

# The Design of Java Applets for Vibration Teaching on the WWW

N W Scott and B J Stone

Department of Mechanical and Materials Engineering, The University of Western Australia.

## Abstract

A search of the WWW reveals very little available material for teaching vibration that includes animations of motion. In the past such animations have been developed for other delivery platforms and have proved to be very useful in allowing students to gain a good understanding of vibration. This paper describes the results of a project aimed at writing Java applets that should have a wide range of applications. The design objectives are first discussed and it is concluded that such applets should encompass what has been found to be useful in previous non-WWW platforms. Thus animation and the ability to vary parameters are a prime consideration. However the ability to animate some shapes on a computer screen can be abused; a static diagram is open to misinterpretation and a moving diagram even more so. Each new animation has to be thoroughly tested and revised to ensure that students learn what was intended. The latter part of the paper gives helpful hints on writing Java applets which include animations.

## 1. Introduction

In a companion paper [1 - in these proceedings] a description is given of the current state of teaching vibration via the WWW. A comprehensive set of WWW notes, animations and quizzes is described in that paper. It was stated that,

*With so much to be gained from illustrating vibration by means of animations it is surprising that apparently there are very few examples on the WWW. This may be the result of a lack of Java skills.*

This paper describes some Java applets that present animations of vibration and allow parameter variation. The basis for the code is described. In order to appreciate the code the final application will also be described so that the product of the code may be seen.

It is important to be very clear on what is required before starting to write code. The experience from the use of some Hypercard material [2] written in 1988 helped set targets since it was an aim of the current work to overcome the known deficiencies in the earlier programs. Also the early work had of necessity been in black and white and colour was considered to be useful and attractive. A typical program that has been used for over 10 years is shown in Figure 1. This shows an animation with parameter variation. As with all the examples given it is hard to appreciate an animation in a static medium, which is of course why animations can be very helpful.

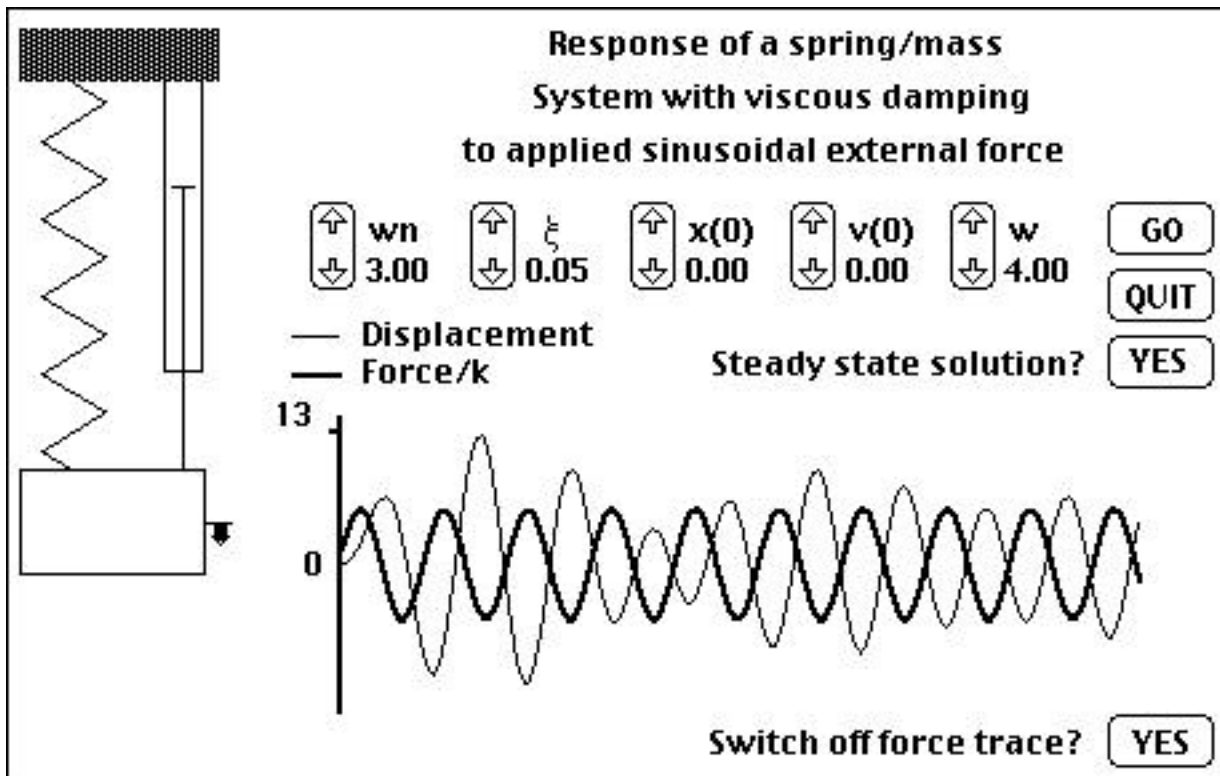


Figure 1 An early animation example.

The students who have used the 1988 material did so as either a supplement to conventional lectures or as a means to bridge the gap for an advanced vibration course. In addition the animations were used with good effect in lectures. The main criticism of the animation was that some students did not see the connection between the time trace and the motion of the spring/mass system; in other words they did not realise that the sinusoidal trace was being generated by the mass. As a result it was decided that the 1999 version should use a "pen" sticking out of the mass to draw on a moving trace, like the pen on a chart recorder. Further, by using the same *colour* for the mass and trace, it was anticipated that it would be clear that the trace was a record of the motion was that of the mass. To help with reinforcement it was all masses in *allprograms* have the same colour. The excitation force (where appropriate) was also of a standard colour and the force trace of the same colour.

A further drawback to the program shown in Figure 1 was that the time trace was of limited length. If a student wished to see how long it took for a steady state to be achieved then this was not possible. For the trace shown the steady state has still not been achieved by the end of the trace. If the trace is programmed to cover a longer period of time then the waves become too close together and clarity is lost. The requirement was for a longer time trace, but how to accommodate this in the width of a typical monitor?

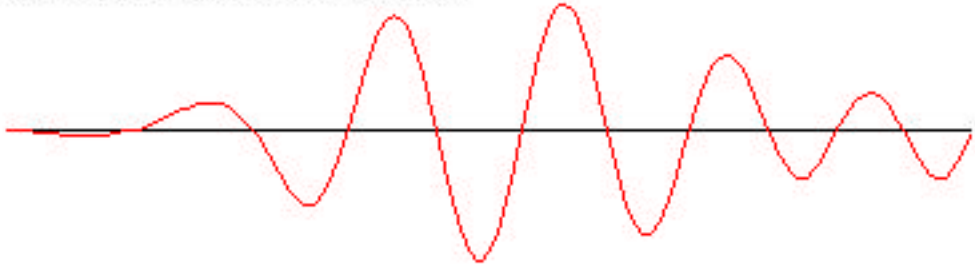
The parameter variation buttons in the 1988 version had been very useful. They are very easy to use and allow the associated number to be increased or decreased without typing. In moving this interface feature to the 1999 version we decided not to use the Java class Scrollbar because we found it to be unreliable on some computer platforms, so we wrote our own replacement for it which has given us a lot more control over how it behaves.

## 2. Example

The example chosen for this paper is that of out-of-balance (oob) excitation of a single degree of freedom (DOF) system. To set the context the WWW page that introduces the animation program is shown in Figure 2.

**Starting up**

The motion that occurs depends on the steady rotational speed ( $\omega$ ) and the rate of angular acceleration ( $\alpha$ ) from rest to the steady rotational speed. The initial part of a typical response is shown below.



It is best to investigate the effects of different start up accelerations using the [animation program](#).

It should be noted that the total mass ( $m' + m$ ), i.e the o-o-b mass ( $m'$ ) plus the rigid mass ( $m$ ), is used in the calculation of the [undamped natural frequency](#) and the [damping ratio](#).

Most texts and teachers do not cover the gradual start up illustrated here. It is common to assume that the rotating o-o-b mass instantly achieves [constant running speed](#). However, in practice, it is often the case that the steady running speed is above the resonant frequency and the rate of acceleration through resonance is crucial.

**NOTE:**  
When you use the animation program you may choose an excitation which results in the displacement being excessive. The viscous damper will then reach the end of its stroke and be restricted.

Figure 2. WWW page introducing the animation program

It needs to be stressed that there is a considerable amount of material presented before this page. The [links](#) to the definition of terms are to material that has already been covered. The page stresses the importance of "start up" and notes that this is usually omitted from texts. If the [animation program](#) link is selected then the Java Applet runs. A typical run of this program is shown in Figure 3 and the program has been paused part way through the run.

The mass and its trace have been chosen to be red (in other programs force is blue and abutment motion the same colour as the abutment shown). The large trace is stationary and the mass at rest when the applet is first loaded. When the run button is selected the large trace (chart recorder paper) starts to move to the left and the pen leaves a red trace. The start of the rotation of the mass is delayed so that a reference red line is drawn. Then the oob

mass starts to rotate as a result of its angular acceleration. This acceleration is effective until the running speed ( ) is achieved thereafter the speed remains constant.

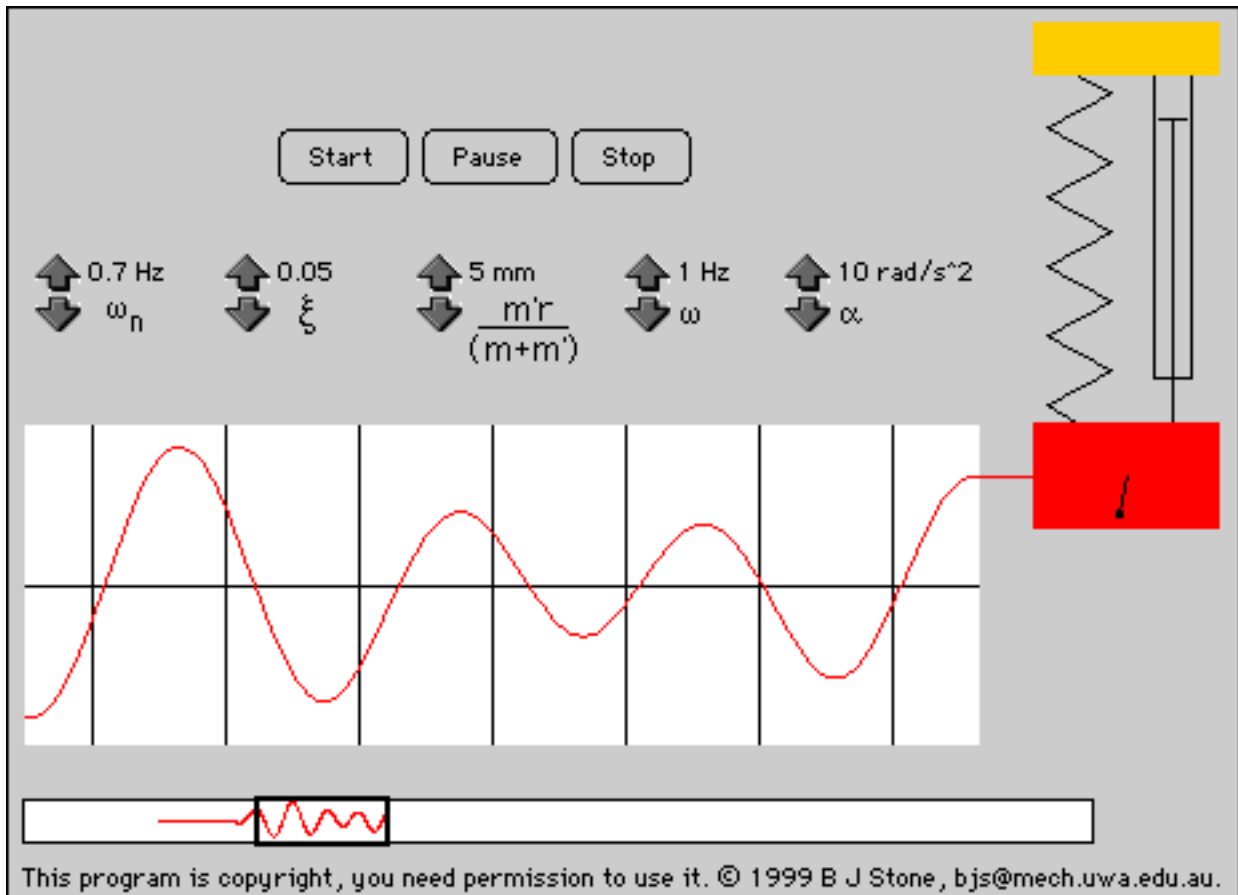


Figure 3. Typical output from out of balance Java applet (Truncated to fit on page and retain clarity)

To allow for a long time trace there is a thumbnail version of the trace along the bottom of the applet screen and a cursor (the heavy black box on the thumbnail trace) moves to the right as the main trace moves to the left. When paused or stopped it is possible to view an earlier portion of the trace by dragging the cursor to the left with the mouse. The same effect can also be achieved by dragging the large trace to the right. However to appreciate these features it is best to try the program, which is at,

<http://www.mech.uwa.edu.au/bjs/Vibration/OneDOF/OOB/Starting/OOB.html>

The text from which the applet is called may be found at.

<http://www.mech.uwa.edu.au/bjs/Vibration/OneDOF/OOB/Starting/>

The pages are freely accessible. The copyright is to ensure that acknowledgment is given. Permission to use the applets will be given on request.

The parameter buttons have been found to conserve space and may be accurately positioned. The values that may be selected are between set limits. In addition the applet runs in real time and the frequency and acceleration values that are chosen are those that will be seen.

To aid users, information boxes appear when the mouse is positioned over any part of the output. Two examples are shown in Figure 4. Note these information boxes would not be visible at the same time.

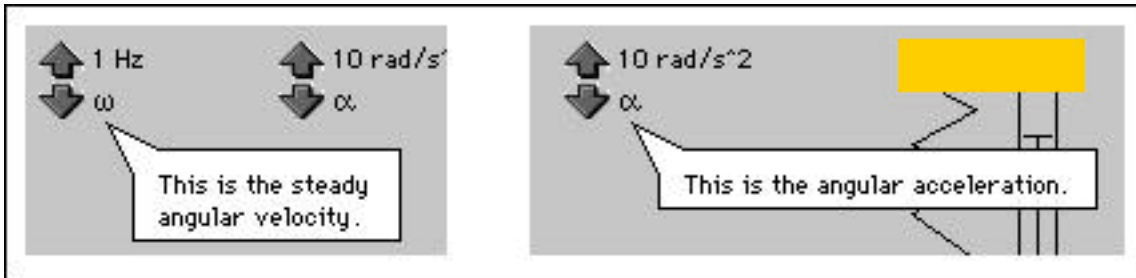


Figure 4. Information boxes visible when the mouse cursor is placed over a any item.

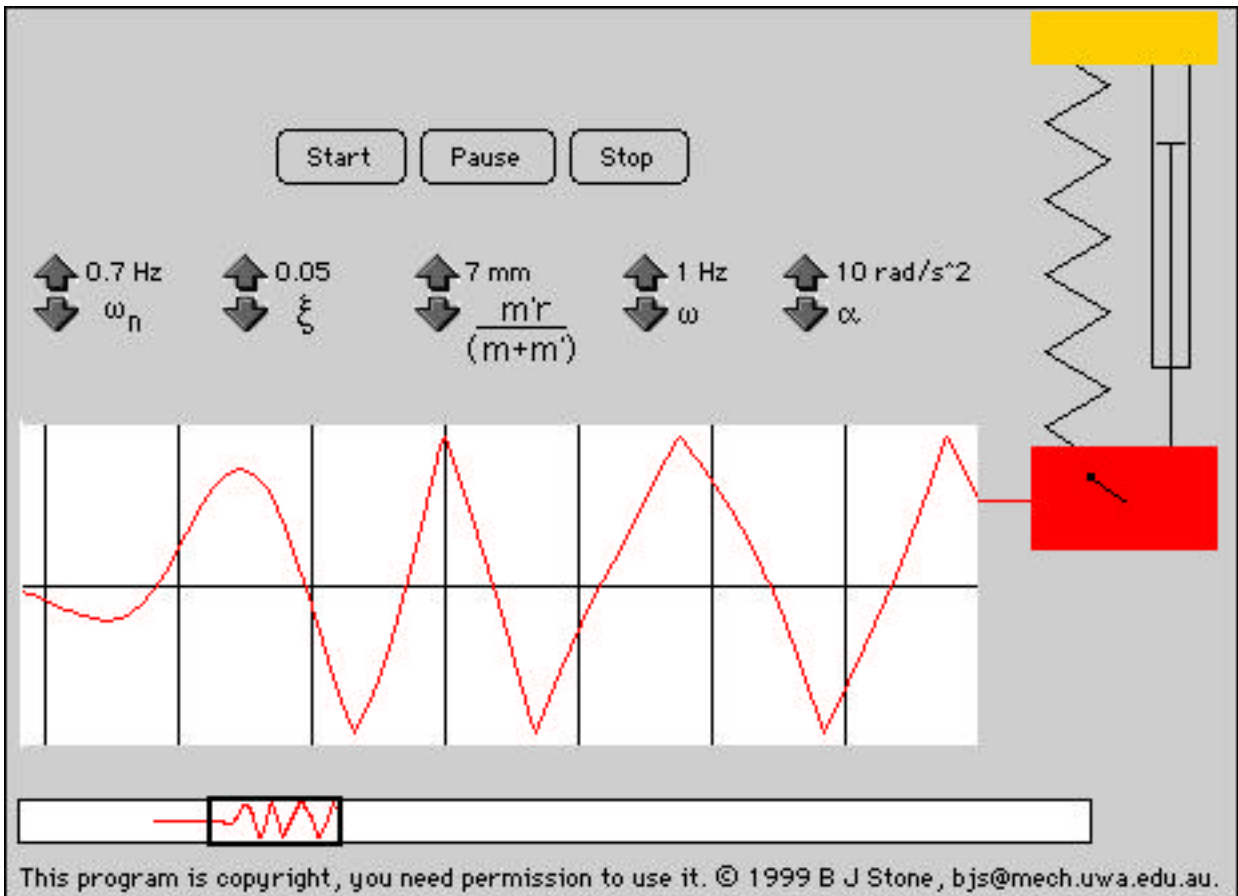


Figure 5. Illustration of non-linear effects due to impacts at end of damper.

The topic of non-linear vibration is not often covered even in advanced texts. As the applet has real frequencies and specific values for the other parameters it is possible to select a set of values that cause the damper to reach the limits of its travel. The code has been written so that when the mass "hits the stops" there is a large change in momentum – hence the sharp change in direction at the upper and lower ends of travel in Figure 5. Note that only one parameter has been changed between Figures 4 and 5.

Similar programs have been written for transient motion, forced motion involving an external force and by abutment excitation. All the programs use a common shell, which is described below.

#### 4. Writing code for easy re-use

It has been said that everyone has at least one good book in them. So it may be that most academics in the physical sciences have at least one good animation they would love to write. The problem is that most academics do not know Java and cannot afford to pay someone else to do it for them. Actually the problem is even worse than this, because experience has shown that "pure" programmers often do not understand the academic matter being animated and consequently tend to produce software that can lead students astray. It is like asking a non-Mandarin speaker to write street signs in Mandarin from a handwritten sketch: any stroke out of place could make the sign incomprehensible to an experienced reader, let alone a novice.

So, ideally, we would like academics with a vision to be able to express it themselves. In this case Scott was somewhat acquainted with the vibration material although not in the same depth as Stone, and Stone was somewhat aware of what Java could do, but not as well as Scott. So Scott wrote a "shell" for Stone – an animation program that had buttons to change the numeric parameters of a fairly simple dynamic system. The design was expressed as a series of related Java objects in an event-driven model. Stone subsequently used the shell and inserted the "vibration" code for several applications.

##### 4.1 *Java objects*

It is easy for learners in this area – i.e. those wanting to learn how to write Java – to get discouraged by the jargon surrounding the subject. The modern programming concept of *objects* is a good example. Many academics come from a background with a traditional non-object-oriented programming language such as Fortran, Pascal or C. The key to understanding objects is to see them as blocks of structured data (called records in Pascal and structs in C). In fact an object can be used purely as a way of storing data. However objects also have two other useful properties: methods and inheritance.

An *object method* is a procedure or function (a block of code) which is only defined in the context of a specific object type. Consider the following object, called a CRectangle:

```

class CRectangle
{
    // define some variables for the object to store and use int left, top, width, height;
    // data is stored in these object variables

    // now define a method of the CRectangle object
    public void OffsetRect(int dx, int dy)
    {
        left = left + dx;
        top = top + dy;
        // note that we can refer to the object variables if we want to. It is understood\
        // that these are the variables of THIS object and not some other object.
    }
}

```

The advantage of defining a CRectangle object in this way is that it becomes a neat package containing the important data (left, top, width, height) as well as utilities for manipulating the data. This neatness, if exploited intelligently in the design of a program, can make it much easier to keep track of a complex set of data structures. This has advantages in both maintenance and re-use of code.

If objects only provided data and methods they would probably still be worthwhile. However they also provide a wonderful behaviour called *inheritance*. It is often the case that we wish to have several objects that are quite similar in their data elements or function, but are also different in important ways. We could invent fresh objects by copying the code of old ones, but this could be a problem if the older code needs to be revised. Inheritance allows us to define a series of objects that *build* data elements and functions on top of simpler objects. Consider the following class, written to draw a rectangle with a word printed inside it:

```

class CWordRectangle extends CRectangle
{
    // because this class extends CRectangle, we still have access to the data
    // left, top, width and height. We do not need to re-define these things, and in fact
    // it would be a disaster if we tried to do so. However we can add new data "slots":
    int wordX, wordY; // position of word
    String word; // the word we will draw

    // if we used the method OffsetRect as it was in the older CRectangle object, the new
    // data elements wordX and wordY would not be altered correctly. So we override the
    // older method and add some new functionality to it
    public void OffsetRect(int dx, int dy)
    {
        super.OffsetRect(dx, dy); // execute the code of the older method

        // now do some new things
    }
}

```

```

    wordX += dx;
    wordY += dy;
}
// the real object would have to define additional methods to actually draw the word
// on request
}

```

The terminology associated with inheritance is confusing. As new objects are defined, they *extend* the function of the simpler objects and are therefore larger and more complex. Confusingly a derived object is often referred to as a *child* or *subclass* of the simpler one. The simpler object is also sometimes called the *parent* or *ancestor* of the derived object.

Java defines a huge set of useful object types which can be used in programming, so it is not always necessary to invent new objects "from scratch". The defined object types can also be used as the basis for more sophisticated ones. In particular, Java defines a standard way of dividing up the visual area, based on classes derived from the predefined class `Frame`. However the built-in `Frame` class has some serious limitations and we decided to invent our own replacement for it.

The animation shell defines a class called a `CFrame` which extends a fundamental Java class called a `Rectangle` (not the same thing as the `CRectangle` example above, but similar). The `CFrame` object represents, in an abstract way, anything that can appear on the visual area. See Table 1.

	Java class <code>Frame</code>	Our class <code>CFrame</code>
Positioning	Uses complicated objects called Layout Managers. Results unpredictable and often inappropriate.	Allows precise (x, y) positioning of all screen elements
Overlap	Each part of the visual area is owned by an unambiguous list of nested <code>Frame</code> objects i.e. of <code>Frames</code> that contain <code>Frames</code> . Peer <code>Frames</code> do not overlap.	<code>CFrame</code> objects are stored in a flat list, can overlap and have the option of masking or revealing lower <code>CFrames</code> .
Events	<code>Frame</code> objects have a separate method to respond to each kind of mouse event, and they do not define a standard way to respond to other events.	<code>CFrame</code> objects can respond to events through a single <code>MouseEvent</code> method. Other data can be sent to a <code>CFrame</code> through its <code>ControlMessage</code> method.

Table 1 Features of the predefined `Frame` class compared to our version.

This is not to say that the predefined `Frame` class is *bad*, simply that it was not close enough to what we wanted. Java provides both convenience and flexibility.



We also invented a class called a CFramePanel, which is a single Frame descendant that looks after all the CFrames; it is the intermediary between the predefined visual heirarchy and our own version. The CFramePanel object type also maintains an off-screen bitmap so that the screen refreshes smoothly. The 'smooth redraw' code is thus hidden away where the designer of the animation does not need to see it.

#### 4.2 Event model

Like objects, the idea of an event-driven program can intimidate the new learner of Java. The problem is that Fortran, Pascal and C all have an explicit "main program". The code in this program is executed one line at a time, from top to bottom. If subroutines are called these are executed in a precisely known sequence. The significant difficulty Stone faced in starting to use Scott's shell was to see how the *program flow* worked. He was frustrated in his understanding because he could see bits of code that made sense, embedded in the various objects, but it was not clear when each bit of code would run. Did they all run at once? From top to bottom, perhaps? It seemed illogical.

In an event-driven program it is often the case that the main program is extremely small, and can consist of just a few lines of code (expressed here in words):

```
Do the following until the program quits:
{
  Receive an event from the keyboard or mouse;
  Process the event.
}
```

To confuse matters further, in Java the main program is hidden away in a compiled object (like a library) – it cannot be seen at all. But it is still there.

The function of a particular event-driven program is expressed in the way the program responds to each event. Examples of events include

- A key on the keyboard has been pressed down;
- The mouse button has been pressed down;
- The mouse has moved and the new coordinates are (x, y)

It will be seen that the program cannot anticipate what the user will do. The user might click the keys on the keyboard in any order, or might use the mouse. So the program no longer has a set order in which the procedures will execute. All it can do is to wait patiently for an event and then to do something in response to the event. For example, if the user clicks a button on the screen, the program might change the state of some variables and then run a subroutine to process some information.

In the case of the animation shell it was desirable to invent a refinement of the basic Java event model. We wanted the animated object(s) to respond to events, which means that we

must inform the objects that the events have occurred, and we also wanted to define new event types. For example we wanted an event type that meant "x seconds have elapsed in real time", and another event that meant "increase this parameter by x". Class CFrame defines a simple message-passing event system. Every CFrame descendant overrides

```
public void ControlMessage(int code, double val)
```

In other words, at any time we can tell a given CFrame object to change something or do something by calling, for example,

```
myCFrameInstance.ControlMessage(23, 0.2727);
```

The animation shell is actually a completely generic refinement of the basic Java environment. To make a specific animation, some descendants of the CFrame object must be defined and positioned on the screen. There is no clear distinction between a CFrame descendant which does animation, and one which is merely a control or interface feature. All the CFrame objects have a presence on the screen, and all of them can send and receive control messages. The shell keeps all the CFrame descendants informed about events such as mouse clicks. The vibration example animation can now be explained in terms of the CFrame descendants that make it up. See Figure 6.

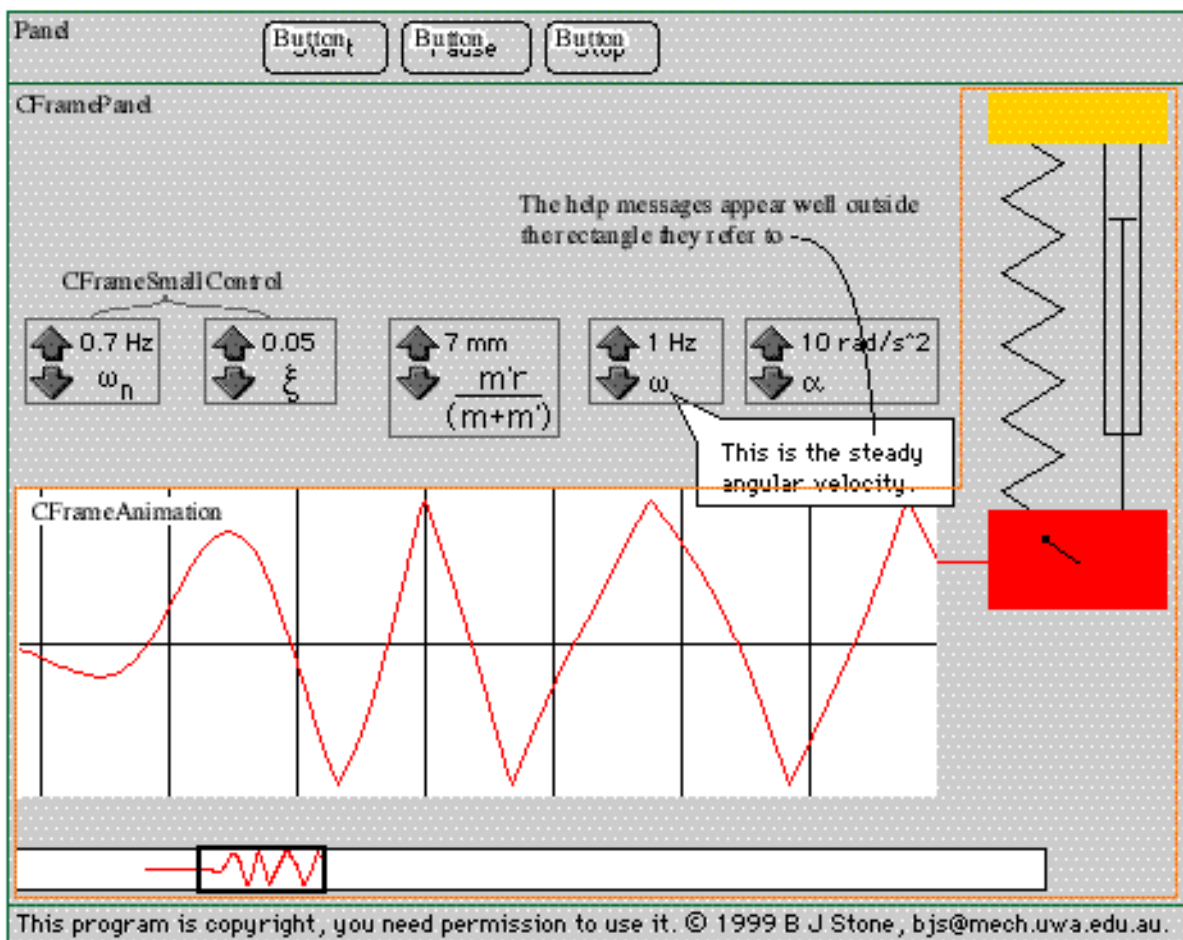


Figure 6 Each part of the screen has an associated object, and each object is of a certain type

The workhorse of the applet is the CFrameAnimation object. In a sense this object is not an “animation” at all, since at any given instant it displays a static image. It is an animation because it responds to a regular control message from the main program: the elapsed time message. If this message is received, the CFrameAnimation object recalculates the position of the spring-mass-damper system at the new time, and then redraws the screen. The CFrameSmallControl objects are configured to send other control messages to the CFrameAnimation object. For example, if a mouse click event happens in the CFrameSmallControl that governs  $\omega_n$ , a message containing the new value is sent to the CFrameAnimation object, which takes note of the new value and stores it internally. The screen appearance of the system might not change immediately but the new parameter value may produce a change in the behaviour of the system over time. The message paths can be seen in Figure 7.

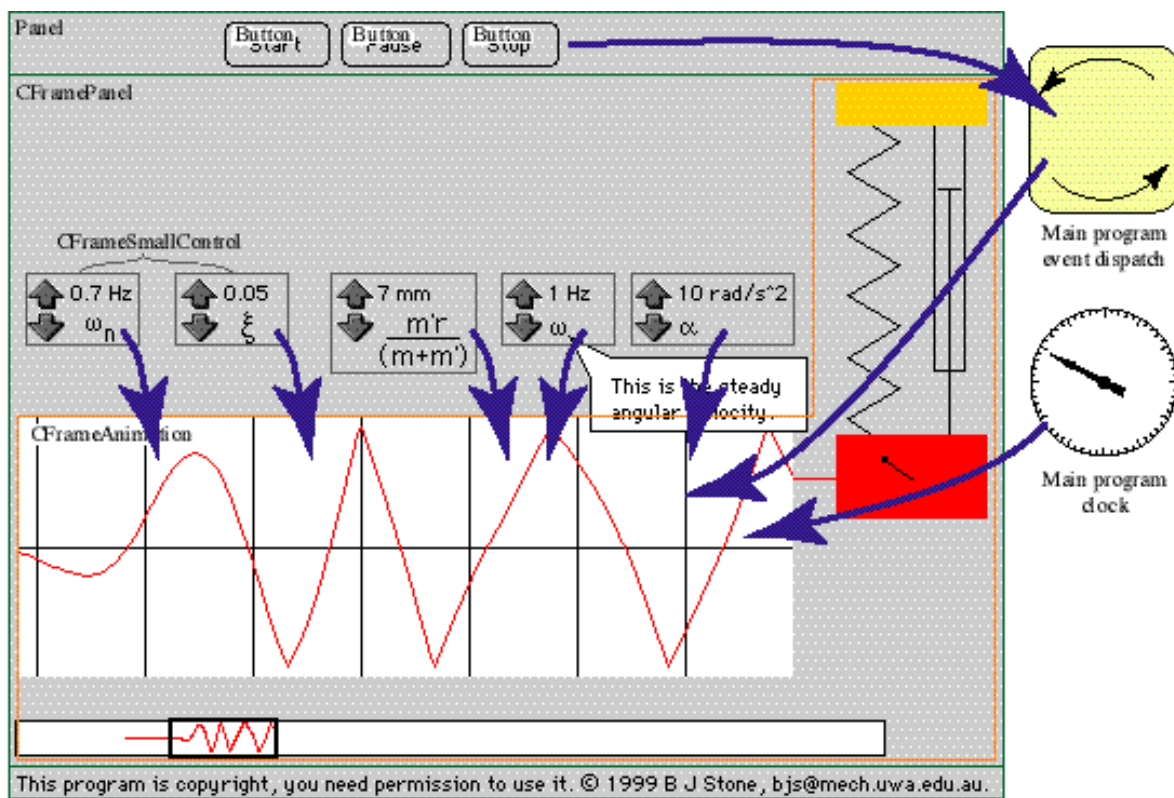


Figure 7 Flow of control information within the program

## 5. Recycling code and adding new objects

The original animation shell written by Scott did not look much like Figure 5. It had most of the graphical elements but not the detail of the spring-mass damper system or the many controllable parameters. The shell has proven to be a success because it is clear that Stone can add CFrame descendants easily, can adjust the positions and sizes of the on-screen elements to achieve a desired effect, and can concentrate on the engineering and educational aspects of the design.

We considered putting the whole source code for the example in these proceedings, but decided not to do so because it is quite long and, although it is heavily commented, is quite hard to follow in printed form. If you are interested in getting a copy of the source files in order to make your own animation, contact Dr Scott by email.

## **Bibliography**

Paper 1 below contains an extended list of further references.

1. Scott, N. W., Hirannah, S., Mannan, M. A. and Stone B.J., 'Teaching One degree-of-freedom vibration on the WWW', ASEE 2000 Annual Conference session 3220.
2. Li, X., and Stone, B. J., 'The Teaching of Vibration by means of Self-teach Computer Programs and Laboratories' Experimental and Theoretical Mechanics Conference, ETM93 Bandung, Indonesia, 1993, 426-431.

### **N.W. SCOTT**

Dr Nathan Scott is a Lecturer in the Department of Mechanical & Materials Engineering, The University of Western Australia. He spends a great deal of time in front of a computer, developing and maintaining computer-based resources such as animations and tutorial systems. He is always willing to correspond by email: [nscott@mech.uwa.edu.au](mailto:nscott@mech.uwa.edu.au)

### **B.J. STONE**

Professor Brian Stone has held the Chair in Mechanical Engineering at the University of Western Australia since 1981. He has been writing teaching software since 1987. In 1997 he was named the best Engineering teacher in Australia by a federal committee. His research interests include vibration suppression and computer simulation of dynamic systems.