

## **Livecode: Missing the Point**

Richmond Mathewson

I have recently been going through the final projects of children (9-13) who have completed an 8 week course entitled “An Introduction to Livecode Programming”.

An extremely awkward problem arose because of a misunderstanding about an important feature of Livecode.

I started learning computer programming languages at the age of 13 (1975), and until 1993, when I went to the United States and encountered HyperCard they were largely the same insofar as a program consisted on one long list of code (often with line numbers). It is only with hindsight that I value the concept of subroutines which could be written as addenda after the main code.

A week ago (September 2016) I was examining a pupil's 'game' and trying to fathom out where the code went wrong. I was unable to find where the code went wrong, while being perfectly capable of criticising where the game's functionality did not work. I realised the reason I was unable to work out what was not working was because the pupil had written his code as a monolithic lump exactly the way I was taught to write code in MINIFORTAN in 1975, and that was completely missing one of the greatest strengths of Livecode, the possibility to modularise code.

The ability to modularise code means that isolating problems is considerably easier than when one has a single, long line of code.

This pupil had 2 objects on-screen that were animated by 2 end-users via the computer keyboard. There were various 'collision' routines in case the two objects overlapped.

However, on pressing a key that should have animated one of the objects, both the objects animated incorrectly. The pupil then attempted to modify his code, and the result was that neither animated. The reason he was unable to modify his code successfully was that his long line of code meant that his 2 animation routines (that should have been implemented as quite distinct routines) were mixed up with each other and he could not work out “what referred to what”.

I told the pupil that the best way to sort out the non-functioning of his program was to throw all his code away completely and sit down at a large table and play with objects on that table. Understandably he was not happy about my suggestion.

We sat down at the table and, using a board marker, marked out a section of the table to represent his Livecode window. We then selected various chess pieces to represent the objects he wished to animate. I asked him to tell me what the chess pieces were meant to do on the table.

The point of this exercise was to get away from the boy's code, which was “blinding” him, and to re-examine the desired functionality.

One of the great strengths of Livecode is that it has a WYSIWYG development interface. Therefore, having determined a program's desired functionality with toys on a table one is able to assemble a virtual “table with toys” on-screen directly.

Having made our table-top prototype I presented the pupil with a set of small (A6) pieces of paper: one for each object, and a larger piece of paper (A4) for the Livecode windows (the 'card'). He was invited to write pseudo-code on each piece of paper to represent the functionality of that object.

As each of the objects to be animated consisted of a virtual display that, as well as animating, displayed a series of images (like an animated cartoon), I suggested that he tear each piece of paper for each object in half: one for the animation effected by an end-user hitting keys, and one for the cartoon animation displayed in the object.

We then made a virtual “table with toys” on a computer (making sure the pupil has no access whatsoever to his original offering) that was an exact copy of that that we had assembled on the physical table, even to the extent of using images of chess pieces.

Then the pupil had to convert his pseudo-code for animating each object into Livecode and write that into the individual code spaces of those objects. We then tested the functionality of that code; finding that, as there was no “linking code” (card script) nothing happened. We then wrote linking code in the code space of the Livecode window (card script) to call the code in the objects, and functionality was achieved.

We then replaced the images of chess pieces by display objects (graphic objects).

We then imported the image sequences we required for our animated cartoon effects into the Livecode document and named them sequentially. A button to hold each animation sequence code was created. Code was written in each button's code space to send an image sequence to each display object. Linking code was inserted into the code space of the Livecode window (card script) so that when the display objects were animated via key commands the cartoon animations would activate.

Then we compared the functionality of the finished work with that of the pupil's original work.

The essential difference was that by modularising code it became significantly easier to isolate trouble spots. While in Livecode modularisation is carried to an extreme degree this is effectively the same thing as setting up a series of subroutines. That these subroutines can be modularised (i.e. hived off into separate objects) in Livecode makes things far, far easier than other programming languages that do not have this feature.

Livecode's greatest strengths are its WYSIWYG interface and the way code can be modularised within it. The latter point is almost entirely overlooked when it is advertised and discussed.