is that only one `Tempo` class can be active at any one time. In this case this is not actually a problem. If it was then the `static` pointer could be changed, to a list of pointers, for example.

8.3.2 The display classes: `Dancers` and `Sets`

Sets

It may be thought that a set is just a collection of dancers, and a dancer is a dancer, but even here there is justification for a hierarchy. Sets may or may not be arranged into couples—the Duke of Perth does have the dancers arranged in pairs of men and women [4], but the Dashing White Sergeant has groups of three (two women and a man or two men and a woman) facing each other [6]. Even in sets with couples the couples may be arranged in a longwise set (Duke of Perth), in two lines of couples facing each other (La Tempête) [5] or in a square (Clutha [10]) or triangular (The Wind On Loch Fyne [15]) set.

The *inheritance* and *polymorphic* features of C++ have again been used to good effect here. All sets have a collection of `Dancer`s and an associated display window. This is a `TWindow` as supplied in the *ObjectWindows* class library (section 7.2). There are also the `virtual` functions `ResetSet` to restore the dancers to their initial positions, `Display` to display the dancers on the window, and `Clear` to remove the displayed dancer preparatory to displaying her in a new location.

The `protected` functions `AllocateDancers`, `StartUp` and `InitialiseSet` are used when an instance of the class is first created. These have the following uses:

`AllocateDancers` This function is responsible for creating the `Dancer`s for the set. This is redefined in derived classes to allow different types of `Dancer`s to be created—four women and four men in a square set, or four women and two men for the Dashing White Sergeant, for example.

`InitialiseSet` This function takes the `Dancer`s created by the previous function, and initialises their locations appropriately—in two rows with men facing women for a longwise set, or around the perimeter of a circle, with men on their partner's left in a circular set, for example.

`StartUp` This function is used in the work-around for the problem described below.

There was a problem with initialising the set when first created. Ideally the constructor of the base class would call the functions `AllocateDancers` and `InitialiseSet`. These calls would then be mapped to the virtual functions of the derived class being instantiated. However the `vtable`, the table of pointers to functions used in resolving calls to virtual functions, is only updated on entry to the body of the appropriate class constructor. The constructors of base classes are called before the body is entered, and so the table gets filled in from the bottom up: each class calls the constructor(s) of the class(es) below, which fills in the `vtable` up to that point, does any initialisation and returns. The `vtable` is then updated for the current class, before its constructor body is called. This means that at the time of the base constructor being called, the `vtable` only contains the pointers corresponding to the base class, and so the constructor does not know about redefinitions later in the hierarchy.

To get around this, each constructor takes a flag as an optional parameter, defaulting to `FALSE`. A derived class calls the constructor below it with the flag set to `TRUE`. If the flag is `FALSE` this class is the topmost in the hierarchy, and so it calls its `StartUp` function which does the necessary initialisation. The `vtable` is up to date because there are no derived classes to redefine any of the functions and so the correct version of the function is called. If the flag is `TRUE` the constructor does no initialisation (other than that provided by the compiler, such as `vtable` initialisation) and returns. This work-around is less than ideal. It is difficult to see how the compiler could efficiently act otherwise, however.

Dancers

The dancers do have a lot in common, obviously. They have a position, and a number indicating to which couple/group of the set they belong. However, the display functions for men and women need to be different—men are shown as circles, women as squares, by convention. So instead of building this into a display routine, why not have separate classes with a virtual function `Display` which is defined differently for each class.

This polymorphism bears fruit when the figures involving couples acting as a unit, such as an *Allemande* or a *Promenade*, or even a standard movement with a couple acting as one of the participating dancers, such as in *A Tribute to the Borders* [11]. A new type of dancer can be defined which is composed of two dancers joined together. These can then be moved around exactly as if they were one dancer, and the `SetPosition` routine of the new `Dancer` class will handle the positioning of the two members of the couple.

9. The derivation of the atomic figure functions

This chapter describes the atomic figures developed, together with the mathematical background to each and a list of their various options and parameters. The coordinate system used is also discussed.

It is assumed that the reader is familiar with the dancing terminology in section 3.1.

9.1 The coordinate system

In this section the coordinate systems developed for the different types of starting formation are laid out, and their validity is argued.

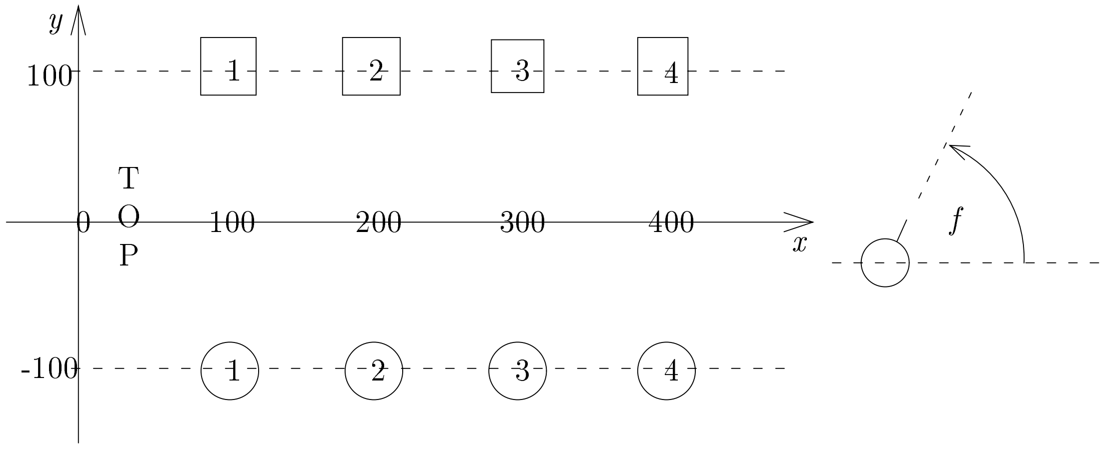


Figure 9.1: The longwise set coordinate system

Each dancer has three positional coordinates, (x, y, f). The x and y coordinates are the standard cartesian coordinates, and the f coordinate indicates which direction the dancer is facing, measured as an anticlockwise angle from parallel to the x-axis.

Angles are measured in degrees in the files and in radians internally, and can be fractional.

Figures 9.1 and 9.2 show the coordinate system used for longwise and square sets respectively.

9.1.1 Longwise sets

In a longwise set the women are at positions $(\langle\text{number} \times 100\rangle, 100)$, with the men opposite them at $(\langle\text{number} \times 100\rangle, -100)$. This has the following advantages:

- The diagrams have the same orientation as in Pilling, with the top of the set on the left of the diagram. [2].
- The positions can be read off directly from the x-coordinates—a coordinate of 100 is in first place, while a coordinate of 250 is half way between second and third places. Also, positive y-coordinates are on the women's side of centre, negative on the men's, with 100 on the sideline.

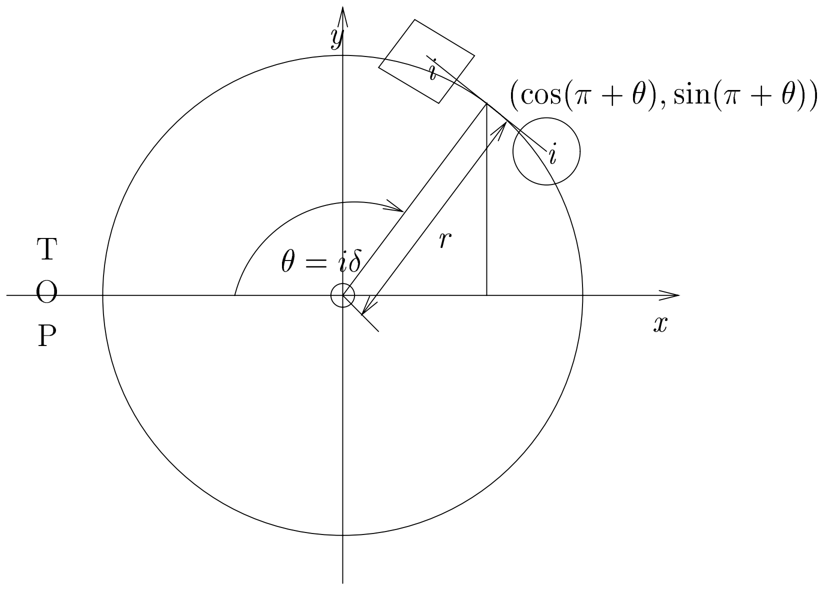


Figure 9.2: The square set coordinate system

- The spacing is close to the actual spacing used when dancing, where the space between the two lines is roughly twice the space between people in a line.
- The centreline of the set runs down the x-axis. This means that the symmetry inherent in the set layout is also inherent in the coordinate system.
- The f coordinate—the angle faced—is measured as an angle anticlockwise from facing directly down the set. This is intuitively reasonable.

9.1.2 Square, triangular and other polygonal sets

In a "polygonal" set, by which I mean any set arranged in a circular fashion, so square sets, triangular sets, hexagonal sets, and so on, the couples are arranged at equally spaced intervals around the perimeter of a circle centered on the origin, and face towards the origin.

The advantages of this layout are:

- It generalises readily to any number of couples.
- By continuing to use cartesian coordinates the straight-line movements can be carried straight across, whereas using polar coordinates would call for some horrendous maths even to describe a straight line if it did not pass through the origin. (I know—I worked it out).
- By having the set centered on the origin the coordinates can easily be converted to polar coordinates where this makes the calculations simpler.
- The diagrams again follow the Pilling standard, with the top of the set on the left of the diagram.
- People standing next to each other (partners in this case) have the same distance between them as in a longwise set.

To be more precise about the positioning, the midpoint between the two members of a couple is on the circle, and the two dancers are 100 units apart along the tangent to the circle at that point. The couples are numbered clockwise from the couple at the top of the set, which is again on the left. After some experimentation the radius of the circle was set to be 50 × (number-of-couples − 1), a value which was found to give a convincing spacing for the numbers of couples tried.

Working through this, let N be the number of couples in the set, and let all angles be measured in radians (since these calculations are only relevant internally). We then have

$$r = 50 \times (N - 1) \tag{9.1}$$

Also

$$TopOfSet = \pi \qquad \longleftarrow \qquad \text{at the negative end of the } x\text{-axis} \tag{9.2}$$

The angle δ between couples is

$$\delta = \frac{2 \times \pi}{N} \tag{9.3}$$

and so the angle of rotation θ_i for couple i, measured anticlockwise from the x-axis is

$$\theta_i = TopOfSet - ((N - 1) \times \delta) \tag{9.4}$$

From this the centre point C_i for couple i is

$$C_i = (r \cos\theta_i, r \sin\theta_i, \theta_i + \pi) \tag{9.5}$$

The tangent is at right angles to the radius, and hence the f-coordinate, so at $\theta_i{+}\pi/2$. The woman is 50 units along the tangent anitclockwise from the point of intersection, and the man the same distance clockwise. This offset Δ_i is

$$\Delta_i = (50 \cos(\theta + \pi/2), 50 \sin(\theta + \pi/2), 0) \tag{9.6}$$

The woman's position L_i is $C_i + \Delta_i$ and the man's, M_i is $C_i - \Delta_i$, which gives us the final equations

$$\begin{aligned} L_i &= (r\cos\theta_i + 50\cos(\theta + \pi/2), r\sin\theta_i + 50\sin(\theta + \pi/2), \theta_i + \pi) && (9.7) \\ M_i &= (r\cos\theta_i - 50\cos(\theta + \pi/2), r\sin\theta_i - 50\sin(\theta + \pi/2), \theta_i + \pi) && (9.8) \end{aligned}$$

9.2 The atomic figures

This describes the actual figures. All the figures are defined as functions to allow arbitrary precision in the calculation of locations. They receive the current time in bars, counted from the start of the animation and, with one exception, convert this into a fraction between 0 and 1 of the overall duration of the movement.

9.2.1 Straight-line movements

Command: `Move` or `MV` and `MoveTo` or `MVT`.

The figures `Move` and `MoveTo` encapsulate a movement in a straight line. `Move` is a

Figure:	Parameters	Description
`Move`:	x, y and f displacement. Default: 0 0 0.	A relative movement in a straight line.
`Curve`:	x and y displacement, plus angle subtended by arc. Default: 0 0 90.	A relative movement in a curve.
`CurveTo`:	x and y location of endpoint, plus angle as above. Default: 0 0 90.	An absolute movement in a curve.
`Cross`:	The hand grip used (LH, RH, BH or N[one]), and x and y displacements to be added to the destination. Default: N 0 0.	Exchanges the position of the two dancers using a 90° curve.
`CrossDown`:	The hand grip used, and the number of places moved down. Default: N 0.	As above, but the move appropriate to the number of places is added to the destination.
`CrossUp`:	The hand grip used, and the number of places moved up. Default: N 0.	As above, but the move is in the opposite direction.
`Cast`:	The number of places cast. Default: 1.	A cast off by the number of places. A negative number of places gives a cast up.
`CircularFigure`:	The direction (L/R/B) and percentage of a complete circle. Default: B 100.	Moves all the participants in a circle around the average of the starting positions.
`Circle`:	As Above.	As above, but has the participants holding hands around the circle.
`Wheel`:	As above.	As above, but has the participants' hands meeting at the centre point.
`Reel`:	The number of participants, the starting shoulder and the percentage of a complete reel. Default: nParticipants R 100.	Does an appropriate reel with the axis set by the end participants.
`FigureEight`:	The start (XU/XD/CU/CD) and the percentage. Default: XD 100.	Based on a reel of three.
`StepDown`:	The number of places. Default: 1.	Step down the number of places in four steps taking two bars.
`StepUp`:	The number of places. Default: 1.	Step up the number of places in four steps taking two bars.

Table 9.1: The atomic figures defined—quick reference.

relative movement, where the parameters are added onto the starting position, while `MoveTo` is an absolute movement, leaving the dancer at the specified location at the end of the movement. `MoveTo` converts the absolute coordinates to relative coordinates for each dancer in the list and passes them on to `Move`. This uses simple linear interpolation on the three coordinates independently to calculate the new coordinates.

So for a dancer at (x_d, y_d, f_d) with input (x_t, y_t, f_t) `MoveTo` would pass `Move` the relative coordinates $(x_r, y_r, f_r) = (x_t - x_d, y_t - y_d, f_t - f_d)$. To calculate the position at time $\Delta t \in [0, 1]$ `Move` would use the equations

$$x = x_d + (\Delta t \times x_r) \tag{9.9}$$
$$y = y_d + (\Delta t \times y_r) \tag{9.10}$$
$$f = f_d + (\Delta t \times f_r) \tag{9.11}$$

If more than one dancer is specified, this procedure is carried out for each dancer given. If more than one set of parameters is passed to `Move` they are cycled through until all dancers have had positions calculated. For instance, if six parameters were passed (two sets) and there were three dancers listed, the first dancer would get the first set of parameters, the second dancer would get the second set of parameters, and the third dancer would get the first set of parameters again.

Any missing parameters are set to zero. So if the parameters passed were 1,2,3,4 `Move` would have two sets of parameters: (1,2,3) and (4,0,0).

These are features built into the program to make it more error-tolerant, and to allow the devisor to take shortcuts where she wishes.

9.2.2 Curved movements

Command: `Curve` or `CV` and `CurveTo` or `CVT`.

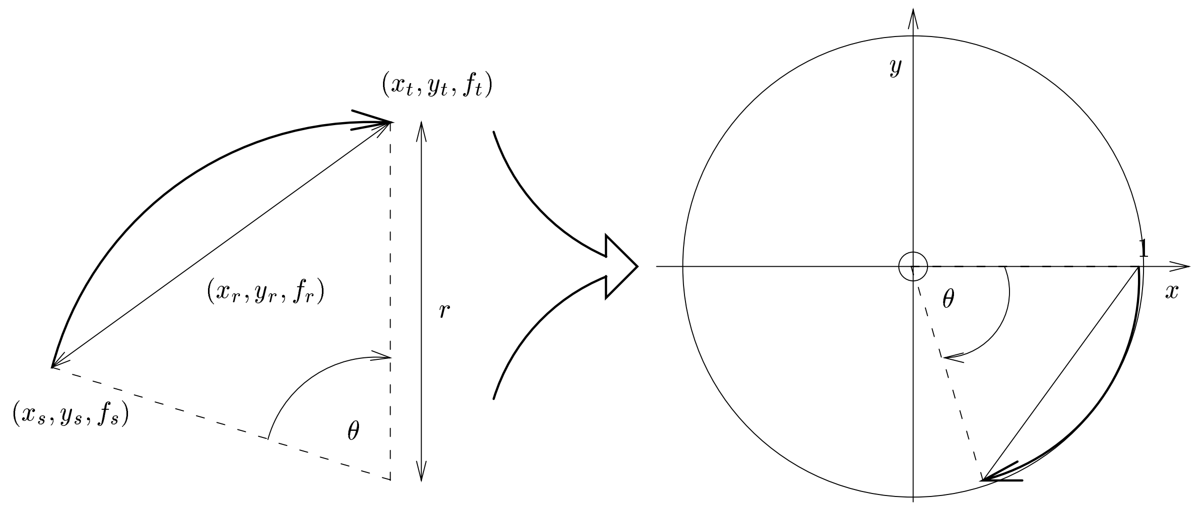


Figure 9.3: The mapping for the maths of the `Curve` figure.

An absolute curve, `CurveTo`, is converted internally to the corresponding relative curve. For instance, if the dancer is at (x_d, y_d, f_d) and the parameters are x_t, y_t and θ the parameters are converted to $x_t - x_d$, $y_t - y_d$, *theta*. The relative curve `Curve` is then called with these parameters.

A relative curve is specified by its starting position, the displacement of the endpoint, and the angle of curvature. The starting position is set by the position of the dancer at the start of the movement, and the x and y components of the displacement and the angle of curvature are given as parameters to the figure. If not specified, the x and y coordinates are assumed to be zero and the angle is assumed to be 90°, which was found to give a realistic curve.

The way this function works is to map the curve onto an arc on the unit circle with the starting point mapped to $(1, 0)$. The position along this arc is then calculated, and the result rotated and scaled to give the true coordinates. The mapping, which is constant throughout the figure, is calculated when the figure is initialised, and so much less work needs to be done at each step of the animation.

The calculation of the values used in the mapping is done as follows. Given an offset $(\Delta x, \Delta y)$ and an angle of curvature θ the angle is first converted from degrees to radians.

The length of the chord C_b whose angle subtended at the centre is θ is the first thing calculated:

$$C_b = \sqrt{}$$ [illegible] S

$$S = 2\frac{C_r}{C_b}$$

The anticlockwise angle from the positive end of the x-a
is stored in ϕ:

$$\phi = \tan^{-1}\left(\frac{y}{x}\right)$$

and from this the angle of rotation in the mapping, ψ, i

$$\psi = \begin{cases} \phi - \frac{1}{2}(\pi + \theta) & : \theta \geq 0 \\ \phi + \frac{1}{2}(\pi + \theta) & : \theta < 0 \end{cases}$$

All the preliminary calulations have now been done. W
is called at time $\Delta t \in [0, 1]$ only a couple of calculations n
the point on the unit circle and rotate and scale it up. But
at time Δt is calculated:

$$\theta' = \theta \times \Delta t$$

This makes the point on the unit circle

$$p_u = (\cos\theta', \sin\theta')$$

This is now to be rotated about the endpoint $(1, 0)$ or
origin must first be moved to $(1, 0)$. This gives

$$p_i = (\cos\theta' - 1, \sin\theta')$$

Now for the rotation itself. Here the rotation matrices used in Computer Graphics [23, 24, 25] come in useful. To rotate a point (x, y) by an angle θ anticlockwise about the origin the calculation is as follows:

$$(x', y') = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{9.20}$$

Multiplying out, with p_i our initial point and ψ our angle of rotation, and scaling up by S we get

$$x' = S\left(\cos\psi\left(\cos\theta' - 1\right) - \sin\psi \sin\theta'\right) \tag{9.21}$$

$$y' = S\left(\sin\psi\left(\cos\theta' - 1\right) + \cos\psi \sin\theta'\right) \tag{9.22}$$

This is the displacement from the starting point (x_d, y_d, f_d). The actual position can be found by adding the two together. The dancer is assumed to be facing along the tangent at the point, so the f-coordinate is

$$f' = \tan^{-1}\left(\frac{y}{x}\right) \tag{9.23}$$

The calculated point is then

$$(x, y, f) = (x_d + x', y_d + y', f') \tag{9.24}$$

Note that the only calculations made at time Δt are θ' (9.17), x' and y' (9.21 and 9.22), f' (9.23) and the final point (9.24).

9.2.3 Crosses

Command: `Cross` or `C`, `CrossUp` or `XU` and `CrossDown` or `XD`.

A cross is based on an absolute curve. It exchanges the positions of the two dancers using a 90° curve, adding the optional offset to both end positions. So for dancer A at (x_A, y_A, f_A) and dancer B at (x_B, y_B, f_B), with offsets Δx and Δy, dancer A would have a `CurveTo` with parameters $x_B + \Delta x$, $y_B + \Delta y$, 90, and dancer B would have a `CurveTo` with parameters $x_A + \Delta x$, $y_A + \Delta y$, 90.

`CrossDown` is implemented using a cross with offset $nPlaces \times 100$, 0, where $nPlaces$ is the second parameter to the figure, while `CrossUp` is passed straight on to `CrossDown` with the parameter negated.

The first parameter to all the variants gives the hand grip used in the cross. This may be `L`, `R` or `B` meaning left hand, right hand and both hands respectively. Each of these may be followed by an `H`, meaning hand or an `S` meaning shoulder. In the second case no hands are given. The parameter may also be `N`, which may be followed by an `H` and an `R` or an `L` indicating the shoulder. If no shoulder is indicated it is assumed to be right.

If no parameter is given, `NHR` is taken by default.

The angle of the curve is set by the side of passing: a left hand pass calls for a 90° curve, while a right hand or both hand pass calls for a −90° curve.

9.2.4 Casting off and up

Command: `Cast` or `C`.

A cast involves the dancer curving out behind the other dancers on that side of the set, dancing up or down the set the requisite number of places, and then curving back into the sidelines.

If casting off on the womens' side or up on the mens' side this start with a 90°curve to the right—a $-90°$ anticlockwise curve. If casting up on the womens' side or off on the mens' this calls for a $+90°$ curve at the start. This curve has x-offset 50 for casting off and -50 for casting up, with y-offset 50 on the womens' side, and -50 on the mens'. It is followed by a `Move` of $(100(N-1), 0)$ for casting off or $(-100(N-1), 0)$ if casting up, where N is the number of places cast. The movement finishes with another curve identical to the first, but with the y-coordinate negated.

9.2.5 Circular figures

Command: There is no corresponding command.

This is an internal figure used by `Circles` and `Wheels` (sections 9.2.5 and 9.2.5). It moves the dancers around in a circle, centred on the midpoint of the dancers, and of radius $r = 25N$. The midpoint (x_C, y_C) is calculated by taking the average of the x- and y-coordinates of the dancers.

The dancers may be given in any order, but are sorted internally in order anticlockwise from a line running from the midpoint parallel to the positive end of the x-axis.

If the dancers are placed with the first dancer on the x-axis, and the others equally spaced around the perimeter of a circle, the calculations are easy—just add an offset multiplied by the appropriate amount to a calculated starting angle, and take the cosine and sine of the resulting angle for the x- and y-coordinates. However it is more usual for the dancers to be offset from the canonic positions. To allow for this, an angular offset is also calculated when the figure is initialised. This is calculated as follows, where N is the number of dancers.

The angle between dancers, δ, is

$$\delta = 2\pi/N \tag{9.25}$$

The offset for dancer i, ϕ_i is then the calculated angle minus the actual angle, or, if (x_d, y_d, f_d) is the initial position

$$\phi_i = i\delta - \tan^{-1}\left(\frac{y_d}{x_d}\right) \tag{9.26}$$

The offset used in the calculations, ϕ_{Av} is then

$$\phi_{Av} = \frac{1}{N}\sum_{i=1}^{N}\phi_i \tag{9.27}$$

When the positions are calculated, the base angle of rotation, θ, is first calculated. For a circle one way only, this is $\Delta t \times 2\pi p/100$ for a circle to the left, or the negative of this for a circle to the right. For a circle round and back it is

$$\theta = \begin{cases} \Delta t \times 4\pi p/100 & : \Delta t < \frac{1}{2} \\ (1-\Delta t) \times 4\pi p/100 & : \Delta t \geq \frac{1}{2} \end{cases} \tag{9.28}$$

So for the first half the angle increases, and for the second half it decreases back to zero. In these equations p indicates the percentage of a complete circle, and is given in the second parameter. If not specified, it defaults to 100.

From this angle θ the positions can be calculated. For dancer i the coordinates are:

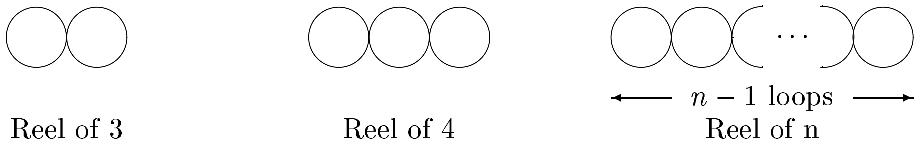


Figure 9.4: Reels of three, four and more.

$$
\begin{aligned}
x &= x_C + r\cos(\theta + i\delta) \qquad (9.29)\\
y &= y_C + r\sin(\theta + i\delta) \qquad (9.30)\\
f &= -\tan^{-1}\left(\frac{y - y_C}{x - x_C}\right) \qquad (9.31)
\end{aligned}
$$

If the circle is not a complete circle, the dancers are left in a position based on their starting positions, but rotated through $2\pi p/100$. This uses the standard rotation as described in section 9.2.2. So for a dancer at (x_d, y_d, f_d), after $p\%$ of a circle, the final position is calculated as follows (for a circle to the right):

$$
\begin{aligned}
\theta_i &= 2\pi p/100 + \tan^{-1}\left(\frac{y_d - y_C}{x_d - x_C}\right) \qquad (9.32)\\
r_i &= \sqrt{(x_d - x_C)^2 + (y_d - y_C)^2} \qquad (9.33)\\
x_i &= r_i\cos\theta_i + x_C \qquad (9.34)\\
y_i &= r_i\sin\theta_i + y_C \qquad (9.35)\\
f_i &= -\theta_i \qquad (9.36)
\end{aligned}
$$

Circles

Command: `Circle` or `O`.

This figure is based on the circular figure but has the participants holding nearer hands while performing it.

Wheels, stars and turns

Command: `Wheel` or `W`, `HandsAcross` or `HA` and `Turn` or `T`.

This is based on the circular figure (section 9.2.5), but has the dancers all with their right hands in the middle when circling to the left, or their left hands in the middle when circling to the right.

Note that a single handed turn is a wheel with only two dancers. Note also that a left-handed wheel is a wheel to the right, and vice versa.

9.2.6 Reels

Command: `Reel` or `R`.

A *reel*, known as a *hay* in English Country Dancing, is a figure in which the dancers weave in and out up and down a line back to their original starting place, passing other dancers alternately by the right and left shoulder. All dancers are travelling along the same path; the only difference between them is where on the path they start. For instance, in a *reel of three*, involving three dancers, each dancer describes a figure 8 on the floor. In a *reel of four*, an extra loop is added. In general, for N dancers, there are $N-1$ loops (see figure 9.4).

The mathematical object which most closely matches this is a *Lissajou figure*. This is formed by plotting $\sin(k_1 t)$ against $\sin(k_2 t + c)$, where k_1, k_2 and c are constants. For n loops k_1 and k_2 should be chosen so that $k_1/k_2 = n$.

The basic Lissajou figure has its endpoints at $(\pm 1, 0)$. This means a scaling and rotation must again be performed, as with the `Curve`. The base points of the reel, matching the endpoints of the Lissajou figure, are taken to be the positions of the endmost dancers. For a reel of three these are the second and third dancers; for all other reels these are the first and last dancers. Let them be at positions (x_a, y_a, f_a) and (x_b, y_b, f_b).

The scale is set from the ratio of the sizes of the axes in the two figures: C_b in the base figure and C_r in the actual reel. In these calculations all working is done in screen units, so C_b is $2 \times 100 = 200$. C_r is

$$P_\Delta = (\Delta x, \Delta y) = (x_b - x_a, y_b - y_a) \tag{9.37}$$

$$C_r = \sqrt{\Delta x^2 + \Delta y^2} \tag{9.38}$$

and so the scale S is $C_r/C_b = C_r/200$. The angle of rotation, ψ, and the centre point of the reel, P_C, can also be calculated from P_Δ:

$$\psi = \tan^{-1}\left(\frac{\Delta y}{\Delta x}\right) \tag{9.39}$$

$$P_C = (x_C, y_C,) = \left(x_a + \frac{1}{2}\Delta x, y_a + \frac{1}{2}\Delta y\right) \tag{9.40}$$

That is all the preparatory calculation done. The positions of the dancers are worked out by calculating the positions on the canonical reel, $P_r = (x_r, y_r, f_r)$, and then rotating these by ψ and scaling by S:

$$x_i = x_C + S(x_r \cos\psi - y_r \sin\psi) \tag{9.41}$$

$$y_i = y_C + S(x_r \sin\psi + y_r \cos\psi) \tag{9.42}$$

$$f_i = f_r + \psi \tag{9.43}$$

The calculations for P_r are done in a subroutine. The base angle θ used in generating the position in the Lissajou figure at time $\Delta t \in [0,1]$ is $2\pi \times \Delta t$. First an offset ϕ to this basic angle is calculated depending on where the dancer started the reel. For a reel of three, for instance, this is

$$\phi_3 = \begin{cases} \textbf{Offset} : \textbf{Starting location} \\ 0 : \text{Centre} \\ \frac{\pi}{2} : \text{Leading end} \\ \frac{3\pi}{2} : \text{Trailing end} \\ \frac{\pi}{4} : \text{Casting down} \\ \frac{5\pi}{4} : \text{Casting up} \\ \frac{3\pi}{4} : \text{Crossing down} \\ \frac{7\pi}{4} : \text{Crossing up} \end{cases} \tag{9.44}$$

The last four cases are used in figures of eight. For a reel of four it is simpler:

$$\phi_4 = \begin{cases} \textbf{Offset} : \textbf{Dancer number} \\ \frac{3\pi}{2} : 1 \\ \frac{7\pi}{6} : 2 \\ \frac{\pi}{6} : 3 \\ \frac{\pi}{2} : 4 \end{cases} \tag{9.45}$$

In a reel of three two of the dancers collide in the centre if they don't avoid each other. To prevent this, one of them has priority over the other. The dancer with priority speeds up slightly while the other slows down when they pass through the centre, and they then return to their normal speeds. I have simulated this by perturbing the base angle by a sinusoidal offset δ when these dancers are near the centre. δ_L signifies the leading dancer's offset, and δ_T represents the trailing dancer's. These are:

$$\delta_L = \begin{cases} \frac{1}{5}\cos(2\theta) : \begin{cases} -\frac{\pi}{4} < \theta < \frac{\pi}{4} \\ \frac{3\pi}{4} < \theta < \frac{5\pi}{4} \\ \frac{7\pi}{4} < \theta < \frac{9\pi}{4} \end{cases} \\ 0 : \begin{cases} \frac{\pi}{4} \le \theta \le \frac{3\pi}{4} \\ \frac{5\pi}{4} \le \theta \le \frac{7\pi}{4} \end{cases} \end{cases} \tag{9.46}$$

$$\delta_T = -\delta_L \tag{9.47}$$

The values $\frac{1}{5}$ and 2 were derived by experimentation. For the other dancer, and for other reels, $\delta = 0$. From this we can now calculate the positions on the canonical figure, $P_r = (x_r, y_r, f_r)$.

$$x_r = \sin(\theta + \delta) \tag{9.48}$$

$$y_{r3} = \frac{1}{2}\sin((n-1)\theta) \tag{9.49}$$

$$y_{r4} = -\frac{1}{2}\sin\left((n-1)\theta + \frac{\pi}{2}\right) \tag{9.50}$$

$$f_r = \tan^{-1}\frac{(n-1)\cos((n-1)\theta)}{\cos\theta} \tag{9.51}$$

9.2.7 Figure of Eight

Command: `FigureEight` or `FIG8`.

The `FigureEight` figure uses the calculations for the reel of three. The initial setup is as described there, with the exception that the scale is multiplied by $\sqrt{2}$. The `Evaluate` function calls the `Reel` subroutine to calculate the canonical positions, and then scales and rotates this exactly as described in equations 9.41–9.43.

9.2.8 Stepping up and down

Command: `StepUp` or `SU` and `StepDown` or `SD`.

This is described by the RSCDS [14] as follows:

> This movement is used to leave a space for the dancing couple or to shorten the track over which they must dance. ...
>
> *Steps:* Four even steps to four counts.
>
> *Bars of Music:* Two.
>
> A: Stepping Up.
>
> 1. Step up to the side with the foot nearer the top of the set.
> 2. Step across in front of it with the other foot.
> 3. Step up to the side again with the first foot.
> 4. Close the other foot into first position.
>
> B: Stepping Down.
>
> This is the same movement as stepping up, but begins with the foot nearer the bottom of the set.
>
> *Note:* When two or more couples step up or down, they join nearer hands at shoulder height on the side lines. To assist a casting couple, it is helpful to step in and up on count 1 and back to the side line on count 3.

This is the only figure which does not convert the current time (in bars) into a fraction of the overall duration.

It takes one parameter, N, indicating the number of places to step up or down. If no parameter is passed it defaults to 1.

Note: `StepUp` is implemented by negating the parameter N and passing it on to `StepDown`.

The function acts as follows:

- If the time is between 0 and 0.5 bars after the start of the movement the function does nothing.
- If the time is between 0.5 and 1.0 bars after the start of the movement the function alters the position by $(25 \times N, 15, 0)$.
- If the time is between 1.0 and 1.5 bars after the start of the movement the function alters the position by $(50 \times N, 20, 0)$.
- If the time is between 1.5 and 2.0 bars after the start of the movement the function alters the position by $(75 \times N, 15, 0)$.
- If the time is over 2.0 bars after the start of the movement the function alters the position by $(100 \times N, 0, 0)$.

The x value is added for `StepDown` and subtracted for `StepUp`. The y value is added for men and subtracted for women—always moving towards the middle of the set.

9.2.9 Petronella

Command: `Petronella` or `PTR`.

The dancers perform a $-90°$ curve 144 units along a line at $45°$ clockwise to the direction they are facing.

9.3 Couple figures

The following figures all treat couples as a unit.

9.3.1 Promenade

Command: `Promenade` or `PROM`.

9.3.2 Allemande

Command: `Allemande` or `ALL`.

9.3.3 Poussette

Command: `Poussette` or `PST`.

This has several variants depending on the number of couples involved and the tempo of the dance (reel/jig or strathspey).

10. The dance compiler

This section provides a brief description of the approach taken to converting dances described in a file into the internal data structures used by all other modules of the program.

It was decided to implement a compiler by hand rather than using tools such as Lex and Yacc [28], since the input language was relatively simple, and more time might have been spent in learning how to use the tools than in writing it from scratch.

Most of the functions were member functions of the appropriate C++ classes. This approach had some benefits, notably in a clean approach to the coding where the object has control of its own modifications, and some drawbacks. The drawbacks stemmed mainly from a lack of global data in the lower level functions, making repeated sections of a dance less easy to implement, for instance.

Each function takes a C++ stream as its parameter and sets or updates the various data items of the class appropriately. This is fine when only one item needs to be set, but when there is a list of items with some common structures, such as in repeated figures, the class needs to create the extra structures and return them somehow. This was achieved by creating a copy of the class item, chaining it onto a list, and updating the current item.

With the above provisos, the compiler implemented was based on a standard recursive-descent parser [28], with each function constructing a token and calling other functions to construct component tokens. A `FigureDefs` structure was produced at the end, holding the relevant definitions. The fact that the parsing functions are member functions of C++ classes makes no difference to the approach.

11. The animation

Two possible approaches to producing an animation from the data structures developed were considered.

The first used the fact that the figure definitions produce a tree, with composite figures at the nodes, and atomic figures at the leaves. The active figures at various stages of the dance *The Duke Of Perth* are highlighted in figure 11.1.

This approach effectively performed a lookup on the data in the tree at each timestep. All child figures which started before the current time and finished after would be recursively evaluated. This requires the whole tree to be instantiated throughout the animation, even though there are nodes which play no part in the evaluation process at the time. This could be thought of as a repeated query.

The second approach had a list of active figures, along the lines of Stéphane Chatty's musical metaphor (section 8.1.3 and [17]). When a child figure was due to start, the parent figure would place it on the active list. Once a figure was past its end time, it would remove itself from the active list. This approach could be thought of as a single tree traversal. It has the benefit of requiring less storage space, since the figures only need to be instantiated while they are actually active. This was the approach selected and implemented.

As described in chapter 8, this led to the development of a class `Tempo` to receive timer events at set intervals. This class held a circular doubly linked list of active figures with a single dummy figure to provide a handle. New figures were inserted after the dummy figure. The figures themselves held the pointers and thus could be removed from the list with no need for a (linear time) search. This allowed insertion and deletion both to be performed in constant time. Lookup of arbitrary figures was not an issue since the list was traversed completely and each figure `Evaluate`d at each timestep.

11.1 Running an animation

To produce an animation the procedure is as follows:

1. A file containing one or more dances is loaded into the `FigureDefs` structure.
2. A list of available figures (remember a dance is just a figure) would be offered to the user. She would select one for running.
3. A `FigureBase` structure would be created from the appropriate definition record and added to the `Tempo`'s active list. Note that this may be a `CompositeFigure` or one of the various `AtomicFigure`s (see section 8.1).
4. A `Set` corresponding to the set type in the figure definition would be created and initialised. This would then be displayed in a window on screen.
5. The user would click on the `Play` button. This would cause a message to be sent to the `Tempo` to start up the timer.

Figure 11.1: The active figures during an animation of the dance *The Duke of Perth*, viewed as a tree. The time increases from left to right within each figure. Simultaneous figures are joined to their parent figures at the same point.

6. Various timer events would occur, so `Figure`s would be `Evaluate`d, start other figures, remove themselves, and update `Dancer` positions. At each timestep the new configuration of the `Dancer`s would be displayed on the window.
7. The user could stop, fast forward or rewind the animation. This is achieved by changing the amount by which the bar count is incremented at each timestep. If the animation is stopped the timer would also be stopped.
8. Eventually the last `Figure` would finish, leaving the active list empty. The timer would then be stopped.

11.2 Problems encountered

This section describes some of the problems encountered when producing the animations, and the solutions developed.

11.2.1 Screen refresh

The initial approach to the animation was made as simple as possible to ensure that all the mechanisms worked. The window was completely cleared at each timestep and the dancers displayed in their new positions. Not surprisingly, this approach was extremely slow and flickery. Various optimisations were tried.

XOR drawing The dancers were displayed using exclusive-or plotting. At each timestep the dancers would be redisplayed at their old positions to return the display at that point to its original state: $(A \operatorname{XOR} B) \operatorname{XOR} B = A$. They would then be displayed at their new positions. For some undetermined reason the XOR drawing did not work, and all drawing was done in “set” mode.

Memory BitBlts All drawing was done on a memory bitmap compatible with the window, and then this bitmap was copied to the screen using a bit-block transfer, or BitBlt. The bitmap was again wiped clean at each timestep by drawing a white rectangle the size of the bitmap on it. This cured the flicker, but still suffered from performance problems. It was decided to use a variant of this to avoid the flicker, and to try further optimisations.

Mini-cls By not clearing the screen at all, and just drawing the dancers in their new positions it was determined that this screen-clearing formed a large part of the overhead of the display. This suggested shrinking the area cleared to just that appropriate to the dancers. The `Dancer` classes were each given an extra `Clear` member function used to remove them from the display. These drew a white rectangle just large enough to cover the original display of the dancer, over the last location of the dancer. The memory drawing approach proved a boon here, since the “screen” could be cleared straight after it had been displayed, without causing flicker. This meant there was no need for storing the previous state of the dancers.

This approach finally bore fruit, allowing a reasonable refresh rate.

11.2.2 Finishing and synchronisation

The first animations produced were a mess. The first figure would be fine, but would not leave the dancers in the right positions. Since each figure works on the location of the dancers at the time when it is started, the second figure would have its dancers starting slightly off from where they should have been, and leave them even further away. The error would propagate in this way, similarly to round-off error, until by the end the dancers would be so out of place that the figures were unrecognisable.

The problem was that figures did not finish cleanly. To counteract this an extra `virtual` member function was added to the `AtomicFigure` classes which calculated the correct positions of the dancers at the end of the figure. This function, `EndLocations`, was then called when the figure was finishing. This made sure that the dancers ended in the correct places.

11.2.3 Phrasing

The `EndLocations` function made sure that dancers ended in the correct places when the figures finished. This led to another problem. The functions for some figures do not naturally leave their participating dancers in an appropriate place—the circular figures are an excellent example. This can lead to a sudden jump at the end of the figure.

Another related problem is when one figure needs the dancers in different locations to those in which the previous figure leaves them. An example would be where two reels of three on the sides, finishing with all the dancers on the sidelines, is followed by a figure involving corners, with the dancing couple in the middle of the dance to start. This was catered for by allowing the dance devisor to specify where the dancers should finish the figure and hence where they should start the next figure. However, it again causes the dancers to make a prodigious leap as the figure ends.

The solution implemented took advantage of the multiple inheritance features of C++: a class can inherit properties from more than one parent class. A new member function, `PhrasedEvaluate`, was added to the `FigureBase` class, which accepted the same parameters as the `Evaluate` function, and did nothing except pass these parameters straight through. A new class, `PhrasedFigure`, was developed. This was derived from `FigureBase` and overrode this `PhrasedEvaluate` function. Instead of simply calling `Evaluate` with all parameters unchanged, this function first checks the time. If it is between one and two bars from the end of the figure, it halves the time left. This means that the figure finishes one bar early. If there is less than a bar left, the function calculates the position on a straight line between the end of the figure and the specified ending location, using linear interpolation exactly as described in section 9.2.1. This copes with the most common case where the dancer must speed up and cover extra ground. It does not handle the case where the finishing position is on the track of the dancer, and the dancer must slow down to avoid arriving too early.

Figures for which phrasing was appropriate, such as `Wheel`, `Circle` and `CompositeFigure` had `PhrasedFigure` added to the list of parent classes, thus causing the new version of `PhrasedEvaluate` to override the version from `AtomicFigure`.

11.2.4 Reverse playback

Providing reverse playback proved awkward due to the sequential approach to animating dances. The functions were geared to calculating an offset from the initial starting positions. When the figures are being displayed backwards, with the time decreasing as the animation progresses, the positions of the dancers on entering the figure are the finishing positions, and so the offsets are based on the wrong position.

The solution to this was to run through the figures at the start of the dance, calculating and storing starting positions for each. The calculations could then use these starting positions whether the dance was being run forwards or backwards. A further benefit of this approach was that the user could jump in at any point in the dance, and the figures would still know where the dancers were supposed to be.

12. Conclusions

A file format has been defined, a data structure developed, and a parser written to build the data structure from files in the given format. An animation module has also been developed running under MS-Windows.

The implementation language, C++, allowed the implementation to model the problem quite naturally. The inheritance and polymorphism allowed clean efficient coding of related structures, and the templates provided a useful shortcut to generating similar data structures differing only in base type. There were various problems, described in the relevant sections, but on the whole the language was a help rather than a hindrance. A particularly useful feature was the class library provided with the compiler which allowed the interface to be built without reinventing the window.

12.1 Further work

Although the modules developed form a completed unit, there are various possible avenues for further exploration. These are exactly those modules described initially which were not implemented.

They are:

- A parser to handle cribs [very near to] the text format.
- Automatic generation of text and Pilling cribs from the dance description.
- A devisor's workshop along the lines of a drawing package.
- Databasing functions—search for dances by name, source, tune, figures, or a combination of these.
- A ball programme editor using the databasing functions.
- Music.

These are described in greater detail below.

12.1.1 A parser for RSCDS-style cribs

At first sight this might seem extremely ambitious. After all, the RSCDS cribs are written in English prose. However, closer examination reveals a definite structure to the cribs. This could be exploited in the definition of a language which is very close to the RSCDS cribs and yet is rigorous enough for a computer to parse.

For instance, looking at the sample RSCDS-style crib given in figure 3.1, it can be seen that it breaks down into two sections: a preamble giving the dance name, the source or devisor, the type, the initial configuration and the tunes; and the description of the dance.

The first eight bars provide a good example of the structure of the descriptions.

> 1–8 1st and 3rd couples dance a double figure of eight, 1st couple crossing down and 3rd couple casting up to start.

It starts with an indication of the bars on which the figure starts and ends. The participating dancers are then enumerated, followed by the figure danced and any qualifying information. Here the figure starts on bar 1 and runs until bar 8. First and third couples are involved, and they dance a double figure of eight. First couple start the figure by crossing down and third couple start by casting up.

Bars 17 to 24 have a figure which is defined in the description—in a manner completely analogous to the definition of the whole dance:

17–24 1st couple with 2nd and 3rd couples dance the Millwheel:

- 1–2 1st lady with 2nd couple and 1st man with 3rd couple dance right hands across in a wheel half way round.
- 3–4 2nd and 3rd couples dance left hands across in a wheel half way round *while* 1st couple chase 1/4 of the way clockwise round the set.
- 5–6 1st lady with 3rd and 2nd ladies, 1st man with 3rd and 2nd men dance right hands across in a wheel half way round.
- 7–8 1st couple turn 1 1/4 by the left hand to face first corners.

Bars 3–4 of this subdefinition illustrate how simultaneous actions are indicated: the word *while* separates two descriptions.

There is quite a lot of structure already, and I feel that with only a small amount of extra formality a full parser could be built.

12.1.2 Automated crib generation

The idea here is to take the dances held internally in the data structure and generate cribs on screen or on a printer in either the RSCDS-style or Pilling-style formats.

This involves mainly a look-up table of outputs for the different figures, together with some way of specifying the participants. Some elements of typesetting would need to be incorporated to get a reasonable display.

12.1.3 A devising package

This would be a module to help in devising a new dance. It would probably be in the style of a drawing package, with palettes or menus of figures which are dragged and dropped into place. These figures could then have their participants and parameters edited. This module would probably build on the Pilling-style diagram module.

12.1.4 A dance database

Since the dances are on computer, it should be a relatively simple task to create a database of the dances. This would allow dances to be found by name, devisor, tune, set type, component figures, or a mixture of attributes. It could be used to generate an index of formations and associated dances similar to that published by the RSCDS [12].

12.1.5 A ball programme editor

This would be a module to aid the writing of ball programmes. There are guidelines for putting together such programmes, most of which are common sense. For instance it is recommended to aim for a fast, fast, slow, fast, fast, slow, ..., fast, fast ordering of dances. Also a mix of dances is recommended, with as wide a spread of figures as is possible. If the same figure comes up too many times people get bored. One particularly wants to avoid having dances which have a large degree of overlap on the same programme. I remember once dancing *The Deil Amang the Tailors* [8] followed by *Berwick Johnnie* [9]. These dances differ only in their last eight bars and one is a reel and the other a jig; in fact a reel was played for the jig as well. People were not amused. A database such as described in 12.1.4 above would help avoid such occurences.

12.1.6 Music

Scottish Country Dancesare always done to music. Accordingly it would be good to allow music to be stored in the database and/or played in time to the animation. It would probably be quite a large task to add this capability, however. Particular thought would need to be paid as to how the music would be entered. Would a language be developed, or would an interactive score editor be written, or what?

Bibliography

[1] F. L. Pilling. *Scottish Country Dances in Diagrams.* Leeds, 1955.

[2] C. N. Ribbeck, J. P. Duckett, S. Duckett, J. B. Elsley and H. Williams. *Scottish Country Dances in Diagrams.* Sixth Edition, Chester, 1992.

[3] *The Scottish Country Dance Books 1–37, assorted other books of dances.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, 1927–1993.

[4] *Book 1: The Scottish Country Dance Book.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, 1927, reprinted Hawick, 1985.

[5] *Book 2: The Scottish Country Dance Book.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, reprinted 1985.

[6] *Book 3: The Scottish Country Dance Book.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, reprinted 1985.

[7] *Book 11: The Scottish Country Dance Book.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, reprinted 1986.

[8] *Book 14: The Scottish Country Dance Book.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, reprinted 1986.

[9] *The Book of Graded Scottish Country Dances.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF.

[10] *Book 31: The Scottish Country Dance Book.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, 1982.

[11] *Assorted leaflets containing single Scottish Country Dances.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, various dates.

[12] E. Callander-Sharp. *Formation Index.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, 1978, revised 1981, 1983, 1988, 1992.

[13] J. Milligan. *Won't you join the dance?.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, 1976.

[14] *The Manual of Scottish Country Dancing.* The Royal Scottish Country Dance Society, 12 Coates Crescent, Edinburgh EH3 7AF, 1992.

[15] *Dunedin Dances.* Dunedin Dancers, Edinburgh. Reprinted 1993.

[16] R. N. Goss. *Computer Plotting of Country Dance Figures.* MSc Thesis, St. Andrews, July, 1984.

[17] S. Chatty. *Defining the dynamic behaviour of animated interfaces.* Engineering for Human-Computer Interaction 95–109, Elsevier Science Publishers B.V.(North Holland), 1992.

[18] *Borland C++ Version 4.0 Programmer's Guide.* Borland International, Inc, 100 Borland Way, Scotts Valley, California, 1993.

[19] *Borland ObjectWindows for C++ 2.0—Programmer's Guide.* Borland International, Inc, 100 Borland Way, Scotts Valley, California, 1993.

[20] *Borland ObjectWindows for C++ 2.0—Reference Guide.* Borland International, Inc, 100 Borland Way, Scotts Valley, California, 1993.

[21] C. Petzold. *Programming Windows, edition 2.* Microsoft Press, Washington, 1990.

[22] A. Schulman, D. Maxey, and M. Pietrek. *Undocumented Windows.* Addison-Wesley, June, 1992.

[23] E. MacKenzie. *Computer Graphics.* Lecture Notes, Dept. of Computer Science, University of Edinburgh, October, 1993.

[24] J. Foley, A. van Dam, S. Feiner and J. Hughes. *Computer Graphics: Principles and Practice.* Addison Wesley, 1990, reprinted with corrections, November, 1991.

[25] J. McGregor and A. Watt. *The Art of Graphics for the IBM PC.* Addison Wesley, 1986.

[26] L. Atkinson and M. Atkinson. *Using C.* Que Corporation, 11711 N. College Avenue, Carmel, IN 46032, 1990.

[27] P. Wegner. *Learning the Language.* Article in Byte Magazine volume 14, no 3, March 1989.

[28] A. Aho, R. Sethi and J. Ullman. *Compilers—Principles, Techniques and Tools.* Addison-Wesley, 1986.