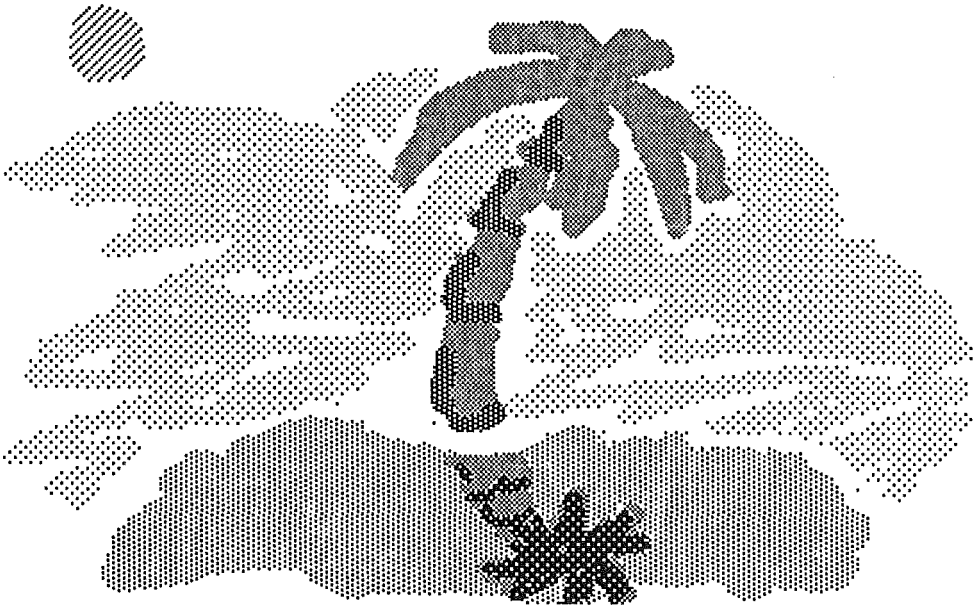
OASIS:

An Object Oriented Animation System In Scheme



By Peter Munnings

Contents

Section 1

Background

1.1 Object-oriented programming

Object oriented languages provide a particularly good environment for writing modular programs. An object is an independent module of a program, just as a procedure is an independent part of a Pascal program. Objects are much more flexible than procedures, however, and this added flexibility can result in a radically altered approach to programming.

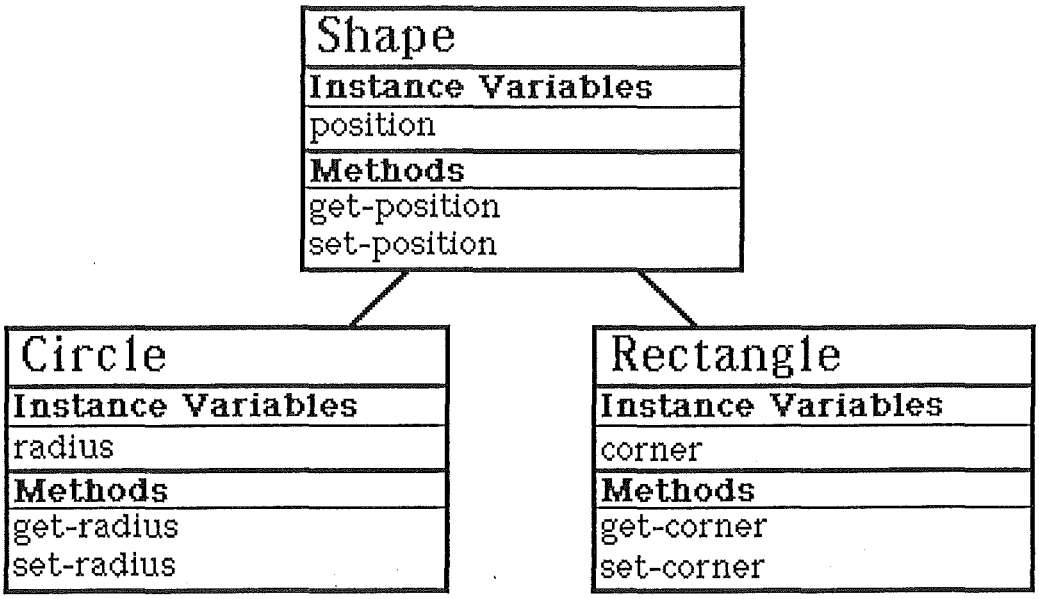
Unlike a procedure, an object can perform several possible actions. These actions are defined in an object's *methods*. Each method of an object is essentially a Pascal-style function in its own right. It can contain its own local variables. It can invoke other methods, either inside the same object, or belonging to other objects. It can return a result to its caller. Methods are invoked by *message passing*. Most object oriented programming languages provide a pseudo object called *self* to allow messages to be passed to other methods contained in the same object as the calling method.

An object can also maintain information about its state between invocations of its methods. This state information is contained in the object's *instance variables*. An instance variable acts like a global variable that is only visible to the object's own methods. This means that external access to instance variables can only be achieved by invoking methods - an object has complete control over its own integrity.Several different *instances* of an object can be created - each with the same methods and instance variables, but independent of each other, because the values of instance variables of two instances of the same object may be distinct.

The 'object' abstraction means that, when a model of a system is being implemented as a program, it can be done in a very natural way. Most systems are modelled on paper as a set of entities with some relationships defined between them. The entities of the system may have some actions associated with them, and may also have some internal information about their current state. There may be several equivalent entities with the same basic properties, but independent of each other. The entities in the model may operate by passing messages to each other about what to do next.

Each entity in the system could be implemented as an object, and any actions associated with the system as a whole could be performed by message passing between objects. Each object can be defined without having to consider side effects caused by other objects - the only visible properties of other objects are defined by their methods. It is also easy to have several independent instances of an object, corresponding to equivalent entities in the original model. The resulting close correspondence between the model of the system on paper and the object oriented implementation not only makes it easier to implement the model as a program, but can make it more elegant and easier to debug.

In order to make the specification of objects easier, many object-oriented programming languages also provide for the *inheritance* of the definitions of instance variables and methods from other *classes* of object. Usually this is implemented by maintaining a 'tree' of object classes, in which child classes inherit the instance variables and methods defined in their parent class.



Consider the above example of a class hierarchy. Objects of class *Shape* have the attribute *position,* which represents their position in some coordinate space. The method *get-position* might return the current position of the shape; *set-position* might set it. A *Circle* would have all of the properties of a *Shape*, but in addition it might have an attribute representing its radius. If the inheritance of methods and instance variables was not supported, it would be necessary to duplicate the definitions of the position attribute of *Shape* and its associated methods. In this system, however, the instance variable and method definitions from the class *Shape* are automatically inherited by the class *Circle.* Similarly, these definitions are inherited by the class *Rectangle.*

1.2 The flavour system

The object oriented language used in this project is a *flavour* system, lying on top of the Scheme programming language. A flavour system supports the most important features of a good object oriented programming language. Object flavours are formed into a class hierarchy, with automatic inheritance as described earlier. At the top of the hierarchy is the flavour *vanilla*. This particular system was originally written by Greg Ewing, and was ported to MacScheme on the Macintosh Plus by me in late March.

In order to get a feel for how this particular system works, an example of the system in operation is provided here.

```
(defflavour shape vanilla ((position (list 0 0))))
```

The *defflavour* function is used to define new flavours of object. In this case, the flavour *shape* has been defined as a component flavour of vanilla, with a single instance variable representing its position.

```
(defmethod shape set-position (x y)
 (set! position (list x y))
 (self 'get-position))

(defmethod shape get-position ()
 position)
```

The *defmethod* function defines a method for a particular flavour of object. Here methods have been defined to set the position of a shape, and to return its current position. Note that a message has been sent to *self* in order to invoke the *get-position* method from within the *set-position* method.

```
(defflavour circle shape ((radius 42)))

(defmethod circle set-radius (x)
 (set! radius x))

(defmethod circle get-radius ()
 radius)
```

Now a circle has been defined as a kind of shape with an added attribute for its radius, and some methods to set and fetch the current radius. Objects of flavour *circle* will also automatically inherit the attribute *position* and the methods *get-position* and *set-position* from the flavour *shape*.

```
(define fred (circle 'new))
```

Here, fred has been defined as an instance of *circle*.

```
(fred 'get-position)
--> (0 0)

(fred 'set-position 42 42)
-->(42 42)

(fred 'set-radius 23)
-->23

(fred 'get-radius)
-->23
```

Here, several messages have been sent to the object *fred*, and appropriate responses have been returned. Note that the radius of the circle was maintained between the sending of the set-radius message and the get-radius message. The flavour system used in this project is not only quite powerful, but also extremely easy to use. This was a primary motivation in using it to implement the animation system that is the basis for my project. Object oriented programming in standard Scheme (without a flavour system) is described in *Structure and Interpretation of Computer Programs* [2]. The best way to get started in Scheme programming on the Macintosh is probably to experiment with MacScheme using *MacScheme* [4] (the reference manual) as a guide to the language.

Section 2

Aims and Objectives

" *An object-oriented animation system*
This project will use the SCHEME (a Lisp dialect) programming system to design and build a single animation package for some micro-world (i.e. geometric objects). It will first require a port of locally developed class and flavour packages from ChezScheme (on the Vax) to MacScheme (on a Macintosh) " - original project specification

The main aim of this project is to develop an animation system for simple geometric objects. Of course this is not a very clear definition of what is required. While working on this project, the objectives to be reached have gradually evolved, and have become much more specific.

The animation system resulting from this project should allow someone using it to build arbitrarily complex animation sequences using a series of simple commands. The key to achieving this is to ensure that the user need not plan an entire animation sequence in advance. It should be possible to divide the implementation of a sequence into parts, each of which can be implemented with relative ease. The objects in the system should be persistent - an object should be an independent entity which responds to messages passed to it by performing animation sequences. Ideally, a geometric figure in the workspace should always correspond to the same object in the animation system. Such a close correspondence would increase the consistency of the interface between the animation system and the user. When a user builds a more complex figure out of a series of simple animation objects, it should be possible to define this as an object in its own right, and manipulate it accordingly. It should also be possible to build still more complex objects out of several complex ones, so that objects of arbitrary complexity can be constructed in a flexible manner. This increases the usefulness of the system, because any collection of objects that the user perceives as an object in its own right can be treated as such by the system.

Messages would normally be passed to the objects in the animation system through the Scheme transcript window. Effectively a user would be executing the methods of various objects in the system just as they would be executed from any Scheme program. This approach has the advantage that the same system acts both as an interactive program and a library of animation functions. Few users of the system would really need so much flexibility, however. A graphical user interface would be a very useful addition to the animation system, as it would make communication between the user and the system more flexible. The user could enter information by typing in characters, selecting a menu item, or indicating a screen position with the mouse. Menus of the available animation functions would be provided. A novice could then learn to use the system by browsing through the menus and trying various commands.

In the ordinary version of the system, the position of an object must be specified either in terms of cartesian coordinates, or by specifying its new position relative to the old. This makes the placement of objects a trial-and-error process, because it is very difficult to visualise a shape when it is expressed in cartesian coordinates. The menu system could allow a user to directly specify the position of objects in the animation workspace, using the Macintosh mouse. This would give him or her immediate visual verification that an object is correctly placed.

An important requirement not actually stated in the original specification of the project is that the system be as usable as possible. While *usability* is difficult to quantify, it seems clear that there are two basic principles adhered to (as much as possible) by this system which generally make a program easier to use. Firstly, it is important that the system should allow problems to be broken up into smaller and smaller pieces, which can be solved individually much more easily than the problem could be solved as a whole. Secondly, it is important that the user interface provided by the system should match as closely as possible what the user perceives as being the model's current state. The more closely the user and machine views of the animation system correspond, the less likely the user is to be confused by the way in which the model interprets a particular command.

Section 3

What has been achieved

3.1 The design of the system

When I began the design of OASIS, it was very difficult to define exactly what the system should provide. Without some basis for experimentation, it was not possible to decide precisely what constituted a good user environment for an animation system. OASIS was eventually designed by developing successively more sophisticated versions of the program until all of the original aims of the project were met.

3.1.1 Exploratory programming

The design of OASIS relied heavily on exploratory programming. The combination of the MacScheme programming environment and the flavour system meant that it was easy to change fundamental aspects of the animation system's design without rewriting large amounts of code. The encapsulation of methods and instance variables in objects made this even easier because the side effects caused by modifying an object were minimised. After each version of OASIS was implemented, it was tested and evaluated to find weaknesses in its design. These weaknesses were then corrected in the next version of the system. The final version (upon which the menu system was built) actually represents the fourth major revision of the animation system's design.

An exploratory approach to the design of OASIS was definitely worthwhile. The most important point in its favour is the distinct lack of similarity between the first and final versions of the animation system. Comparing the two, it is obvious that the first approach to the design of OASIS was vastly inferior to the last, but it is doubtful that this would have been evident without the valuable lessons learned from each successive version.

3.1.2 OASIS version 1

The first version of OASIS seemed to provide all of the basic features mentioned in the original specification of the project. This system was based around the concept of an *animated object*. In order to use the system, a new object would first be created. Several different types of object were possible; circles, squares, rectangles, ovals, and polylines (a collection of points joined by a continuous line) had been provided. Messages could then be sent to the new object to indicate its initial position, and any animated movements required of it. The object could then be passed the message *go*, at which point the animated sequence built up for it would be displayed on the screen. When several simple animated objects had been defined, they could be combined using a *composite* object, which allowed all of its component objects to move simultaneously.

The animation sequences themselves were stored in *animation lists* - lists of the drawing commands needed to produce each frame in the animated sequence. In order to make animation as smooth as possible, these drawing commands could be stored in a QuickDraw 'picture' data structure, so that the speed of animation was not limited by the speed of MacScheme. The simultaneous animation of multiple objects was achieved by combining their animation lists into one big list, and then drawing the resulting animation on the screen.

This version of OASIS allowed quite complex animations to be created using simple commands. The resulting animation was extremely smooth, especially when QuickDraw pictures were used to display the animation sequences. Unfortunately, the system used a very limiting definition of animated objects. Each object in the system was really an *animated shape*. This meant that it was meaningful to talk about a 'moving square' as an object, but not meaningful to consider the square itself as an object. Each object had by definition only one possible action associated with it. This greatly reduced the flexibility of the system. When a user sees a rectangle on the screen, he perceives it not as an 'animated rectangle' but as an object in its own right. It would seem natural that the rectangle could then *shrink* or *moveRight* depending on some message passed to it. In the case of this system, while a shrinking rectangle and one moving to the right might *appear* to be the same object, they would be different objects as far as the system was concerned. It seemed that it would be simple to modify OASIS to support multiple actions for each object.

3.1.3 OASIS version 2

The new version of OASIS had a fundamentally different view of animation objects. A simple object was now regarded as a geometric shape, which had several possible actions associated with it. A separate animation sequence was built up for each action. This made OASIS much easier to understand. The objects on the screen could now be regarded as permanent entities, responding to messages sent to them by performing various feats of animation. When using this system, however, it became obvious that the support for the movement of multiple objects simultaneously was not adequate.

When a user builds up a composite object from several simple objects (like a stick man built up of lines and circles), it is convenient if this more complex object can be treated in the same way as simple objects. The user now perceives this combination of simple objects as an object in its own right. It would then seem natural to be able to move this object around the screen as a whole, in the same way that a simple object could be moved around. Once this composite object had been moved, its components should still behave in an intuitive way. In the second version of OASIS, this was not the case. It was not possible to move composite objects at all. As a result, it was necessary to plan any actions performed by a compound object in advance, and implement them at the level of the object's individual components. This made the system extremely difficult to use.
It seemed that the problems of OASIS could be corrected by modifying the specifications of compound objects to allow their movement. The most important modification was to add some form of local coordinate system to the program. The positions of the components of a compound object should be measured relative to the parent object's current position, so that any change in a compound object's current position automatically adjusted the position at which its components would be displayed.

3.1.4 OASIS version 3

This system was never completed. When the changes suggested by the second version of OASIS were considered in more detail, a whole host of implementation problems arose. In the first three versions of the system, animation sequences were maintained internally as lists of the drawing commands for each frame of the animation sequence. The movement of compound objects was achieved by interleaving the animation lists of the component objects. These animation lists were very large, as they essentially duplicated all of the components' actions. Trying to implement a local coordinate system on top of this structure was very difficult. The individual drawing commands in an animation list were simply QuickDraw commands, and were based on a global coordinate system. This meant that a system in which components had their own local coordinates would have to be implemented implicitly by adjusting global coordinates when necessary. Moving a compound object then meant that all of the coordinates in the animation lists for each component object would have to be adjusted to ensure that the components' actions would occur in the right place. Even this problem is simple compared to the situation in which a component is supposed to be performing some action while its entire parent object is moving.

At this stage it was obviously necessary to consider carefully whether it was sensible to store animation sequences as frame-by-frame animation lists. It seemed that all of the problems of version 3 would be greatly simplified if animation sequences were stored as a series of animation primitives (like *move* or *put*) which would be expanded frame by frame when the sequence was displayed. Unfortunately, the animated pictures produced by this system would not be as smooth as in the earlier systems because the position of each object for each frame would be calculated 'on the fly' instead of being calculated in advance. However, this problem seemed unimportant when weighed against ease of implementation and the elegance of the resulting program. Eventually it was decided to redesign OASIS around the concept of animation primitives.

3.2 OASIS.

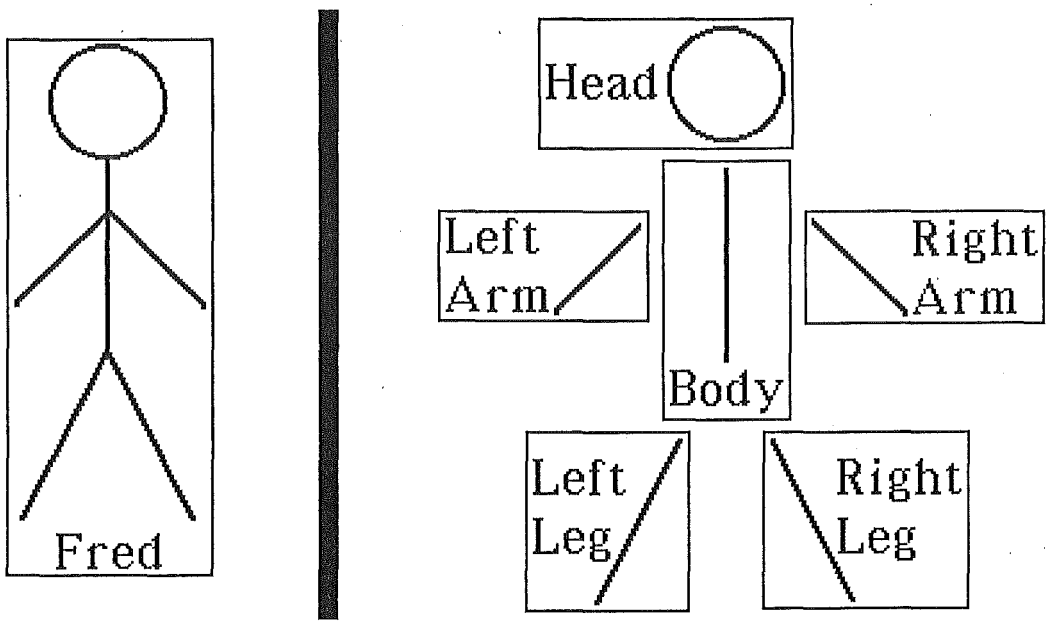
This system is sufficiently powerful for a user to manipulate arbitrarily complex objects in a manner consistent with the manipulation of simple objects. It provides all of the features intended for the third version of OASIS, but in a much more elegant fashion than would have been possible using frame-by-frame lists for animation sequences. The system is built on a few important concepts, which are:

1. An animation object hierarchy
2. Message passing
3. Animation primitives
4. Local coordinate systems
5. A flavour hierarchy

See *Computer Graphics* [1] for a discussion of hierarchies of graphical objects and local coordinate systems in Chapter 15, MODELING METHODS.

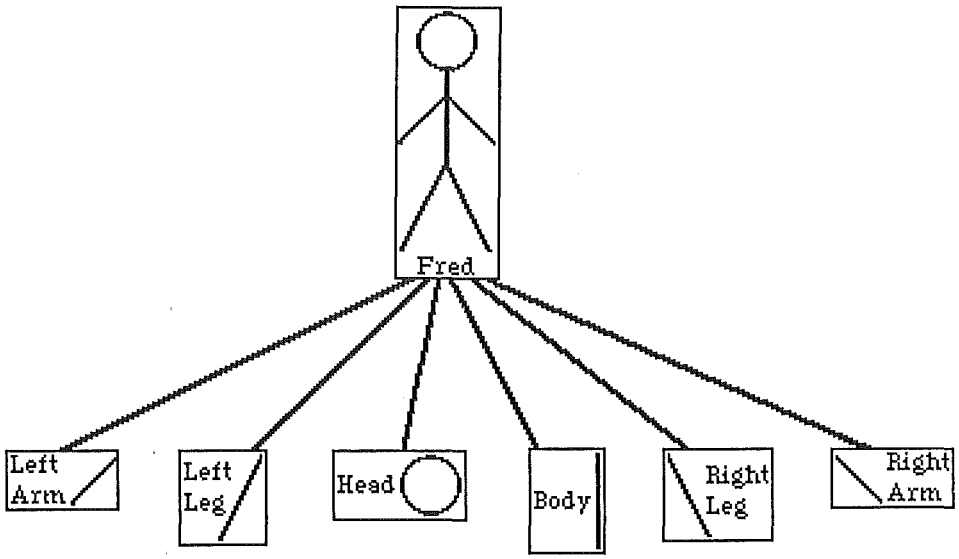
3.2.1 An animation object hierarchy

OASIS allows a user to define complex objects in terms of simple components. In order to allow this, a hierarchy of objects is maintained.



Consider the complex object *fred*, depicted above. In one sense, fred can be considered to be a 'stick man', but in another sense, fred consists of five lines and one circle, a collection of simple component objects. It could be said that fred *owns* his left arm, in the sense that the actions taken by fred affect the environment of fred's arm - if fred moves to the right, his arm will also move to the right. It would not be true to say that fred's arm owns fred, which would in effect be like saying 'the tail is wagging the dog'.

The drawing below represents an object hierarchy corresponding to fred.



In practice, the complex object fred stores a list of its component objects in one of its instance variables, but fred's components have no explicit knowledge that they are owned by fred. In order to create an object such as fred from several components, a user of the animation system would probably enter the sequence of commands below.

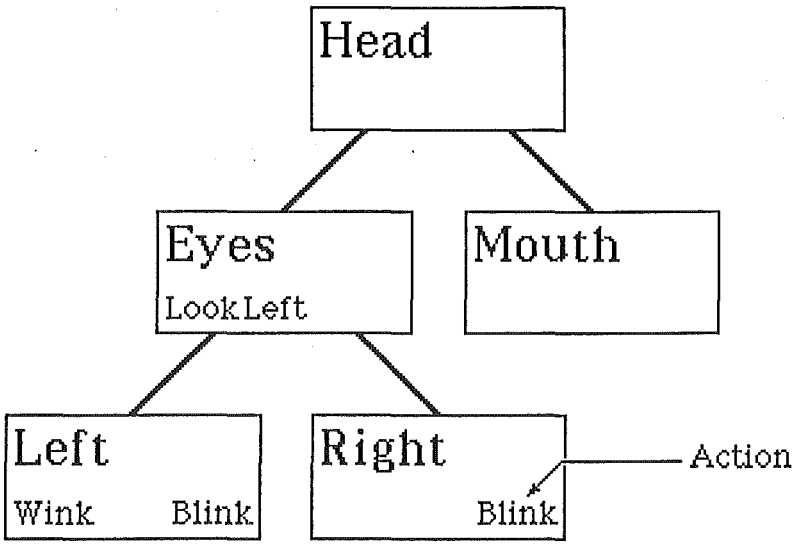
```
; create component objects
(define head (circle 'new))
(define leftArm (polyline 'new))
(define leftLeg (polyline 'new))
(define body (polyline 'new))
(define rightArm (polyline 'new))
(define rightLeg (polyline 'new))

; create a new compound object
(define fred (multiple 'new))

; add body parts to the list
; of fred's component objects
(fred 'owns leftArm)
(fred 'owns leftLeg)
(fred 'owns head)
(fred 'owns body)
(fred 'owns rightLeg)
(fred 'owns rightArm)
```

Here the *owns* message indicates ownership of an object. This message requires one parameter, the component object which is to be 'owned'. This simple way of indicating ownership of objects allows an arbitrarily complex hierarchy of objects to be constructed.

3.2.2 Message passing

The object hierarchy used in OASIS reproduces animation sequences through a powerful message passing protocol. Having defined some action for a composite object, a user of the system may request that it be displayed on the screen. This would be done by sending the message *go* to the object involved. The system's response may involve the animation of several component objects simultaneously. There are essentially two phases to the message passing protocol used to display animated sequences. In the first phase, each object concerned with the desired action is set up up to execute it. In the second phase messages are passed to perform each frame of the animation sequence.

Consider the composite object *Head*, illustrated below. This object has two components. Of these, *Mouth* is a simple object, but the object *Eyes* is a composite object, consisting of a left eye *Left* and a right eye *Right*. A few of these objects have actions associated with them. The action *Wink* has been set up to make the left eye wink. The action *Blink* is defined for both eyes - this should allow them to close simultaneously, producing a blinking effect. The action *LookLeft* is intended to make both eyes move to the left, and has been defined for the composite object *Eyes*.



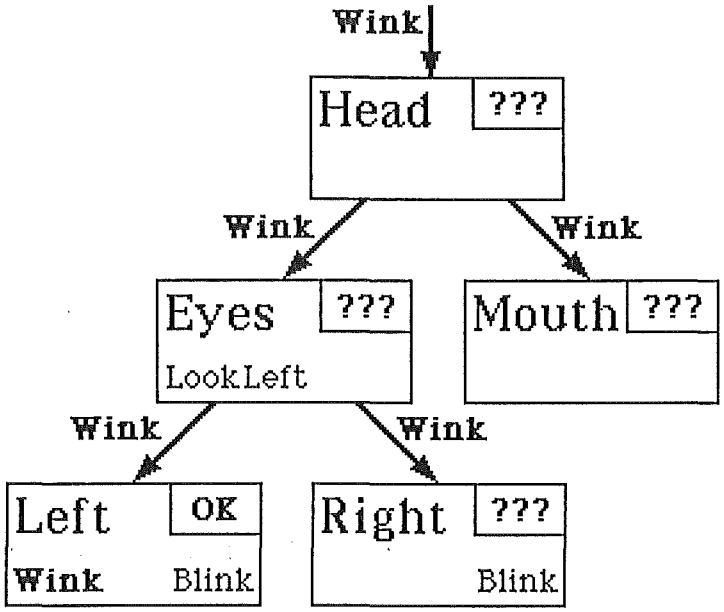
The *Wink* action may well have been defined as shown below.

```
; create new action
(Left 'action 'Wink)

; Start by changing eye into horizontal line
; (by flattening the oval)
(Left 'add 'Wink 'moveto 5 0 10 20 10)

; Now restore to eye shape
(Left 'add 'Wink 'moveto 5 0 0 20 20)
```

Suppose that the OASIS user now wants the object *Head* to *Wink*:

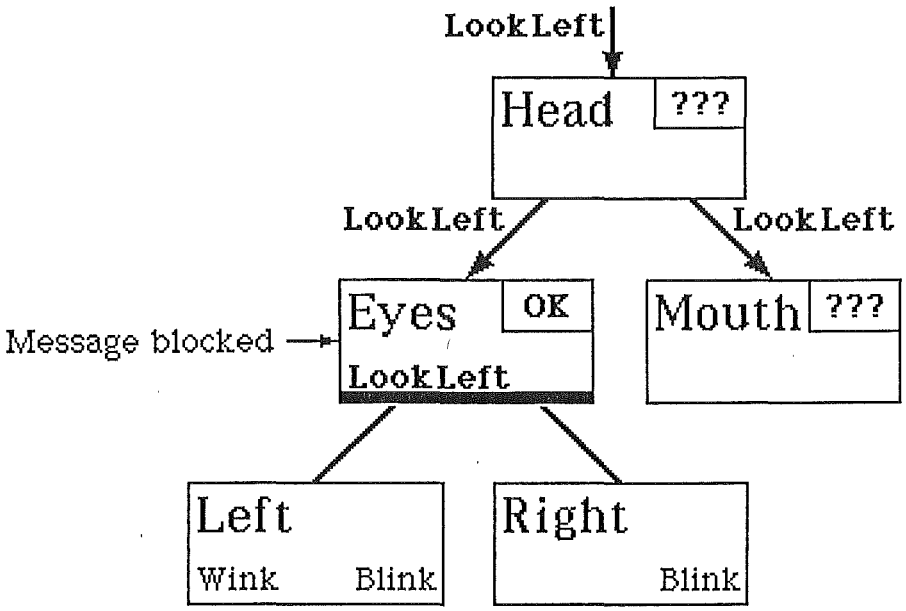
```
(head 'go 'Wink)
```

The message passing that results is shown in the diagram below.



An important feature of this system is the way in which messages are handled when they are not understood by a particular object. In many cases, the fact that some object in the hierarchy failed to understand an action does not mean that no object would understand it. For this reason it is important that composite objects relay an unrecognised message to all of their component objects, in case the message was intended for one of them. In this example, the object *Head* has been passed the message *Wink*. *Head* has no *Wink* action defined for it, and so it relays the message to its component objects. When *Mouth* receives the message, it too cannot understand it. As *Mouth* is a simple object it has no component objects, and so the message stops there. The object *Eyes* will relay the message to *Left* and *Right*. Of these, the object *Left* understands the message, and sets itself up to execute the corresponding list of animation primitives. In this case, the protocol has allowed a message to filter down through the entire object hierarchy so that it reaches the object that it was intended for. Of course this example is rather contrived. The user of the system could just as well send the message *Wink* directly to *Left*, which would produce the same result.

Often it is not necessary (and sometimes not even appropriate) for messages to be passed all of the way down the hierarchy. If a composite object understands a message, then it should have control over the execution of the associated action. It would not be worthwhile to pass this message to components of the composite object, because this could actually interfere with the normal execution of the corresponding animation sequence. In the case depicted on the following page, the user has decided that *Head* should perform the action *LookLeft*. Here the message is understood by the composite object *Eyes* and is not passed to its components.



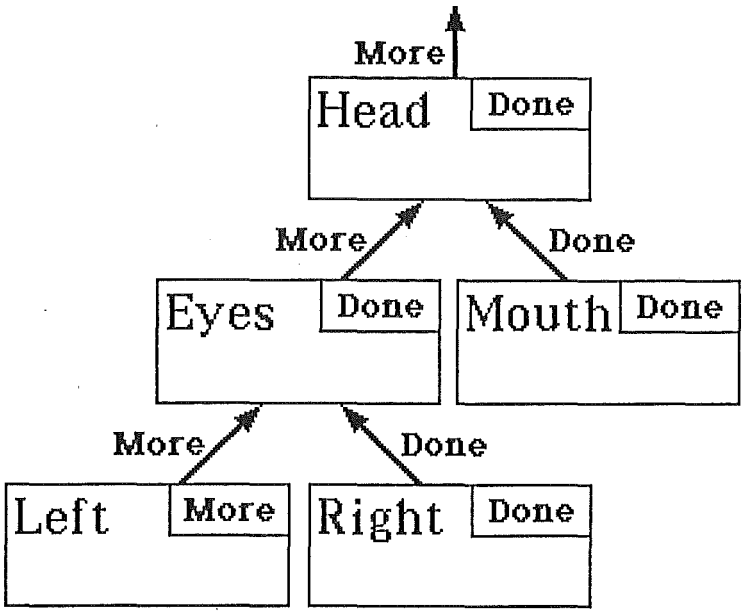
In addition to all of the message passing going on already, new actions can be initiated during the execution of an animation sequence. If a composite object executing an animation sequence finds a primitive that is not a normal one (such as *move*, *put* etc) but is an action name (e.g. *Wink*) then the action will be passed to the components of the composite object, overriding their current actions with the new one. The composite object then *immediately* fetches the next primitive and starts to execute it. This would allow, for example, the object *Head* to move to the left while blinking its eyes, by passing *Blink* to both eyes, and immediately starting to move to the left.

```
(Head 'action 'MoveBlink)
(Head 'add 'MoveBlink 'Blink)
(Head 'add 'MoveBlink 'move 10 -50 0)
```

Of course, all that the message passing covered so far achieves is to inform the appropriate objects of the action to be performed. Once this has been done, the chosen action must still be depicted. In OASIS, this is executed by a simple loop of the form

```
repeat do-next-frame until done
```

which will loop repeatedly, drawing the animation sequence frame by frame until the sequence is finished. The pseudo-procedure *do-next-frame* has been implemented by passing messages to each object in the animation hierarchy to make them perform whatever actions are necessary to draw the next frame of the animation sequence. Each object then returns a value indicating whether or not the animation sequence has been completed. In practice the message *iterate* is sent repeatedly to the object at the top of the hierarchy, until the Scheme value *#f* is returned, to indicate the end of the animated sequence. Inside the hierarchy, the iteration message is passed to every object in the hierarchy, and each object then returns *#t* or *#f* to indicate whether or not there is more to be done.

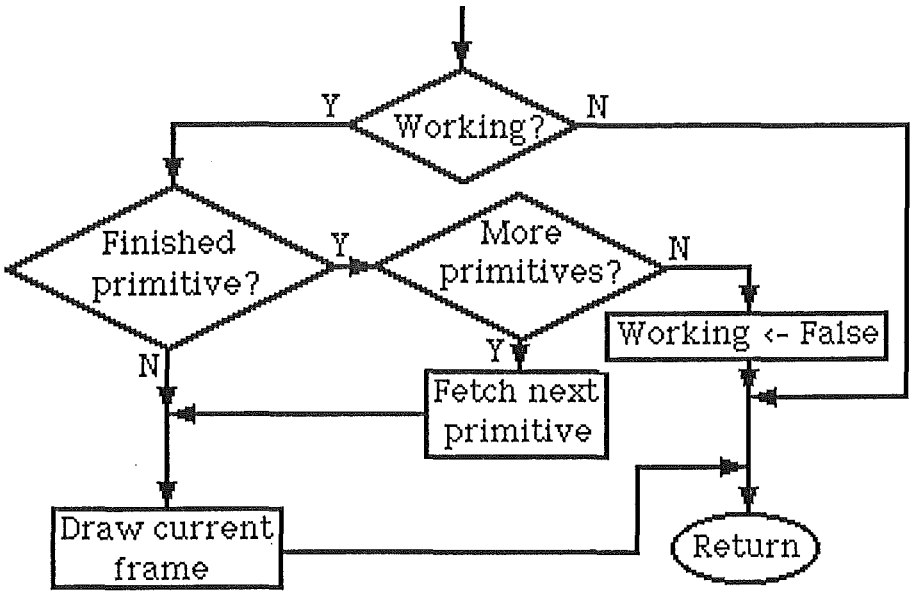


This is a typical example of how a returned value is generated. Each of the objects in this hierarchy has just completed a frame of the current animation sequence, and the boxes inset into each object indicate its internal status. For example, the object *Eyes* has completed all of the animation primitives associated with the current action, and so its *internal* state is 'Done'. A simple object indicates that it is finished if it has completed the action that it is directly concerned with. A composite object, however, only returns *Done* if it is finished *and so are all of its component objects*. This ensures that all objects in the hierarchy are finished before the animation sequence terminates. In the diagram above, *Eyes* returns *More* to its owner even though it has consumed its own list of animation primitives. This is because the message received from its component, *Left*, indicates that *Left* has not yet completed its own action.

3.2.3 Animation primitives

At this stage, the iteration method of an object appears as a kind of black box that miraculously performs whatever actions are necessary for each frame of an animation sequence. In reality, the situation is rather more complicated. In OASIS, the actions of an animation object are defined in terms of *animation primitives*, simple steps in an animated sequence. These animation primitives must be expanded by the iteration method to form a frame by frame sequence of drawing commands. As an example, the animation primitive *move* indicates that an object should be moved from its current position to a destination position in some specified number of frames. The iteration method must convert this into a series of small changes in the position of the object, and move the object one small step for each frame of the animation sequence.

A (greatly) simplified flowchart for the iteration method is depicted below.

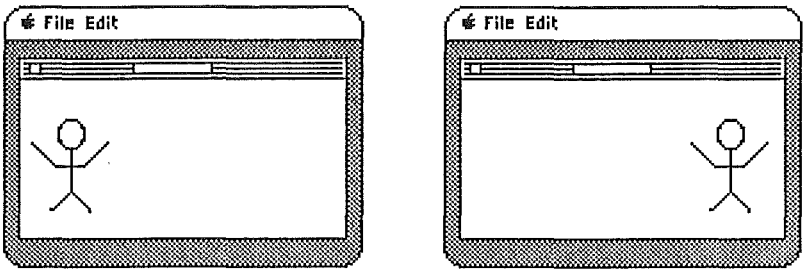


While the list of primitives corresponding to the selected action is being executed, the iteration method is essentially expanding each primitive in turn. It is possible of course that the iteration method may be invoked when there are no more animation primitives left to execute for the current action. In this case, the object merely adjusts its position according to any changes passed to it from its owning object (see section 3.2.4 for a discussion of the coordinate system used by the animation system). There are several details of the iteration process that have been omitted from the flowchart above for simplicity's sake. When an animation primitive is fetched and is found to be a message for lower levels in the animation hierarchy, the fetching procedure automatically passes it down the hierarchy and fetches the next primitive in the list. This simplifies the fetching process for the main iteration method, which is assured of receiving a primitive that must be expanded into a frame by frame sequence. When the primitive is fetched, a method is then invoked which sets up some parameters indicating what is to be done for each frame over the period in which the primitive is to be executed. For a simple object, drawing the current frame involves invoking its internal drawing method, but for a composite object this is achieved by passing the iteration message to each of its components.

Before the iteration method returns to its calling object, it is necessary to calculate the value that should be passed up the hierarchy to indicate whether or not the current animation sequence has been completed. In simple objects, this is the value of *Working*, but composite objects only indicate completion if all of their component objects have also completed their animation sequences.

3.2.4 Local Coordinate Systems

When drawing the objects involved in an animation sequence, the final position of an object on the screen must take into account the position of its parent object. If the parent object moves, so should its components. Consider the two situations depicted below.

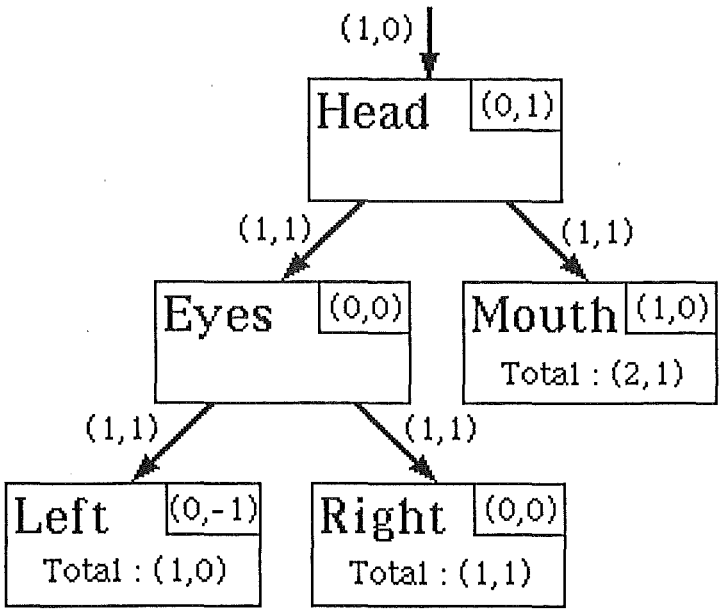


In the first picture, there is a composite object in the left half of the OASIS workspace window. A user of the system might send a message to this stick figure instructing it to wave its arm, and OASIS would respond with the appropriate action. In the second picture, the same stick figure has been moved to the right half of the window. The user of the system would still expect that the object would wave its arm in the same way as in the first situation, but the lines representing the arm for each frame of the 'wave' now have to be drawn in a different part of the window. In order to make this work, OASIS maintains a hierarchy of local coordinate systems corresponding to the hierarchy of animation objects. Each object maintains the position of the origin of its own coordinate system (which is expressed in terms of the window's coordinate system) and the positions of all component objects (or positional attributes of a simple object) are expressed as offsets from this origin.

This means that there are really two ways of moving an object. The owner of the object can move it by relocating the origin of its coordinate system. This moves the entire object at once, since all component (or attribute) positions are relative to the origin of an object's internal coordinate system. An object can also move itself, by changing its position relative to the origin of its local coordinate system.

This divides the movement of the object into two components, one due to internal movements, and one due to external movements. For each frame of an animation sequence, each object is passed the change in the position of its origin due to any movement by its parent object, as an argument to its iteration method. When the object calls the iteration methods of its component objects, it passes to them the *sum* of its own movements and those passed to it.

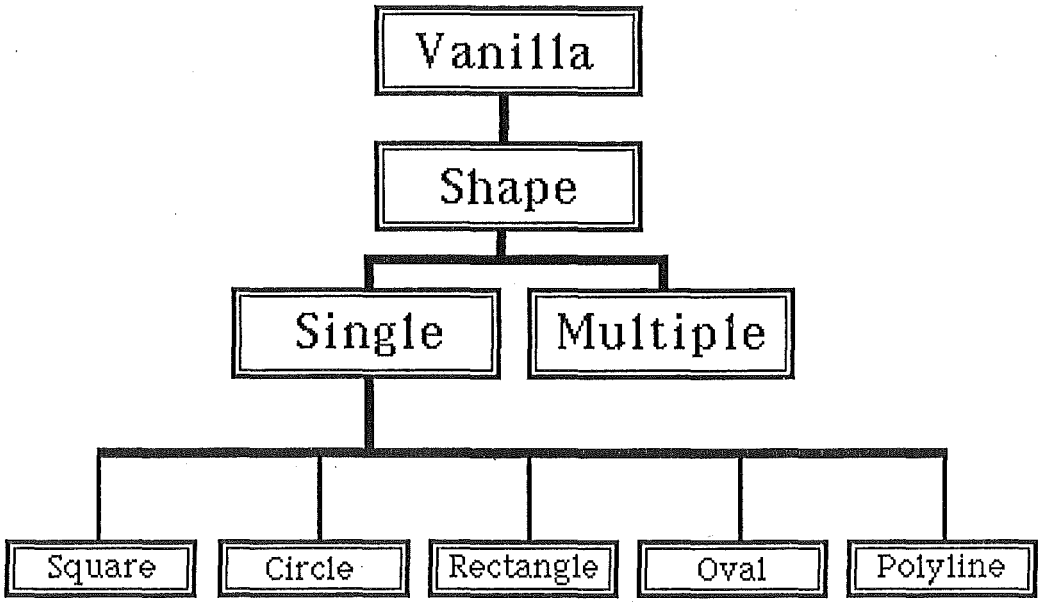
The diagram below gives an example of this process in action.



The object *Head* has been passed the initial movement (1,0). This might correspond to a movement of one pixel to the right. The action currently being performed by *Head* dictates an internal movement of (0,1), and so *Head* passes the resultant movement (1,1) to each component. Note the totals calulated in each simple object in this hierarchy. These have been included to show the final movement of the individual objects for this frame.

3.2.5 A flavour hierarchy

During the discussion of this system, it should have become obvious that there are at least two distinct types (or *flavours*) of animation object. The first type of object is the *composite object*, which must have the ability to relay messages to component objects, and to combine their responses. The second type is the *simple object*, which concerns itself mainly with the animation of a simple geometric figure. A more complete diagram of the flavour hierarchy used in OASIS is shown below.



At the top of this flavour hierarchy is, naturally enough, the flavour *Vanilla*. The highest level flavour defined for OASIS is *Shape*. This flavour contains the animation methods and instance variables common to all animated objects, and has two components. One of these, the flavour *Multiple*, contains all of the methods and instance variables explicitly concerned with the animation of composite objects. The other, *Single*, is concerned with the properties common to all simple objects (including animation primitives such as *move* or *put*). The component flavours of *Single* are concerned with the drawing and erasing of the various simple objects on the screen. In this way, a flavour like *Circle* would contain the methods to draw and erase itself, would inherit the methods and instance variables concerned with simple objects, and would also inherit the methods and instance variables concerned with general animated objects. This results in a great reduction in the amount of code that is duplicated in the various flavours making up OASIS. It also minimises the amount of work involved in adding new geometric primitives to the system. Only three methods need be defined for any subflavour of *Single*. A method must be provided to draw the object in the animation window, one to erase the object, and one to report the object's current position. A user of the animation system creates new instances of these animation objects using the message *new*. For example,

```
(define fred (circle 'new))
```

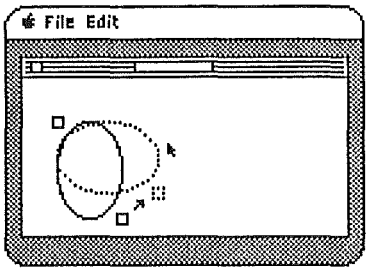
defines fred as a new instance of the flavour *Circle*.

3.3 The menu system

3.3.1 Features of the menu system

Having completed OASIS, it was time to address the second part of this project - the implementation of a *graphical front end* which would make the system easier to use. OASIS basically works by allowing the user to pass messages to objects using the normal MacScheme interface. Since OASIS has no real user interface of its own, it is possible to make use of the system from within another Scheme program. The menu system uses OASIS to provide all animation functions, which makes its implementation very easy.

In the context of an animation system, a graphical user interface has several advantages over a purely command-line based system. The most important advantage is undoubtedly the ability to specify the position and shape of an object visually instead of expressing it in terms of cartesian coordinates. Consider the diagram below.



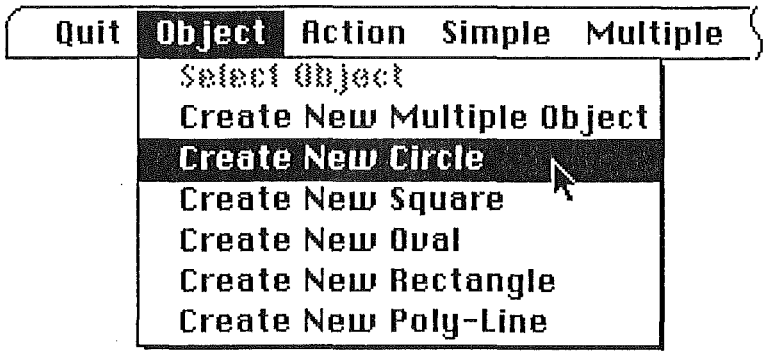

```
(circle 'add 'put 40 50 130 100)
```

Which is better, entering coordinates from the keyboard, or 'dragging' the object with the mouse until it is in the correct position? Clearly it is difficult to visualise the exact position of an object in cartesian coordinates.

The menu system has some other advantages also. A menu bar is provided which gives access to all of the operations of the ordinary animation system. This means that a novice user need not continually refer to the documentation to see what operations are permissible. If a particular menu option is not applicable at some point, then it is disabled. A user should be able to learn how to use most of the features of the system by browsing through the various menus and trying things. There are also some features that can be provided by the menu system which are not practical for the command line driven system. The command line driven system simply returns *#f* if an operation could not be performed successfully, so that a calling function can handle errors flexibly. Since the menu system need not have this flexibility, full reporting of error messages (through alert boxes) has been implemented. This makes it much easier for novice users to determine what they may have done wrong. The menu driven system can also take advantage of some special features of MacScheme which allow the animation workspace to be redrawn automatically after it has been obscured by another window.

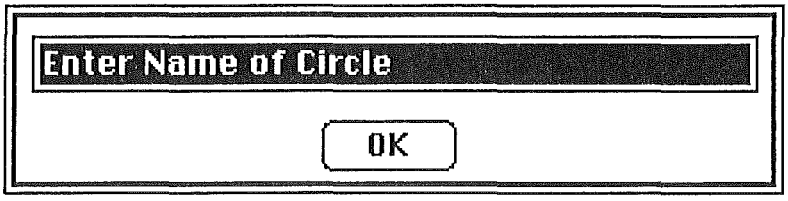
At this stage, it would be useful to provide a very short example of the menu system in action.

Suppose a user of the menu system wishes to create a circle, called *fred*, and create a short animation sequence in which it moves across the screen. Once inside the menu system, the user would start by selecting the *Create New Circle* option from the *Object* menu.

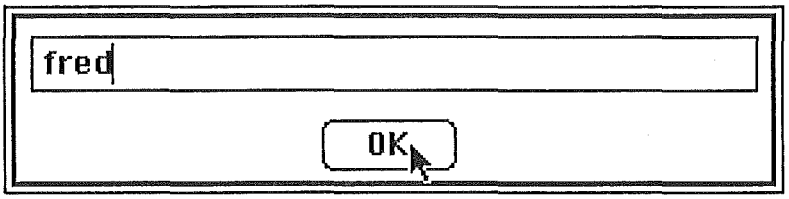


Note that at this stage the *Select Object* option has been disabled. Since there are no objects in the system yet, it is clearly not possible to select one of them. Although not shown in this diagram, all of the options in the *Action*, *Simple*, and *Multiple* menus would also be disabled at this stage.

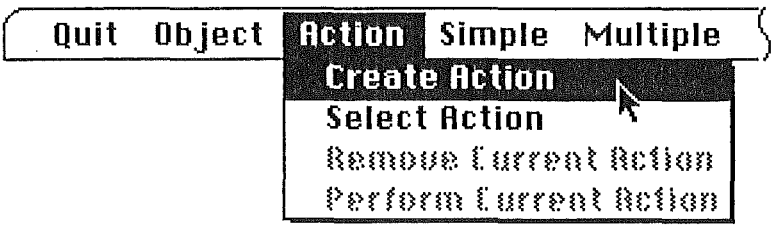
After the menu selection has been made, the dialog box shown here opens up.



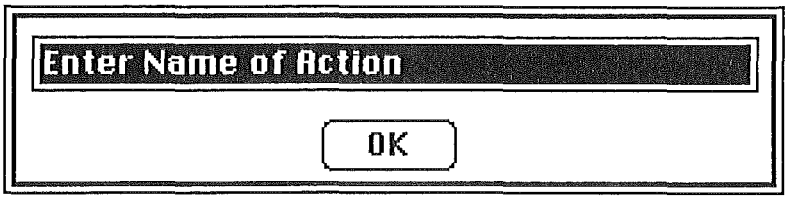
The user then enters the name of the newly created object, and clicks the mouse on the *OK* button.



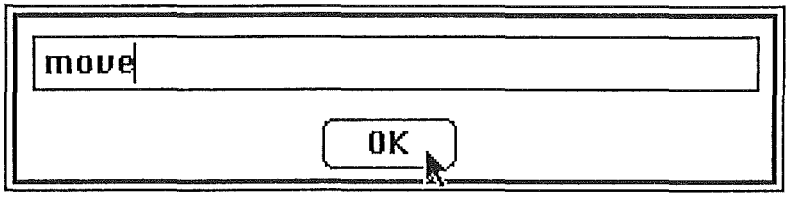
Having created a new object, an action must still be created for it if the user wants it to actually do anything. In this case, a new action *move* will be created to make *fred* move across the screen. The user selects *Create Action* from the *Action* menu,



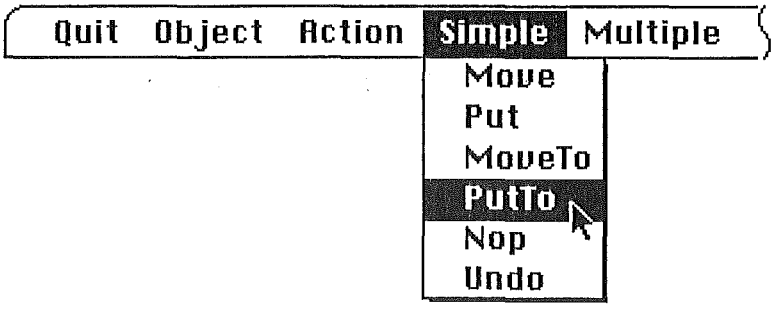
a dialog box pops up,



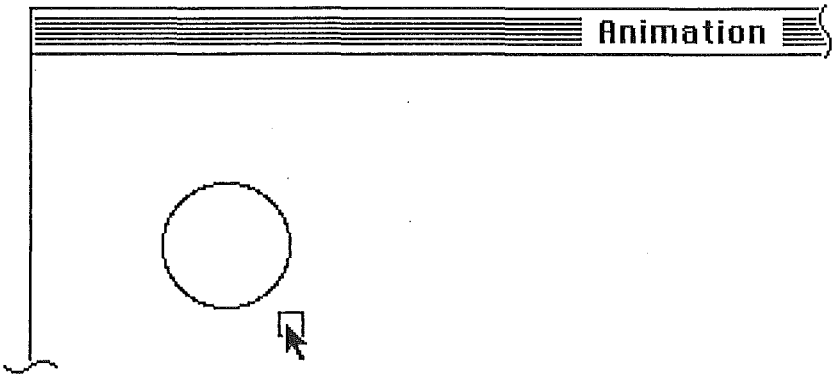
and the name of the new action is entered.



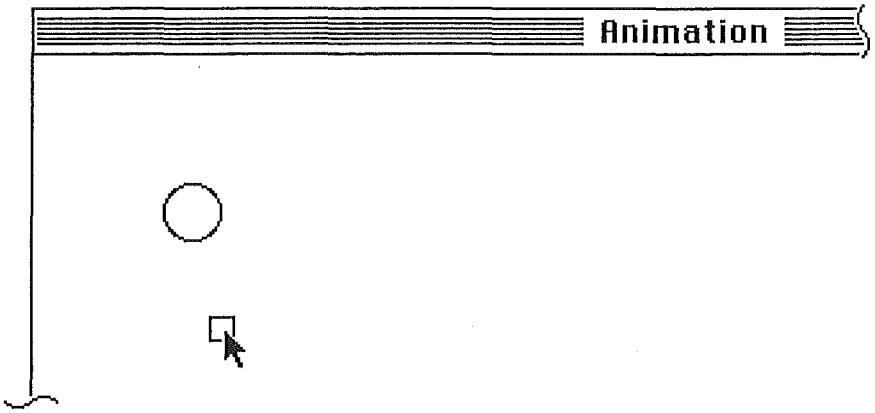
Now the user can specify the animation primitives involved in performing *move*. The first primitive in this case is *PutTo*, which puts an object at some position on the screen (the primitive *Put* does much the same but moves the object relative to its last position).



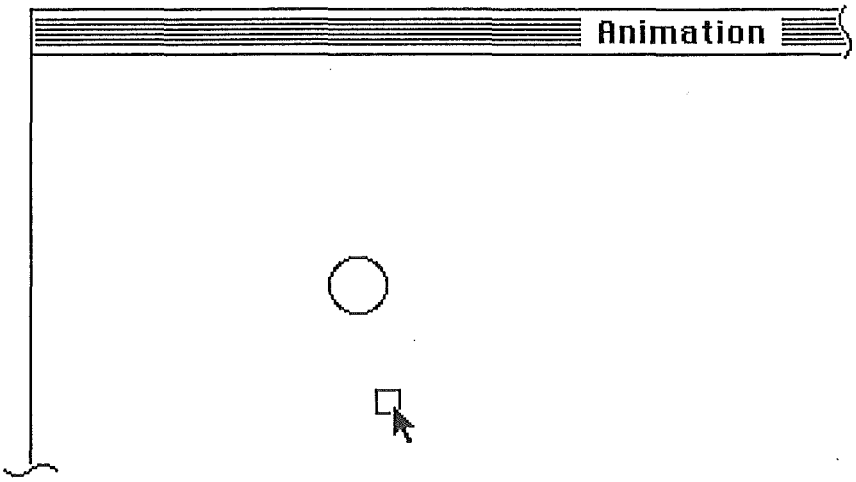
When this option is selected, the menu system allows the user to specify the circle's position interactively. The circle is drawn on the screen, with a small square (or *drag box*) at its lower right corner.



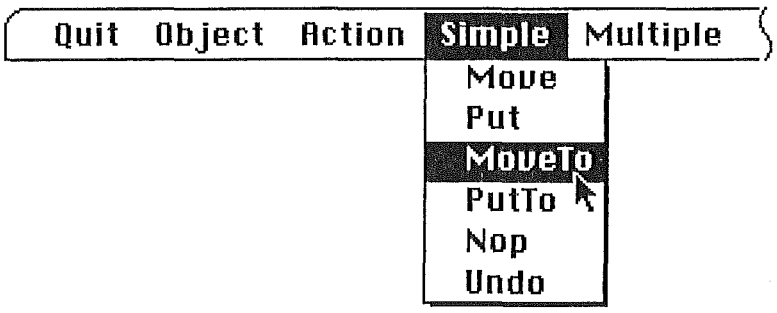
By dragging this box with the mouse, the size of the circle can be changed.



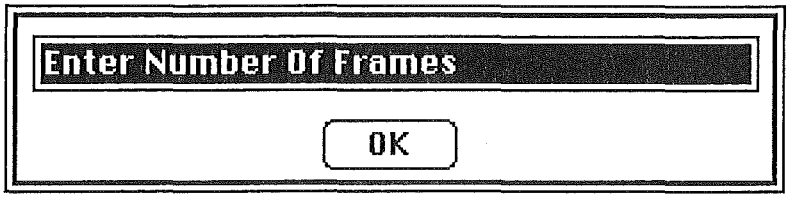
If the shift key is held down when the box is dragged, then the entire circle moves at once, allowing the user to change its position rather than its shape.



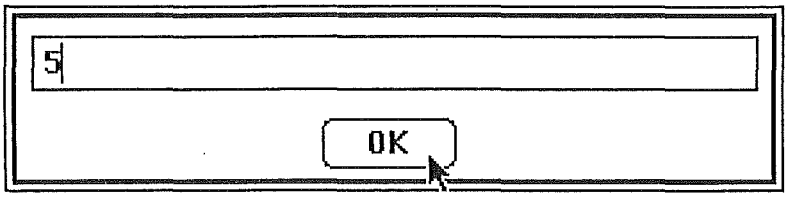
When the user is happy with the placement of the circle, the mouse is clicked outside the drag box, at which stage the primitive has been completely specified. Now that a starting position for the circle has been defined, it is time to try moving it across the screen. In this example, the *MoveTo* primitive is used.



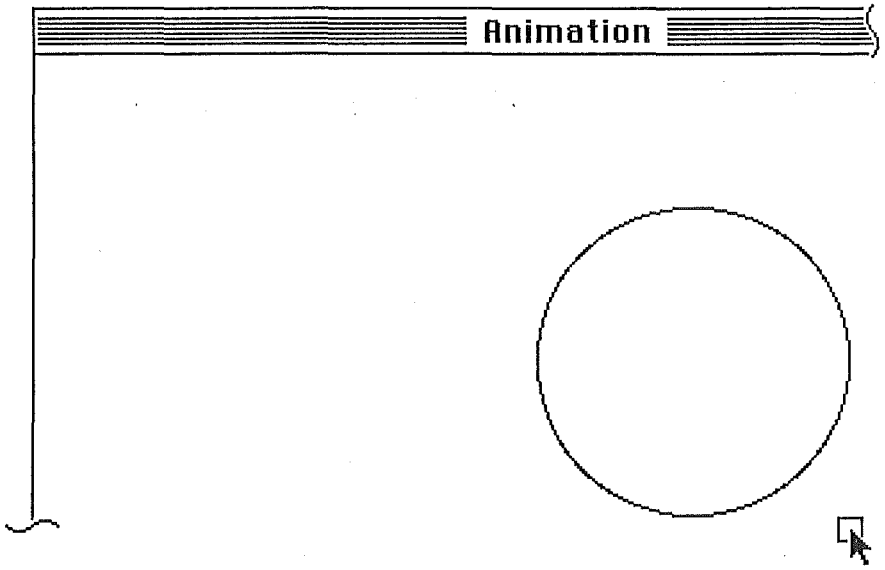
The *MoveTo* primitive requires the number of frames over which the object is to move as an argument.



In this case, the user types 5, and clicks in the *OK* box.



The position that the circle should move to is now specified in the same way as for the *PutTo* primitive.



The action has now been completely specified, and the user can view the finished product by selecting *Perform Current Action* from the *Action* menu.

3.3.2 Implementation of the menu system

Because MacScheme provides a highly interactive programming environment, the designers of the system were faced with an interesting problem when they took on the task of writing a Scheme interface to the Macintosh Toolbox routines. When a user is developing a Scheme application that uses windows and menus, it is highly desirable that he should still be able to access the normal Scheme windows and menus while his program is running. This would not only make debugging of a program easier, but it would also allow programs to be written which extended the MacScheme environment. In order to allow the coexistence of Scheme and application menus and windows, the designers of the system provided for the handling of windows, menus and events by implementing their own high level functions, only relying on standard toolbox calls for dialog boxes, QuickDraw commands etc.

For this reason, those parts of the OASIS menu system that deal with windows, menus, or mouse events (for dragging of objects in the animation workspace) are not implemented in the same way as they would be in another programming language on the Macintosh. Another minor incompatability occurs when it is necessary to interface Scheme data structures to those used in the toolbox routines. Apart from these qualifications, however, the menu system is a relatively straightforward collection of Toolbox calls. A good source of information on the Macintosh Toolbox is *Macintosh revealed* [3]. For MacScheme-specific information concerning the high level handling of windows, menus and events, refer to the *MacScheme+Toolsmith Preliminary Manual* [5].

Section 4

Possible Improvements

The design of OASIS provides a good basic framework for some possible improvements. Some of the changes suggested below were not implemented because of limited time, but some others were not implemented because preliminary experiments found them to be unacceptably slow in the MacScheme environment.

In the current menu system, there is no way that a user can save his work and return to it later. This limitation could be overcome by writing some representation of the objects in the system to a file on disk, and reloading it on request. A related issue concerns 'cloning' of objects. When a user has completely defined an animation object, he may want to have several copies of it in the workspace at the same time (e.g. turning a *person* into a *crowd*). The system could allow for this by copying the values of the object's instance variables across to a newly created instance of the same flavour.

In the current system, objects can only move in straight lines, or move between arbitrary points in a single frame. It would not be difficult to add to the system the ability to specify the function to be applied to determine where an object should move to for each frame of an animation sequence. This would allow objects to move along curved lines or sine waves for example.

In OASIS, there is no provision for the rotation or scaling of objects. The only attributes provided are positional. While this greatly simplifies the implementation of the local coordinate system used in the project, it greatly reduces the flexibility of the program. This limitation could be overcome quite easily by using homogeneous coordinate transformation matrices instead of positional offsets to transform between local and global coordinates. Unfortunately, a preliminary study of this technique showed that the additional overhead due to 3x3 matrix multiplication was unacceptable in the MacScheme environment, slowing the frame rate to the point where an animation sequence was more like a slide show.

The design of OASIS depends much more heavily on the principles of object oriented programming than on the unusual flexibility of the MacScheme programming environment. As a result of this it would be feasible to port the animation system to a faster, but less flexible, object oriented language (even many object oriented extensions to Pascal would be sufficient to support an implementation of the system). This could have great advantages in terms of the speed at which an animation sequence would be executed. It would also be feasible to port the system to a faster computer (such as a Sun), since the graphics functions required by the animation system are by no means unique to the Macintosh.

Section 5

References

[1]. Hearn, D. & Baker, M.P.; *Computer Graphics.*
Prentice-Hall, 1986.

[2]. Abelson, H. & Sussman, G.J. & Sussman, J; *Structure and Interpretation of Computer Programs.*
The MIT Press / McGraw-Hill, 1985.

[3]. Chernicoff, S; *Macintosh revealed.*
Hayden, 1985 (in two volumes).

[4]. *MacScheme .*
Semantic Microsystems, 1986 (the MacScheme reference manual).

[5]. *MacScheme+Toolsmith Preliminary Manual.*
Semantic Microsystems, 1986.

Appendix 1

Using OASIS

OASIS, as handed in with this project report, comprises three disks:

OASIS 1 - This disk contains an executable image of the OASIS menu system. Double click on the file *OASIS+menus* to start the program. The file *animateresource* contains the resource definitions of dialog and alert boxes, and must be in the same folder as *OASIS+menus*.

OASIS 2 - This disk contains the latest version of MacScheme+Toolsmith, and a dumped heap image of the basic OASIS system. Double-click on the heap image *OASIS* to execute. Also included are two sample files, *fredhead* and *man*, that demonstrate some of the features of the animation system. These can be executed from within the animation system using *load* or by opening them into windows, selecting the code, and using the *Eval* menu option.

OASIS 3 - This disk contains all of the sources for the animation system and menu system. To build the dumped heap image of the animation system from the source code, MacScheme+Toolsmith must first be started with the heap called *toolsmith.heap* . The following sequence of commands must then be entered (commands marked [*] are optional but probably a good idea).

```
[*]     (set! include-lambda-list? #f)
[*]     (set! include-source-code? #f)
[*]     (set! include-procedure-name? #f)
        (load "class.sch")
        (load "flavour.sch")
        (load "p2.sch")
[*]     (set! d (multiple 'new))
[*]     (set! d (square 'new))
[*]     (set! d (circle 'new))
[*]     (set! d (rectangle 'new))
[*]     (set! d (oval 'new))
[*]     (set! d (polyline 'new))
[*]     (set! include-lambda-list? #t)
[*]     (set! include-source-code? #t)
[*]     (set! include-procedure-name? #t)
        (dumpheap)
```

To build the menu system application, start with the menu system loaded into Scheme, then enter

```
(load "loader.sch")
```

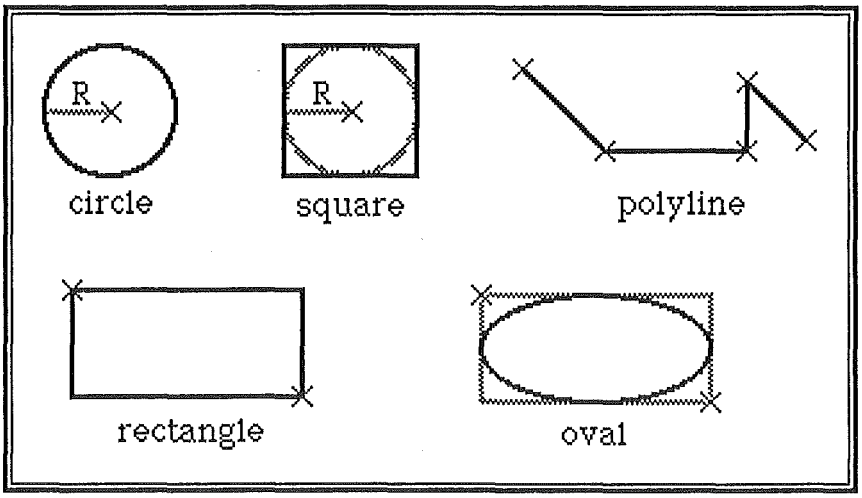
1.1 Using OASIS

1.1.1 The Macintosh coordinate system

Before delving into OASIS, it is important to bear in mind the underlying coordinate system used to represent the positions of objects in the animation workspace. As the horizontal coordinate of a point increases, the point moves to the right, as one would expect in a cartesian coordinate system. When the vertical coordinate of a point increases, the point moves ***downward***, not upward as one might expect. This is very important as it means that points with smaller vertical coordinates are always ***above*** those with larger vertical coordinates.

1.1.2 Animation object types

OASIS supports six basic types of objects. Five of these are simple geometric shapes - the sixth is a composite object which consists of many components.



1. CIRCLE

A circle has three attributes. The first two attributes represent the position of the centre of the circle in the animation workspace, and the third is the circle's radius. For example, attribute values of 100,120, and 30 would represent a circle at the position (100h, 120v) with radius 30.

2. SQUARE

A square has three attributes, essentially the same as for the circle above. The radius attribute of a square represents the radius of the largest circle enclosed by the square. For example, attributes of 100, 120 and 30 would put the corners at (70h, 90v), (130h, 90v), (130h, 150v) and (130h, 90v).

3. RECTANGLE

A rectangle has four attributes. The first two of these define the position of the upper-left corner of the rectangle, the second two the lower-right corner of the rectangle. For example, a rectangle with attributes 70, 90, 130, and 150 would have its four corner points at (70h, 90v), (130h, 90v), (130h, 150v) and (130h, 90v).

4. OVAL

An oval is to a rectangle what a circle is to a square. An oval has four attributes, which essentially define a rectangle in the workspace. The oval drawn is the largest that will fit within this bounding box.

5. POLYLINE

A polyline has an number of attributes which should be even, and at least 4. The attributes of this object describe a series of points in the animation workspace which are joined by line segments. For example, attributes of 100, 100, 120, 120, 130 and 150 would result in a line from (100h, 100v) to (120h, 120v) and one from (120h, 120v) to (130h, 150v).

6. MULTIPLE

A multiple object consists of several components. It has two attributes, which define its position in the animation workspace.

1.1.3 OASIS commands

`(start)`

This command opens the workspace window in which all OASIS objects are displayed. It is usually entered at the beginning of an OASIS session. The workspace window must be open when an animation is performed, otherwise an error will occur.

`(end)`

This command closes the workspace window, and is usually entered at the end of an OASIS session.

`(object-type 'new)`

When a new instance of an OASIS object is needed, the message *new* is passed to the appropriate flavour. For example, `(define fred (circle 'new))` defines fred as an instance of a circle.

`(object 'owns another-object)`

This message is only applicable to objects of type *multiple*. It specifies that the current object owns the specified object. Using this message, composite objects are constructed out of simple components. For example, `(barney 'owns fred)` defines the object *barney* as owning the circle *fred.*

`(object 'refresh)`

If another window is moved on top of the workspace window, portions of an animation object may become obscured. If the object later has to be re-drawn, then the message *refresh* can be sent to it to restore its original appearance.

`(object 'action action-symbol)`

This message creates a new action for *object*, giving it the name passed as *action-symbol*. For example, `(fred 'action 'yahoo)` would create the new action *yahoo* for the object *fred*.

`(object 'remove action-symbol)`

This message removes the action specified by *action-symbol*.

`(object 'add action-symbol <animation primitive>)`

This message adds an animation primitive to the list of primitives associated with the specified action (the available primitives are discussed in section 1.1.4).

`(object 'undo action-symbol)`

The last animation primitive is removed from the list associated with the specified action.

`(object 'go action-symbol)`

The specified action is performed by the system. The resulting animation sequence is displayed in the workspace window, which should be open before this command is used.

1.1.4 Animation primitives

`'put <attributes>`

Changes the attributes of the object by the amounts specified. This changes the shape and position of the object relative to its current shape and position. *put* takes one frame to execute. For example, `(fred 'add 'put 50 50 10)` would (when executed) simultaneously move the circle *fred* 50 pixels to the right, 50 pixels down, and increase its radius by 10 pixels.

`'move frames <attributes>`

Changes the attributes of the object by the amounts specified. Unlike *put*, the change is spread (using linear interpolation) over the specified number of frames. The number of frames specified should be at least one. For example,
`(fred 'add 'move 5 50 50 10)` would do the same as the previous example, but the change would be spread over five frames instead of a single frame.

`'putTo <attributes>`

Changes the attributes of the object to the values specified. This sets the shape and position of the object. *putTo* takes one frame to execute.
For example, `(fred 'add 'putTo 50 50 10)` would simultaneously move the circle to the position (50h, 50v), and set its radius to 10 pixels.

'moveTo frames <attributes>

Changes the attributes of the object to the values specified. Unlike *putTo*, the change is spread (using linear interpolation) over the specified number of frames. The number of frames specified should be at least one. For example,
`(fred 'add 'moveTo 5 50 50 10)` would do the same as the previous example, but the change would be spread over five frames instead of a single frame.

'nop frames

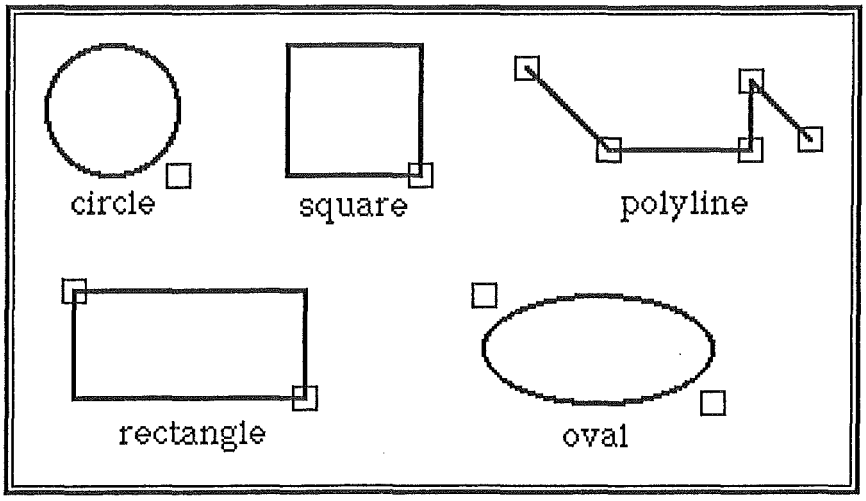
The object does nothing for the specified number of frames. The number of frames should be at least one.

'<action>

This 'primitive' only applies to objects of type *multiple*. It causes all of the components of the current object to begin executing *action*. This command takes **no** frames to execute.

1.2 Notes on the menu system

The menu system presents OASIS commands and animation primitives through Macintosh-style pull down menus. Most of this system should therefore be self explanatory, with the primitives and commands doing the same things as in ordinary OASIS. The main innovation of the program needing some explanation is the way in which the position of an object in the animation workspace is specified. Basically, a geometric figure is put in the animation workspace that has one or more *drag boxes* associated with it.



The drag boxes are the small squares attached to the simple geometric shapes above. There are two ways of using these boxes. When the boxes are dragged with the mouse (by clicking the mouse inside one, holding the button down, and moving the mouse) then the shape of the object associated with them changes. When the shift key is held down and a box is dragged, the whole object moves without changing its shape. Try working through the example in section 3.3.1.

When a composite object is to be moved, a different system is used.

A simple line segment is displayed in the workspace which can be dragged with the mouse. The position of this segment is not important. It really represents a *vector*, indicating the direction and extent of movement of the composite object. The user of the system therefore positions the tail of the line on some convenient part of the object (using shift-drag) and then drags the head of the line until the direction and extent of movement of the object is satisfactory.