1

Microl

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

An Object-Oriented Modelled Animation Control Editor

by

Rónald L. Huizen B.C.S.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario
August 14, 1987

© copyright
1987, R. L. Huizen

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-39415-3

6

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis

**An Object-Oriented
Modelled Animation
Control Editor**

submitted by Ronald L. Huizen, B.C.S.,
in partial fulfillment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Chairman, School of Computer Science
Carleton University

Date

Abstract

This thesis describes the design and implementation of the Modelled Animation Control Editor (MACE), a system comprised of a graphics style editor used for controlling a built-in three dimensional modelled animation system. The editor is used for controlling the system itself, as well as for specifying and controlling motion in the animation. The main emphasis of MACE is in the specification, control, and synchronization of motion.

MACE allows the user to visually manipulate not only the animated objects but also the objects used to control the animation. A small set of simple yet powerful control objects is provided which can be used for creating complex control structures. As a part of its interface MACE employs a three-dimensional cursor which allows for the selection and manipulation of objects in three dimensional space. MACE also introduces the concept of alter values, which are used to synchronize complex control strategies. Alter values allow for multiple actions to occur on an object simultaneously.

MACE was implemented in Smalltalk/V, an implementation of the Smalltalk-80 programming environment. This thesis discusses the advantages and disadvantages of using Smalltalk as the implementation language, lending further support for the use of object-oriented languages as base languages for animation systems.

Acknowledgements

First and foremost I thank my wife, Jayne, for her endless encouragement and support, as well as her help in preparing the final drafts of this document. I couldn't have done it without you Jayne.

I also wish to thank my supervisor, Wilf Lalonde, and my pseudo-supervisor, John Pugh, for their support (both financial and otherwise). They have provided guidance as well as many useful ideas and discussions.

Special thanks to John Duimovich and Mike Wilson for their help in answering endless Smalltalk questions, Paul White and Dave Thomas for their input in various informal discussions, and Kent Johnson for various system support activities above and beyond the call of duty.

Finally, I would like to acknowledge the funding provided by Carleton University.

Table of Contents

	Page
Acceptance Sheet	ii
Abstract	iii
Acknowledgements	iv
Table of Contents	v
List of Figures	viii
1. Introduction	1
1.1 Motivation	1
1.2 Thesis and Contribution	3
1.3 Overview	4
2. Modelled Animation Systems	6
2.1 Modelled Animation	6
2.2 Related Systems	8
2.2.1 MIRANIM	8
2.2.2 DIRECTOR	10
2.2.3 ANIMUS	11
2.2.4 ARK	12
2.2.5 STELLA	14
2.2.6 Others	15
2.3 The Advantages of Smalltalk	16
2.4 Summary	19
3. An Overview of Modelled Animation Control Editor	24
3.1 The MACE Environment	24
3.2 The Three Dimensional Cursor	26

Table of Contents

3.3	Reactions	28
3.4	Animation Clocks	28
3.5	Alter Values	30
3.6	Animation Laws	34
3.7	Connectors	34
3.8	Combiners	36
3.9	Animation Events	38
3.10	Animated Objects	40
3.11	Virtual Cameras	41
3.12	Animation Worlds	42
3.13	Animation Browsers	43
4.	An Example Animation Created With MACE	45
4.1	Defining the Model	45
4.2	Defining the Control	52
4.3	Creating the Animation	60
5.	The Implementation of MACE	65
5.1	Smalltalk	65
5.2	The MACE Environment	66
5.3	The Three Dimensional Cursor	66
5.4	Animation Clocks	67
5.5	Alter Values	68
5.6	Animation Laws	69
5.7	Virtual Cameras	70
5.8	Animation Browsers	71
6.	Conclusion	72
6.1	Future Work	74

Table of Contents

References	77
Appendices	79
Appendix A: MACE Commands	79
Appendix B: Smalltalk/V Source Code for MACE	83

List of Figures

Figure	Description	Page
3.1	The Basic MACE Environment	25
3.2	Using Connectors	35
3.3	Using a Combiner	37
3.4	Using an Event	39
4.1	The Initial Animation Browser	48
4.2	The Box	50
4.3	The Box and Balls	52
4.4	An Animation Law Browser	53
4.5	The Ball Collision Event	55
4.6	The Box-Balls Collision Event	56
4.7	The Two Virtual Cameras	58
4.8	The Complete Control Structure	60
4.9	A Frame from the Animation	63

Introduction

Chapter 1

1.1 Motivation

The process of creating animations involves the creation of an extremely large number of actual pictures, or frames. To create the illusion of movement, we must display these frames in rapid succession, where each frame differs from its predecessor very slightly. The generally accepted rate of display to produce smooth motion is 24 frames per second for video and 30 for film. Rates lower than this allow the eye to see the jumps between the frames. Therefore, to produce just one minute of animation on video, we must create 1440 frames (24 frames/second X 60 seconds).

As you might suspect, the animation process lends itself well to computer automation. In fact, many animation systems are available today that have become great assistants to the animator. These packages allow the animator to "paint" the frames on the computer, with most letting the animator paint only the "key frames". Key frames are those in the animation that show major parts of the motion. Generally, we might need a key frame every second or so. The system then uses interpolation methods to generate the "in between" frames, thus taking over the job of the animator's assistant (also known as the inbetweener).

These types of systems, used at such famous animation studios as Lucasfilm and Disney, come under the classification of computer assisted animation. The other area of computer animation is modelled animation.

Modelled animation involves modelling the world that we are trying to create an animation of. Generally, we create representations for objects in that world, and allow operations on these objects. In a way, these systems can be grouped with simulation systems.

There are three main activities involved in modelled animation: object modelling, motion specification and synchronization, and image rendering. **Object modelling** involves the description and construction of the objects to be used in the animation. **Motion specification and synchronization** is the process of giving motion to the objects and synchronizing these motions over time. **Image rendering** consists of creating realistic images of the objects.

The majority of current modelled animation systems place their emphasis on two of the three basic activities: object modelling and image rendering. While these two tasks are very important, and necessary for creating realistic animations, the remaining task of motion specification and synchronization, is also very important. The key feature of animation systems is the creation of the illusion of movement. Hence it is logical that the main task of a modelled animation system should be the specification and synchronization of this motion. Although there are some systems that emphasize this aspect, none provide the end user with control that is simple and straightforward while maintaining the flexibility and power necessary for a general purpose modelled animation system.

1.2 Thesis and Contribution

It is our thesis that the combination of techniques and concepts described herein, and implemented in the **Modelled Animation Control Editor (MACE)**, provide the end user with control that is simple and straightforward while maintaining the flexibility and power required by a general purpose modelled animation system. While other systems allow the user to visually manipulate the animated objects, MACE also provides the user with visual representations of its main control concepts, allowing them to be manipulated. To our knowledge, no other system provides this capability.

The main contributions of this thesis are the design and implementation of MACE's visual interface and the introduction and implementation of alter values, a method of synchronizing complex control strategies. In addition, this thesis lends further support for the use of object-oriented languages as base languages for modelled animation systems, demonstrating the flexibility and power these languages provide.

The visual interface is achieved by a graphics style editor. It is comprised of a command area and a view area, which is a window onto the three dimensional world being modelled. A three dimensional cursor is developed that is used for manipulating objects in the view.

Alter values provide us with the ability to average out simultaneous independent changes to an object. Basically, an alter value is a replacement value for an attribute of an object. When a new value for the attribute is computed, it is recorded as an alter value without changing the original. Independent modifications can each recompute their own

alter values. At the end of each time step when all computations have been performed, the alter values are resolved into a single unique value which replaces the original attribute value. Usually, the resolution of alter values is a simple averaging process but other resolution techniques are permitted.

1.3 Overview

This thesis describes the design and implementation of MACE. The effort expended in MACE is involved almost solely in motion specification, control, and synchronization. This thesis focuses on these issues, briefly discussing the minimal amount of effort put into object modelling and image rendering.

Overall, MACE is a modelled animation system designed to control simple user specified micro-worlds. Its intended use is the creation of scientific and educational animations. The objects in the micro-worlds are the main entities. They control their own individual motion. Global control is necessary to synchronize these modelled objects. The user may create simple animations merely by using the mouse with very little keyboard input, or may create complex animations through programming refinements and extensions to MACE's animated and control objects. Once defined, these new entities are available for future use.

Chapter 2 will present basic concepts of modelled animation. An overview of related systems will be given, highlighting the major features of each. The advantages and disadvantages of using Smalltalk will be discussed, followed by a summary of how MACE relates to the discussed systems.

The MACE system will be introduced in Chapter 3. The major components of MACE will be presented, and the underlying concepts behind each will be explained.

Chapter 4 goes through a specific example animation, showing how it was created with MACE. This example demonstrates the use of MACE's major components.

Implementation issues will be dealt with in Chapter 5. Methods used to implement the components of MACE in an object-oriented programming language will be discussed.

Finally, Chapter 6 provides a conclusion to this thesis. Suggestions for future work and extensions are given.

Modelled Animation Systems

Chapter 2

This chapter discusses modelled animation systems. It begins with an introduction to some of the basic principles of modelled animation, and then goes on to give a brief look at some of the more powerful systems that exist. A discussion of the advantages and disadvantages gained by using the Smalltalk programming environment is given, followed by an overall summary of how MACE relates to the other systems presented.

2.1 Modelled Animation

Modelled animation can be defined as the process of creating animations by developing computer representations of objects and then manipulating these objects over time. The major difference between modelled and key-framed systems is the fact that we manipulate objects as opposed to images of objects.

Key-frame systems allow the user to "paint" the key, or major, frames of an animation. The system then generates the inbetween frames automatically. The major technique used to accomplish the generation of the inbetween frames is interpolation. These systems are very useful for creating production animations since they reduce the amount of time the animator must spend drawing similar pictures. These systems also tend to be easier to introduce into an already existing animation studio since they do not drastically change the way in which animations are created.

Modelled animation systems, however, do change the method by which animations are created. Instead of creating countless numbers of images, animators model the major objects in the images. Once modelled correctly, an animation is produced by manipulating and displaying the objects.

As stated in chapter 1, there are three main activities involved in modelled animation: object modelling, motion specification and synchronization, and image rendering. The role of the user of a modelled animation system is therefore threefold. First, the user is a producer, getting all the necessary resources into place. Since the objects must be modelled before they can be used, he can design them as part of this task. Second, the user is a director, specifying and synchronizing the various resources the producer has provided. Finally, the user is an editor, taking what the director has provided and producing the final animation from the various pieces.

We can see that modelled animation systems not only model the objects to be animated, they also model the way in which motion pictures are created. For this reason, we feel that a modelled animation system would be easier to introduce into a new studio since the required personnel to use a powerful system effectively would not be present in a traditional studio.

Also for the same reasons, modelled animation systems are more widely accepted in the scientific world, where the users are familiar with the process of modelling and simulation, and tend to use it (one way or another) in most of their work. Modelled animation systems provide them with a method of putting their work into animations, to be used either as a research tool, or as a learning tool.

Modelled animation systems seem to split themselves into two groups: those concerned with producing realistic looking animations, and those more concerned with producing very accurate animations. Systems such as ANIM8 [3] provide a great deal of accuracy by employing numerical integration techniques to generate the values over time, instead of using discrete techniques that most other systems employ. Systems such as MIRANIM [14,16] tend to concern themselves more with producing realistic animations and shortening the time required to create an animation.

2.2 Related Systems

We have presented a brief view of the basics of modelled animation systems, and can now go on to look at some of the related systems that MACE draws from.

2.2.1 MIRANIM

MIRANIM [14,16] is an extensible director-oriented three dimensional modelled animation system, developed at the University of Montreal. It combines an animator-oriented system, ANIMEDIT, with an animation sublanguage, CINEMIRA-2. CINEMIRA-2 is a subset of CINEMIRA [14], limited to programming entities to be used by ANIMEDIT. Users extend MIRANIM by programming the extensions in CINEMIRA-2. ANIMEDIT provides the front-end, allowing the user to create animations by issuing various commands.

MIRANIM provides motion control by letting the user define animated variables, which can be governed by predefined laws. To control an object users do not attach laws

to objects, but rather attach them to variables, which are then associated with attributes of specific objects.

Objects in MIRANIM can be read in from predefined object files. Once read in, they can be rotated, transformed, scaled, and colored. As well as basic animated objects, MIRANIM recognizes decor objects, which do not have any animation characteristics associated with them (i.e. they don't change over time). MIRANIM supports over 16 types of transformations on objects, from simple operations such as translation, to more complex ones such as shears. The user can create animated objects and specify the transformations to be applied to each.

As with other systems, MIRANIM supports virtual cameras, and allows the user to define one or more cameras and associate animated variables with them. It also supports zooming and spinning. MIRANIM supports light sources and allows the user to specify motion for these sources. This is a powerful feature of MIRANIM, not offered in many systems. The final creation of an animation with MIRANIM involves using the director mode to specify starting time and duration of the animated objects, cameras, and decor, as well as control of shooting and playback.

The CINEMIRA-2 sublanguage allows the user to program new transformations and laws that can then be accessed from ANIMEDIT. As stated previously, CINEMIRA-2 is limited to allowing the definition of entities that can be used in ANIMEDIT.

Overall, MIRANIM provides a powerful and flexible system. It is oriented to the non-computer user, but provides a method by which it can be extended. One of its

shortcomings lies in its user interface, which is comprised of textual command input that forces the user to learn a special command syntax to use the system. In our view, such textual command input is in itself a type of programming. Hence it is the job of a director-oriented system to present the director with a programming environment that provides the easiest and fastest method for accomplishing his or her task. While MIRANIM allows the director to accomplish the task, it does so in a cumbersome manner, forcing the director to memorize complex syntax.

Another shortcoming of MIRANIM is the control methods available. The major entities for controlling an animation are the animated variables, but these are limited to being reals or vectors. As well, laws, another major entity, are restricted to functions that return vectors and reals. A result of these limitations is that we cannot create a law that takes into account various attributes of an object easily. We must create a complex network of laws and variables to accomplish this task.

As previously stated, MIRANIM is a very powerful system. It attempts to achieve an object-oriented approach to animation, but suffers due to the fact that its underlying implementation is in Pascal.

2.2.2 DIRECTOR

DIRECTOR [11], developed at MIT, is an object-oriented modelled animation system. It is only two-dimensional but provides some interesting ideas. DIRECTOR is implemented in LISP, and draws heavily on the turtle geometry concepts first introduced in LOGO. It also draws directly from Hewitt's actor theory. Every object can receive a basic

set of messages, including the basic turtle commands, used for controlling the animation.

The messages sent to the objects comprise the animation, and the user creates a script that directs the overall flow of events. DIRECTOR is basically event driven. The user can plan to take some action at pre-specified points in time. It supports messages such as "move 100 over 10 ticks starting in 30 ticks", which causes the object to move 10 each tick for 10 successive ticks starting 30 ticks from the current time. We can expand the set of messages an object receives, and ask it to remember values.

The most powerful concept DIRECTOR introduces to animation systems is the idea of attaching behaviour to objects. We can tell an object how to react to certain stimuli, where the stimuli are introduced as messages to the object. For example, we could tell an object that when someone asks (sends a message that says) "what is your name", answer "Joe Smith". With this power, we can model objects in a way that far surpasses the ability to simply model the way an object moves. We can attach specific behaviour to objects.

2.2.3 ANIMUS

ANIMUS [6] is an animation system developed as an extension to ThingLab [2]. ThingLab is a constraint based system that allows for the creation of prototypical graphical objects. The major feature of ANIMUS is the method by which control is achieved, namely constraints.

Constraints are an interesting and very powerful means for controlling an animation. Basically, a **constraint** "constrains" one object (or attribute of an object) to be some

function of another, and vice versa. For example, a mathematical formula such as $4x = y$ is an example of a simple constraint. If we say x is 4 then y must be 16, or if y is 24, then x is 6.

ANIMUS extends constraints to handle time. The result is temporal constraints, which can be used to constrain various objects and their attributes to provide the necessary control over an animation. ANIMUS seems to be geared more for scientific use since constraints tend to lend themselves well to simulations. As well, it becomes quite complex when large numbers of objects are introduced into the system.

An interesting problem when dealing with constraints is what to do when all constraints cannot be met. In many instances, the system could simply complain, since the situation being depicted is not possible. In other cases, however, we may wish to resolve the conflicts through some known method, such as a type of best-fit approach (where we resolve each as much as possible). ANIMUS allows for the resolution of such conflicts in a user defined manner. However, the resolution methods are quite complex.

ANIMUS also provides a good graphical interface (inherited mainly from ThingLab). This interface uses Smalltalk concepts such as browsers to allow us to manipulate animations and the objects in them.

2.2.4 ARK

The Alternate Reality Kit (ARK) [19] is an animated environment for creating interactive simulations. It is currently under development at the Xerox Palo Alto Research

Center. ARK provides users with an extremely friendly and intuitive system for interacting with simulations. The user plays with alternate realities (thus the name) in order to achieve a better understanding of the reality being simulated. For example, if the simulation is of the solar system, the user could adjust the gravitational forces to create an alternate reality in which the planets would move in different orbits.

ARK can be thought of as a type of visual programming. It allows users to program the system through graphical means. ARK's main concern is to use as little "magic" as necessary to provide its required functionality. Magic, in ARK terms, is something which is not present in the literal sense, and provides added functionality. For example, a pop-up menu is a magical entity. We do not see or use them in the real world, but in ARK they provide us with some powerful functionality. Actions or entities which behave in a real world way are said to be "literal". An example of a literal entity would be ARK's hand, which is used for grasping objects.

ARK only makes use of magic where it provides enough enhanced functionality to warrant the extra learning time the user would require to make use of it. Otherwise, it makes use of literals. An interesting point is that there are some things which are not magic since they add no functionality, but are also not literal. An example of this would be using the mouse to move the hand. It is not literal but provides ease of manipulation rather than extra functionality. This type of entity or action is said to be an external factor. External factors are generally limitations imposed by input devices, computing power, and other things which are beyond the control of the system designer.

ARK provides us with extended research into the area of usability of a system. ARK aims itself at novice users, and appears to have successfully achieved its goals. Application level users can generally make use of only literal actions to interact with the simulation.

The area of ARK which is not explained in great detail is the environment provided for simulation builders. Since ARK is implemented in the Smalltalk environment, one may assume the simulation builder is provided with an extended Smalltalk environment. This environment must be mastered if the system needs to be extended.

Although it is not fair to compare ARK to general modelled animation systems since it is not one, we can make use of ARK's concepts in the design of the user interface to a general modelled animation system.

2.2.5 STELLA

The Structural Thinking Experiential Learning Laboratory with Animation (STELLA) [9] is a simulation system that employs a graphical user interface to allow its users to create and view simulations. It employs the concepts of general purpose simulation languages, allowing its users to graphically program a simulation visually.

STELLA provides a toolbox containing the basic simulation tools. The user selects tools and places them in the view area. By connecting various tools together, the user creates a simulation. The user may also specify various statistics that should be gathered, and graphs that should be produced from these statistics.

The animation STELLA provides is very simple, but does aid the user in understanding the run-time behaviour of the simulation. We can view the graphs being created as the simulation runs, or view entities such as stocks being filled and emptied.

STELLA provides useful input for our design of MACE since any modelled animation system is a type of simulation system. The major concept we draw from STELLA is the way in which it allows entities to be connected together. The user can combine various attributes of entities and put the computed value into another entity. STELLA provides a calculator style interface that allows the user to specify the expression to compute. As well as standard math operators and digits, the calculator supplies a list of the attributes of the entities we are drawing values from. The expression is specified by selecting from the operators, digits, and attributes.

2.2.6 Others

There are, of course, countless other animation systems available that provide a variety of features. The ones presented here are those that have influenced the design of MACE. I will very briefly go over some others that have been explored.

ANIM8 [3] is a powerful system implemented in FORTRAN. Its most important features are its use of numerical integration techniques and its concept of using block diagrams to program an animation. ANIM8 was designed to be used for educational purposes, and as such tries to stay as simple as possible while providing very accurate motion.

ASAS (Actor Scriptor Animation System) [18] is not as much a system as a programming language for graphics and animation. The user writes a "script" which specifies control of the animation. It is actor based, and employs three dimensional turtle concepts.

Small-Director [10] is a Smalltalk implementation of a subset of DIRECTOR. It provided some guidelines for an animation system implemented in Smalltalk.

2.3 The Advantages of Smalltalk

Smalltalk [5,8] is an extremely powerful object-oriented programming environment. This section will highlight some of the major advantages gained by using object-oriented programming in general, and Smalltalk in particular.

Modelled animation involves modelling objects. It is therefore quite natural to use an object-oriented language to implement this. An object-oriented system provides us with a means to implement the objects that we are modelling as true objects, with attributes and behaviour.

Smalltalk allows us to program objects to react to certain stimuli (messages) with set behaviours (methods). We specify behaviour for types (classes) of objects, and then create instances of these (for example, once we have modelled a sphere and defined it in the system, we can create as many spheres as we desire, and each will have its own attributes). We can base behaviour on the value of the object's attributes, thus allowing two objects of the same class to behave differently under a specific stimulus based on their

current state.

One of the most powerful concepts of Smalltalk is the class structure. This allows us to define subclasses of an existing class. The subclass automatically inherits all attributes and behaviour of the superclass (the class it is a subclass of). We can then add new behaviour and attributes, or modify existing ones. This class structure allows us to define a generic class which the user defines subclasses of. For example, MACE defines the general behaviour and attributes for all animated objects in the three dimensional object class. The user defines a subclass of this when modelling a new type of object, thus the basic attributes (position and a coordinate system) as well as the basic behaviour (movement, rotation, alter values, etc.) are automatically present for the new type of object.

The properties listed above that we gain automatically with Smalltalk could require a large amount of work in a procedural programming language. Object-oriented languages seem to be very well suited to modelled animation systems since they allow us to model and implement the objects in the animation in a manner consistent with the real world.

Smalltalk also allows us to create composite objects. The attributes of an object are objects themselves since everything in Smalltalk is an object. We therefore automatically inherit the ability to create complex composite objects.

Smalltalk uses bit-mapped graphics, providing the user with a powerful set of operations. The system allows us to add our extensions easily. As well, the inheritance structure used in this part of the system makes it quite easy to maintain and enhance.

The user interface Smalltalk presents is a well laid-out windowing system with various predefined concepts such as browsers and inspectors, as well as run-time debuggers. We can clone any of these, or modify them to achieve our own goals (for example, the animation browser discussed in the next chapter draws the majority of its windowing and scheduling directly from the class browser that is already defined in the Smalltalk/V environment). Pop-up menus are an integral part of the Smalltalk environment. The interface provided by MACE is an extension of the Smalltalk interface. In this way we save endless hours of development time, since the majority of the interface features we desire are already implemented, and those that are not can be implemented as modifications to already existing objects.

It would not be fair to list the many advantages of using Smalltalk without discussing the disadvantages as well. Most of these are minor complaints, such as the fact that common algorithms must be converted from the procedural (or functional) approach to the object-oriented approach. Another complaint is the small number of Smalltalk implementations in existence.

A major shortcoming of Smalltalk is that it does not support prototype instances (the ability to create specializations of instances). This shortcoming is overcome by the ACTRA [12] environment. In Smalltalk, we would have to define a new class in order to do the same. A big advantage of prototypes is that the user can create a sample without creating a class for it, manipulate it through editors, etc., and then use it as a base for creating more instances (for example the user could create a sphere, size it to the exact radius desired, and then create as many just like this as desired). The biggest advantage of prototypes, however, is that we can have instances with different behaviours without having to make

use of state variable to do so (i.e. we can supply methods for each specific instance as well as common ones for all related instances).

We adopted Smalltalk/V as the system to use to implement MACE. It won over the MacIntosh version of Smalltalk-80 because of its speed and because it runs on IBM PC compatibles, which are more flexible as far as adding displays and other peripherals than the MacIntosh. In the long run, this will give us real animation ability by allowing MACE to be interfaced (via Smalltalk) to high quality displays and video recorders for viewing and filming the animations (it is also hoped that the high quality displays can be used for controlling the animation as well as just viewing it).

2.4 Summary

Having discussed modelled animation, a number of animation and simulation systems, and the Smalltalk programming environment, we will now describe how these all relate to MACE, and the concepts MACE has drawn from the discussed systems.

MACE is a modelled animation system, and as such incorporates the major concepts of one. It models objects and allows for the manipulation of these objects in order to create an animation. MACE allows the user to perform the major tasks of modelled animation through a graphical interface but also allows the user to exploit the power of the Smalltalk programming environment in order to enhance and extend MACE.

MACE uses the actor based approach found in MIRANIM, DIRECTOR, ANIMUS, ARK, and ASAS. This approach appears to be dominating the modelled animation field, as it provides us with great power, not only by representing the real objects we model as objects (actors), but also by allowing for the interaction of the objects.

Laws, such as those found in MIRANIM, are a powerful control structure. They allow us to attach behaviour to objects in a easy manner. MACE employs the concept of laws, providing a more powerful version than MIRANIM. While MIRANIM laws work on animated variables, which can be attributes of some object, MACE **animation laws** work on the objects themselves. It makes more sense conceptually for a law to act on an object, and also provides more power. For example, the basic law of motion acts on the objects, not on some animated variable which happens to be the position of some object.

Like DIRECTOR, ASAS, and ARK, MACE allows the user to specify how actors (objects) react to messages (stimuli). By specifying **reactions**, users can create general behaviours for objects. As well, the reaction of one object may involve stimulating another, thus we can tie the behaviour of various objects together.

ANIMUS introduces the concept of temporal constraints as the basis for the control and synchronization of motion. While this is a powerful concept, it is difficult to implement and also difficult to use for complex control. MACE uses **alter values** and **connectors** to provide the same type of control structures temporal constraints provide. These are easy to implement, and provide interesting behaviours not found with constraints. The user is referred to Chapter 3 for more details on alter values and connectors.

The ARK concepts of magic and literalism have been an influence on the design of various MACE concepts and aspects of its interface. We confess that we tend to use too much magic, but it is very difficult not to when we wish to provide a powerful and flexible system. Wherever possible, the interactions between the user and MACE have been kept as literal as possible.

The concept of events is present in most animation systems, although in some they are used more for internal control than user specified control. In implementing Small-DIRECTOR, we found that events were used extensively for the internal control of the animation, although the user was not provided with a means of specifying events directly. In MACE, we deemed user control of animation events to be a requirement since they prove useful for fine-tuning an animation and implementing entities such as collision detectors.

Present in all animation systems is the concept of time. Most make use of clocks to control the timing aspect of the animation. MACE allows for multiple animation clocks, which can be set to different resolutions. These clocks control the time aspect of the animation for mutually exclusive sets of objects i.e. an object can be controlled by one clock only.

As with most animation systems, MACE supplies virtual cameras. These cameras are used for viewing the animation. MACE virtual cameras are treated as all other animated objects; i.e., they can be controlled in the same manner.

STELLA provides the concept of connecting entities together. From this concept, we developed **connectors** and **combiners**. Connectors allow for the behaviour of objects to be connected together by extracting a value from one object and passing it to another. Connectors are an often used MACE concept since their behaviour proves very useful. As well, they are simple enough to be easily understood. Combiners were developed for more complex interconnection of objects. This allowed us to keep connectors as simple as they are. Basically, a combiner is a function, taking an arbitrary number of inputs and producing an output. Connectors and combiners both use expressions, the former for extracting the value and the latter for computing the output. Presently, these expressions are input textually, although it would be very nice if they made use of a calculator-like entity as in STELLA.

ANIM8 shows the accuracy gained by employing numerical integration techniques for calculating time dependent functions. While we desired accuracy with MACE, we were not willing to pay the high cost of using these techniques. As well as the sheer complexity of implementation, we felt it would restrict other control methods since the interactions of these with the integrator might have proved impossible.

MACE employs the Smalltalk concepts of the class structure, browsers, inspectors, windowing, pop-up menus, and various other entities. Many of these were directly inherited from Smalltalk, while others provided ideas for MACE entities such as the animation browser. While the majority of these are user interface concepts, the class structure provides a very powerful implementation base for MACE. It allows us to define generic object types (classes), which the user may define subclasses of. These subclasses automatically inherit the behaviour and attributes of the generic class. Multiple instances of

a class can be created, with each having its own set of values for its attributes.

Smalltalk also provides for the implementation of reactions automatically since this is the way in which Smalltalk works. It is inherently message driven, and the user supplies behaviour for objects by specifying their reactions to stimuli (messages).

The next chapter describes the major concepts of MACE in more detail, discussing them in terms of a continuing example.

An Overview of the Modelled Animation Control Editor

Chapter 3

In this chapter we present an overview of the Modelled Animation Control Editor (MACE). The overall MACE environment will first be discussed, followed by the major concepts and objects that are used in MACE. Throughout, examples of the use of the concepts and objects will be given in the context of a continuing example.

The continuing example is concerned with three balls that manipulate each other's positions. This example will demonstrate the use of MACE's major concepts.

3.1 The MACE Environment

The MACE environment consists of a graphical user interface coupled with an extensible set of basic object types. The interface is used to create and manipulate the objects which can be either actual objects or control objects. **Actual objects** are those that are visible in the animation, while **control objects** are invisible meta-objects that are used strictly for control of the animation. Throughout the thesis, the terms **animated object**, **three dimensional object**, and **actual object** are used as synonyms. As well, **non-animated object**, **meta-object**, and **control object** are synonyms.

Figure 3.1 shows the MACE animation browser containing four control objects, a virtual camera, and two actual objects.

Figure 3.1 The Basic MACE Environment

The set of object types can be easily user expanded by either refining present object types or defining completely new ones. MACE automatically allows the user to access these new object types through the interface.

MACE is implemented as an extension to the Smalltalk/V programming environment. Therefore the user is provided with a complete object-oriented programming language which can be used to extend the functionality of MACE as needed.

By implementing MACE in this way, we provide a very flexible animation system. The basic system support and control methodologies are provided, with a small set of initial actual object types. It is envisioned that MACE will be tailored to individual users. Once a new object type is defined, it may be used from that point on. As well, the behaviour of existing object types may be extended or modified.

3.2 The Three Dimensional Cursor

As a part of the graphical user interface, the user is provided with a working view of the animation being created. We felt it a basic requirement for the user to be able to select and manipulate objects in this view in a visual manner. Having found no proper yet easy to implement technique for true three dimensional selection, we developed the concept of a three dimensional cursor.

The three dimensional cursor extends the normal cursor concepts from two to three dimensions. Instead of only being able to position the cursor in a two dimensional position, the user can position the cursor in a three dimensional position. Depth is indicated by the cursor changing size, i.e. moving the cursor farther away causes it to become smaller.

The user makes use of a mouse to control movement in the normal two dimensions, and uses the keyboard to control depth movement. Note that this is not a true three dimensional input device, but merely a three dimensional cursor controlled by a two dimensional input device and a keyboard (which is a one-dimensional input device).

When the user selects at a specific position, the object nearest to the cursor is selected. While this sometimes causes problems because what seems to be the nearest in the view may not be nearest in the three dimensional world, it is nevertheless more powerful than a traditional two dimensional user. For instance, it permits him to manipulate objects that are occluded by moving the mouse to the back of the occluding object. A wire-frame display of the objects permits the user to "see through them".

We investigated other methods that could be used for the same purpose as the three dimensional cursor. One of these involved using a robotic-like arm attached to a corner of the view window. The user would use the mouse and keyboard to move the arm and hand, grasping objects with it. The arm would be a three dimensional object, thus its appearance would indicate depth. The problem with this concept is that the time required to redraw the arm every time we moved it would be so long as to make it unusable. With a more powerful system that could allow us to move the arm in real time, this concept becomes more attractive.

Another method envisioned involved displaying a grid structure to show current position. This would resemble a cube, where the current x, y, and z positions would be indicated by highlighting that area of the grid. This idea was dismissed because we felt that the user would become confused by the number of lines, and generally would find it hard

to use. As well, we would have to redraw the grid lines each time we moved, causing more than slight delays.

3.3 Reactions

A reaction is an object's response to a certain stimulus. In MACE, the user can specify the exact reaction an object will have to specific stimuli. Basically, the user states the stimulus and the reaction, based on the object type. The reaction can use the state of the object to decide the exact behaviour. As well, the reaction of one object to a stimulus may cause another object to be stimulated.

Reactions in our example are used for system control. Standard predefined reactions are used for such things as positioning a ball (in which the ball reacts to the "position" stimulus), running the animation (where the animation clock stimulates each object with the "tick" stimulus on every clock tick), and displaying the objects (where the animation world reacts to the "display" stimulus). MACE provides many predefined reactions for all animated objects, including those used for movement (forward, backward, left, right, up, down) and those used for rotating an object (pitch, yaw, roll).

3.4 Animation Clocks

Animation is the act of creating the appearance of motion by varying object attributes over time. Time is the basis of all animation, whether it be real time or conceptual time. MACE makes use of animation clocks to control time in animations.

An animation clock is an object that ticks at a specified rate. When it ticks, all objects that use the clock are informed, and they take the actions defined for them for that point in time. The user can set the rate at which a clock ticks, and it is in terms of ticks per second (though the animation does not run in real time). Seconds are used because they are the defined units of time. If, however, we wished to use some other time unit, such as eons, it is merely a change in terminology.

The majority of animation systems support one clock that usually run at 24 or 30 frames per second. This clock is the sole controller of time in the animation. MACE allows the user to create as many clocks as is desired. However, each object can be controlled only by one clock. We are not creating multiple definitions of time, only breaking time up into different sized units. An easy way to view multiple animation clocks is as alarm clocks that sound at specific rates. The act of sounding the alarm is the clock ticking, which all objects it controls will hear.

When an object is defined, the clock that controls the object is selected. From this point on, whenever the selected clock ticks, the object will update itself for that point in time.

One of the main advantages of multiple clocks is that we can update objects at different rates. For example, if we wanted one ball's trajectory to be twice as accurate, we could attach it to a clock running at twice the speed of the default clock. The reason that it would be more accurate is that we assume constant values between ticks for simplicity even though the values should be continuous. Therefore the faster the clock runs, the more accurate the result.

Multiple clocks allow us more complex control strategies, and are another factor contributing to the flexibility of MACE. For most animations, multiple clocks will not be used, with the one clock being the default animation clock. The power multiple clocks give are reason enough to include them in MACE.

In our example, we will only make use of one clock, which MACE provides automatically as the default clock. The visual representation for a clock can be seen in figure 3.1.

3.5 Alter Values

Any type of object has a defined set of attributes. The value of these attributes determine such things as position of the object, velocity, mass, etc. Basically, these attributes are the state variables required by the object. Animation is produced by changing the value of some attributes of some objects over time. For example, to make a ball move we change its position slightly each frame.

In MACE, we provide great flexibility in the way in which attributes can be changed. Therefore, we must provide some type of mechanism to control the value of the attribute when it has been changed by more than one object during the same clock tick. For example, if we use physical laws to control motion, an object may be acted on by the law of motion, the laws of gravity, and various other laws. According to physics, we take the sum of the forces that are being exerted on the object. If we merely changed the value of the attribute after each law was applied, the next law that is applied would use the new value to compute the force it should exert, when it was the original value that should have

been used. For each law we must use the original values, and then sum the results with the original value to produce the new value for the attribute.

In order to handle these types of situations, MACE introduces the concept of "alter values". Basically, an **alter value** is another value for an attribute of an object. The original value of the attribute remains the same, while many alter values for this attribute may be specified. When the object is updated for that point in time, it resolves the list of alter values for an attribute to produce a new value.

The formula by which alter values are resolved can be specified for each type of object. The default resolution method does not assume that the values are physical forces, therefore it doesn't merely sum them. Instead it takes the average of the alter values. The reason for this is simplicity. Once resolved, all alter values are discarded since they have been used. Again, they are collected until the next time they are resolved.

Although alter values do solve many difficulties, they also create some. Generally, they are closer to reality than other methods since conceptually they allow multiple actions to occur on an object at exactly the same point in time. The main problem they create is that many complex events are solved by a method that assumes only one thing happens at a time, i.e. if they change the value of an attribute and then look at the value of the attribute, they expect it to be the new value. With alter values this is not the case since the original value remains until the object updates itself.

In our example, when the clock ticks it first sends the tick message to each connector because MACE assumes that connectors are at a higher priority than other objects. The

connector is designed to set the position of the ball it is connected to. This causes an alter value for the ball's position to be created. The actual position is updated when the ball itself receives a clock tick and resolves its alter values. In this way we do not have to worry about the order in which the connectors run for that clock run since they will extract the actual position from each ball, not the new positions. The only worry we have is to make sure all the objects (connectors and balls) use the same clock, otherwise we may get undesired behaviour. This is a risk involved with using multiple clocks since an animated object may be updated (sent the tick message) before a control object, causing strange behaviour. It is recommended that multiple clocks only be used where absolutely necessary, or where this type of behaviour is desired.

Our example reveals a side effect of alter values, that of delay. In our example, ball 1 sets ball 2's position, which sets ball 3's position. If ball 1 is moved, it will set ball 2's position, which takes one clock tick to take effect due to the fact that ball 2 must resolve its alter values before the new value is used. Then ball 3's position will be set, which again will take effect the next clock tick. We have therefore introduced a delay by which a change in ball 1 takes two time units to affect ball 3.

If constraints were used for the same situation, the effect would be immediate. The question that must be answered is whether or not this behaviour of alter values is acceptable. We are trying to constrain the objects such that a change in one is reflected in another. In the real world change is not immediate. However, the delay is often so small that it is not noticeable. For example, if we tied the balls together with string, and then moved ball 1, ball 3 would not move immediately, although we wouldn't be able to detect the delay. The delay would be dependent on the elasticity of the string, and would be twice

as long for ball 3 as for ball 2. Alter values have the same behaviour. We can set the length of the delay by adjusting the granularity of the clock used to control the objects. In our example, the delay is two clock ticks. If we set our clock to run at one tick per second, then the delay is quite large. If, however, we set our clock to run at 1,000,000 ticks per second (and only view the animation 24 times per second), the delay would not be visible.

From the above argument, we see that alter values actually provide us with more realistic actions than constraints. We are also given the power to control the actions since we can adjust the introduced delay.

Of course, an important consideration for the use of alter values is that they can be implemented easily. They also work in the presence of conflicting changes since the results are usually averaged. Implementing more accurate constraints in the presence of conflicts is difficult if not impossible. An example handled by alter values but not handled by constraints is the situation where ball 1 tries to stay 1 meter from ball 2, while ball 2 simultaneously tries to stay 2 meters from ball 1. With alter values, an oscillating motion results. If we initially set the balls 1 meter apart, ball 2 pushes ball 1 away the first clock tick. The next clock tick ball 1 pulls ball 2 towards it. The process then repeats itself forever. A constraint system would not allow this same situation because the constraints are insatisfiable. Basically, alter values provide approximate solutions to situations where resolving constraints would be difficult if not impossible.

3.6 Animation Laws

MACE provides the capability of attaching laws of behaviour to animated objects. For example, the basic law of motion computes a new position based on an object's previous position, velocity, and acceleration. The laws that have been attached to an object are executed each time the clock controlling the object ticks. If we wanted a ball to move according to the basic law of motion, we attach this law to the ball, and it will automatically be invoked on each clock tick.

Laws can also be associated with classes of objects. Any laws applying to a class also apply to all its subclasses. In order to provide greater flexibility, subclasses as well as instances have the ability to cancel out any laws inherited from superclasses. In a normal world, we would expect the law of motion to be applied to every object. We therefore wish to specify this with one gesture for the class, not one per object.

In our example, the basic law of motion applies to the balls. In order to watch the effects of alter values, however, the velocity of each ball is set to be zero. To experiment, we could set the velocities to some other values, causing the balls to be affected by basic motion as well as each other.

3.7 Connectors

Connectors are control objects that allows pairs of objects to be connected together. Basically, a connector extracts a value from the attributes of one object and sets an attribute of another object to this value. For example, if we wish to connect the balls such that

adjacent balls affect each other, we use a connector to extract the position of a ball, add some offset to it, and set the position of the adjacent ball.

Figure 3.2 shows how connectors are used such that ball 1 and ball 2 affect each other, and ball 2 and ball 3 affect each other. When the clock ticks, the connectors will set the respective ball's position.

Figure 3.2 Using Connectors

The user selects the object to extract attributes from, and is then prompted for the expression which computes the value. The user then selects the object to pass the value to,

and is given the list of the object's attributes from which the user may select the one he wishes to set.

A connector is restricted to transferring a value from one object's attribute to another. More complex operations can be achieved through the use of a combiner, which is described in the next section.

3.8 Combiners

Combiners are control objects that take a set of inputs and generate an output. This function can be thought of as combining the inputs to produce the output, hence the name. A combiner supports an arbitrary number of inputs, but only one output. A combiner, when told its controlling clock has ticked, resolves its input values and uses its expression to generate the output value.

A combiner does not do anything with the output value. It is up to the user to use a connector to extract this value and pass it to other objects. As well, a combiner does not extract values for its inputs. It expects these values to be set by other objects, such as connectors. A combiner does, however, support alter values for its inputs. This is so that more than one object can set an input. The combiner resolves the alter values for an input before it uses the input to generate the output.

To use a combiner, the user first creates one, and then gives the expression used to compute the output. Since this is the basic function of the combiner (producing an output), this is all that is required as far as defining the combiner is concerned. What is left is to connect inputs and connect the output to various objects. This is done by creating connectors (see previous section). When the user selects the object the connector is to pass the value to, and the combiner is selected, a list of the current inputs will be presented. The user may select one of these, or request that a new input be created. This will be the input that the connector will set. The user would then connect the output of the combiner to any other object in the animation that desires it.

Figure 3.3 Using A Combiner

Coupled with connectors, combiners provide the user with a powerful and flexible control mechanism for tying behaviours of objects together. In our example, we use a combiner to compute a position for the camera used to view the animation. Figure 3.3 shows how we can use connectors to extract values from the balls, combine them with a combiner, and then use another connector to take the computed value and set the camera's position to this value. We have now tied the position of the camera to the position of the three balls.

3.9 Animation Events

Behaviour can be defined as reactions to stimuli; i.e., taking specific actions when certain events occur. With reactions, we allow the user to specify in great detail how an object will react to a certain stimulus. We now need some means by which to cause various stimuli to occur.

For this purpose, MACE uses animation events. These events trigger various stimuli when they are caused to occur. We will call this stimulation the action of the event. General animation events are time driven, i.e. they are set to occur at a specific point in time. The user specifies the time at which to invoke the event, and the action to take upon invocation. When the time specified for the event is reached, the action will occur.

As well as the generic event, MACE refines this to provide conditional, repeating, and repeating conditional events. Conditional events are composed of a condition and an action. When the condition becomes true, the action occurs. Repeating events are like the generic event, but repeat as often as specified. Basically, the user specifies how often the

event should occur. Repeating conditional events are ones which combine the properties of conditional events and repeating events. They are exactly like conditional events with the added behaviour of repeating. A conditional event is finished once the condition becomes true; a repeating conditional event continues at the specified rate.

In our example, we use an event to turn the camera on. This event merely triggers the "on" stimulus of the camera at the beginning of the animation. Figure 3.4 shows the visual representation of this event.

Figure 3.4 Using an Event

3.10 Animated Objects

All objects other than control objects are animated objects. Currently, MACE supports three types of animated objects: virtual cameras, boxes, and spheres. The animated object class is used to denote all animated objects, and as well defines some generic behaviours and attributes for animated objects.

MACE assumes all animated objects have a position in three dimensional space, as well as their own coordinate system. Having each object have its own coordinate system simplifies control of the animation. For example, it makes three dimensional rotations easy to implement.

Additionally, we can express behaviour in terms of the object as opposed to in terms of some standard system. For example, if we wish a virtual camera to move forward, we can say "move forward" as opposed to figuring out which direction forward is (based on the standard system) and saying "move in this direction". Using this, we could tell an object to move forward at some speed. If some other force causes it to turn or rotate, as far as the object is concerned it will still be moving forward.

Also, we can easily describe objects in relation to other objects. For example, if a ball is in a box, we can define the ball's position relative to the box. Now, whenever the box moves, the ball automatically moves with it since it has been defined relative to it. This is most useful when trying to model complex objects that are made up of simpler pieces.

The user creates new types of animated objects by refining current ones or the generic "animated object". This generic object type provides basic functionality for the new object; e.g., movement, alter values; position, and its coordinate system.

In our example, we make use of three spheres, as well as a virtual camera. Virtual cameras are special animated objects, and will be covered in the next section.

3.11 Virtual Cameras

Virtual cameras are objects used to view the animation. They produce a two dimensional picture of the three dimensional animation world. The cameras are animated objects, not control objects. As such, they behave like any other animated object. The reason for this is that they are not used for control and provide a functionality that is performed in the real world by real cameras. Since we can move real cameras around; and turn them on and off, etc., we can do the same with virtual cameras.

The user may create as many virtual cameras as desired. They can be controlled with the control objects, using events to turn them on and off. If more than one camera is turned on, each still produces its image on the view, with each image being super-imposed on the others.

By having virtual cameras behave like other animated objects, we provide a very useful tool for animations. In fact, quite a few animations are created by merely moving the virtual camera, not the objects themselves. For example, if the animation world was a city, the objects in the city (buildings, streets, etc.) would not move, but our camera would.

In our example, we use only one camera, as can be seen in figure 3.3 and 3.4. This camera will be turned on by the event shown in figure 3.4 and its movement controlled by the position of the three balls by way of the connectors and combiners shown in figure 3.3.

3.12 Animation Worlds

Having defined the various concepts MACE uses to allow the user to create, manipulate, and control animations, it is important that we have some mechanism for bringing the various pieces of an animation together into a whole. This is accomplished by the animation world.

An **animation world** is a collection of objects which make up an animation. These objects can be animation clocks, animated objects, virtual cameras, connectors, combiners, and various events. As well, the list can include any new types of objects the user defines. By default, every animation world contains a clock. At the moment, there are no other default objects. It is up to the user to make use of the animation browser (described in the next section) to create the objects that make up the animation. Once an animation has been defined, the animation world allows the user to save the work as a whole. In this way, we can have several animations in the system at the same time, with each being an animation world.

Animation worlds are also required so that there is a defined environment for an animation. It is currently not possible to cross or mix animation worlds, although the idea is appealing and would most certainly be an interesting avenue of research. In our

example, we create the entire example in the context of a world. The name of this world is "Example", as can be seen in the title bar of the animation browser shown in all the figures in this chapter.

3.13 Animation Browsers

Animation browsers are the vehicle through which a user creates, views and manipulates animations. Specifically, an animation browser allows the user to "browse" an animation world, where browsing involves manipulating as well as viewing.

The browser provides a graphical user interface that is made up of a command area and the view. The **command area** contains all commands necessary to create and manipulate objects and the animation. The **view** is a window onto the three dimensional animation world. Referring to figure 3.4, the command area contains the various MACE commands, the view contains the actual objects and meta-objects currently in the world, and the title bar contains the name of the world we are browsing.

A normal cursor is used in the command area and title bar but when the cursor enters the view, it becomes a three dimensional cursor. We can think of this as the cursor "entering" the animation world. It may then be used for manipulating objects in the world.

Distinct pop-up menus are provided depending on whether the cursor is in the view, the command area, or the title area of the browser. These menus allow various area specific functions to be performed. Also, the pop-up menu for the view is dependent on the type of object currently selected. For example, if we selected a connector with the cursor and mouse and then popped up the menu, it would contain commands relevant to a connector. The default pop-up menu allows the basic functionality that most objects support.

While some activities involve the use of the Smalltalk built-in browsers (such as defining a new class of object), most control of an animation is achieved through the animation browser. It is hoped that further research will allow us to create our own specialized browsers for these activities as well. One that has already been created is the animation law browser, which can be used for creating animation laws. This browser lets us view and create animation laws. It is basically the same as a Smalltalk class browser, so it will not be discussed further.

The example animation is now complete, as depicted in figure 3.4. We can now run this animation to view its behaviour. By modifying original positions, velocities, and expressions of the connectors and combiners, we can learn more about the underlying control structures of MACE.

This completes the overview of MACE. The next chapter will step through a different example animation created with MACE, concentrating on the actual interactions required to accomplish the task.

An Example Animation Created With MACE

Chapter 4

In this chapter we present an example animation created with MACE. The entire process of creating the animation will be discussed, including defining the model, defining the control, and finally creating the animation.

The example animation we will create will be a box filled with balls, with the balls moving inside the box and bouncing off each other and the walls of the box. A more complex example could be given here instead, but it is felt that the more complex the animation presented, the more difficult it will be to understand the interworkings of MACE. We have therefore used a minimal example that will exercise MACE's functionality. This should hopefully give a better understanding of MACE.

4.1 Defining the Model

The first step in creating a modelled animation is to define the model. This involves defining each piece of the whole, and then combining the components.

For our example we require a box, some balls, and some virtual cameras to film the animation with. Virtual cameras are a standard MACE object, and therefore need no definition. Boxes and spheres must be added to MACE if they have not already been defined (in fact they have, but we will pretend they have not to show how this can be

achieved).

We begin with a ball. MACE provides the generic animated object which we can refine to model a ball. The generic object provides us with position and the common movement functions. What we need to define is the new variables a ball requires and how the ball can display itself. To begin with, we must decide how to model the ball. A position and radius is sufficient. To display itself, it merely draws a circle using the three dimensional pen that the virtual camera provides. We now have a model for a ball. While its display method is simplistic, its representation is completely accurate for a true sphere.

In order to add the model of the ball to MACE, we make use of the Smalltalk user interface to add "sphere" (ball) as a subclass of "three dimensional object" (animated object). We then specify that it has a variable in which it maintains its radius. We now define the method by which it displays itself. This method takes as a parameter a virtual camera. The method uses the camera's pen to draw a circle. As well, we add a method which will create a new sphere with a default position, say (0,0,0), and a default radius, say 25.

Let us now decide on a model for a box. One way to model a box is as a collection of points which are connected by lines. If we look at this collection of points, we note that we need only know two values to represent a box, its position (either its center or a corner) and its width, height, and depth. From this we can draw a wireframe outline of the box.

We again make use of the Smalltalk interface to add "box" as a subclass of "three dimensional object", giving it one instance variable, its three dimensional extent (we already have position since it is a three dimensional object). We then provide the method to

display a box, which takes a virtual camera as input. The method uses the box's position and extent to draw a wireframe outline of itself. As well, we add a method that will create a new box with default position (0,0,0) and extent (50,50,50).

We have now added the two required object types to MACE and we can now create instances of these to use in any animation we desire. We now move on to creating the necessary boxes, balls, and cameras.

We first need an animation world to work in. From the Smalltalk interface, we pop up the system menu and select the "browse animation" command. Up pops a list of all known animation worlds in the system. Since this is our first, the list is empty except for one called "new world". Selecting this causes a prompt to appear asking us to enter a name for our new world. Let us call it "ExampleWorld".

After entering the name, a window appears with the title "ExampleWorld | Animation Browser", as can be seen in figure 4.1. Inside this window are two main areas, a command area and a view area. Within the command area are a list of available MACE commands to select from. The view is currently empty, except for the default animation clock, which appears in the upper left corner of the view.

Figure 4.1 The Initial Animation Browser

If we experiment with moving the cursor around, we notice that if it enters the view area, it takes a drastic change. What is happening is that the normal two dimensional cursor is entering the three dimensional animation world, and as such, it becomes three dimensional. If we hit the "<" or ">" keys while it is in the view area, it moves towards us or away from us respectively. This can be visually seen since it increases or decreases in size.

A pop-up menu is associated with each of the three areas of the window. In the title bar, the menu contains commands used for saving the animation world being browsed,

closing the browser, etc. The command area's menu is currently unused. In the view, the pop up menu is dependent on the current object selected. If the selected object has not specified its own menu or no object is currently selected, the default menu that allows for standard operations is used.

While we could discuss MACE's extensive command set in more intricate detail, it doesn't contribute to understanding the thesis issues. A brief description is provided in the appendix.

If we look at the command area, we can select the command "createObject" to force MACE to explore the system looking for all types of animated objects. It pops up a list of these, which includes boxes, spheres, and virtual cameras. To add a box we select "box" from the list. A box with the default attributes appears in the centre of the view. This box is highlighted with thicker lines to show that it is the current object. If we pop up the menu, we will be able to select the "position" command. This allows us to move the box to a new position by moving the cursor to the desired position and selecting (depressing the mouse button). Note that if we move the box away from us it gets smaller because we have placed it further away from the camera being used to view the animation world. After positioning the box slightly below the center of the view, we have the situation depicted in figure 4.2.

Figure 4.2 The Box

We repeat the same sequence used for the box, but select "sphere" after issuing the **createObject** command. A new sphere appears in the center of the view with the default parameters. If we pop up the menu and select the **relative to** command, we can specify which object is to be the sphere's origin. After issuing the command, we move the cursor to the box and select it. This causes the sphere to be defined relative to the box. We repeat this procedure twice to create a total of three spheres.

We now have a box with three balls defined relative to it. We then position the balls inside the box, resizing them if necessary. Resizing is an optional command for an animated object. We can add specialized resize methods to spheres and boxes using the Smalltalk browser. For spheres, it merely sets the radius to the distance between the new point and the center of the sphere. For boxes, the extent is set to be the difference between the new point and the box's position.

To manipulate attributes of objects, the user may inspect an animated object by first selecting it and then using the pop-up menu to issue the "inspect" command. This will create an inspector for the selected object. For more details on inspectors, see [5,8].

It would be desirable if the user could specify all values by pointing or other graphical means. However, at times specifying values via textual input is simpler. The ideal system should provide a combination of graphical and textual input for values. MACE currently relies too heavily on textual input.

We have now added the required animated objects to the animation. The current state of our animation world can be seen in figure 4.3. It is now time to define the control.

Figure 4.3 The Box and Balls

4.2 Defining the Control

Now that the basic objects have been added to the animation, we must define the control objects to produce the desired results. We will step through each aspect of control individually.

First, we need clocking. By default, all animated objects are run by the default animation clock. We must decide on a rate for this clock to tick. For simplicity we will leave it at the default rate of 24 ticks per second.

Alter values are also required. Fortunately, MACE automatically provides these so they are hidden from the user.

Let us now look at the animation laws we need. The main law that could be used is the basic law of motion. This happens to be already defined in MACE, but if it wasn't we would select the "browse laws" command, which causes an animation law browser to be created. This browser allows the user to view and alter all existing laws as well as add new ones. Figure 4.4 shows an animation law browser being used to create a new law.

Figure 4.4 An Animation Law Browser

There are various ways we could attach this law to the balls and boxes. One way would be to attach it to each object individually. Another would be to attach it to the object class, i.e. attach it to all boxes and all spheres. Still yet an easier method would be to attach it to all three dimensional objects, which means it would automatically apply to all boxes, spheres, and even virtual cameras. It so happens that this law is already attached to all three dimensional objects.

All three dimensional objects have a velocity. Since we have already attached the basic law of motion to all three dimensional objects, all we need to do is specify the velocity for each of the objects (it is (0,0,0) by default).

We will use various events to help control the animation. These will be used to detect collisions between the balls, collisions between the balls and the box, and also to turn cameras on and off.

The first event used will be a repeating animation event which will detect and take action upon the collision of balls. We could implement this as a conditional repeating event, but it is easier to implement it as a repeating event. We select "createMetaObj" from the command area which causes MACE to explore the system for all meta-object types. We are presented with a list of these that includes "repeating event". Selecting this causes the representation of an event to appear in the center of the view. Through the use of the pop-up menu, we issue the command that allows us to specify the action to invoke each time the event is signalled. This action will call a method which detects collision between spheres and takes appropriate action. This collision detector is already written, but if it weren't we would add it as a method for an animation event. This event checks for

collision between all pairs of spheres it applies to. We must specify which spheres the event applies to, so we select the "setAppliesTo" command from the pop-up menu for the event. We then select each of the spheres. The corrective action we decided to take on detecting a collision for this animation is to have the two balls exchange velocities (regardless of the angle they collide at). More complex actions could be specified if the user desired. Remember that once an entity such as an event action, a type of object, or a law is added to MACE, it is available for use by any animation. Users are therefore encouraged to add generic entities to the system to maximize the benefit of their work.

Figure 4.5 shows the new event we have created.

Figure 4.5 The Ball Collision Event

The second event will be used to detect collisions between a ball and the box it is defined relative to. The corrective action is to negate the value of the velocity for the axis that the side of the box being hit is parallel to. Since the balls are relative to the box, the box's sides are parallel to the major axes, therefore if the ball hits the side where x is constant, we negate the x value of the ball's velocity, thus causing it to rebound perfectly off the wall. The event detects and corrects collisions between a box and a group of balls. We must set the box and balls this event applies to, using the "setAppliesTo" command as before. Figure 4.6 shows our world after we have added this event.

Figure 4.6 The Box-Balls Collision Event

The final animation event will be used for control of virtual cameras which we will now create. In order to view the animation, we need at least one virtual camera. In our earlier work, we had been using the working camera, a special virtual camera used to view the animation world when we are manipulating it. We now need a camera as part of the animation world to view the animation with.

We select the "createObject" command, and select "virtual camera" when the list of animated objects is presented. This will cause a virtual camera to be created and placed in the center of the working view. We can then position the camera into a position so that it can view the box and balls.

Once in the desired position, we can see what the view from the camera is by selecting the "show view" command from the pop up menu for the virtual camera. This will cause the view area to be filled with the view of the animation world as seen from the virtual camera. We can then pitch, roll, and yaw the camera so that it is in a desirable orientation (pitch, roll, and yaw rotate the camera in various ways). In order for the camera to be used during the run of the animation, it must be turned on. We can do this by creating an event which turns it on.

While one camera is sufficient, we can do more with two cameras. We create another virtual camera the same way as the first. We position this camera differently from the first so that their views are different. Let us put camera 1 above the box and camera 2 off to the right of the box. We also change the event that turned camera 1 on and off so that it also controls camera 2, i.e. when it turns camera 1 off, it turns camera 2 on. Let us make

the event a repeating event that repeats every 10 seconds, turning off the current camera and turning on the other. Figure 4.7 shows our world at this stage.

Figure 4.7 The Two Virtual Cameras

Since our box is moving through space, it would be desirable to have one of our cameras follow it. This can be accomplished by connecting the position of the box to the position of the camera. Select "createMetaObject" from the command area and select "connector" from the list of meta-objects presented. Position the connector so that it is below the box. Pop up the view menu and select the "set from" command. This allows us to select which object the connector should extract from. Use the cursor to select the box.

Pop up the menu again, and select the "set to" command. This allows us to select the object and attribute the connector should set. Use the cursor to select the camera, which causes a list of the camera's attributes to appear. Select "origin" from this list.

All that is left for our connector to work is to specify the expression that extracts the value from the box. Since we want the camera to follow the box, but at an offset from its origin, we make the expression "origin + (100,50,100)". Pop up the menu, select the "expression" command, and enter the above expression. Our connector is now complete, and the camera's position will now follow the box's position at the specified offset.

Our world is now almost complete. We have the box, the balls, and the cameras, one of which follows the box. What does the other camera do? Let us make it follow the balls. The only problem is that there are three balls, so we cannot just connect the balls to the camera. We will need to use a combiner.

Select the "createMetaObj" command, and then select "combiner" from the list. Position the combiner in the upper right corner of the view. We will use this combiner to combine the three positions of the balls. We therefore need to connect the balls to the combiner.

Create a connector, set the "from" object to be one of the balls and the "to" object and attribute to be a new input on the combiner. Set the expression to be "self.worldCoordinateFor: origin", which gives us the ball's origin in world coordinates (we need this since the ball is defined relative to the box). Repeat this process for the other two balls.

The combiner now has three inputs. The value computed can be made the average of these three inputs. Therefore set the combiner's expression to be $(\text{input1} + \text{input2} + \text{input3}) / 3$. This combiner will now compute this value every clock tick. We now must connect the value of the combiner to the origin of the camera. Create a connector to do this (its expression will be "value"). Our control is now complete and it is time to create the actual animation. Figure 4.8 shows our world as it currently exists.

Figure 4.8 The Complete Control Structure

4.3 Creating the Animation

We have now defined the model and the control necessary for our animation. We must now create the actual animation by running what we have already defined. Although

MACE does not currently directly support a recorder, it does allow for frame by frame display of the animation, and could easily be adapted to control a recorder.

If we now select "run for frames" from the command area, we will be prompted for the number of frames to run for. Enter 24 (which is one second, since the default clock's speed is 1/24 second). We will get 24 frames displayed, with each being the animation world as viewed 1/24 of a second later than the previous frame.

Another option would be to tell the animation to run for a number of seconds. We can also step through frame by frame. One thing MACE does not currently support is back-stepping. This would be extremely complex, if not impossible, since many control actions in MACE are not solely dependent on time. We could possibly achieve it by keeping a history for each object, but this would prove very expensive (although it would be very nice, as we could do things to an object like say "go back to where you were two seconds ago").

In order for MACE to allow for an animation to be restarted, it automatically creates events which set objects to their initial position (these events are transparent to the user). As well, if we stop the animation part-way through and reposition an object, this is stored as an event for that point in time. Anytime that we rerun the animation, the object will be set to that position at that point in time.

As with most animation systems, using MACE involves fine-tuning control through trial and error. This, however, is dependent on what is desired from the animation. If we are trying to create a certain outcome, we usually have to either model very precisely or

fine-tune. If, however, we merely want to see what will happen, there is usually no fine-tuning (except possibly for refining the model further). We desire the animation system to run with no known outcome (often so that we will learn what the outcome is). This is one of the most useful features of modelled animation systems and is what gives them great power over the simplistic key frame animation systems.

We now have our complete animation, and can run it as long as we wish. The balls will forever bounce around inside of the box which can be given a velocity so that it moves as well. We could extend the animation in various ways. Figure 4.9 shows one frame of the animation, as seen from one of our virtual cameras.

An easy way to expand this animation would be to add more cameras, switching back and forth between them. We could also add more balls, or change the action that occurs when two balls collide (for example, we could have one consume the other).

Figure 4.9 A Frame from the Animation

Another interesting extension would be a recursive world based on our current world, where our box is inside a larger ball, which is with other similar balls inside of another yet larger box. This box would then be inside of a yet larger ball, which is inside another box with other similar balls, etc. Of course we would have to stop the recursion at some maximum and minimum level, but we could provide enough levels to make a very nice animation by moving up and down levels of the recursion.

Overall, this example has shown the basic operation of MACE. It is hoped that it has shown the power and flexibility of MACE to a certain degree, although the full power

and flexibility of MACE is too fertile a ground to cover in this work. We hope that the readers will take it upon themselves to discover the underlying power behind the concepts brought forth in MACE.

The Implementation of MACE

Chapter 5

This chapter describes the implementation of the Modelled Animation Control Editor (MACE). MACE is implemented in the Smalltalk programming language, and therefore a very brief introduction to Smalltalk is given. Further sections deal with implementation issues concerning the MACE concepts discussed in previous chapters.

5.1 Smalltalk

Smalltalk is an object-oriented programming language, and as such incorporates the idea of objects as the mainstream of its programming model. For those unfamiliar with Smalltalk or object-oriented programming in general, refer to [5,8].

Smalltalk incorporates a very friendly graphical user interface as one of its mainstays. Without its graphical interface Smalltalk would still be a powerful programming *language*, but would not be the powerful, flexible, and organized programming *environment* that it is.

MACE draws much of its interface directly from the Smalltalk interface. In fact, parts of MACE's functionality are achieved by using the Smalltalk interface directly, since MACE is merely an extension of the Smalltalk environment. It is not necessary for users of MACE to know the Smalltalk language if they restrict themselves to predefined classes of objects and behaviours.

5.2 The MACE Environment

As discussed in Chapter 3, the MACE environment is an extension of the Smalltalk environment. The extra features MACE adds are its various concepts, such as animation browsers, law browsers, and three dimensional cursors. Further sections of this chapter deal explicitly with the implementation of these concepts.

5.3 The Three Dimensional Cursor

The three dimensional cursor is implemented in the main processing loop of the view window. This loop is the one that checks for mouse movement and selection, as well as keyboard input.

The three dimensional cursor is actually a collection of bitmap representations. Each successive bitmap is a representation of the cursor at a greater depth. These bitmaps are precomputed. Currently, we use 16 different size bitmaps to show depth.

An earlier implementation actually shrunk or magnified the cursors bitmap according to depth and then displayed it. While this showed depth more accurately, the time involved in shrinking or magnifying caused great delays when moving the cursor.

Basically, we make the normal cursor invisible and display one of our own bitmaps instead. The bitmap displayed is selected from a calculation based on the current depth of the cursor. We project the cursor's three dimensional position and display the appropriate

bitmap at the projected position.

The main reason for implementing the cursor as a bitmap is speed. If our machine was fast enough, a better three dimensional cursor would be an actual three dimensional object. The problem is that this takes too long for it to be usable. Every time we move the cursor, we would have to wait while it was projected to two dimensions and redrawn. As it is, we can notice flicker when we move the cursor, and it is even more noticable when we adjust its depth.

5.4 Animation Clocks

In discussing the implementation of animation clocks, we inherently describe the underlying strategy MACE uses for controlling the running of animations. The reason for this is that clocks are the main drivers behind an animation since all animations are based on time.

Most animation systems use one clock. This clock is commonly used for the run-time control of the animation. The usual method is to have an event queue based on time. Each event has a time stamp, and we process all events stamped to run at the time the clock says it is. When these have all been run, we can display the frame and record it. We then increment the clock and repeat the process.

Conceptually, MACE has clocks as the main drivers. Since MACE supports multiple clocks, it is not enough to process events based on the current time of one clock. Instead, clocks are implemented through events. As previously stated, the event queue is based on time stamps. The event queue manager takes the next event off of the queue, sets the

system time to the time stamp of the event, and runs the event. This process is repeated until we empty the queue or reach a specified time.

A clock is implemented by inserting a repeating event into the queue with the action to make the clock tick. For example, if a clock ticks 24 times a second, it would insert a repeating event into the queue to be run at every $1/24$ of a second. This event would cause the clock to tick, which sends the tick message to all objects using this clock for control. Once this is accomplished, the event is reinserted into the queue for the next time the clock should tick (note this is automatic since it is a repeating event).

In this way, MACE is really an event driven system, not a time driven system. Conceptually, however, it can still be said to be time driven, for without time, nothing would happen.

In order to achieve the display of the frames at the desired rate, MACE makes use of a display clock. Each time the display clock ticks, all objects in the animation world are requested to display themselves. Once they have, the frame is displayed on the screen and can be recorded. We therefore set the display rate (or number of frames per second) by adjusting the speed of the display clock.

5.5 Alter Values

The implementation of alter values is quite simple. Alter values for an object are stored in a dictionary, where a key is the name of an attribute of the object, and the entry for the key is the collection of alter values for that attribute. The resolution of alter values is

accomplished by a method which operates on this dictionary, deleting the entries as they are resolved. The default resolution method simply averages the alter values to produce the new value.

5.6 Animation Laws

Animation laws are simply class methods for the class "Animation Laws". Creating new ones through the animation law browser is the same as writing a method for the class. The complexity in animation laws lies in the way they are attached to objects.

As shown in the previous chapter, we can attach a law to a specific object, or to a class of objects. A law that applies to a class is inherited by all of its subclasses (that is, it applies to the subclasses as well). In order to provide more power, MACE also allows us to disinherit an inherited law.

Each object stores a collection of laws that have been specifically attached to it, as well as a list of inherited laws that should be disherited. Each class of object also has the same collections. When an object updates itself for some point in time, it first must run all laws that apply to it. The algorithm for doing this involves traversing the class structure of the object. The algorithm is as follows:

- perform all laws attached specifically to the object
- create a collection of non-laws
- add all laws stated that this object should disinherit to the collection of non-laws
- move up to the object's class

- Repeat
 - step through the laws attached to this class, performing the law only if it is not in the collection of non-laws
 - add all laws that should be disinherited by this class to the collection of non-laws
 - move up to the superclass of this object
- Until we are at the top of the class structure

5.7 Virtual Cameras

Virtual cameras are implemented as three dimensional objects, since they share the same basic properties of all three dimensional objects. The underlying algorithms for virtual cameras has been taken from [1].

A virtual camera provides a three dimensional pen for objects to draw themselves with. This pen automatically clips the drawing to the view, performing the clipping in three dimensions as per the algorithm of [13]. For complex graphics, the objects can look at the various attributes of the virtual camera, such as position, orientation, focal length, etc. in order to draw themselves.

MACE leaves the actual job of displaying an object up to the object itself. We feel this makes sense conceptually since the only thing that should have to know what an object looks like is the object itself. This also allows us to use different underlying representations for objects since the display of the object is not accomplished by a predefined method (for example, some systems assume all objects are represented as polygon meshes, thereby automatically restricting how we represent objects). It is up to the

object to display itself in the camera's view. If we did wish to implement complex display algorithms for various object representations, we would define a general one for animated objects.

5.8 Animation Browsers

Animation browsers are implemented as a subclass of the Smalltalk/V class browser. They inherit their basic scheduling from the class browser, adding their own functionality as required. The main changes that were required were to add the special view window and the special command area window.

Conclusion

Chapter 6

We have described the design and implementation of MACE, a modelled animation control editor. The visual interface MACE provides has been discussed, and the concept of alter values has been introduced. As well, we have described how MACE uses the Smalltalk programming environment.

We have shown that the visual interface MACE implements provides for basic manipulation of both the animated and control objects. The set of control objects proposed allows for the creation of complex control structures, while not being complex themselves. MACE is by no means a complete system, but is a preliminary system that provides guidelines and suggestions for future evolution. Many interface details have not been implemented but they do not detract from the contribution of this thesis since the concepts involved are not new.

The three dimensional cursor developed allows for selection and manipulation of objects in three dimensional space. Although the current implementation is not perfect, it does demonstrate the basic concept. The method of displaying different bitmaps dependent on depth works well. However, if the computing power necessary to display a moving three dimensional object in real time was available, we would implement the cursor as a true three dimensional object. The same applies to the visual representations of the control objects.

We have shown that alter values provide an interesting and powerful method for the underlying synchronization of motion. These can be used in conjunction with connectors to provide control strategies similar to those achieved by using temporal constraints, although the resulting behaviour will be slightly different due to the delays introduced by alter values. These delays are not, however, a detraction of alter values. They are, in fact, an attraction since we can control the amount of delay. With constraints all effects are immediate.

Our discussion in Chapter 2 of the advantages and disadvantages of using Smalltalk and the description of implementations in Chapter 5 show that we have obtained a great deal of required functionality for free by making use of the Smalltalk programming environment. MACE has also provided further support for the actor-based approach to modelled animation systems.

It is hoped that a MACE user will explore the system in depth and be able to tap what we feel is a very powerful system. MACE grows with use. The more it is used, the more it offers. Currently, only the basics are offered but these should provide a solid starting point for creating and controlling animations.

It is also hoped that MACE will inspire the development of additional concepts for controlling animations and that the many animation systems currently available (or being developed) will benefit from the concepts introduced in MACE. MACE's focus is in the control of an animation, not on issues regarding graphics and real time. While we admit that the need for realistic images as well as near real time for developing animations is a critical part of an animation system, we feel that the control used in the animation itself and

in the development of the animation are just as important.

6.1 Future Work

There is a great deal more work that could be done on MACE to enhance it. The majority of the work lies in completing the implementation of existing concepts. Other work involves pure research.

Obviously, MACE could benefit by having traditional graphic features added. Hidden-line and hidden-surface algorithms, as well as surface displaying algorithms would prove useful in allowing MACE to produce realistic images.

Another feature that would enhance MACE would be to have it use a three dimensional graphics editor to allow users to create new objects. These editors are starting to appear on the market (see [4,9]). They allow a user to create three dimensional objects interactively without programming. Another interesting thing to do would be to integrate ThingLab's approach to object construction. Unlike Easy-3D, ThingLab permits the definition of object prototypes (not just individual objects).

Yet another area in which MACE could be enhanced is the method by which time dependent values are calculated from a continuous function. For example, if we have an object accelerating, we calculate how much it has moved between clock ticks by assuming that speed was constant over that time span. The speed is then updated to the new value based on the acceleration. The proper way to implement this would be with an integrater

which systems such as Anim8 claim as one of their main sources of power. While many systems make the same concessions that MACE currently does, the accuracy gained by using an integrater would be worth the complexity if we wished to use MACE to model worlds very accurately. For many animations, however, we can afford the loss of accuracy since it is not visually noticeable.

MACE's interface could also be enhanced to further eliminate non-graphical interactions. Things such as sliders and scales for inputting values and moving objects would be useful, as well as better ways of highlighting selected objects. As well, MACE could use specialized browsers for adding new types of objects and defining behaviour (methods) for the objects.

Apart from the above mentioned enhancements, which are mainly implementation details and not theoretical issues, MACE does open some further avenues of research. One very interesting area has been mentioned in a previous chapter, that of combining and crossing animation worlds. This concept could possibly add some very interesting effects to MACE. Like the recursive animation suggested in chapter 4, we could have recursive worlds, where every world is a part of a bigger world, just the way in which our world is a part of our solar system, which is a part of our galaxy, which is a part of our universe. We could have objects leave one world and enter a different world where all physical laws may be drastically different. Note that MACE as it exists could support this kind of animation base but it would not be as clean an implementation as could be achieved with the ability to mix worlds. In the current system a planet would be an object, not an animation world.

Research into adding the ability to use constraints for controlling special cases could prove quite interesting as well as beneficial to MACE. This thesis has shown how alter values and constraints differ, and also pointed out some deficiencies of constraints. We cannot, however, disregard the power achieved through constraints. The ability to use constraints and alter values in the same animation could prove very useful.

An area that has yet to be explored are the problems created by the fact that animated objects move throughout the run of the animation but control objects do not. If we attach some control object to an animated object, and then run the animation for a while, the animated object may have moved quite far away from the control object. It therefore becomes difficult to view the control objects being used to control an animated object. It would be easy to have the control objects follow the objects they control but this becomes difficult when one control object controls many animated objects. Which should it follow?

Another area that has not been explored is the concept of general purpose recorders. These recorders would record various attributes of an animation. For example, a virtual camera is a special type of recorder that displays a view of the animation every clock tick. Another type of recorder could be one that gathers information to plot graphs. Yet another could record only the position of each object to show trajectories.

References

1. Abelson, H. and diSessa, A., Turtle Geometry: The Computer as a Medium for Exploring Mathematics, The MIT Press, 1981.
2. Borning, A.H., "ThingLab - A Constraint-Oriented Simulation Laboratory", Ph.D. Thesis, Stanford University, 1979.
3. Cavers, J.K., and Woodside, C.M., "A Block Diagram Computer Language for Educational Animation", IEEE Transactions on Education, Vol. 19, No. 4, November 1976, pp. 133-138.
4. Challenger Software Corporation, Mac3D, 1985.
5. Digitalk Inc., Smalltalk/V Tutorial and Programming Handbook, 1986.
6. Duisberg, R., "Constraint Based Animation: The Implementation of Temporal Constraints in the ANIMUS System", Ph.D. Thesis, University of Washington, 1986.
7. Enabling Technologies, Easy-3D, 1985.
8. Goldberg, A. and Robson, D., Smalltalk-80: The Language and its Implementation, Addison-Wesley Publishing Company, 1983.
9. High Performance Systems Inc., A Users's Guide to STELLA, 1985.
10. Huizen, R.L., "Small-DIRECTOR", MCS Course Requirement, 1986.
11. Kahn, K.M., "DIRECTOR Guide", MIT AI Memo No. 482B, 1979.

12. Lalonde, W., Thomas, D, and Pugh, J., "An Exemplar Based Smalltalk", OOPSLA'86, September 1986, pp.322-330.
13. Liang, Y., and Barsky, B., "A New Concept and Method for Line Clipping", ACM Transactions on Graphics, Vol.3., No.1, January 1984, pp.1-22.
14. Magnenat-Thalmann, N., and Thalmann, D., Computer Animation Theory and Practice, Springer-Verlag, 1985.
15. Magnenat-Thalmann, N., and Thalmann, D., "Three Dimensional Computer Animation: More an Evolution than a Motion Problem", IEEE Computer Graphics and Applications, October 1985.
16. Magnenat-Thalmann, N., Thalmann, D., and Fortin, M., "MIRANIM: An Extensible Director-Oriented System for the Animation of Realistic Images", IEEE Computer Graphics and Applications, March 1985.
17. Papert, S., "Teaching Children Thinking", Proceedings IFIP World Conference on Computer Education, 1970, pp.I/73 - I/78.
18. Reynolds, C.W., "Computer Animation with Scripts and Actors", Proceedings SIGGRAPH 1982, Computer Graphics, Vol.16, No.3, pp.289-296.
19. Smith, R.B., "Experiences with The Alternate Reality Kit: An Example of the Tension Between Literalism and Magic", CHI & GI, 1987.

Appendix A

MACE Commands

This appendix lists the special MACE commands. Commands that are inherited from the normal Smalltalk environment are not included.

A.1 Command Area

depthSwitch	toggle that turns on/off a depth aid. The aid is a tunnel drawn in dashed lines
displayAll	toggle that turns on/off display of all objects. If off, only animated objects are displayed
browseLaws	open an animation law browser
run	run the animation for an amount of time that will be prompted for
nextFrame	display the next frame of the animation
framesPerSecond	set the number of frames displayed per second
film	record the animation for an amount of time. Length of time to film and film to record on will be prompted for. The film is currently in memory.
inspect	inspect the animation world being browsed
createObject	create an animated object. A list of all animated object types currently in the system will be displayed to select from.
createMetaObj	create a meta-object. A list of all meta-objects types currently in the system will be displayed to select from.
refreshDisplay	redraw the display. Needs to be used at times to clean up a messy display that is due to the fact that we don't redraw everything in the display on every interaction to save time.
restart	restart the animation. Sets time to 0.

◆ A.2 View Area Pop-Up Menus

A.2.1 Default

inspect	inspect the selected object
position	set the position of the selected object
deSelect	de-select the selected object
remove	remove the selected object from the animation world
connectors	select all connectors attached to the selected object
showLaws	display all the laws that apply to the selected object
setClock	set the clock that controls the selected object
positionCamera	position the working camera

A.2.2 For Boxes and Spheres

inspect	same as default
position	same as default
resize	resize the selected box or sphere
deSelect	same as default
remove	same as default
connectors	same as default
showLaws	same as default
setClock	same as default
positionCamera	same as default
setRelativeTo	set the object relative to another animated object

A.2.3 For AnimationConnections

inspect	same as default
position	same as default

deselect	same as default
remove	same as default
expression	set the expression the connection uses to extract its value
name	set the name of the connection
setToObject	set the object and attribute of that object the connection passes its value to
setFromObject	set the object that the connection extracts its value from
setClock	same as default
positionCamera	same as default

A.2.4 For AnimationCombiners

inspect	same as default
position	same as default
deselect	same as default
remove	same as default
expression	set the expression the combiner uses to compute its value
name	set the name of the combiner
setClock	same as default
positionCamera	same as default

A.2.5 For AnimationEvents

inspect	same as default
position	same as default
deselect	same as default
remove	same as default
expression	set the action expression of the event
name	set the name of the combiner
setAppliesTo	set the objects the event applies to

positionCamera same as default

A.2.6 For All Other AnimationMetaObjects

inspect same as default

position same as default

deSelect same as default

remove same as default

name set the name of the combiner

setClock same as default

positionCamera same as default

A.2.7 For CameraPens (virtual cameras)

inspect same as default

position same as default

deSelect same as default

remove same as default

name set the name of the combiner

showView show the current view from the selected camera

setClock same as default

positionCamera same as default

setRelativeTo set the camera relative to another animated object

A.3 Animation Law Browser Pop-Up Menu

removeLaw remove the currently selected law

newLaw create a new animation law

usage show the objects in the animation world that use the selected law

classUsage show the classes of objects that use the selected law

Appendix B

Smalltalk/V Source Code for MACE

This appendix lists the Smalltalk/v Source code used to implement MACE. Source code that is copyright of Digital Incorporated is not included. The code is listed by class, with the order being the same as that viewed with the Smalltalk/V browser.

SortedCollection variableSubclass: #AnimationEventQueue

instanceVariableNames: "

classVariableNames: "

poolDictionaries: "

^comment:

 This class represents queues for AnimationEvents.

 The events are sorted on time first and priority second,
 with lower times and higher priorities prevailing.

 The basic calls to instances of this class are

 "add: anAnimationEvent" and "removeFirst", which
 removes and answers the first event in the queue !

!AnimationEventQueue class methods !

new

 "Return a new instance of myself. My entries are AnimationEvents, which
 are sorted on time first and priority second"

^super new sortBlock:

 [:a :b]

 ((a time) = (b time))

 ifTrue: [(a priority) >= (b priority)]

 ifFalse: [(a time) < (b time)] ! !

!AnimationEventQueue methods !

addNoSort: anObject

 "Add an object to the queue without putting it in
 the proper spot. Implemented for efficiency purposes
 only - so that one can add many items and then force
 a sort, instead of sorting each time an item is added

 , Answer anObject."

 endPosition = self basicSize

 ifTrue: [self putSpaceAtEnd].

 endPosition := endPosition + 1.

 ^self basicAt: endPosition put: anObject!

firstIndex

 "Return the index of my first entry"

 ^startPosition!

lastIndex

 "Return the index of my last entry"

 ^endPosition!

next

"Remove and answer my first event"

^self removeFirst!

removeFirst

"Remove and answer my first element.

If I'm empty, return nil"

| answer |

startPosition > endPosition

ifTrue: [^nil],

answer := self basicAt: startPosition.

self basicAt: startPosition put: nil.

startPosition := startPosition + 1.

^answer! !

Object subclass: #AnimationEventQueueManager

instanceVariableNames:

'theQueue currentTime'

classVariableNames: "

poolDictionaries: "

comment:

"This class represents managers for an animation event queue. The basic operation of the manager is to remove the first event from the queue and tell it to perform its action, answering the time stamp of the event. As well, the manager can process all events, returning when the queue is empty. The manager also keeps track of the current time' !

!AnimationEventQueueManager class methods !

new

"Answer a new event queue manager initialized"

^super new initialize! !

!AnimationEventQueueManager methods !

currentTime

"Answer the current time"

^currentTime!

currentTime: aNumber

"Set the current time - this is invalid, as time is determined by the last event to be processed and cannot be adjusted manually "

^self error: 'Cannot set current time - solely dependent on events'!

eventQueue


```
"Answer my queue"

^theQueue!

eventQueue: anAnimationEventQueue
    "Set my queue to anAnimationEventQueue"

    (anAnimationEventQueue isKindOf: AnimationEventQueue)
        ifFalse:[^self error:'The queue must be an AnimationEventQueue'].

    theQueue:= anAnimationEventQueue!

initialize
    "Initialize myself by resetting my event queue and time"

    theQueue:= AnimationEventQueue new.
    currentTime:=0.0.!

initializeQueue
    "Create my queue and initialize it"

    theQueue:= AnimationEventQueue new.!

initializeQueueTo: aCollection
    "Create my queue and initialize it with copies of all the
    events found in aCollection. Set my current time to 0.0"

    | temp |

    theQueue:= AnimationEventQueue new.
    aCollection do:[ :each |
        theQueue addNoSort:
            ((each deepCopy) queue: theQueue).
    ].
    theQueue reSort.
    currentTime:= 0.0.!

processAll
    "Process all the events in the queue"

    [(self processNext) isNil]
        whileFalse:[]!

processNext
    "Process the next event, removing it from
    the queue. Answer the current event. If the queue is empty
    answer nil"

    | timeStamp nextEvent |

    nextEvent:= theQueue next.
```

```

nextEvent isNil
  ifTrue:[^nil].
currentTime:= nextEvent time.
nextEvent performAction.
^nextEvent!

```

```

processNextTick
  "Process the next event, and all other events
  scheduled for the same time. If the queue is empty
  answer nil"

```

```

  | timeStamp nextEvent |

```

```

  "Process the next event, setting current time to its time"
  nextEvent:= theQueue next.
  nextEvent isNil
    ifTrue:[^nil].
  currentTime:= nextEvent time.
  nextEvent performAction.

```

```

  "Process all other events at this same time"
  [((nextEvent:=theQueue next) isNil)
   or: { (nextEvent time) > currentTime}]
  whileFalse:[ nextEvent performAction].

```

```

  "If not at end of queue, add back the event which was not processed"
  nextEvent isNil
    ifFalse:[ theQueue add: nextEvent ].

```

```

  ^nextEvent!

```

```

processUntilTime: aNumber
  "Process the next event, and all other events
  scheduled for times less than or equal to aNumber.
  If the queue is empty answer nil"

```

```

  | nextEvent |

```

```

  "Process all events up to and including time aNumber"

```

```

  [((nextEvent:= theQueue next) isNil)
   or: { (nextEvent time) > aNumber}]
  whileFalse:[ currentTime:= nextEvent time.
               nextEvent performAction].

```

```

  "If not at end of queue, add back the event which was not processed"
  nextEvent isNil
    ifFalse:[ theQueue add: nextEvent ].

```

2

Micro-D

```

^nextEvent!

queue
  "Answer my queue"

^theQueue!

queue: anAnimationEventQueue
  "Set my queue to anAnimationEventQueue"

(anAnimationEventQueue isKindOf: AnimationEventQueue)
  ifFalse:[^self error:'The queue must be an AnimationEventQueue']

theQueue:= anAnimationEventQueue!

Object subclass: #AnimationWorld
  instanceVariableNames:
    'connections animatedObjects nonAnimatedObjects activeCamera clocks activeDisplayClock
    eventManager eventQueue'
  classVariableNames: "
  poolDictionaries: "
  comment:
    "This is the basic class for management of an animation.
    An instance of this class owns all the objects of the animation.
    These are: connections - which allow interconnection of objects
    and their attributes; animatedObjects - which are the basic objects
    viewable in the animation plus cameras, mainly objects with class
    ThreeDObject in their ancestry; nonAnimatedObjects - which are used
    for control, and are objects such as combiners; an activeCamera -
    which is the camera or cameras currently used for viewing the world;
    an eventManager - which is responsible for managing the event queue;
    an animation event queue - which contains the predetermined events
    in the animation world (such as clocks ticks, etc); clocks - which
    control an objects updates, i.e. it tells all dependent objects to
    update on each tick; activeDisplayClock - is the clock used for
    generating the frames. This clocks purpose is to tell the animation
    world to display itself when it ticks"!

!AnimationWorld class methods!

new
  "Return an initialized AnimationWorld"

  ^super new initialize!

!AnimationWorld methods!

activeCamera
  "Return the activeCamera"

```

```
^activeCamera!

activeCamera: aCameraPen
    "Set the activeCamera - if one is already
    active, turn it off"

    (activeCamera = nil)
        ifFalse: [ activeCamera off].
    activeCamera := aCameraPen.
    activeCamera on.!

activeDisplayClock
    "Answer my active display clock"

^activeDisplayClock!

activeDisplayClock: anAnimationDisplayClock
    "Set my active display clock"

    activeDisplayClock := anAnimationDisplayClock!

add: aThreeDObject
    "Add aThreeDObject to my collection
    of animated objects if it isn't already present"

    (animatedObjects includes: aThreeDObject)
        ifFalse: [animatedObjects add: aThreeDObject].!

addAnimatedObject: aThreeDObject
    "Add aThreeDObject to my collection
    of animated objects if it isn't already present"

    (animatedObjects includes: aThreeDObject)
        ifFalse: [animatedObjects add: aThreeDObject].!

addClock: anAnimationClock
    "Add anAnimationClock to my collection
    of clocks if it isn't already present"

    (clocks includes: anAnimationClock)
        ifFalse: [clocks add: anAnimationClock].!

addConnection: aConnection
    "Add aConnection to my collection of connections"

    connections add: aConnection!

addMetaObject: anObject
    "Add anObject to my collection
    of non-animated objects if it isn't already present"
```

```

(anObject isKindOf: AnimationConnection)
  ifTrue:[^self addConnection: anObject].
(anObject isKindOf: AnimationClock)
  ifTrue:[^self addClock: anObject].
(anObject isKindOf: AnimationEvent)
  ifTrue:[eventQueue add: anObject.
    ^self addNonAnimatedObject: anObject].
^self addNonAnimatedObject: anObject!

addNonAnimatedObject: anObject
  "Add anObject to my collection
  of non-animated objects if it isn't already present"

  (nonAnimatedObjects includes: anObject)
    ifFalse: [nonAnimatedObjects add: anObject]!.

animatedObjectNearestTo: aVector
  "Answer the animated object in my world that is nearest to
  aVector, where aVector is a aVector defined in this
  worlds space."

  | min nearest distance|

  min:= 999999999.
  nearest:=nil.
  animatedObjects do: [:each |
    distance:= each worldDistanceFrom: aVector.
    (distance < min)
      ifTrue:[ nearest:= each.
        min:= distance]].

  ^nearest!

animatedObjects
  "Return my animated objects"
  ^animatedObjects!

clockNearestTo: aVector
  "Answer the animation clock in my world that is
  nearest to aVector, where aVector is a aVector defined
  in this worlds space."

  | min nearest distance|

  min:= 999999999.
  nearest:=nil.
  clocks do: [:each |
    distance:= each worldDistanceFrom: aVector.
    (distance < min)
      ifTrue:[ nearest:= each.

```

```

min:= distance]].

^nearest!

clocks
"Answer my clocks"

^clocks!

connectionNearestTo: aVector
"Answer the animation connection in my world that is
nearest to aVector, where aVector is a aVector defined
in this worlds space."

I min nearest distance

min:= 999999999.
nearest:=nil.
connections do: [:each I
    distance:= each worldDistanceFrom: aVector.
    (distance < min)
    ifTrue:[ nearest:= each.
        min:= distance]].

^nearest!

connections
"Answer my collection of connections"

^connections!

defaultClock
"Answer the deafult clock. This is the first clock in
my collection of clocks"

^clocks at: 1!

display
"Display the view from all active cameras. Assumes the cameras'
destForms have been set. Only displays animated objects"

animatedObjects do: [:camera I
    (camera isKindOf: CameraPen)
    ifTrue:[
        camera active
        ifTrue:[
            animatedObjects do:[:eachI
                each displayWith: camera]]]]!

```

displayAllOn: aForm

"Display the view from the activeCamera on aForm. This method displays all objects i.e. connections, non-animated objects and animated objects, not just the animated objects "

```
(activeCamera = nil)
  ifTrue: [self error: 'There is no active camera']
  ifFalse:
```

```
  activeCamera displayOn: aForm.
```

```
  animatedObjects do: [:each |
    each displayWith: activeCamera].
  nonAnimatedObjects do: [:each |
    each displayWith: activeCamera].
  connections do: [:each |
    each displayWith: activeCamera].
  clocks do: [:each |
    each displayWith: activeCamera]]!
```

displayAllOn: aForm withCamera: aCameraPen

"Display the view from aCameraPen on aForm. This method displays all objects i.e. connections, non-animated objects and animated objects, not just the animated objects "

```
(aCameraPen isKindOf: CameraPen)
  ifFalse: [self error: 'Can only display with a CameraPen']
  ifTrue:
```

```
  aCameraPen displayOn: aForm.
```

```
  animatedObjects do: [:each |
    each displayWith: aCameraPen].
  nonAnimatedObjects do: [:each |
    each displayWith: aCameraPen].
  connections do: [:each |
    each displayWith: aCameraPen].
  clocks do: [:each |
    each displayWith: aCameraPen]]!
```

displayOn: aForm

"Display the view from the activeCamera on aForm"

```
(activeCamera = nil)
  ifTrue: [self error: 'There is no active camera']
  ifFalse:
```

```
  activeCamera displayOn: aForm.
```

```
  animatedObjects do: [:each |
    each displayWith: activeCamera]]!
```

displayOn: aForm withCamera: aCameraPen

```
"Display the view from aCameraPen on aForm"

(aCameraPen isKindOf: CameraPen)
  ifFalse: [self error: 'Can only display with a CameraPen']
  ifTrue: [
    aCameraPen displayOn: aForm.

    animatedObjects do: [ :each |
      each displayWith: aCameraPen]]!

eventManager
  "Answer my event queue manager"

  ^eventManager!

eventQueue
  "Answer my event queue"

  ^eventQueue!

initialize
  "Initialize myself"

  animatedObjects:= OrderedCollection new.
  nonAnimatedObjects:= OrderedCollection new.
  connections:= OrderedCollection new.
  clocks:= OrderedCollection new.

  activeCamera:=nil.
  eventManager:= AnimationEventManager new.
  eventQueue:= AnimationEventQueue new.
  activeDisplayClock:= AnimationDisplayClock withAnimationWorld: self.
  clocks add:
    ((AnimationClock withAnimationWorld: self)
     name:'DClock';
     setOrigin: (1,1,100)).!

initializeGraphPane: aBox
  "Return a form for the GraphPane"
  | tempForm |
  Display fill: aBox
    rule: Form over
    mask: Form white.
  tempForm:= Form new
    width: (aBox width)
    height: (aBox height)
    initialByte: 16rFF.
  self displayOn: tempForm.
  tempForm displayAt: (aBox origin).
  ^tempForm!
```

newdisplay

"Display the view from all active cameras. Assumes the cameras' destForms have been set. Only displays animated objects"

```

animatedObjects do: [:camera |
  (camera isKindOf: CameraPen)
  ifTrue:[
    camera active
    ifTrue:[
      animatedObjects do:[each |
        (each isKindOf: Sphere)
        ifTrue:[
          each displayWith: camera]]]]!

```

nonAnimatedObjectNearestTo: aVector

"Answer the non-animated object in my world that is nearest to aVector, where aVector is a aVector defined in this worlds space."

| min nearest distance|

```

min:= 999999999.
nearest:=nil.
nonAnimatedObjects do: [:each |
  distance:= each worldDistanceFrom: aVector.
  (distance < min)
  ifTrue:[ nearest:= each.
    min:= distance]].

```

^nearest!

nonAnimatedObjects

"Return my non-animated objects"

^nonAnimatedObjects!

objectNearestTo: aVector

"Answer the object in my world that is nearest to aVector, where aVector is a aVector defined in this worlds space. The nearest object can be a connection, animatedObject, or nonAnimatedObject"

| min nearest distance|

```

min:= 999999999.
nearest:=nil.
animatedObjects do: [:each |
  distance:= each worldDistanceFrom: aVector.
  (distance < min)

```

```

        ifTrue:[ nearest:= each.
            min:= distance]].
connections do: [:each |
    distance:= each distanceFrom: aVector.
    (distance < min)
        ifTrue:[ nearest:= each.
            min:= distance]].
nonAnimatedObjects do: [:each |
    distance:= each distanceFrom: aVector.
    (distance < min)
        ifTrue:[ nearest:= each.
            min:= distance]].
clocks do: [:each |
    distance:= each distanceFrom: aVector.
    (distance < min)
        ifTrue:[ nearest:= each.
            min:= distance]].
^nearest!

remove: anObject
    "Remove anObject from my world. It can be a Connection,
    an AnimatedObject (ThreeDObject), a Combiner, or
    a clock"

    (anObject isKindOf: AnimationConnection)
        ifTrue:[ ^connections remove: anObject ifAbsent: nil].
    (anObject isKindOf: ThreeDObject)
        ifTrue:[ ^animatedObjects remove: anObject ifAbsent: [].
    (anObject isKindOf: AnimationClock)
        ifTrue:[clocks remove: anObject ifAbsent: [].
            ^eventManager eventQueue
                remove:(anObject ticker) ifAbsent:[]].
    (anObject isKindOf: AnimationEvent)
        ifTrue:[nonAnimatedObjects remove: anObject ifAbsent: [].
            eventQueue remove: anObject ifAbsent: [].
            ^eventManager eventQueue remove: anObject ifAbsent:[]].

    nonAnimatedObjects remove: anObject ifAbsent:[].!

removeAnimatedObject: aThreeDObject
    "Remove aThreeDObject from my collection
    of animated objects if it is present"

    animatedObjects remove: aThreeDObject ifAbsent: [].!

removeClock: anAnimationClock
    "Remove anAnimationClock from my collection
    of clocks if it is present"

    clocks remove: anAnimationClock ifAbsent: [].!

```

```

removeConnection: aConnection
    "Remove aConnection from my collection
    of connections if it is present"

    connections remove: aConnection ifAbsent: [].!

removeNonAnimatedObject: anObject
    "Remove anObject from my collection
    of non-animated objects if it is present"

    nonAnimatedObjects remove: anObject ifAbsent: [].!

resetEventQueue
    "Reset my event queue"

    eventQueue:= AnimationEventQueue new!

restart
    "Restart the animation by initializing the event manager
    and his queue, as well as the clocks tickers. The process
    all events scheduled for time 0.0, i.e. the initializing
    events"

    eventManager initializeQueueTo: eventQueue.
    clocks do: [:each| each startUp].
    activeDisplayClock startUp.
    eventManager processUntilTime:0.0.!

runForFrames: anInteger
    "Run the animation for anInteger frames"

    | eventProcessed |

    1 to: anInteger do: [:i |
        (eventProcessed:=
            eventManager processUntilTime:
                (activeDisplayClock ticker time)) isNil
            ifTrue:[^self error:'Run out of events'].

        ]!

    setCamerasDestForm: aForm
        "Set all the cameras to use aForm to produce their view
        on"

    animatedObjects do: [:each|
        (each isKindOf: CameraPen)

```

```
ifTrue:(each displayOn: aForm))! !
```

```
Object subclass: #AnimationLaws
```

```
instanceVariableNames: "
```

```
classVariableNames:
```

```
  'TheAnimatedObject'
```

```
poolDictionaries: "
```

```
comment:
```

```
  'This class provides all the animation laws used by animated  
  objects, as class methods' !
```

```
'AnimationLaws class methods !
```

```
basicMotionLaw
```

```
  "This law performs basic motion,  
  i.e. change= velocity * dt"
```

```
TheAnimatedObject origin:
```

```
((TheAnimatedObject origin) +
```

```
((TheAnimatedObject velocity) *
```

```
(TheAnimatedObject clock dt)))!
```

```
lawExample
```

```
  "This is an example of how a law is to be  
  written. The object that the law is being  
  applied to is referred to as TheAnimatedObject"
```

```
TheAnimatedObject origin: (TheAnimatedObject  
  origin + (TheAnimatedObject extent))!
```

```
setAnimatedObject: anAnimatedObject
```

```
  "PRIVATE - Set the class variable TheAnimatedObject  
  to anAnimatedObject. Any law performed will be  
  performed on this animated object"
```

```
TheAnimatedObject := anAnimatedObject! !
```

```
'AnimationLaws methods ! !
```

```

Object subclass: #AnimationMetaObject
instanceVariableNames:
'origin scaleFactor normalForm scaledForm name'
classVariableNames: ''
poolDictionaries: ''
comment:
'This class represents animation meta objects (control objects)
that have a representation and position in three dimensional
world space. They can be displayed with a CameraPen' !

!AnimationMetaObject class methods ! !

!AnimationMetaObject methods !

displayWith: aCameraPen
'Display myself with aCameraPen. My representation
is in form, so I must shrink it to the proper size
depending on depth, if the scaleFactor has changed'

| scale transformedPos viewPosition |

transformedPos:= aCameraPen coordinateFor: origin.

"If I'm behind the camera, don't display myself"
((transformedPos z) <= 0.0)
ifTrue:[^nil].
scale:= (1 / ((aCameraPen focalLength) / (transformedPos z)))
asFloat.
(scaleFactor = scale)
ifFalse:[ scaleFactor:=scale.
scaledForm:= normalForm shrink: (0@0 extent: normalForm extent)
by: scaleFactor @ scaleFactor].
viewPosition:= aCameraPen
rightViewingCoordinateFor: origin.

viewPosition:= (aCameraPen project: viewPosition) rounded.
viewPosition:=(aCameraPen twoDPen destForm center) +
(viewPosition x @ (viewPosition y negated)).
viewPosition:= viewPosition -
((scaledForm.extent x // 2)@(scaledForm extent y // 3)).
((aCameraPen twoDPen mask) == (Form white))
ifTrue:[
scaledForm reverse.
aCameraPen twoDPen destForm
copy: (scaledForm boundingBox)
from: scaledForm
to: viewPosition

```

```
rule: Form orRule.  
scaledForm reverse]  
ifFalse:[  
  aCameraPen twoDPen destForm  
  copy: (scaledForm boundingBox).  
  from: scaledForm  
  to: viewPosition  
  rule: Form andRule].!  
  
distanceFrom: aVector  
  "Return the distance I am from aVector. This  
  method returns the distance my origin is from  
  aVector. More accurate distances may be obtained  
  from subclass implementation. Assumes aVector is  
  in world coordinates"  
  
  (aVector isKindOf: Vector)  
  ifFalse:[^self error:'parameter must be a Vector'].  
  ^(origin distanceFrom: aVector)!  
  
form  
  "Answer my normalForm"  
  
  ^normalForm!  
  
name  
  "Answer my name"  
  
  ^name!  
  
name: aString  
  "Set my name"  
  
  name:= aString.  
  self updateForm!  
  
origin  
  "Answer my origin "  
  
  ^origin!  
  
originAsWorld  
  "Answer my origin "  
  
  ^origin!  
  
posiuon: aVector  
  "Set my origin to aVector"  
  
  origin:= aVector!
```

relativeTo

"Answer the coordinate system I am defined in. This is the world coordinate system, which correspond to nil"

^nil!

scaledForm

"Answer my scaledForm"

^scaledForm!

scaleFactor

"Answer my scaleFactor"

^scaleFactor!

setOrigin: aVector

"Set my origin to aVector - used for positioning"

origin:=aVector!

setWorldOrigin: aVector

"Set my origin to aVector - used for positioning"

origin:=aVector!

updateForm

"Create my form. This must be implemented by my subclasses"

^self subclassResponsibility!

worldDistanceFrom: aVector

"Return the distance I am from aVector. This method returns the distance my origin is from aVector. More accurate distances may be obtained from subclass implementation. Assumes aVector is in world coordinates"

(aVector isKindOf: Vector)

ifFalse: (^self error: 'parameter must be a Vector').

^(origin distanceFrom: aVector)!!

AnimationMetaObject subclass: #AnimationCombiner

instanceVariableNames:

'inputs value expression translatedExpression valueBlock alterValues clock'

classVariableNames: "

poolDictionaries: "

comment:

'An animation combiner accepts an OrderedCollection of inputs, which are combined in the expression to produce a value. When asked

for its value, a Combiner returns the current value. When asked to update its value, the combiner updates all inputs. Each input has an associated collection of alterValues, which are resolved in specific ways to produce the new value. In behaviour these inputs are much like attributes of ThreeDObjects. When accessed they return the current value, but when set, they set an alter value. The inputs can come from anywhere, but are usually the output of Connections. A Combiner also knows the clock that controls itself !

!AnimationCombiner class methods !

new

"Answer an initialized combiner"

^super new initialize! !

!AnimationCombiner methods !

addInput: initialValue

"Add a new input into the inputs collection, with its initial value as initialValue. Answer the index of the new input"

inputs addLast: initialValue!

addInputWithValue: initialValue

"Add a new input into the inputs collection, with its initial value as initialValue. Answer the index of the new input"

inputs addLast: initialValue.

^inputs size!

alterVariables

"Answer an array of my instance variables which can accept new values, and resolve them on an update message. This is my collection of inputs"

| inputArray |

inputArray:= Array new: (inputs size + 1).

1 to: ((inputs size) + 1) do[:i] :

inputArray at: i put:('input', (i printString)).!

clock

"Answer the clock which controls me"

^clock!

expression

"Answer my expression"

^expression!

expression>aString

"Set my expression string to aString. Since it has changed, I must ask the compiler to create a new valueBlock"

(aString isKindOf: String)

ifFalse:[^self error:'Expression must be a String'].

expression:= aString.

self translateExpression.

valueBlock:= (Compiler evaluate: (['',translatedExpression,'])

in: (self class)

to: self

notifying: self

ifFail[[]].!

initialize

"Initialize myself"

alterValues:=IdentityDictionary new.

inputs:= OrderedCollection new.

expression:=

valueBlock:=[].!

inputs

"Answer my OrderedCollection of inputs"

^inputs!

inputs: anOrderedCollection

"Set my ordered collection of inputs"

(anOrderedCollection isKindOf: OrderedCollection)

ifFalse:[^self error:'Inputs must be an OrderedCollection'].

inputs:= anOrderedCollection!

promptExpression

"Prompt the user for the expression. This allows him to see it and modify it"

newExpression!

newExpression:=Prompter-prompt:' Combiner Expression '
default: expression.

newExpression isNil

ifFalse:[self expression: newExpression]!

resetAlterValues

"Reset the alter values dictionary"

alterValues := IdentityDictionary new!

resolveAlterValues

"Resolve the alter values to get the new values.

This version simply sums all the alter values
and averages them"

| sum values count symbol|

count := 0.

inputs do: [:name |

count := count + 1.

symbol := ('input',(count printString)) asSymbol.

(alterValues includesKey: symbol)

ifTrue: [values := alterValues at: symbol.

sum := values at: 1.

2 to: (values size) do: [:i |

sum := sum + (values at: i)].

sum := sum / (values size).

inputs at: count put: sum]].!

resolveAndResetAlterValues

"Resolve and reset the alterValues"

self resolveAlterValues; resetAlterValues!

set: symbol to: newValue

"Add newValue to the alter values of the
input with symbol symbol"

(alterValues at: symbol

ifAbsent: [

alterValues at: symbol

put: OrderedCollection new])

addLast: newValue!

setClock: aClock

"Set the clock which controls me, adding myself to its
dependent objects. If I am currently controlled by a clock,
remove myself from its dependent objects"

clock isNil

ifFalse:[clock removeDependentObject: self].

```
clock:= aClock.
clock addDependentObject: self!
```

```
setInput
```

```
"Pop up a menu with my inputs plus a new one,
and answer the one chosen"
```

```
| inputArray selectorArray index |
```

```
inputArray:= Array new: (inputs size + 1).
selectorArray:= Array new: (inputs size + 1).
1 to: ((inputs size) + 1) do:[:i|
    inputArray at: i put:('input', (i printString)).
    selectorArray at: i put: i].
index:=
    (Menu
     labelArray: inputArray
     lines: Array new
     selectors: selectorArray)
    popUpAt: Cursor offset.
index isNil
    ifTrue:[^nil].
(index = (inputs size + 1))
    ifTrue: [self addInput: nil].
^('input',(index printString))!
```

```
translateExpression
```

```
"Translate the expression to standard Smalltalk
code. The only difference is the user can refer
to inputs as 'input1', 'input2', .., 'inputN'. These
have to be translated into '(inputs at: 1)',
'(inputs at: 2)', .., '(inputs at: N)' "
```

```
| subStrings searchString size |
```

```
subStrings:= expression asArrayOfSubstrings.
translatedExpression:= ".
1 to: (inputs size) do:
    [ : inputIndex |
        searchString:= 'input', (inputIndex printString).
        [subStrings includes: searchString]
        whileTrue:
            [subStrings
             at: (subStrings indexOf: searchString)
             put: '(inputs at:', (inputIndex printString), ')'].
    ]
1 to: (subStrings size) do:
    [:i| translatedExpression:= translatedExpression ,
        (subStrings at: i), ' ' ]!
```

```
update
```

"Update myself. Involves resolving all the alter values
for my inputs, then updating my value"

```
self resolveAndResetAlterValues.
self updateValue!
```

updateForm

"Create my form. I am represented by a solid square
with a name. Since this changes my form, I'll set
scaleFactor to nil to ensure the change is immediate, i.e.
the scaledForm will be updated on the next display"

```
! scanner !

normalForm:= Form new extent: 50@50.
normalForm copy: (0@0 extent: 50@50)
  from: StandardAnimationCombinerForm
  to: 0@0
  rule: Form over.
scanner:= CharacterScanner new
  initialize: (0@35 extent: 50@15 )
  font: Font eightLine
  dest: normalForm.
name isNil
  ifTrue:{name:=??}.
scanner display: name
  at: (25 - ((Font eightLine stringWidth: name)
    // 2))@0.
scaleFactor:=nil!
```

updateValue

"Update my value"

```
value:= valueBlock value!
```

value

"Answer my value"

```
^value! !
```

AnimationMetaObject subclass: #AnimationConnection

instanceVariableNames:

'fromObject expression toObject toObjectIndex valueBlock clock'

classVariableNames: "

poolDictionaries: "

comment:

This class represents connections. A connection

has an associated fromObject, expression, and toObject.

The fromObject is where the input of the connection comes from.

The expression is evaluated in the context of the fromObject
to obtain the valueBlock, which is a context. The value of the

valueBlock is the output of the connection. The toObject is either a Combiner or any object. The toObjectIndex gives the index of the toObjects attribute that the output is connected to. In the case of it being a Combiner, this is an index into the Combiners collection of inputs. If the toObject is not a Combiner, it is the symbol of an instance variable of the toObject. A Connection also knows the clock which controls it !

!AnimationConnection class methods !

from: aObject to: bObject

index: anIntegerOrSymbol expression: aString

"Answer a new connection from aObject to bObjects input associated with anIntegerOrSymbol, with the expression aString"

```
^super new fromObject: aObject;
  toObject: bObject index: anIntegerOrSymbol;
  expression: aString !
```

!AnimationConnection methods !

clock

"Answer the clock which controls me"

```
^clock!
```

displayWith: aCameraPen

"Display myself with aCameraPen. My representation is in form, so I must shrink it to the proper size depending on depth"

```
I scale transformedPos viewPosition inPos outPos
```

```
transformedPos:= aCameraPen coordinateFor: origin.
```

```
scale:= (1 / ((aCameraPen focalLength) / (transformedPos z)))
  asFloat).
```

```
(scaleFactor:= scale)
```

```
ifFalse:[scaleFactor:= scale.
```

```
scaledForm:= normalForm shrink: (0@0 extent: normalForm extent)
  by: scale @ scale].
```

```
viewPosition:= aCameraPen
```

```
rightViewingCoordinateFor: origin.
```

```
inPos:= viewPosition +
```

```
(aCameraPen rightViewingDisplacementFor:
  (-25 with: 1 with: 0)).
```

```
outPos:= viewPosition +
```

```
(aCameraPen rightViewingDisplacementFor:
```

```

(25 withy: 1 withz: 0)).

viewPosition:= (aCameraPen project: viewPosition) rounded.
viewPosition:=(aCameraPen twoDPen destForm center) +
    (viewPosition x @ (viewPosition y negated)).
viewPosition:= viewPosition -
    ((scaledForm extent x // 2)@(scaledForm extent y // 3)).
((aCameraPen twoDPen mask) == (Form white))
ifTrue:[
    scaledForm reverse.
    aCameraPen twoDPen destForm
    copy: (scaledForm boundingBox)
    from: scaledForm
    to: viewPosition
    rule: Form orRule.
    scaledForm reverse]
ifFalse:[
    aCameraPen twoDPen destForm
    copy: (scaledForm boundingBox)
    from: scaledForm
    to: viewPosition
    rule: Form andRule}.
fromObject isNil
ifFalse:[
    aCameraPen
    clippedLineFrom: inPos
    to: (aCameraPen rightViewingCoordinateFor:
        (fromObject originAsWorld)).
toObject isNil
ifFalse:[
    aCameraPen clippedLineFrom: outPos
    to: (aCameraPen rightViewingCoordinateFor:
        (toObject originAsWorld))].!

expression
    "Answer my expression string"

    ^expression!

expression: aString
    "Set my expression to aString. Since it has been changed,
    I must ask the compiler to give me a new context for the
    valueBlock"

    (aString isKindOf: String)
    ifFalse:{^self error: 'Expression must be a string'}.
    expression:= aString.
    valueBlock := (Compiler evaluate: ('['expression,']'))
    in: (fromObject class)
    to: fromObject
    notifying: self

```

```
ifFail:[].!

fromObject
  "Answer the object my input is connected to"

  ^fromObject!

fromObject: anObject
  "Set the object my input is connected to"

  fromObject:= anObject!

setClock: aClock
  "Set the clock which controls me, adding myself to its
  dependent connections. If I am currently controlled by a clock,
  remove myself from its dependent connections"

  clock isNil
    ifFalse:[clock removeConnection: self].
  clock:= aClock.
  clock addConnection: self!

tick
  "The clock has ticked, set the attribute of toObject indexed
  by toObjectIndex to my value"

  toObject set: toObjectIndex to: (self value)!

toObject
  "Answer the object my output is connected to"

  ^toObject!

toObject: anObject index: anIntegerOrSymbol
  "Set the object my output is connected to"

  toObject:= anObject.
  toObjectIndex:= anIntegerOrSymbol!

update
  "Update myself, i.e. pass my value to my toObject.
  Note: don't pass the value if it is undefined"

  | result |

  result:=self value.
  result isNil
    ifFalse:[toObject set: toObjectIndex to: result].!

updateForm
  "Create my form. I am represented by a solid sphere"
```


with a name. Set scaleFactor to nil to ensure the scaledForm (the one displayed) is updated on the next display"

| pen scanner |

```
normalForm:= Form new extent: 50@50.
normalForm copy: (0@0 extent: 50@50)
  from: StandardAnimationConnecutionForm
  to: 0@0
  rule: Form over.
scanner:= CharacterScanner new
  initialize: (0@35 extent: 50@15 )
  font: Font eightLine
  dest: normalForm.
name isNil
  ifTrue:[name:=??'].
scanner display: name
  at: (25 - ((Font eightLine stringWidth: name)
    // 2))@0.
scaleFactor:=nil!
```

value

"Answer my value. Returns the value of the valueBlock"

^valueBlock value! !

AnimationMetaObject subclass: #AnimationClock

instanceVariableNames:

'initialStartTime initialEndTime initialDt currentTime theAnimationWorld connections otherDependents
ticker'

classVariableNames: "

poolDictionaries: "

comment:

'Basic clock for controlling the time aspect of an animation.

Every clock is associated with an animation world, theAnimationWorld,

and has an initial start time, initial end time, initial Dt value,

and a ticker to provide its control. Basically, when a clock

is told to activate itself, it puts a RepeatingAnimationEvent

(the ticker) in the animation worlds event queue. The tickers

startTime, endTime and dt correspond to the clocks initial values

for these attributes. My instances

can be meta-objects in an animation world, so I can display myself

using a CameraPen, knowing whether to show myself or erase myself.

For these purposes, I have a normalForm, scaledForm, position, and

scale' !

!AnimationClock class methods !

new

"Answer a new AnimationClock with

```
default attributes"

^super new initialize!

withAnimationWorld: anAnimationWorld
  "Answer a new instance for anAnimationWorld,
  with default attributes."

  ^super new animationWorld: anAnimationWorld;
  initialize! !

!AnimationClock methods !

addConnection: aConnection
  "Add a connection to my dependent connections"

  connections add: aConnection!

addDependentObject: anObject
  "Add an object to my dependent objects. If it is a
  connection, add it to my my connections, else add it
  to my otherDependents"

  (anObject isKindOf: AnimationConnection)
    ifTrue:[self addConnection: anObject]
    ifFalse:[self addOtherDependent: anObject]!

addOtherDependent: anObject
  "Add an object to my other dependent objects."

  otherDependents add: anObject!

animationWorld
  "Answer the animation world I'm associated with"

  ^theAnimationWorld!

animationWorld: anAnimationWorld
  "Set the animation world I'm associated with."

  (anAnimationWorld isKindOf: AnimationWorld)
    ifFalse:[^self error:'Parameter must be an AnimationWorld'].

  theAnimationWorld:= anAnimationWorld!

currentTime
  "Answer my current time"

  ^currentTime!
```

```
displayWith: aCameraPen
    "Display myself with aCameraPen. My representation
    is in form, so I must shrink it to the proper size
    depending on depth, if the scaleFactor has changed"
```

```
super displayWith: aCameraPen.
connections isEmpty
ifFalse:[
    connections do:[each|
        aCameraPen
        drawLineFrom: origin
        to: each originAsWorld]].
otherDependents isEmpty
ifFalse:[
    otherDependents do:[each|
        aCameraPen
        drawLineFrom: origin
        to: each originAsWorld]].!
```

```
dt
    "Answer my current dt"
```

```
^ticker dt!
```

```
form
    "Answer my normalForm"
```

```
^normalForm!
```

```
initialize
    "Set default attributes values"
```

```
initialStartTime:=0.0.
initialEndTime:=99999999.0.
initialDt:= (1/24) asFloat.
connections:=OrderedCollection new.
otherDependents:=OrderedCollection new.
self initializeTicker!
```

```
initializeTicker
    "Initialize my ticker"
```

```
ticker:= AnimationClockTicker
```

```
    startTime: ((initialStartTime = 0.0)
        ifTrue:[ initialDt]
        ifFalse:(initialStartTime))
    endTime: initialEndTime
```

```
dt: initialDt
queue: theAnimationWorld eventManager queue
forClock: self!

initialTicksPerSecond
  "Answer my initial number of ticks per second. This
  is my 1 / initialDt"

  ^ (1 / initialDt) asFloat!

name
  "Answer my name"

  ^name!

name: aString
  "Set my name"

  name:= aString.
  self updateForm!

origin
  "Answer my origin"

  ^origin!

origin: aVector
  "Set my origin"

  origin:= aVector!

position: aVector
  "Set my origin to aVector"

  origin:= aVector!

remove: anObject
  "Remove anObject from my connections and dependentObjects"

  connections remove: anObject ifAbsent: [].
  otherDependents remove: anObject ifAbsent: [].!

removeConnection: anObject
  "Remove anObject from my dependent connections"

  connections remove: anObject ifAbsent: [].!

removeDependentObject: anObject
  "Remove anObject from my dependent objects"
```

```
otherDependents remove: anObject ifAbsent:[].!

scaledForm
  "Answer my scaledForm"

  ^scaledForm!

scaleFactor
  "Answer my scaleFactor"

  ^scaleFactor!

setOrigin: aVector
  "Set my origin"

  origin:= aVector!

startUp
  "Activate myself by putting my ticker in the animation
  worlds event queue"

  self initializeTicker.
  currentTime:=0.0!

tick
  "Perform a tick - update current time. Tell my dependent
  connections to update themselves, then tell my other dependents
  to update themselves"

  currentTime:= ticker time.
  connections do:[each!
    each update].
  otherDependents do: [each!
    each update].!

ticker
  "Answer my ticker - the repeating animation event which
  causes me to tick"

  ^ticker!

ticksPerSecond
  "Answer my current number of ticks per second"

  ^ (1 / (ticker dt)) asFloat!

ticksPerSecond: anInteger
  "Set my number of ticks per second"

  ticker dt: (1 / anInteger)!
```

time

"Answer my current time"

^currentTime!

time: aFloat

"Set my time to aFloat seconds - inform
my dependents!"

updateForm

"Create my form. I am represented by a solid square
with a name. Since this changes my form, I'll set
scaleFactor to nil to ensure the change is immediate, i.e.
the scaledForm will be updated on the next display"

| scanner |

normalForm:= Form new extent: 50@50.

normalForm copy: (0@0 extent: 50@50)

from: StandardAnimationClockForm

to: 0@0

rule: Form over.

scanner:= CharacterScanner new

initialize: (0@35 extent: 50@15)

font: Font eightLine

dest: normalForm.

name isNil

ifTrue:[name:=??].

scanner display: name

at: (25 - ((Font eightLine stringWidth: name)

// 2))@0.

scaleFactor:=nil! !

AnimationMetaObject subclass: #AnimationCombiner

instanceVariableNames:

'inputs value expression translatedExpression valueBlock alterValues clock'

classVariableNames: "

poolDictionaries: "

comment:

'An animation combiner accepts an OrderedCollection of inputs, which
are combined in the expression to produce a value. When asked
for its value, a Combiner returns the current value. When
asked to update its value, the combiner updates all inputs.
Each input has an associated collection of alterValues, which
are resolved in specific ways to produce the new value. In behaviour
these inputs are much like attributes of ThreeDObjects. When
accessed they return the current value, but when set, they set
set an alter value. The inputs can come from anywhere, but are
usually the output of Connections. A Combiner also knows the clock
that controls itself !

!AnimationCombiner class methods !

new

"Answer an initialized combiner"

^super new initialize! !

!AnimationCombiner methods !

addInput: initialValue

"Add a new input into the inputs collection, with its initial value as initialValue. Answer the index of the new input"

inputs addLast: initialValue!

addInputWithValue: initialValue

"Add a new input into the inputs collection, with its initial value as initialValue. Answer the index of the new input"

inputs addLast: initialValue.

^inputs size!

alterVariables

"Answer an array of my instance variables which can accept new values, and resolve them on an update message. This is my collection of inputs"

| inputArray |

inputArray:= Array new: (inputs size + 1).

1 to: ((inputs size) + 1) do:[:i

inputArray at: i put:(input', (i printString))].!

clock

"Answer the clock which controls me"

^clock!

expression

"Answer my expression"

^expression!

expression: aString

"Set my expression string to aString Since it has changed, I must ask the compiler to create a new valueBlock"

```

(aString isKindOf: String)
  ifFalse:[^self error:'Expression must be a String'].
  expression:= aString.
  self translateExpression.
  valueBlock:= (Compiler evaluate: (['',translatedExpression,'])
    in: (self class)
    to: self
    notifying: self
    ifFail:[]).!

initialize
  "Initialize myself"

  alterValues:=IdentityDictionary new.
  inputs:= OrderedCollection new.
  expression:= ''.
  valueBlock:=[].!

inputs
  "Answer my OrderedCollection of inputs"

  ^inputs!

inputs: anOrderedCollection
  "Set my ordered collection of inputs"

  (anOrderedCollection isKindOf: OrderedCollection)
    ifFalse:[ ^self error:'Inputs must be an OrderedCollection'].
  inputs:= anOrderedCollection!

promptExpression
  "Prompt the user for the expression. This allows
  him to see it and modify it"

  InewExpression!

  newExpression:=Prompter prompt:'Combiner Expression'
    default: expression.
  newExpression isNil
    ifFalse:[ self expression: newExpression]!

resetAlterValues
  "Reset the alter values dictionary"

  alterValues := IdentityDictionary new!

resolveAlterValues
  "Resolve the alter values to get the new values.
  This version simply sums all the alter values
  and averages them"

```



```

I sum values count symbol

count:= 0.
inputs do: [ :name |
    count:= count + 1.
    symbol:= ('input',(count printString)) asSymbol.
    (alterValues includesKey: symbol)
    ifTrue: [ values:= alterValues at: symbol.
        sum:= values at: 1.
        2 to: (values size) do: [ :i |
            sum:= sum + (values at: i)].
        sum := sum / (values size).
        inputs at: count put: sum]].!

resolveAndResetAlterValues
    "Resolve and reset the alterValues"

    self resolveAlterValues; resetAlterValues!

set: symbol to: newValue

    "Add newValue to the alter values of the
    input with symbol symbol"

    (alterValues at: symbol
    ifAbsent: [
        alterValues at: symbol
        put: OrderedCollection new])
    addLast: newValue!

setClock: aClock
    "Set the clock which controls me, adding myself to its
    dependent objects. If I am currently controlled by a clock,
    remove myself from its dependent objects"

    clock isNil
        ifFalse:[clock removeDependentObject: self].
    clock:= aClock.
    clock addDependentObject: self!

setInput
    "Pop up a menu with my inputs plus a new one,
    and answer the one chosen"

    I inputArray selectorArray indexI

```

```

inputArray:= Array new: (inputs size + 1).
selectorArray:= Array new: (inputs size + 1).
1 to: ((inputs size) + 1) do:[i]
    inputArray at: i put:('input', (i printString)).
    selectorArray at: i put: i].
index:=
    (Menu
     labelArray: inputArray
     lines: Array new
     selectors: selectorArray)
    popUpAt: Cursor offset.
index isNil
    ifTrue:[^nil].
(index = (inputs size + 1))
    ifTrue: {self addInput: nil}.
^('input',(index printString))!

```

translateExpression

"Translate the expression to standard Smalltalk code. The only difference is the user can refer to inputs as 'input1', 'input2', .., 'inputN'. These have to be translated into '(inputs at: 1)', '(inputs at: 2)', .., '(inputs at: N)'."

```

1 subStrings searchString size!

```

```

subStrings:= expression asArrayOfSubstrings.
translatedExpression:= ".
1 to: (inputs size) do:
    [ : inputIndex |
        searchString:= 'input' , (inputIndex printString).
        [subStrings includes: searchString]
        whileTrue:
            [subStrings
             at: (subStrings indexOf: searchString)
             put: '(inputs at: ' , (inputIndex printString) , ')']].

1 to: (subStrings size) do:
    [:i| translatedExpression:= translatedExpression ,
        (subStrings at: i), ' '].!

```

update

"Update myself. Involves resolving all the alter values for my inputs, then updating my value"

```

self resolveAndResetAlterValues.
self updateValue!

```

updateForm

"Create my form. I am represented by a solid square with a name. Since this changes my form, I'll set

scaleFactor to nil to ensure the change is immediate, i.e.
the scaledForm will be updated on the next display"

| scanner |

```
normalForm:= Form new extent: 50@50.
normalForm copy: (0@0 extent: 50@50)
  from: StandardAnimationCombinerForm
  to: 0@0
  rule: Form over.
scanner:= CharacterScanner new
  initialize: (0@35 extent: 50@15 )
  font: Font eightLine
  dest: normalForm.
name isNil
  ifTrue:(name:=??).
scanner display: name
  at: (25 - ((Font eightLine stringWidth: name)
    // 2))@0.
scaleFactor:=nil!
```

updateValue

"Update my value"

value:= valueBlock value!

value

"Answer my value"

^value! !

AnimationMetaObject subclass: #AnimationConnection

instanceVariableNames:

'fromObject expression toObject toObjectIndex valueBlock clock'

classVariableNames: "

poolDictionaries: "

comment:

"This class represents connections. A connection
has an associated fromObject, expression, and toObject.
The fromObject is where the input of the connection comes from.
The expression is evaluated in the context of the fromObject
to obtain the valueBlock, which is a context. The value of the
valueBlock is the output of the connection. The toObject is either a
Combiner or any object. The toObjectIndex gives the index of the
toObjects attribute that the output is connected to. In the
case of it being a Combiner, this is an index into the Combiners
collection of inputs. If the toObject is not a Combiner, it is
the symbol of an instance variable of the toObject. A Connection
also knows the clock which controls it !

!AnimationConnection class methods !

```

from: aObject to: bObject
    index: anIntegerOrSymbol expression: aString

"Answer a new connection from aObject to
bObjects input associated with anIntegerOrSymbol, with
the expression aString"

^super new fromObject: aObject;
    toObject: bObject index: anIntegerOrSymbol,
    expression: aString! !

!AnimationConnection methods !

clock
    "Answer the clock which controls me"

    ^clock!

displayWith: aCameraPen
    "Display myself with aCameraPen. My representation
    is in form, so I must shrink it to the proper size
    depending on depth"

    | scale transformedPos viewPosition inPos outPos |

    transformedPos:= aCameraPen coordinateFor: origin.
    scale:= (1 / ((aCameraPen focalLength) / (transformedPos z)))
        asFloat .
    (scaleFactor = scale)
        ifFalse:[scaleFactor:= scale.
            scaledForm:= 'normalForm shrink: (0@0 extent: normalForm extent)
                by: scale @ scale].
    viewPosition:= aCameraPen
        rightViewingCoordinateFor: origin.
    inPos:= viewPosition +
        (aCameraPen rightViewingDisplacementFor:
            (-25 withy: 1 withz: 0)).
    outPos:= viewPosition +
        (aCameraPen rightViewingDisplacementFor:
            (25 withy: 1 withz: 0)).

    viewPosition:= (aCameraPen project: viewPosition) rounded.
    viewPosition:=(aCameraPen twoDPen destForm center) +
        (viewPosition x @ (viewPosition y negated)).
    viewPosition:= viewPosition -
        ((scaledForm extent x // 2)@(scaledForm extent y // 3)).
    ((aCameraPen twoDPen mask) == (Form white))
    ifTrue:[

```

```

scaledForm reverse.
aCameraPen twoDPen destForm
copy: (scaledForm boundingBox)
from: scaledForm
to: viewPosition
rule: Form orRule.
scaledForm reverse]
ifFalse:[
aCameraPen twoDPen destForm
copy: (scaledForm boundingBox)
from: scaledForm
to: viewPosition
rule: Form andRule].
fromObject isNil
ifFalse:[
aCameraPen
clippedLineFrom: inPos
to: (aCameraPen rightViewingCoordinateFor:
(fromObject originAsWorld))].
toObject isNil
ifFalse:[
aCameraPen clippedLineFrom: outPos
to: (aCameraPen rightViewingCoordinateFor:
(toObject originAsWorld))].!

expression
"Answer my expression string"

^expression!

expression: aString
"Set my expression to aString. Since it has been changed,
I must ask the compiler to give me a new context for the
valueBlock"

(aString isKindOf: String)
ifFalse:[^self error: 'Expression must be a string'].
expression:= aString.
valueBlock := (Compiler evaluate: (['',expression,''])
in: (fromObject class)
to: fromObject
notifying: self
ifFail:[]).!

fromObject
"Answer the object my input is connected to"

^fromObject!

fromObject: anObject
"Set the object my input is connected to"

```

```
fromObject:= anObject!

setClock: aClock
    "Set the clock which controls me, adding myself to its
    dependent connections. If I am currently controlled by a clock,
    remove myself from its dependent connections"

    clock isNil
        ifFalse:[clock removeConnection: self].
    clock:= aClock.
    clock addConnection: self!

tick
    "The clock has ticked, set the attribute of toObject indexed
    by toObjectIndex to my value"

    toObject set: toObjectIndex to: (self value)!

toObject
    "Answer the object my output is connected to"

    ^toObject!

toObject: anObject index: anIntegerOrSymbol
    "Set the object my output is connected to"

    toObject:= anObject.
    toObjectIndex:= anIntegerOrSymbol!

update
    "Update myself, i.e. pass my value to my toObject.
    Note: don't pass the value if it is undefined"

    | result |

    result:=self value.
    result isNil
        ifFalse:[toObject set: toObjectIndex to: result].!

updateForm
    "Create my form. I am represented by a solid sphere
    with a name. Set scaleFactor to nil to ensure the
    scaledForm (the one displayed) is updated on the
    next display"

    | pen scanner |

    normalForm:= Form new extent: 50@50.
    normalForm copy: (0@0 extent: 50@50)
        from: StandardAnimationConnectionForm
```

```

to: 0@0
rule: Form over.
scanner= CharacterScanner new
  initialize: (0@35 extent: 50@15 )
  font: Font eightLine
  dest: normalForm.
name isNil
  ifTrue:{name:='??'}.
scanner display: name
  at: (25 - ((Font eightLine stringWidth: name)
    // 2))x@0.
scaleFactor:=nil!

value
  "Answer my value. Returns the value of the valueBlock"

  ^valueBlock value! !

AnimationMetaObject subclass: #AnimationClock
instanceVariableNames:
  'initialStartTime initialEndTime initialDt currentTime theAnimationWorld connections otherDependents
  ticker'
classVariableNames: "
poolDictionaries: "
comment:
  'Basic clock for controlling the time aspect of an animation.
  Every clock is associated with an animation world, theAnimationWorld,
  and has an initial start time, initial end time, initial Dt value,
  and a ticker to provide its control. Basically, when a clock
  is told to activate itself, it puts a RepeatingAnimationEvent
  ( the ticker) in the animation worlds event queue. The tickers
  startTime, endTime and dt correspond to the clocks initial values
  for these attributes. My instances
  can be meta-objects in an animation world, so I can't display myself
  using a CameraPen, knowing whether to show myself or erase myself.
  For these purposes, I have a normalForm, scaledForm, position, and
  scale' !

!AnimationClock class methods !

new
  "Answer a new AnimationClock with
  default attributes"

  ^super new initialize!

withAnimationWorld: anAnimationWorld
  "Answer a new instance for anAnimationWorld,
  with default attributes."

  ^super new animationWorld: anAnimationWorld;

```

```
initialize! !

!AnimationClock methods !

addConnection: aConnection
    "Add a connection to my dependent connections"

    connections add: aConnection!

addDependentObject: anObject
    "Add an object to my dependent objects. If it is a
    connection, add it to my connections, else add it
    to my otherDependents"

    (anObject isKindOf: AnimationConnection)
        ifTrue:[self addConnection: anObject]
        ifFalse:[self addOtherDependent: anObject]!

addOtherDependent: anObject
    "Add an object to my other dependent objects."

    otherDependents add: anObject!

animationWorld
    "Answer the animation world I'm associated with"

    ^theAnimationWorld!

animationWorld: anAnimationWorld
    "Set the animation world I'm associated with."

    (anAnimationWorld isKindOf: AnimationWorld)
        ifFalse:[^self error:'Parameter must be an AnimationWorld']

    theAnimationWorld:= anAnimationWorld!

currentTime
    "Answer my current time"

    ^currentTime!

displayWith: aCameraPen
    "Display myself with aCameraPen. My representation
    is in form, so I must shrink it to the proper size
    depending on depth, if the scaleFactor has changed"

    super displayWith: aCameraPen.
```



```

connections isEmpty
ifFalse:[
    connections do:[each|
        aCameraPen
        drawLineFrom: origin
        to: each originAsWorld]].
otherDependents isEmpty
ifFalse:[
    otherDependents do:[each|
        aCameraPen
        drawLineFrom: origin
        to: each originAsWorld]].!

dt
    "Answer my current dt"

    ^tucker dt!

form
    "Answer my normalForm"

    ^normalForm!

initialize
    "Set default attributes values"

    initialStartTime:=0.0.
    initialEndTime:=99999999.0.
    initialDt:= (1/24) asFloat.
    connections:=OrderedCollection new.
    otherDependents:=OrderedCollection new.
    self initializeTucker!

initializeTucker
    "Initialize my tucker"

    ticker:= AnimationClockTicker

        startTime: ((initialStartTime = 0.0)
            ifTrue:[ initialDt]
            * ifFalse:[initialStartTime])
        endTime: initialEndTime
        dt: initialDt
        queue: theAnimationWorld eventManager queue
        forClock: self!

initialTicksPerSecond
    "Answer my initial number of ticks per second. This
    is my 1 / initialDt"

    ^ (1 / initialDt) asFloat!

```

```
name
  "Answer my name"

  ^name!

name: aString
  "Set my name"

  name:= aString.
  self updateForm!

origin
  "Answer my origin"

  ^origin!

origin: aVector
  "Set my origin"

  origin:= aVector!

position: aVector
  "Set my origin to aVector"

  origin:= aVector!

remove: anObject
  "Remove anObject from my connections and dependentObjects"

  connections remove: anObject ifAbsent: [].
  otherDependents remove: anObject ifAbsent: [].!

removeConnection: anObject
  "Remove anObject from my dependent connections"

  connections remove: anObject ifAbsent: [].!

removeDependentObject: anObject
  "Remove anObject from my dependent objects"

  otherDependents remove: anObject ifAbsent: [].!

scaledForm
  "Answer my scaledForm"

  ^scaledForm!

scaleFactor
```

```
"Answer my scaleFactor"

^scaleFactor!

setOrigin: aVector
  "Set my origin"

  origin:= aVector!

startUp
  "Activate myself by putting my ticker in the animation
  worlds event queue"

  self initializeTicker.
  currentTime:=0.0!

tick
  "Perform a tick - update current time. Tell my dependent
  connections to update themselves, then tell my other dependents
  to update themselves"

  currentTime:= ticker time.
  connections do:[each!
    each update].
  otherDependents do:[each!
    each update].!

ticker
  "Answer my ticker - the repeating animation event which
  causes me to tick"

  ^ticker!

ticksPerSecond
  "Answer my current number of ticks per second"

  ^ (1 / (ticker dt)) asFloat!

ticksPerSecond: anInteger
  "Set my number of ticks per second"

  ticker dt: (1 / anInteger)!

time
  "Answer my current time"

  ^currentTime!

time: aFloat
  "Set my time to aFloat seconds - inform
  my dependents"!
```

updateForm

"Create my form. I am represented by a solid square with a name. Since this changes my form, I'll set scaleFactor to nil to ensure the change is immediate, i.e. the scaledForm will be updated on the next display"

| scanner |

normalForm:= Form new extent: 50@50.

normalForm copy: (0@0 extent: 50@50)

from: StandardAnimationClockForm

to: 0@0

rule: Form over.

scanner:= CharacterScanner new

initialize: (0@35 extent: 50@15)

font: Font eightLine

dest: normalForm.

name isNil

ifTrue:(name:=??').

scanner display: name

at: (25 - ((Font eightLine stringWidth: name)

// 2))@0.

scaleFactor:=nil! !

AnimationClock subclass: #AnimationDisplayClock

instanceVariableNames: "

classVariableNames: "

poolDictionaries: "

comment:

"This class represents clocks which are used solely for controlling the display of an animation. Each time the clock performs its action, i.e. it ticks, it tells the animation world to display itself. Note that control objects such as connections and combiners are not displayed, only the actual animated objects. The default speed of a display clock is 24 frames/second, but this can be modified to any value. The default priority of a display clock is very low priority so that objects are displayed at the end of a tick, once all other actions have occurred" !

!AnimationDisplayClock class methods ! !

!AnimationDisplayClock methods !

initializeTicker

"Initialize my ticker. The difference in my ticker compared to AnimationClock tickers is that my priority is very low so that I will be the last event processed at a given time"

```

super initializeTicker.
ticker priority: AnimationEvent displayClockPriority!

```

```

tick

```

```

"Perform a tick. Since I'm a display clock,
my only action is telling the animation world
to display itself"

```

```

currentTime := ticker time.
theAnimationWorld display! !

```

```

AnimationMetaObject subclass: #AnimationEvent
instanceVariableNames:

```

```

'time action priority objects expression'

```

```

classVariableNames: "

```

```

poolDictionaries: "

```

```

comment:

```

```

"This class represents animation events, which have
a time, action, priority, and a collection of objects
this event applies to. The time is the time stamp
of when the action is to be performed. The action is either a
block (context), a symbol (representing a method to perform),
or nothing, where the action will be performed within the method
"performAction".

```

```

To process an event, the call "event performAction" is made.
Therefore the code of the method "performAction" determines the
action that is taken. The priority determines the ordering of two
events with the same time, where higher priorities are favoured!

```

```

!AnimationEvent class methods!

```

```

action: aBlock at: aNumber

```

```

"Answer a new animation event with action aBlock
and time stamp aNumber. Give it normal priority"

```

```

^super new

```

```

action: aBlock;

```

```

time: aNumber;

```

```

priority: AnimationEvent normalPriority!

```

```

action: aBlock at: aNumber inQueue: anAnimationEventQueue

```

```

"Insert a new animation event with action aBlock
and time stamp aNumber into anAnimationEventQueue.
Give it normal priority"

```

```

anAnimationEventQueue add:

```

```

(super new

```

```

action: aBlock;

```

```

time: aNumber

```

```

priority: AnimationEvent normalPriority)!

```

```
action: aBlock
  at: aNumber
  inQueue: anAnimationEventQueue
  priority: bNumber

"Insert a new animation event with action aBlock
and time stamp aNumber into anAnimationEventQueue
Give it priority bNumber"

anAnimationEventQueue add:
  (super new
   action: aBlock;
   time: aNumber;
   priority: bNumber)!

action: aBlock at: aNumber priority: bNumber
"Answer a new animation event with action aBlock,
time stamp aNumber, and priority bNumber"

^super new action: aBlock; time: aNumber; priority: bNumber!

clockPriority
"Answer the priority at which clock events are set.
Note this is a high priority, as clock events are usually
the first to occur at a given time"

^100!

displayClockPriority
"Answer the priority at which display clocks
are usually set. This is very low, so that all other
events at that time will be performed first"

^0!

forObject: anObject
  expression: aString
  time: aNumber

"Answer a new ExpressionEvent that uses the object anObject,
with expression aString, time stamp aNumber, and normal priority"

^super new
  addObject: anObject ;
  time: aNumber;
  priority: AnimationEvent normalPriority;
  expression: aString!

forObject: anObject
```

```
expression: aString  
time: aNumber  
priority: bNumber
```

"Answer a new ExpressionEvent that uses the object anObject,
with expression aString, time stamp aNumber, and priority
bNumber"

```
^super new  
  addObject: anObject ;  
  time: aNumber;  
  priority: bNumber;  
  expression: aString!
```

```
forObjects: anOrderedCollection  
expression: aString  
time: aNumber
```

"Answer a new ExpressionEvent that uses the objects in
anOrderedCollection, with expression aString, time stamp
aNumber, and normal priority"

```
^super new  
  objects: anOrderedCollection;  
  time: aNumber;  
  priority: AnimationEvent normalPriority;  
  expression: aString!
```

```
forObjects: anOrderedCollection  
expression: aString  
time: aNumber  
priority: bNumber
```

"Answer a new ExpressionEvent that uses the objects in
anOrderedCollection, with expression aString, time stamp aNumber,
and priority bNumber"

```
^super new  
  objects: anOrderedCollection ;  
  time: aNumber;  
  priority: bNumber;  
  expression: aString!
```

```
lowPriority
```

"Answer a priority which is lower than normal"

```
^25!
```

new

"Answer an initialized animation event"

^super new initialize!

normalPriority

"Answer the priority at which events are set by default.
Note this is a medium priority, to allow events to be set
lower or higher than most"

^50!

topPriority

"Answer a priority higher than clockPriority. This
is used mainly for initializing events, such as the
initial positioning of an object"

^125! !

!AnimationEvent methods !

action

"Answer my action"

^action!

action: anObject

"Set my action"

action:= anObject!

addObject: anObject

"Add anObject to my collection of objects that I apply to, or
can use"

objects isNil

ifTrue:[objects:=OrderedCollection new].

objects add: anObject!

displayWith: aCameraPen

"Display myself with aCameraPen. My representation
is in form, so I must shrink it to the proper size
depending on depth, if the scaleFactor has changed. This
is handled by my super, but I must show who I apply to, so
I draw lines to my objects"


```
super displayWith: aCameraPen.
objects isEmpty
ifFalse:[
    objects do:[each|
        aCameraPen
            drawLineFrom: origin
            to: each originAsWorld]].!

expression
    "Answer my expression string"

    ^expression!

expression: aString
    "Set my expression to aString. This means I must
    transform it into a block, and evaluate it in myself
    to get a Context"

    (aString isKindOf: String)
    ifFalse:[^self error:'Expression must be a string'].

expression:= aString.
self updateAction!

initialize
    "Initialize my objects and my representation"

    objects:=OrderedCollection new.
    time:= 0.0.
    priority:= AnimationEvent normalPriority.
    self updateForm!

objects
    "Answer the objects I apply to or use"

    ^objects!

objects: anOrderedCollection
    "Set the objects I apply to or use to anOrderedCollection"

    (anOrderedCollection isKindOf: OrderedCollection)
    ifFalse:[^self error:'Objects must be an OrderedCollection'].

objects:=anOrderedCollection!

performAction
    "Perform my action. Report an error if it is not
    a Context"

    (action isKindOf: Context)
    ifFalse:[^self error:'Invalid Action'].
```

action value!

priority

"Answer my priority to aNumber. Remember higher priority events are executed first"

^priority!

priority: aNumber

"Set my priority to aNumber. Remember higher priority events are executed first"

(aNumber isKindOf: Number)

ifFalse:[^self error:'Priority must be a Number'].

priority:= aNumber!

queue

"Answer the queue I'm associated with - since I don't keep track of that, answer nil (implemented for compatibility with subclasses, i.e RepeatingAnimationEvents must know which queue they are associated with)"

^nil!

queue: anAnimationEventQueue

"Set the queue I'm associated with - since I don't keep track of that, do nothing (implemented for compatibility with subclasses)"

time

"Answer the time at which my action should occur"

^time!

time: aNumber

"Set the time at which my action should occur"

(aNumber isKindOf: Number)

ifFalse:[^self error:'Time must be a number'].

time:= aNumber!

updateAction

"Update the action so that it corresponds to the expression"

action := (Compiler evaluate: (['expression;'])

in: (self class)

to: self

notifying: self

```

    ifFail:[]).!

updateForm
    "Create my form. I take my representation from
    a global, StandardAnimationEventForm. Since this
    changes my form, I'll set scaleFactor to nil to
    ensure the change is immediate, i.e. the scaledForm
    will be updated on the next display"

    !scanner !

    normalForm:= Form new extent: 50@50.
    normalForm copy: (0@0 extent: 50@50)
        from: StandardAnimationEventForm
        to: 0@0
        rule: Form over.
    scanner:= CharacterScanner new
        initialize: (0@35 extent: 50@15 )
        font: Font eightLine
        dest: normalForm.
    name isNil
        ifTrue:[name:=??'].
    scanner display: name
        at: (25 - ((Font eightLine stringWidth: name)
            // 2))@0.
    scaleFactor:=nil! !

AnimationEvent subclass: #RepeatingAnimationEvent
instanceVariableNames:
    'endTime dt queue '
classVariableNames: "
poolDictionaries: "
comment:
    "This class represents events which repeat themselves.
    The event starts at time and each time it performs its action,
    it schedules itself for time + dt, until endTime is reached" !

!RepeatingAnimationEvent class methods !

action: aBlock
    startTime: aNumber
    endTime: bNumber
    dt: cNumber
    queue: anAnimationEventQueue

    "Create a new RepeatingAnimationEvent that will insert
    itself into anAnimationQueue with time stamp startTime.
    Each time the event performs its action, it will change
    its time stamp to time + dt, and reinsert itself in the
    queue, until endTime is reached. It is given normal
    priority"

```

```
^anAnimationEventQueue add:
  (super new
    action: aBlock;
    time: aNumber;
    endTime: bNumber;
    dt: cNumber;
    priority: AnimationEvent normalPriority;
    queue: anAnimationEventQueue)!
```

```
action: aBlock,
  startTime: aNumber
  endTime: bNumber
  dt: cNumber
  queue: anAnimationEventQueue
  priority: dNumber
```

"Create a new RepeatingAnimationEvent that will insert itself into anAnimationQueue with time stamp startTime. Each time the event performs its action, it will change its time stamp to time + dt, and reinsert itself in the queue, until endTime is reached. It is given priority dNumber"

```
^anAnimationEventQueue add:
  (super new
    action: aBlock;
    time: aNumber;
    endTime: bNumber;
    dt: cNumber;
    priority: dNumber;
    queue: anAnimationEventQueue)!
```

```
action: aBlock
  startTime: aNumber
  endTime: bNumber

  queue: anAnimationEventQueue
```

"Create a new RepeatingAnimationEvent that will insert itself into anAnimationQueue with time stamp startTime. Each time the event performs its action, it will change its time stamp to time + dt, and reinsert itself in the queue, until endTime is reached. It is given normal priority"

```
^anAnimationEventQueue add:
  (super new
    action: aBlock;
    time: aNumber;
    endTime: bNumber;
```

```
dt: (1 / 24) asFloat;  
priority: AnimationEvent normalPriority;  
queue: anAnimationEventQueue)!!
```

!RepeatingAnimationEvent methods !

```
dt  
"Answer my dt, i.e. my step time"  
  
^dt!
```

```
dt: aNumber  
"Set my dt, i.e. my step time"  
  
(aNumber isKindOf: Number)  
  ifFalse:[^self error:'Time must be a Number'].  
dt:=aNumber!
```

```
endTime  
"Answer my endTime"  
  
^endTime!
```

```
endTime: aNumber  
"Set my endTime"  
  
(aNumber isKindOf: Number)  
  ifFalse:[^self error:'Time must be a Number'].  
endTime:= aNumber!
```

```
initialize  
"Initialize myself"  
  
super initialize.  
endTime:= 99999999.0.  
dt = 1/24 asFloat.!
```

```
performAction  
"Perform my action, then update my time stamp and  
reinsert myself into the queue if my endTime has  
not been exceeded"
```

```
action value.  
self repeat!
```

```
queue  
"Answer the queue I insert myself into"  
  
^queue!
```

```
queue: anAnimationEventQueue
```

```
    "Set the queue I insert myself into to anAnimationQueue"
```

```
    queue:=anAnimationEventQueue.!
```

```
repeat
```

```

ClassBrowser subclass: #AnimationBrowser
  instanceVariableNames:
    'depthForm depthActive animationPane selectedObject positioning resizing positioningCamera
  displayAll viewCamera'
  classVariableNames: ''
  poolDictionaries: ''
  comment:
    'A browser for animation worlds - will allow user to develop
    an animation' !

!AnimationBrowser class methods !

!AnimationBrowser methods !

accept: aPoint
  "Accept aPoint as the currently selected Point"!

animationMenu
  "Private - Answer the animation pane menu."

  (selectedObject isKindOf: AnimationConnection)
  ifTrue:[
    ^Menu
    labels:
      'inspect\position\deSelect\remove\expression\name\setToObject\setFromObject\positionCamera\set clock'
      withCrs
    lines: Array new
    selectors: #(inspectSelectedObject setPosition deSelectObjects removeObject connectionExpression
      connectionName setToObject setFromObject positionCamera setClock)
  ].

  (selectedObject isKindOf: AnimationCombiner)
  ifTrue:[
    ^Menu
    labels: 'inspect\position\deSelect\remove\expression\name\positionCamera\set clock' withCrs
    lines: Array new
    selectors: #(inspectSelectedObject setPosition deSelectObjects removeObject connectionExpression
      connectionName positionCamera setClock)
  ].

  (selectedObject isKindOf: AnimationEvent)
  ifTrue:[
    ^Menu
    labels: 'inspect\position\deSelect\remove\expression\name\setAppyToObjects\positionCamera\set
      clock' withCrs
    lines: Array new
    selectors: #(inspectSelectedObject setPosition deSelectObjects removeObject connectionExpression
      connectionName setApplicableObjects positionCamera setClock)
  ].

  (selectedObject isKindOf: AnimationMetaObject)
  ifTrue:[

```

```

^Menu
  labels: 'inspect\position\deSelect\remove\name\positionCamera\set Clock' withCrs
  lines: Array new
  selectors: #(inspectSelectedObject setPosition deSelectObjects removeObject connectionName
positionCamera setClock)
].
(selectedObject isKindOf: CameraPen)
ifTrue:[
  ^Menu
    labels: 'inspect\position\deSelect\remove\name\show View\positionCamera\set Clock' withCrs
    lines: Array new
    selectors: #(inspectSelectedObject setPosition deSelectObjects removeObject connectionName
viewFromCamera positionCamera setClock)
].
^Menu
  labels: 'inspect\position\resize\connectors\showLaws\deSelect\remove\positionCamera\setClock'
withCrs
  lines: Array new
  selectors: #(inspectSelectedObject setPosition resizeObject showConnectors showLaws
deSelectObjects removeObject positionCamera setClock)!

animationPane
  "Answer my animation pane"

  ^animationPane!

browseLaws
  "Open a MotionLawBrowser"

  (MotionLawBrowser new) openOn: AnimationLaws
    withWorld: browsedClass!

camera
  "Answer the view camera"

  ^viewCamera!

connectionExpression
  "Show the selected connections expression. Presently merely
prompts with the expression"

  newExpression!

  newExpression:= Prompter prompt: 'Connection expression'
    default: selectedObject expression.
  newExpression isNil
    ifFalse: [selectedObject expression: newExpression]!.

connectionName
  "Show the selected connections name. Presently merely

```


prompts with the expression"

I newExpression I

```
newExpression := Prompter prompt: 'Name'
                  default: selectedObject name.
newExpression isNil
  ifFalse: [selectedObject name: newExpression].!
```

createClock

"Create a new AnimationClock"

I newClock I

```
newClock := AnimationClock withAnimationWorld: browsedClass.
newClock
```

```
name: '??';
```

```
position: ((viewCamera origin) +
  (0 with: 0 with:
    (viewCamera focalLength))).
```

```
browsedClass clocks add: newClock.
self deShowSelectedObjects.
selectedObject := newClock.
```

```
self showSelectedObjects.!
```

createCombiner

"Create a new AnimationCombiner and allow the user to specify the expression"

I newCombiner I

```
newCombiner := AnimationCombiner new.
newCombiner
```

```
expression: "";
name: '??';
position: ((viewCamera origin) +
  (0 with: 0 with:
    (viewCamera focalLength))).
```

```
browsedClass nonAnimatedObjects add: newCombiner.
newCombiner setClock: browsedClass defaultClock.
self deShowSelectedObjects.
selectedObject := newCombiner.
self showSelectedObjects.!
```

createConnection

"Create a new AnimationConnection and allow the user to specify where its coming from and where its going to, as well as the expression"

```
! newConnection !
```

```
newConnection := AnimationConnection new.
newConnection
    expression: "";
    name: "??";
    position: ((viewCamera origin) +
        (0 withy: 0 withz:
            (viewCamera focalLength))).
browseClass connections add: newConnection.
newConnection setClock: browseClass defaultClock.
self deShowSelectedObjects.
selectedObject:= newConnection.
```

```
self showSelectedObjects.!
```

```
createMetaObject
```

```
"Create a new meta-object in the animation world -
Prompt user to select class of meta-object"
```

```
! answer newObject!
```

```
answer:=((SelectorMenu selectorObjects:
    (AnimationMetaObject allSubclasses)
    lines: Array new
    title: 'What type of meta-object?'
    popUpReturningObjectAt: Cursor offset ).
answer isNil
    ifTrue:[^nil].
```

```
(answer == AnimationConnection)
    ifTrue:[^self createConnection].
(answer == AnimationClock)
    ifTrue:[^self createClock].
(answer == AnimationCombiner)
    ifTrue:[^self createCombiner].
newObject:= (answer new)
    setOrigin:
        ((viewCamera origin) +
            (0 withy: 0 withz:
                (viewCamera focalLength))).
browseClass addMetaObject: newObject.
```

```
self changed: #initializeGraphPane:.
self deShowSelectedObjects.
selectedObject:= newObject.
self showSelectedObjects.
CursorManager normal change.!
```

createObject

"Create a new object in the animation world -
Prompt user to select class of object and origin"

answer newObject!

answer:=((SelectorMenu selectorObjects:
 (ThreeDObject allSubclasses)
 lines: Array new
 title: 'What class of object?')
 popUpReturningObjectAt: Cursor offset)

answer isNil

ifTrue:[^nil].

CursorManager execute change.

newObject:= (answer new)

setOrigin:

((viewCamera origin) +

(0 withy: 0 withz:

(viewCamera focalLength))).

browsedClass add: newObject.

newObject setClock: browsedClass defaultClock.

self changed: #initializeGraphPane:.

self deShowSelectedObjects.

selectedObject:= newObject.

self showSelectedObjects.

CursorManager normal change.!

depthForm

"Answer my depth form"

^depthForm!

depthSwitch

"Toggle the depthActive field - inform dependents.
When depthActive is true, depth planes are shown.
When false they are not"

CursorManager execute change.

depthActive:= depthActive not.

self changed: #initializeGraphPane:.

CursorManager normal change.!

deSelectObjects

"De select the currently selected objects,
de-highlights the selected objects, and sets
selected object to nil."

self deShowSelectedObjects.

selectedObject:= nil.!

deShowSelectedObjects

"Highlight the selected object or objects by displaying them with a thicker pen nib"

| currentCamera |

selectedObject isNil

ifFalse:[

currentCamera:= viewCamera.

(currentCamera twoDPen) defaultNib: 2@2; white.

(selectedObject isKindOf: OrderedCollection)

ifTrue:[

selectedObject do:[each |

each displayWith: currentCamera]]

ifFalse:[selectedObject displayWith:

currentCamera].

(currentCamera twoDPen) defaultNib: 1@1; black.

(selectedObject isKindOf: OrderedCollection)

ifTrue:[

selectedObject do:[each |

each displayWith: currentCamera]]

ifFalse:[selectedObject displayWith:

currentCamera].

animationPane showWindow].!

dictionaries

"Private - Answer an Array of dictionary types (class and instance)."

^(niller browseLaws)!

displayAll

"Toggle the displayAll field - inform dependents.

When displayAll is true, all objects are shown.

When false only animated objects are shown"

CursorManager execute change.

displayAll:= displayAll-not.

self changed: #initializeGraphPane:.

CursorManager normal change.!

displayOn: aForm

"Display the animation world on aForm - if depth mode is active, show the depth planes"

depthActive

ifTrue:[

(depthForm = nil)

```

    ifTrue:[ depthForm:= (Form new) extent: (aForm extent).

        viewCamera displayOn: depthForm.
        viewCamera showDepthPlanes].
    aForm copy: (0@0 extent:(depthForm extent))
    from: depthForm to: 0@0 rule: (Form over).]
    ifFalse:[ aForm white].

displayAll
    ifTrue:[browsedClass displayAllOn: aForm withCamera: viewCamera]
    ifFalse:[browsedClass displayOn: aForm withCamera: viewCamera].
self showSelectedObjects.
self displayTime!

displayTime
    "Show the current time of the animation world in
    the upper right corner of the top pane"

    laStream location time timeString!

    aStream := WriteStream on: (String new: 8).
    time:= browsedClass eventManager currentTime.
    timeString:= time printString.
    aStream nextPutAll: "Time: ".
    1 to: (12 - timeString size) do: [:each| aStream space].
    time printOn: aStream.
    location:= (animationPane frame origin x) @ (animationPane frame origin
        y + animationPane frame extent y + 3).

    aStream contents displayAt:
        location.!

eraseSelectedObjects
    "Erase the selected object or objects by displaying
    them with a white pen nib"

    | currentCamera |

selectedObject isNil
    ifFalse:[
        currentCamera:= viewCamera.

        (currentCamera twoDPen) defaultNib: 2@2; white.

        (selectedObject isKindOf: OrderedCollection)
        ifTrue:[
            selectedObject do:[each |
                each displayWith: currentCamera]]
        ifFalse:[ selectedObject displayWith:
            currentCamera].

```

```
(currentCamera twoDPen) defaultNib: 1@1; black.
```

```
animationPane showWindow].!
```

```
film
```

```
"Run and film the animation. Prompt for the number of
seconds to run for and name for the film"
```

```
| time filmName |
```

```
time:= (Prompter prompt: 'Film for how many seconds ?'
defaultExpression: '1').
```

```
time = nil
```

```
ifFalse:[
```

```
filmName:= (Prompter prompt: 'Name for the film ?'
default: 'MyFilm').
```

```
filmName = nil
```

```
ifFalse:[ CursorManager execute change.
```

```
Smalltalk at: (filmName asSymbol)
```

```
put: (OrderedCollection new).
```

```
filmName:= Smalltalk at: (filmName asSymbol).
```

```
1 to: time do: [ :i |
```

```
1 to: ((browsedClass clock) ticksPerSecond) do: [ :j |
```

```
browsedClass tickFor: 1.
```

```
self changed: #initializeGraphPane:.
```

```
filmName add:
```

```
(Form fromDisplay: (animationPane frame))]]].
```

```
CursorManager normal change.!
```

```
framesPerSecond
```

```
"Show/Set the number of clock ticks per second"
```

```
| newValue |
```

```
newValue:= (Prompter prompt: 'Clock ticks per second (integer only)'
```

```
defaultExpression: (((browsedClass activeDisplayClock)
```

```
ticksPerSecond truncated) printPaddedTo: 0)).
```

```
(newValue isKindOf: Integer)
```

```
ifTrue:[ (browsedClass activeDisplayClock) ticksPerSecond: newValue].!
```

```
initializeGraphPane: aBox
```

```
"Return a form for the GraphPane"
```

```
| tempForm |
```

```
Display fill: aBox
```

```
rule: Form over
```

```
mask: Form white.
```

```
tempForm:= Form new
```

```

        width: (aBox width)
        height: (aBox height)
        initialByte: 16rFF.
        self displayOn: tempForm.
        tempForm displayAt: (aBox origin).
        ^tempForm!

inspectAnimationWorld
    "Open an inspector on the animation world"
    "NOTE: temporarily inspects the browser"
    self inspect
    "browsedClass inspect"!

inspectSelectedObject
    "Inspect the currently selected object. If there is
    not one, do nothing. If it is a collection of objects,
    inspect the collection"

    selectedObject isNil
        ifFalse: [selectedObject inspect]!

nextFrame
    "Produce the next frame as viewed by the view camera"

    CursorManager execute change.
    browsedClass runForFrames: 1.
    self changed: #initializeGraphPane:.
    CursorManager normal change!

notYetImplemented
    "Pop up a message to inform user this
    feature is not yet implemented"

    Menu message: 'Implementation not yet complete'!

open
    "Create an animation browser window on aClass. Define
    the type, behavior and relative size of each
    pane and schedule the window."
    | topPane i |

    topPane := TopPane new
        label: 'Animation Browser!';
        minimumSize: SysFontWidth * 20
            @ (SysFontHeight * 8);
        yourself.
    i := SysFontHeight * 2 + 4.

    selectedDictionary := browsedClass.

```

```

topPane addSubpane:
    (SelectorListPane new
        model: self;
        name: #selectors;
        change: #(nil browseLaws);
        menu: #selectorMenu;
        framingBlock: [:box|
            box origin + (0@0) extent:
                box width // 5 @ (box height)]).

topPane dispatcher open scheduleWindow!

openOn: anAnimationWorld
    "Create an animation browser window on
    anAnimationWorld. Define the type, behavior
    and relative size of each pane and schedule
    the window."

    | topPane i |
    depthActive := false.
    positioning:=false.
    resizing:=false.
    positioningCamera:= false.
    displayAll:= true.
    browsedClass:= anAnimationWorld.
    viewCamera:= CameraPen new.
    topPane := TopPane new
        label: (Smalltalk keyAtValue: anAnimationWorld) asString
        'Animation Browser';
        minimumSize: SysFontWidth * 20
            @ (SysFontHeight * 8);
        yourself.
    i := SysFontHeight * 2 + 4.

    selectedDictionary := browsedClass.
    topPane addSubpane:
        (SelectorListPane new
            model: self;
            name: #selectors;
            change: #(depthSwitch displayAll browseLaws run nextFrame framesPerSecond film
inspectAnimationWorld createObject createMetaObject refreshDisplay restart rollViewCamera
pitchViewCamera yawViewCamera);
            menu: #selectorMenu;
            framingBlock: [:box|
                box origin + (0@0) extent:
                    box width // 5 @ (box height)]).
    animationPane:=
        (AnimationPane new
            model: self;
            name: #initializeGraphPane;;
            change: #accept;;

```



```

        menu: #animationMenu;
        framingBlock: [:box|
            box origin + ((box width // 5) @ 0)
            ^ corner: box corner]).
topPane addSubpane: animationPane.
topPane dispatcher open scheduleWindow!

pitchViewCamera
    "Pitch the view camera, that is, rotate it around z axis"

    | angle |

    angle := Prompter prompt: 'Angle to pitch?'
    defaultExpression: '90'.
    angle isNil
        ifFalse: [viewCamera pitch: angle.
            self refreshDisplay]!

positionCamera
    "Allow the user to position the camera. For now, just
    take the x,y of the cursor location as the x,y for the
    cameras new position"

    positioningCamera := true.!

project: aVector
    "Answer the 2D projection of aVector, as performed
    by the animation worlds active camera"

    ^ viewCamera quickProject: aVector!

refreshDisplay
    "Redisplay the animation pane"

    CursorManager execute change.
    animationPane update.
    CursorManager normal change.!

removeObject
    "Remove the selected objects"

    selectedObject isNil
        ifTrue: {^nil}.
    self eraseSelectedObjects.
    (selectedObject isKindOf: OrderedCollection)
        ifTrue: [
            selectedObject do: [:each|
                browsedClass remove: each]]
        ifFalse: [
            browsedClass remove: selectedObject].

```

```
selectedObject:=nil!  
  
resizeObject  
  "Allow the user to resize the selected object"  
  
  selectedObject isNil  
    ifFalse:[  
      (selectedObject respondsTo: #resize:)  
        ifTrue:[ resizing:=true]]!  
  
restart  
  "Restart the animation"  
  
  CursorManager execute change.  
  browsedClass restart.  
  self changed: #initializeGraphPane:.  
  CursorManager normal change!  
  
revertToViewCamera  
  "Show the view from the view camera"  
  
  CursorManager execute change.  
  self changed: #initializeGraphPane:.  
  CursorManager normal change!  
  
rollViewCamera  
  "Roll the view camera, that is, rotate it around x axis"  
  
  | angle |  
  
  angle:= Prompter prompt: 'Angle to roll?'  
    defaultExpression: '90'.  
  angle isNil  
    ifFalse:[viewCamera roll: angle.  
      self refreshDisplay]!  
  
run  
  "Run the animation. Prompt for the number of  
  frames to run for. Uses the cameras of the animation  
  world, not the viewCamera"  
  
  | time |  
  
  time:= (Prompter prompt: 'Run for how many frames ?'  
    defaultExpression: '24').  
  time = nil  
    ifFalse:[self runForFrames: time].!  
  
runForFrames: anInteger  
  "Produce the next anInteger frames of the animation world."
```

Must tell the animation world to set all its cameras to use the pane's form"

```
CursorManager execute change.
browsedClass setCamerasDestForm: animationPane form.
1 to: anInteger do:[il
    animationPane form white.
    browsedClass runForFrames: 1.
    animationPane showWindow.
    self displayTime].
CursorManager normal change!
```

```
selectAtVector: aVector
    "Return the object nearest to the point defined
    by aVector. If none are within the tolerance
    first assigned to 'min', return nil.
    Since aVector is in the active cameras
    coordinate system, first transform it to world.
    Note: CameraPens are not included in the search"
```

```
1 min nearest distance point!
```

```
point := (viewCamera) worldCoordinateFor: aVector.
```

```
positioningCamera
    ifTrue:[CursorManager execute change.
        (viewCamera) worldOrigin:
            ((point x) withy: (point y)
            withz: (point z)).
        viewCamera resolveAndResetAlterValues.
        viewCamera backward: viewCamera focalLength.
        viewCamera resolveAndResetAlterValues.
        positioningCamera:=false.
        self changed: #initializeGraphPane:.
        CursorManager normal change.
        ^(0 withy: 0
            withz: (viewCamera focalLength))].
```

```
positioning
    ifTrue: [
        self eraseSelectedObjects.
        selectedObject setWorldOrigin: point.
        self showSelectedObjects.
        positioning:= false.
        ^nil].
```

```
resizing
    ifTrue:[
        self eraseSelectedObjects.
```

```

selectedObject resize: point.
self showSelectedObjects.
resizing: false.
^nil].

```

"Not positioning camera, positioning selected object, or
resizing object, then must be selecting an object"

```

displayAll
  ifTrue:[
    nearest:=
      browsedClass objectNearestTo: point]
  ifFalse:[
    nearest:=
      browsedClass animatedObjectNearestTo: point].

```

```

nearest = nil
  ifFalse:[
    (nearest == selectedObject)
      ifFalse:[ self deShowSelectedObjects.
        selectedObject:= nearest.
        self showSelectedObjects].
    ^((viewCamera)
      coordinateFor: (nearest originAsWorld))].
^nil!

```

selectors

"Private - Answer a list of selectors."

^#(depthSwitch displayAll browseLaws run nextFrame framesPerSecond film inspect createObject
createMetaObject refreshDisplay restart rollViewCamera pitchViewCamera yawViewCamera)!

setApplicableObjects

"Allow the user to set the objects that the
selected event can use/apply to. The user may continue to
select until he selects the event itself "

| point applyToObject |

```

(selectedObject isKindOf: AnimationEvent)
  ifTrue:[-
    point:= viewCamera worldCoordinateFor:
      (animationPane dispatcher getPoint).
    applyToObject:= browsedClass objectNearestTo: point.
    [applyToObject == selectedObject]
      whileFalse:[
        applyToObject isNil
          ifFalse: [
            selectedObject addObject: applyToObject.
            self showSelectedObjects].
        point:= viewCamera worldCoordinateFor:

```

```

        (animationPane dispatcher getPoint).
        applyToObject:= browsedClass objectNearestTo: point)]!

setClock
    "Allow the user to set the clock controlling the
    selected object"

    | point selectedClock |

    (selectedObject isKindOf: AnimationClock)
    ifFalse:[
        point:= viewCamera worldCoordinateFor:
            (animationPane dispatcher getPoint).
        selectedClock:= browsedClass clockNearestTo: point.
        selectedClock isNil
        ifFalse:[selectedObject setClock: selectedClock]].
    ^nil!

setFromObject
    "Allow the user to set the fromObject of the
    selected connection"

    | point selectedFromObject |

    (selectedObject isKindOf: AnimationConnection)
    ifTrue:{
        point:= viewCamera worldCoordinateFor:
            (animationPane dispatcher getPoint).
        selectedFromObject:= browsedClass objectNearestTo: point.
        selectedFromObject isNil
        ifFalse: [
            self eraseSelectedObjects.
            selectedObject fromObject: selectedFromObject.
            self showSelectedObjects].
        ^nil].!

setPosition
    "Allow the user to set the position of the selected object"

    | point |

    selectedObject isNil
    ifFalse:[
        point:= viewCamera worldCoordinateFor:
            (animationPane dispatcher getPoint).
        self eraseSelectedObjects.
        selectedObject setWorldOrigin: point.

        self showSelectedObjects.
        ^nil].!

```

```
setShowConnectors!
```

```
setShowLaws
```

```
"Display (pop up a window) the laws applying
to the selected object"
```

```
| laws logger extent aString selectors |
```

```
selectedObject isNil
```

```
ifFalse:[
```

```
  aString := 'Laws of Selected Object'.
```

```
  extent := ((Display width // 3 max:
```

```
    aString size * LabelFont width + 6) min.
```

```
  Display width) @ (Display height // 2).
```

```
  logger := TextEditor
```

```
    windowLabeled: aString
```

```
    frame: (Display extent x - extent x @ 0
```

```
      extent: extent).
```

```
  CursorManager execute change.
```

```
  laws := selectedObject allLaws.
```

```
  laws do: [:each |
```

```
    logger nextPutAll:
```

```
      (each asString);
```

```
      cr.].
```

```
  logger positionAtBeginning.
```

```
  CursorManager normal change.
```

```
  logger topDispatcher scheduleWindow].!
```

```
setToObject
```

```
"Allow the user to set the toObject of the
selected connection"
```

```
| point selectedToObject index |
```

```
(selectedObject isKindOf: AnimationConnection)
```

```
ifTrue:[
```

```
  point := viewCamera worldCoordinateFor:
```

```
    (animationPane dispatcher getPoint).
```

```
  selectedToObject := browsedClass objectNearestTo: point.
```

```
selectedToObject isNil
```

```
ifFalse: [
```

```
  (selectedToObject isKindOf: AnimationCombiner)
```

```
  ifTrue:[index := selectedToObject setInput]
```

```
  ifFalse:[
```

```
    index :=
```

```
      (Menu labelArray:
```

```
        (selectedToObject class allInstVarNames)
```

```
        lines: Array new
```

```

        selectors:
        (selectedToObject class allInstVarNames))
        popUpAt: Cursor offset].
    index isNil ifFalse:[index:= index asSymbol].
    self eraseSelectedObjects.
    selectedObject toObject: selectedToObject
        index: index.

    self showSelectedObjects].
^nil].!

setViewCameraToSelectedCamera
    "Private - set the origin, orientation, and focal length
    of the view camera to be similar to those of the selected
    camera"

    CursorManager execute change.
    viewCamera focalLength: selectedObject focalLength.
    viewCamera upVector: selectedObject upVector.
    viewCamera rightVector: selectedObject rightVector.
    viewCamera normalVector: selectedObject normalVector.
    viewCamera origin: selectedObject origin.
    viewCamera cotViewingAngleOverTwo: selectedObject cotViewingAngleOverTwo.
    viewCamera resolveAndResetAlterValues.
    self changed: #initializeGraphPane:.
    CursorManager normal change.

showConnection
    "Show the selected connection. Presently merely
    prompts with the expression"

    | newExpression |

    newExpression:= Prompter prompt: 'Connection expression'
        default: selectedObject expression.
    newExpression isNil
        ifFalse: (selectedObject expression: newExpression].!

showConnectors
    "Show the connectors of the selected object by making
    them the selected objects"

    | connectors |

    connectors:=OrderedCollection new.

    (selectedObject isKindOf: ThreeDObject)
    ifTrue:[
        self deShowSelectedObjects.
        browsedClass connections do:[each]
            (((each fromObject) == selectedObject) or:

```

```

    [(each toObject) == selectedObject])
    ifTrue: [connectors add: each]].
(connectors isEmpty)
ifFalse: [ selectedObject:= connectors.
    self showSelectedObjects]]!

```

showLaws

"Display (pop up a window) the laws applying to the selected object"

```

| laws logger extent aString selectors |

```

```

selectedObject isNil

```

```

ifFalse:[

```

```

    aString := 'Laws of Selected Object'.
    extent := ((Display width // 3 max:
        aString size * LabelFont width + 6) min:
        Display width) @ (Display height // 2).
    logger := TextEditor
        windowLabeled: aString
        frame: (Display extent x - extent x @ 0
            extent: extent).

```

```

    CursorManager execute change.

```

```

    laws:= selectedObject allLaws.

```

```

    laws do: [:each |

```

```

        logger nextPutAll:
            (each asString);
        cr.].

```

```

    logger positionAtBeginning.

```

```

    CursorManager normal change.

```

```

    logger topDispatcher scheduleWindow].!

```

showSelectedObjects

"Highlight the selected object or objects by displaying them with a thicker pen nib"

```

| currentCamera |

```

```

selectedObject isNil

```

```

ifFalse:[

```

```

    currentCamera:= viewCamera.

```

```

    (currentCamera twoDPen) defaultNib: 2@2; black.

```

```

    (selectedObject isKindOf: OrderedCollection)

```

```

    ifTrue:[

```

```

        selectedObject do: [:each |
            each displayWith: currentCamera]]

```



```

    ifFalse:[ selectedObject displayWith:
               currentCamera].
    animationPane showWindow].!

tick
    "Run the animation for 1 frame"

    CursorManager execute change.
    browsedClass nextFrame.
    self changed: #initializeGraphPane:.
    CursorManager normal change.!

ticksPerSecond
    "Show/Set the number of clock ticks per second"

    | newValue |

    newValue:= (Prompter prompt: 'Clock ticks per second (integer only)'
                      defaultExpression: (((browsedClass displayClock)
                      ticksPerSecond) printPaddedTo:0)).
    (newValue isKindOf: Integer)
    ifTrue:[ (browsedClass displayClock) ticksPerSecond: newValue].!

viewFromCamera
    "Show the view from the selected camera pen. Then allow the
    user to either revert to last view, or move view camera to
    the position and orientation of the selected camera"
    | aForm location theFrame |

    theFrame:= animationPane frame.
    location:= (theFrame origin x)
        @ ((theFrame origin y) + (theFrame extent y) - 10).
    CursorManager execute change.
    aForm:= animationPane form.
    depthActive
    ifTrue:[
        (depthForm = nil)
        ifTrue:[ depthForm:= (Form new) extent: (aForm extent).

            selectedObject displayOn: depthForm.
            selectedObject showDepthPlanes].
        aForm copy: (0@0 extent:(depthForm extent))
            from: depthForm to: 0@0 rule: (Form over).]
    ifFalse:[ aForm white].

displayAll
    ifTrue:[browsedClass displayAllOn: aForm withCamera: selectedObject]
    ifFalse:[browsedClass displayOn: aForm withCamera: selectedObject].
    animationPane showWindow.
    self displayTime.
    CursorManager normal change.

```

```

(Menu
  labels: 'set viewCamera to current orientation\revert to last orientation' withC15
  lines: Array new
  selectors: #(setViewCameraToSelectedCamera revertToViewCamera))
popUpAt: location for: self.

yawViewCamera
  "Yaw the view camera, that is, rotate it around y axis"

  | angle |

  angle := Prompter prompt: 'Angle to yaw?'
    defaultExpression: '90'.
  angle isNil
    ifFalse: [viewCamera yaw: angle.
      self refreshDisplay]!.

ClassBrowser subclass: #MotionLawBrowser
  instanceVariableNames:
    'theAnimationWorld'
  classVariableNames: "
  poolDictionaries: "
  comment:
    'This class implements animation law browsers'!

!MotionLawBrowser class methods!!

!MotionLawBrowser methods!

classUsageOf
  "Pop-up a window with the classes which use the
  animation law selected."
  | logger extent aString selectors |

  aString := 'Class Usage of ', selectedMethod printString, '.
  extent := ((Display width // 3 max:
    aString size * LabelFont width + 6) min:
    Display width) @ (Display height // 2).
  logger := TextEditor
    windowLabeled: aString
    frame: (Display extent x - extent x @ 0
      extent: extent).
  CursorManager execute change.

  (ThreeDObject withAllSubclasses) do: [:each |
    (each usesLaw: selectedMethod)
      ifTrue: [logger nextPutAll:
        (each name);
        cr.]].

```

```

logger positionAtBeginning.
CursorManager normal change.
logger topDispatcher scheduleWindow!

```

```
newLaw
```

```

"Private - Display the text for a new
law template in the text pane."

```

```

self
  changed: #selectors;
  selector: nil;
  changed: #text with: #homeCursor!

```

```
openOn: aClass
```

```

"Create a Motion Law browser window on aClass. Define
the type, behavior and relative size of each
pane and schedule the window."

```

```

| topPane |
(aClass isKindOf: Class)
  ifFalse: [^nil].
browsedClass := aClass.
topPane := TopPane new
  label: aClass name, ' | Animation Law Browser';
  minimumSize: SysFontWidth * 20
    @ (SysFontHeight * 8);
  yourself.
t := SysFontHeight * 2 + 4.

```

```
selectedDictionary := browsedClass class.
```

```
topPane addSubpane:
```

```

(ListPane new
  model: self;
  name: #selectors;
  change: #selector;;
  menu: #selectorMenu;
  framingBlock: [:box|
    box origin + (0@0) extent:
      box width // 5 @ (box height))).

```

```
topPane addSubpane:
```

```

(TextPane new
  model: self;
  name: #text;
  change: #accept:from;;
  framingBlock: [:box|
    box origin + ((box width // 5) @ 0)
    corner: box corner])).

```

```
topPane dispatcher open scheduleWindow!
```

```
openOn: aClass withWorld: anAnimationWorld
```

"Create a Motion Law browser window on aClass. Define the type, behavior and relative size of each pane and schedule the window. Need to know the animation world being used for various features, such as users of an animation law"

```

| topPane i |
(aClass isKindOf: Class)
  ifFalse: [^nil].
browsedClass := aClass.
theAnimationWorld := anAnimationWorld.
topPane := TopPane new
  label: aClass name, ' | Animation Law Browser';
  minimumSize: SysFontWidth * 20
    @ (SysFontHeight * 8);
  yourself.
i := SysFontHeight * 2 + 4.

selectedDictionary := browsedClass class.
topPane addSubpane:
  (ListPane new
    model: self;
    name: #selectors;
    change: #selector;;
    menu: #selectorMenu;
    framingBlock: [:box|
      box origin + (0@0) extent:
        box width // 5 @ (box height))).
topPane addSubpane:
  (TextPane new
    model: self;
    name: #text;
    change: #accept:from;;
    framingBlock: [:box|
      box origin + ((box width // 5) @ 0)
        corner: box corner)).

topPane dispatcher open scheduleWindow!

selectorMenu
  "Private - Answer the selector pane menu."
  ^Menu
    labels: 'new law\remove law\users\class usage' withCrs
    lines: Array new
    selectors: #(newLaw removeSelector usersOf classUsageOf)!

template
  "Private - Answer the
  template for animation laws."
  ^newLawName
  "comment and assumptions re connections"

```

```

| temporaries |

statements'!

text'
    "Private - Answer the source text for
    the selected method - gives template
    if selectedMethod is nil"
    | answer aClass |

    aClass := browsedClass class.
    selectedMethod isNil
        ifTrue: [^self template]
        ifFalse: [^aClass sourceCodeAt: selectedMethod]!.

usersOf
    "Pop-up a window with the users of the
    animation law selected."
    | instances logger extent aString selectors anAnimationLaw |
    anAnimationLaw := selectedMethod.
    aString := ' ', (Smalltalk keyAtValue: theAnimationWorld) asString,
        ' | Users of ', anAnimationLaw printString, ' '.
    extent := ((Display width // 3 max:
        aString size * LabelFont width + 6) min:
        Display width) @ (Display height // 2).
    logger := TextEditor
        windowLabeled: aString
        frame: (Display extent x - extent x @ 0
            extent: extent).
    CursorManager execute change.
    instances := theAnimationWorld animatedObjects.
    instances do: [:each |
        (each usesLaw: anAnimationLaw)
            ifTrue: [ logger nextPutAll:
                (each printString);
                cr.]].

    logger positionAtBeginning.
    CursorManager normal change.
    logger topDispatcher scheduleWindow! !

```

```

GraphPane subclass: #AnimationPane
instanceVariableNames: "
classVariableNames: "
poolDictionaries: "
comment:
  'Used only for animations - displays
  the model (an animation world) by asking it
  to display itself on this panes form' !

!AnimationPane class methods ! !

!AnimationPane methods !

defaultDispatcherClass
  "Answer AnimationDispatcher which is the
  default dispatcher of an AnimationPane."
  ^AnimationDispatcher!

graySelection
  "Private - window has been deactivated. I don't want
  to save the pane contents, as they are already in the
  backup form, and if I saved the contents, I'd get the
  image of the cursor, which I don't want.
  Therefore I'll do nothing but over ride the message"!

projectToScreen: aVector
  "Answer the screen coordinates of the 3D vector
  aVector. Performed by asking the model to
  project it into normal coordinates, i.e.
  0@0 as the center, and then transforming this
  into screen coordinates"

  | normPos |

  normPos:= model project: aVector.
  ^ ((frame center) + ((normPos x) @ (normPos y negated)))!

selectAtVector: aVector
  "Ask the model to select the object nearest the
  point defined by aVector. Note that aVector is in
  the active camera coordinates.
  Note: Temporarily a vector is returned from the model,
  and I set my dispatchers cursor to this vector"

  | vector |

  vector:= model selectAtVector: aVector.
  (vector == nil)
    ifFalse:[ dispatcher position: vector].!

```

```

showWindow
    "Draw the borders of the pane
    and copy the backup form to the pane."
    self border.
    paneScanner isNil
        ifTrue: [
            formHolder isNil ifTrue: [
                formHolder := model perform: name with: frame].
            self initTopCorner.
            paneScanner := BitBlt new
                combinationRule: Form over;
                extent: frame extent.
            self scrollBarInit]
        ifFalse: [
            (formHolder width < frame width or:
             formHolder height < frame height) ifTrue: [
                Display fill: frame
                clippingBox: frame
                rule: Form over
                mask: Form white].
            paneScanner
                sourceOrigin: topCorner;
                copyBits].
    dispatcher isNil ifFalse: [dispatcher resetBackForm].!

```

```

update
    "Ask the model to redisplay
    itself on my form - if my form isn't nil"
    (formHolder = nil)
    ifTrue: [self showWindow]
    ifFalse: [model displayOn: formHolder.
              self showWindow]! !

```

```

Dispatcher subclass: #AnimationDispatcher
instanceVariableNames:
    'position backForm'
classVariableNames: ''
poolDictionaries:
    'CharacterConstants FunctionKeys'
comment:
    'A Dispatcher for animation panes. This dispatcher
    provides a three dimensional cursor on the animation pane,
    keeping track of its position in the variable position.
    The cursor is controlled from the keyboard, with the following
    layout:
        i (up) o (forward)
        j (left) k (right)
        n (backward) m (down)

```

There are 30 different cursor sizes to indicate depth, ranging

from 30x30 to 1x1. Presently they are linearly mapped to the depth (z coordinate) as follows: depth = 0, cursor = 30x30; depth = 10, cursor = 29x29; .. depth >= 290, cursor = 1x1.

When moving the cursor, the 3D position is updated, and this is projected into a 2D screen coordinate, where the cursor is displayed, and the actual mouse (although it is hidden) is set to this point as well. By setting the real mouse to this point, we can use the standard way of releasing control, that is when the cursor goes outside the animation pane.!!

!AnimationDispatcher class methods !!

!AnimationDispatcher methods !

cursorBackward

"Decrement the 3D cursors z position"

position := (position x) withy: (position y)
withz: (position z - 1)!

cursorBackward: amount

"Decrement the 3D cursors z position by amount"

position := (position x) withy: (position y)
withz: (position z - amount)!

cursorDown

"Decrement the 3D cursors y position"

position := (position x) withy: (position y - 1)
withz: (position z)!

cursorDown: amount

"Decrement the 3D cursors y position by amount"

position := (position x) withy: (position y - amount)
withz: (position z)!

cursorForward

"Increment the 3D cursors z position"

position := (position x) withy: (position y)
withz: (position z + 1)!

cursorForward: amount

"Increment the 3D cursors z position by amount"

position := (position x) withy: (position y)
withz: (position z + amount)!

cursorLeft

"Decrement the 3D cursors x position"

position := (position x - 1) withy: (position y)
withz: (position z)!

cursorLeft: amount

"Decrement the 3D cursors x position by amount"

position := (position x - amount) withy: (position y)
withz: (position z)!

cursorRight

"Increment the 3D cursors x position"

position := (position x + 1) withy: (position y)
withz: (position z)!

cursorRight: amount

"Increment the 3D cursors x position by amount"

position := (position x + amount) withy: (position y)
withz: (position z)!

cursorUp

"Increment the 3D cursors y position"

position := (position x) withy: (position y + 1)
withz: (position z)!

cursorUp: amount

"Increment the 3D cursors y position by amount"

position := (position x) withy: (position y + amount)
withz: (position z)!

getPoint

"Continue the main process loop, answering the position
when the select function is issued. Only stops for select
function"

oldPos delta lastPos temp cursorForm lastChar

Cursor hide.

```

lastPos:=position.
temp:= self positionProjectedToScreen.
oldPos:= temp.

cursorForm:= CursorForms3 at:
    (((30 - ((position z rounded)//10))
    max: 1)
    min: 30).
backForm isNil
    ifFalse:[backForm display.
backForm:= Form fromDisplay: (oldPos extent: (cursorForm extent)).
cursorForm displayAt: temp rule: 1.
Cursor offset: temp.

[(lastChar:= Terminal read) == EndSelectFunction]
whileFalse: [

    self specialProcessKey: lastChar.
    delta := (Cursor offset) - oldPos.
    delta := (delta x) @ (delta y negated).
    (delta = (0@0))
    ifFalse:[ position:= ((position x) + (delta x))
        withy: ((position y) + (delta y))
        withz: (position z)].
    (position = lastPos)
    ifFalse:[ backForm display.
    temp:= self positionProjectedToScreen.

    cursorForm:= CursorForms3 at:
        (((30 - ((position z rounded)//10))
        max: 1)
        min: 30).

    Cursor offset: temp.
    oldPos:= temp.
    lastPos := position.

    backForm:= Form fromDisplay: (oldPos
        extent:(cursorForm extent)).
    cursorForm displayAt: temp rule: 1. ].
Cursor display.
^position!

position
"Answer my 3D position - its in the active cameras
coordinate system"

^position!

position: aVector

```

"Set my 3D position - its in the active cameras
coordinate system"

position:= aVector!

positionProjectedToScreen

"Answer my 3D position projected into 2D and transformed
into screen coordinates - this is where the cursor will be
on the screen. Implemented by asking my animation pane to
do it"

^(pane projectToScreen: position) rounded.!

processFunctionKey: aCharacter

"Private - Process the keyboard function keys
and mouse events represented by aCharacter."

aCharacter == SetLoc ifTrue: [^self].

(Cursor cursorKey: aCharacter) isNil

ifFalse: [^self].

aCharacter == EndSelectFunction

ifTrue: [

self select.

^self].

aCharacter == SelectFunction ifTrue: [^self].

aCharacter == CycleFunction ifTrue: [^self cycle].

aCharacter == PaneCycleFunction ifTrue: [^self cyclePane].

WindowMenuRequest == aCharacter

ifTrue: [^pane topPane performMenu].

PaneMenuRequest == aCharacter

ifTrue: [

Cursor display.

pane performMenu.

^Cursor hide].

ControlBreak == aCharacter

ifTrue: [^self].

self doesNotHandle!

processInput

"Private - Activate the receiver pane.

Read and process user input as long
as the cursor remains inside the pane.

This is the main processing loop for
the active pane. Uses save of back form method
for displaying cursor"

| temp oldPos cursorForm lastPos delta entryPoint |

pane activatePane.

Cursor hide.

```

entryPoint:= (Cursor offset) - (pane frame center).
entryPoint:= (entryPoint x) @ ((entryPoint y) negated).
position:=
  ((entryPoint x))
  withy: ((entryPoint y) )
  withz: (pane model camera focalLength).
lastPos:=position.
temp:= self positionProjectedToScreen.
oldPos:= temp.

cursorForm:= CursorForms3 at:
  (((30 - ((position z rounded)//10))
    max: 1)
    min: 30).
backForm isNil
  ifFalse:[backForm display].
backForm:= Form fromDisplay: (oldPos extent: (cursorForm extent)).
cursorForm displayAt: temp rule: 1.
Cursor offset: temp.

[self isControlActive]
  whileTrue: [

    self processKey: Terminal read.

    delta := (Cursor offset) - oldPos.
    delta := (delta x) @ (delta y negated).
    (delta = (0@0))
    ifFalse:[ position:= ((position x) + (delta x))
              withy: ((position y) + (delta y))
              withz: (position z)].
    (position = lastPos)
    ifFalse:[ backForm isNil ifFalse:[backForm display].
              temp:= self positionProjectedToScreen.

              cursorForm:= CursorForms3 at:
                (((30 - ((position z rounded)//10))
                  max: 1)
                  min: 30).

              Cursor offset: temp.
              oldPos:= temp.
              lastPos := position.

              backForm:= Form fromDisplay: (oldPos
                extent:(cursorForm extent)).
              cursorForm displayAt: temp rule: 1. ]].

    backForm isNil ifFalse:[backForm display].
    pane deactivatePane.
  ]

```

Cursor display!

processInputKey: aChar

"Process the input key aChar. Remember that

i = up; m = down; j = left; k = right; o = forward;

and n = backward

Also: up, down, left and right are mapped to the real mouse, and forward and backward are mapped to '.' and ',' respectively (on most keyboards the shift of these are

'<' and '>', which can be thought of as moving away and moving closer. The shifted forward and backward, '<' and '>', provide

faster depth movement. They move the cursor 10 times the normal amount"

"forward"

(aChar = \$.)

ifTrue:[^self cursorForward].

"backward"

(aChar = \$.)

ifTrue:[^self cursorBackward].

"fast forward"

(aChar = \$>)

ifTrue:[^self cursorForward:10].

"fast backward"

(aChar = \$<)

ifTrue:[^self cursorBackward: 10].

"up"

((aChar = \$i) or: [aChar = \$I])

ifTrue:[^self cursorUp].

"down"

((aChar = \$m) or: [aChar = \$M])

ifTrue:[^self cursorDown].

"left"

((aChar = \$j) or: [aChar = \$J])

ifTrue:[^self cursorLeft].

"right"

((aChar = \$k) or: [aChar = \$K])

ifTrue:[^self cursorRight].

"forward"

((aChar = \$o) or: [aChar = \$O])

ifTrue:[^self cursorForward].

"backward"

((aChar = \$n) or: [aChar = \$N])

ifTrue:[^self cursorBackward].

"none of the above - invalid"

self doesNotHandle!

resetBackForm

"Set my backFrom to nil"

backForm:=nil!

saveFormprocessInput

"Private - Activate the receiver pane.
Read and process user input as long
as the cursor remains inside the pane.
This is the main processing loop for
the active pane. Uses save of back form method
for displaying cursor"

| temp oldPos cursorForm lastPos delta |
pane activatePane.
Cursor hide.

position:= 0 withy: 0 withz: 628/3.
lastPos:=position.
temp:= self positionProjectedToScreen.
oldPos:= temp.
cursorForm := CursorForms3 at: 15.
backForm isNil
ifFalse:[backForm display].
backForm:= Form fromDisplay: (oldPos extent: (cursorForm extent)).
cursorForm displayAt: temp rule: 1.
Cursor offset: temp.

[self isControlActive]
whileTrue: [

self processKey: Terminal read.

delta := (Cursor offset) - oldPos.
delta := (delta x) @ (delta y negated).
(delta = (0@0))
ifFalse:[position:= ((position x) + (delta x))
withy: ((position y) + (delta y))
withz: (position z)].

(position = lastPos)
ifFalse:[backForm display.
temp:= self positionProjectedToScreen.

cursorForm:= CursorForms3 at:
(((30 - ((position z rounded)//10))
max: 1)
min: 30).

Cursor offset: temp.
oldPos:= temp.
lastPos := position.

backForm:= Form fromDisplay: (oldPos
extent:(cursorForm extent)).
cursorForm displayAt: temp rule: 1.]].

```
backForm display.
pane deactivatePane.
Cursor display!
```

```
select
  "Inform the pane that the select function has
  been issued and give it the current 3D position
  of the cursor"
```

```
pane selectAtVector: position.!
```

```
specialProcessFunctionKey: aCharacter
  "Private - Process the keyboard function keys
  and mouse events represented by aCharacter.
  Special - only allows for mouse movement and selection"
```

```
aCharacter == SetLoc    ifTrue: [^self].
(Cursor cursorKey: aCharacter) isNil
  ifFalse: [^self].
aCharacter == SelectFunction
  ifTrue: [^self].
aCharacter == EndSelectFunction ifTrue: [^self].
```

```
ControlBreak == aCharacter
  ifTrue: [^self].
  self doesNotHandle!
```

```
specialProcessKey: aCharacter
  "Private - Categorize the input aCharacter
  and invoke the appropriate method to
  process aCharacter. Special method - does not
  allow for most functions - only movement and select"
```

```
ClockEvent ifTrue: [^self processClock: aCharacter].
(MouseEvent or: [FunctionKey])
  ifTrue: [^self specialProcessFunctionKey: aCharacter].
(aCharacter between: Space and: $)
  ifTrue: [self processInputKey: aCharacter]
  ifFalse: [self processControlKey: aCharacter]!
```

```
xORprocessInput
  "Private - Activate the receiver pane.
  Read and process user input as long
  as the cursor remains inside the pane.
  This is the main processing loop for
  the active pane. Uses XOR method of displaying
```

```

cursor"

| temp oldPos cursorForm lastPos delta |
pane activatePane.
Cursor hide.

position:= 0 with: 0 with: 628/3.
lastPos:=position.
temp:= self positionProjectedToScreen.
oldPos:= temp.
cursorForm := CursorForms3 at: 15.
cursorForm displayAt: temp rule: 6.
Cursor offset: temp.

[self isControlActive]
whileTrue: [

    self processKey: Terminal read.

    delta := (Cursor offset) - oldPos.
    delta := (delta x) @ (delta y negated).
    (delta = (0@0))
    ifFalse:[ position:= ((position x) + (delta x))
              with: ((position y) + (delta y))
              with: (position z)].
    (position = lastPos)
    ifFalse:[ cursorForm displayAt: oldPos rule: 6.
            temp:= self positionProjectedToScreen.

            cursorForm:= CursorForms3 at:
                (((30 - ((position z rounded)//10))
                 max: 1)
                 min: 30).

            Cursor offset: temp.
            oldPos:= temp.
            lastPos := position.
            cursorForm displayAt: temp rule: 6]].

cursorForm displayAt: oldPos rule: 6.
pane deactivatePane.
Cursor display! !

```

```

Menu subclass: #SelectorMenu
instanceVariableNames: "
classVariableNames: "
poolDictionaries:
'CharacterConstants FunctionKeys'
comment:
'A combination of a selector list and a menu.
The labels are usually objects to be selected

```


from. The objects name are displayed as a menu
and the selected object is returned. No selection
results in a response of nil' !

'SelectorMenu class methods !

```
classCollection: anOrderedCollection lines: lineArray
    "Answer a menu with anOrderedCollection for the items,
    which is a collection of class's,
    and lines drawn
    under the item numbers contained in lineArray."
    llabeledArray l
```

```
    labeledArray:= Array new: (anOrderedCollection size).
    1 to: (labeledArray size) do:[i l
        labeledArray at: i put:
            ((anOrderedCollection at: i) symbol)].
```

```
    ^self new
        labels: labeledArray lines: lineArray;
        selectors: labeledArray;
        yourself!
```

```
labeledArray: labeledArray lines: lineArray
    "Answer a menu with labeledArray for the items,
    and lines drawn
    under the item numbers contained in lineArray." 2
```

```
    ^self new
        labels: labeledArray lines: lineArray;
        selectors: labeledArray;
        yourself!
```

```
labels: aString lines: lineArray
    "Answer a selector menu with aString for the items
    and lines drawn under the item numbers contained
    in lineArray."
```

```
    l anArray aStream l
    anArray := aString asArrayOfSubstrings.
    ^self new
        labels: anArray lines: lineArray;
        selectors: anArray;
        yourself!
```

```
selectorObjects: anOrderedCollection lines: lineArray
    "Answer a selector menu with anOrderedCollection
    for the objects to be selected from, and lines
    drawn under the item numbers contained in lineArray."
    llabeledArray l
```

```
    labeledArray:= Array new: (anOrderedCollection size).
    1 to: (labeledArray size) do:[i l
```

```

labelArray at: i put:
(Smalltalk keyAtValue:
(anOrderedCollection at: i))).

```

```

^self new
  labels: labelArray lines: lineArray;
  selectors: labelArray;
  yourself!

```

```

selectorObjects: anOrderedCollection lines: lineArray title: aTitleString

```

```

"Answer a selector menu with anOrderedCollection
for the objects to be selected from, and lines
drawn under the item numbers contained in lineArray."

```

```

labelArray secLine 1

```

```

secLine:=String new: (aTitleString size).
secLine replaceFrom: 1 to: (aTitleString size) withObject: S~.
secLine:= secLine trimBlanks.
labelArray:= Array new: ((anOrderedCollection size) + 2).
labelArray at: 1 put: aTitleString; at:2 put: secLine.
3 to: (labelArray size) do:[i]
  labelArray at: i put:
  (Smalltalk keyAtValue:
  (anOrderedCollection at: (i - 2))).

```

```

^self new
  labels: labelArray lines: lineArray;
  selectors: labelArray;
  yourself!

```

```

!SelectorMenu methods!

```

```

displayAt: aPoint

```

```

"Private - Display the receiver at aPoint
and save the image underneath."

```

```

| aRectangle w h newLoc |

```

```

currentLine := nil.

```

```

w := SysFontWidth.

```

```

h := SysFontHeight.

```

```

aRectangle :=

```

```

(aPoint x -

```

```

(MouseEvent ifTrue: [popUpForm width // 2]

```

```

ifFalse: [0])) @

```

```

(aPoint y - (popUpForm height // 2))

```

```

extent: popUpForm extent.

```

```

aRectangle corner < Display extent

```

```

ifFalse: [

```

```

aRectangle corner:

```

```

(Display extent -

```

```

        (w@h) min: aRectangle corner)).
aRectangle origin >= (0 @ 0)
  ifFalse: [aRectangle moveTo:
    (0 @ 0 max: aRectangle origin)].
offset := aRectangle origin + 2.
hiddenArea := Form fromDisplay: aRectangle.
newLoc := (offset x + (frame x // 2))
  @ (offset y +
    (selectors size // 2 * SysFontHeight +
    SysFont basePoint y)).
MouseEvent
  ifFalse: [newLoc x: offset x].
Cursor offset: newLoc.
popUpForm displayAt: aRectangle origin.
self reverseLine: 1.

```

```

popUpAt: aPoint
  "Pop up menu at aPoint, give it control, and
  answer the user response or nil if no response."
  [aCharacter aCursor]
  aCursor := Cursor.
  CursorManager normal change.
  priorCursor := Cursor offset.
  self displayAt: aPoint.
  [true] whileTrue: [
    self showCurrentLine.
    aCharacter := self moveCursor: Terminal read.
    (aCharacter == EndSelectFunction or:
    [aCharacter == SelectFunction])
    ifTrue: [
      self disappear.
      aCursor change.
      * currentLine isNil
      ifTrue: {^nil}
      ifFalse: [
        Cursor offset: priorCursor.
        ^selectors at: currentLine]].
    aCharacter == SetLoc
    ifFalse: [
      (self containsPoint: Cursor offset)
      ifFalse: [
        aCursor change.
        self disappear.
        Cursor offset: priorCursor.
        ^nil]]].

```

```

popUpAt: aPoint for: anObject
  "Pop up menu at aPoint, give it control,
  and return the selected object"
  [answer]

```

```

answer:= (self popUpAt: aPoint).
((answer = nil) | (answer = (selectors at: 1))) |
  (answer = (selectors at: 2)))
ifFalse:[^(Smalltalk at: answer)].
^nil!

```

```

popUpReturningObjectAt: aPoint
  "Pop up menu at aPoint, give it control,
  and return the selected object"
  | answer |
  answer:= (self popUpAt: aPoint).
  ((answer = nil) | (answer = (selectors at: 1))) |
    (answer = (selectors at: 2)))
  ifFalse:[^(Smalltalk at: answer)].
  ^nil!

```

```

showCurrentLine
  "Private - Reverse the line
  that the cursor is on."
  | index aPoint |
  aPoint := Cursor offset.
  (self containsPoint: aPoint)
  ifFalse: [
    (currentLine = nil)
    ifFalse:[
      (currentLine > 2)
      ifTrue:[
        self reverseLine: currentLine].
      currentLine := nil].
    ^self].
  index := aPoint y - offset y
  // SysFontHeight + 1.
  index = currentLine
  ifFalse: [
    (((currentLine = nil) not) and: [currentLine > 2])
    ifTrue:[
      self reverseLine: currentLine].
    index > 2
    ifTrue:[
      self reverseLine: index].
    currentLine := index]! !

```

```

ListPane subclass: #SelectorListPane
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  comment:
    'Combination of a list pane and a menu -
    displays like a list pane, but on a
    selection, it calls the models method
    corresponding to the selection' !

```

!SelectorListPane class methods !!

!SelectorListPane methods !

selectAtCursor

```

    "Set currentLine to the line
    at the cursor position."
    "select the current line - call the method
    of the model corresponding to the selection"
    self findCurrentLine.
    currentLine isNil
    ifTrue: [^self]
    ifFalse: {
        currentLine > list size
        ifTrue: [^self]}.
    currentLine == selection
    ifFalse: {
        self normalLine: selection.
        self boldLine: currentLine.
        selection := currentLine}.
    model
    perform: (changeSelector at: selection).!!

```

Pen subclass: #NormalPen

instanceVariableNames: "

classVariableNames: "

poolDictionaries: "

comment:

'A pen with its coordinate system centered on the dest form' !

!NormalPen class methods !!

!NormalPen methods !

extent

```

    "Answer my extent. Since my origin is at
    the center of my destForm, it is 1/2 of
    my destForm's extent"

```

```

    ^(destForm extent)//2!

```

goto: aPoint

```

    "Move the receiver Pen to aPoint on NormalPen's coordinate
    system, where 0@0 is the center of the form
    on the destination Form."

```

! newPoint !

```

newPoint := (aPoint x) @ (destForm extent y - (aPoint y)).
newPoint := (newPoint x + (destForm center x)) @ (newPoint y - (destForm center y)).
downState
  ifTrue:
    [self
      drawLoopX: newPoint x - destX
      Y: newPoint y - destY].
self place: newPoint!

lineFrom: aPoint to: anotherPoint
  "Draw a line"
  (downState)
  ifTrue:{
    self up;
    goto: aPoint;
    down;
    goto: anotherPoint
  }
  ifFalse:[self goto: anotherPoint].! !

Object subclass: #ThreeDObject
  instanceVariableNames:
    'alterValues plusLaws minusLaws origin upVector rightVector normalVector headingVector velocity
    acceleration mass clock relativeTo'
  classVariableNames:
    'InheritedLaws NonInheritedLaws'
  poolDictionaries: ''
  comment:
    'Abstract class for animated objects, i.e.
    all animated objects must be a subclass of ThreeDObject.
    Every animated object, i.e. ThreeDObject has its own
    coordinate system, origin, velocity, mass, acceleration, and also
    knows the clock controlling it' !

!ThreeDObject class methods !

addAccessingMethods
  "Add accessing methods automatically for
  all instance variables - accesses the actual
  value - sets the alter value"

  | tempString |
  self instVarNames do: [:name |
    tempString := ((name, '\n ^', name) withCrs).
    self compile: tempString.
    (self compiledMethodAt: (name asSymbol))
      sourceString: tempString.
    tempString := ((name, ': new Value\n ',
      'alterValues at: #' , name,
      ' ifAbsent: { \n',

```

```

        alterValues at: #'name,'
        \ put: OrderedCollection new)),
        \ addLast: newValue') withCrs.
self compile: tempString.
(self compiledMethodAt: ((name,':') asSymbol))
  sourceString: tempString ].

tempString := ('initialize \',
  ' self addAccessingMethods')
  withCrs.
(self class) compile: tempString.
((self class) compiledMethodAt:
  ('initialize' asSymbol))
  sourceString: tempString.

addLaw: anAnimationLaw
  "Add anAnimationLaw to the class's
  collection of laws"
  (InheritedLaws at: self ifAbsent: [
    InheritedLaws at: self put: OrderedCollection new])
    add: anAnimationLaw!

addNonLaw: anAnimationLaw
  "Add anAnimationLaw to the class's
  collection of nonlaws"
  (NonInheritedLaws at: self ifAbsent: [
    NonInheritedLaws at: self put: OrderedCollection new])
    add: anAnimationLaw!

initialize

  self addAccessingMethods!

laws
  "Answer a collection of all laws of this class."
  ^InheritedLaws at: self ifAbsent: [^OrderedCollection new]!

new
  "Answer a new instance of ThreeDObject initialized"
  ^super new initialize!

nonLaws
  "Answer a collection of all nonLaws of this class."
  ^NonInheritedLaws at: self ifAbsent: [^OrderedCollection new]!

removeLaw: anAnimationLaw
  "Remove anAnimationLaw as a law of this class."
  | theLaws |
  (theLaws := InheritedLaws at: self ifAbsent: [^self])
    remove: anAnimationLaw.

```

```

theLaws isEmpty
  ifTrue: [InheritedLaws removeKey: self!]

removeNonLaw: anAnimationLaw
  "Remove anAnimationLaw as a non-law of this class."
  | theNonLaws |
  (theNonLaws := NonInheritedLaws at: self ifAbsent: {^self})
    remove: anAnimationLaw.
  theNonLaws isEmpty
    ifTrue: [NonInheritedLaws removeKey: self!]

resetLaws
  "Reset the inherited and non-inherited laws"

  InheritedLaws:= IdentityDictionary new.
  NonInheritedLaws := IdentityDictionary new.

usesLaw: anAnimationLaw
  "Answer true if the receiver class uses the
  animation law anAnimationLaw."

  | goodLaws badLaws currentClass |

  badLaws:= OrderedCollection new.
  currentClass:= self.

  [currentClass = Object] whileFalse:{
    (badLaws includes: anAnimationLaw)
      ifTrue:[^false].
    goodLaws:= InheritedLaws at: currentClass
      ifAbsent: [OrderedCollection new].
    (goodLaws includes: anAnimationLaw)
      ifTrue:[^true].
    badLaws addAllLast: (NonInheritedLaws at: currentClass
      ifAbsent:[OrderedCollection new]).
    currentClass:= currentClass superclass].
  ^false!!

!ThreeDObject methods !

addLaw: anAnimationLaw
  "Add the symbol anAnimationLaw to my plus laws"

  plusLaws addLast: anAnimationLaw!

addNonLaw: anAnimationLaw
  "Add the symbol anAnimationLaw to my minus laws"

  minusLaws addLast: anAnimationLaw!

```


allLaws

"Answer an Ordered collection of all the laws applicable to the receiver. All plus laws of the receiver plus all inherited laws of the receiver if they are not a non-inherited law or a minus law"

I currentClass laws nonLaws allLaws!

currentClass:= self class.
nonLaws := OrderedCollection new.
nonLaws addAllLast: minusLaws.
allLaws := OrderedCollection new.
allLaws addAllLast: plusLaws.

[currentClass = Object]
whileFalse: [
 laws:= InheritedLaws at: currentClass ifAbsent: [nil].
 (laws isNil)
 ifFalse: [
 laws do: [:each |
 (nonLaws includes: each)
 ifFalse:{allLaws add: each}]].
 nonLaws addAllLast:
 (~~NonInheritedLaws~~ at: currentClass
 ifAbsent:[OrderedCollection new]).
 currentClass:= currentClass superclass].
^allLaws!

alterValues

^alterValues!

alterValues: newValue

(alterValues at: #alterValues ifAbsent: [
 alterValues at: #alterValues
 put: OrderedCollection new])
addLast: newValue!

backward: distance

"Move myself backward by distance"

self origin: (origin + (normalVector * (distance negated)))!

clock

"Answer the clock that controls me"

^clock!

coordinateFor: aVector

"Answer the vector representing the world coordinate system aVector in my coordinate system"

^self coordinateForRelative:(relativeTo coordinateFor: aVector)!

coordinateForRelative: aVector

"Answer a vector which represents aVector, in the coordinate system I am relative to, in my coordinate system"

letmp newPosition 1

temp:= aVector + origin.

newPosition :=((temp dotProduct: rightVector)

withy:

(temp dotProduct: upVector)

withz:

(temp dotProduct: normalVector)).

^newPosition!

displaceBy: aVector

"Displace myself by a vector"

self origin: (origin + aVector)!

displacementFor: aVector

"Answer the vector representing the world coordinate system displacement aVector in my coordinate system"

^self displacementForRelative:(relativeTo displacementFor: aVector)!

displacementForRelative: aVector

"Answer a vector which represents aVector in the coordinate system I am relative to, in my coordinate system. Note that it is a displacement"

letmp newPosition 1

temp:= aVector.

newPosition :=((temp dotProduct: rightVector)

withy:

(temp dotProduct: upVector)

withz:

(temp dotProduct: normalVector)).

^newPosition!

displayWith: aCameraPen

"Every threeDObject must be able to display itself with a CameraPen"

^self subclassResponsibility!

distanceFrom: aVector

"Return the distance I am from aVector. This method returns the distance my origin is from aVector. More accurate distances may be obtained from subclass implementation. Assumes aVector is in the coordinate system I am relative to"

(aVector isKindOf: Vector)

ifFalse:[^self error:'parameter must be a Vector'].

^(origin distanceFrom: aVector)!

distanceFromObject: aThreeDObject

"Return the distance I am from aThreeDObject. This method returns the distance my origin is from aThreeDObject's origin. More accurate distances may be obtained from subclass implementation. Assumes aThreeDObject is relative to the same coordinate system I am"

(aThreeDObject isKindOf: ThreeDObject)

ifFalse:[self error:

Parameter must be a ThreeDObject'].

^(origin distanceFrom: (aThreeDObject origin))!

distanceFromVector: aVector

"Return the distance I am from aVector. This method returns the distance my origin is from aVector. More accurate distances may be obtained from subclass implementation. Assumes aVector is defined in the coordinate system I am relative to"

(aVector isKindOf: Vector)

ifFalse:[self error:'Parameter must be a Vector'].

^(origin distanceFrom: aVector)!

distanceFromWorldVector: aVector

"Return the distance I am from aVector. This method returns the distance my origin is from aVector. More accurate distances may be obtained from subclass implementation. Assumes aVector is defined in the world coordinate system"

(aVector isKindOf: Vector)

ifFalse:[self error:'Parameter must be a Vector'].

^((relativeTo worldCoordinateFor: origin) distanceFrom: aVector)!

down: distance

"Move myself down by distance"

self origin: (origin + (upVector * (distance negated)))!

```
forward: distance
    "Move myself forward by distance"

    self origin: (origin + (normalVector * distance)).

goto: aVector
    "Move my origin to a vector"

    self origin: aVector!

headingVector

^headingVector!

headingVector: newValue

(alterValues at: #headingVector ifAbsent: [
    alterValues at: #headingVector
    put: OrderedCollection new
    addLast: newValue!

initialize
    "Initialize myself so that my origin is at the world origin, with the
    major axes as my axes, and my heading is right, my velocity is
    (0,0,0)"

    origin := (0,0,0).
    upVector := (0,1,0).
    rightVector := (1,0,0).
    normalVector := (0,0,1).
    headingVector := (1,0,0).
    velocity := (0,0,0).
    plusLaws := OrderedCollection new.
    minusLaws := OrderedCollection new.
    alterValues := IdentityDictionary new.!

left: distance
    "Move my origin left by distance"

    self origin: (origin + (rightVector * (distance negated)))!

minusLaws

^minusLaws!

minusLaws: newValue

(alterValues at: #minusLaws ifAbsent: [
    alterValues at: #minusLaws
    put: OrderedCollection new])
```

3 of/de 3

Micro-D

```
addLast: newValue!  
  
nextLevelCoordinateFor: aVector  
    "Answer the next level (the system I'm relative to)  
    coordinate for aVector defined in my  
    coordinate system"  
  
    ^(( rightVector * (aVector xComponent))  
      + (upVector * (aVector yComponent))  
      + (normalVector * (aVector zComponent))) + origin!  
  
nextLevelDisplacementFor: aVector  
    "Return aVectors displacement in the coordinate system  
    that I am relative to"  
  
    ^(( rightVector * (aVector xComponent))  
      + (upVector * (aVector yComponent))  
      + (normalVector * (aVector zComponent)))!  
  
normalVector  
  
    ^normalVector!  
  
normalVector: newValue  
  
    (alterValues at: #normalVector ifAbsent: [  
      alterValues at: #normalVector  
      put: OrderedCollection new])  
    addLast: newValue!  
  
origin  
  
    ^origin!  
  
origin: newValue  
  
    (alterValues at: #origin ifAbsent: [  
      alterValues at: #origin  
      put: OrderedCollection new])  
    addLast: newValue!  
  
originAsWorld  
    "Answer my origin in world coordinates"  
  
    ^relativeTo worldCoordinateFor: origin!  
  
originInWorld: newValue  
    "Set the origin to be newValue, which is a world coordinate"  
  
    | temp |
```

```
temp := relativeTo coordinateFor: newValue.
```

```
(alterValues at: #origin ifAbsent: [
  alterValues at: #origin
  put: OrderedCollection new])
addLast: temp!
```

```
performAllLaws
```

```
"Perform all the laws applying to the receiver.
Starts by applying the laws of the receiver, then
going up through the class heirarchy, applying all
the laws that are applicable"
```

```
| currentClass laws nonLaws |
```

```
currentClass := self class.
nonLaws := OrderedCollection new.
nonLaws addAllLast: minusLaws.
```

```
AnimationLaws setAnimatedObject: self.
plusLaws do: [ :each |
  AnimationLaws perform: each].
```

```
[currentClass = Object]
  whileFalse: [
    laws := InheritedLaws at: currentClass ifAbsent: [nil].
    (laws isNil)
    ifFalse: [
      laws do: [ :each |
        (nonLaws includes: each)
        ifFalse: [ AnimationLaws perform: each ]]].
    nonLaws addAllLast:
      (NonInheritedLaws at: currentClass
        ifAbsent: [OrderedCollection new]).
    currentClass := currentClass superclass].
AnimationLaws setAnimatedObject: nil !
```

```
pitch: angle
```

```
"Pitch myself (my coordinate system) by angle (in degrees).
Rotates the x and z axis"
```

```
| temp |
```

```
temp = rightVector rotateWith: normalVector and: angle.
normalVector := normalVector rotateWith: (rightVector negated) and: angle
rightVector := temp!
```

```
plusLaws
```

^plusLaws!

plusLaws: newValue

```
(alterValues at: #plusLaws ifAbsent: [
  alterValues at: #plusLaws
  put: OrderedCollection new])
addLast: newValue!
```

relativeTo

"Answer the object that my coordinate system is relative to"

^relativeTo!

relativeTo: aThreeDObjectOrNil

"Set the object that my coordinate system is relative to. This can be aThreeDObject, or nil"

```
(alterValues at: #relativeTo ifAbsent: [
  alterValues at: #relativeTo
  put: OrderedCollection new])
addLast: aThreeDObjectOrNil!
```

relativeToCoordinateFor: aVector

"Answer the next level (the system I'm relative to) coordinate for aVector, which is defined in my coordinate system"

```
^ (( rightVector * (aVector xComponent))
  + (upVector * (aVector yComponent))
  + (normalVector * (aVector zComponent))) + origin!
```

removeLaw: anAnimatedLaw

"Remove the symbol anAnimatedLaw from my plus laws"

plusLaws remove: anAnimatedLaw ifAbsent: []!

removeNonLaw: anAnimatedLaw

"Remove the symbol anAnimatedLaw from my minus laws"

minusLaws remove: anAnimatedLaw ifAbsent: []!

resetAlterValues

"Reset the alter values dictionary"

alterValues := IdentityDictionary new!

resolveAlterValues

"Resolve the alter values to get the new values
This version simply sums all the alter values"


```

and averages them"

| sum values count |

count:= 0.
(self class allInstVarNames) do: [ :name |
    count:= count + 1.
    (alterValues includesKey: (name asSymbol))
    ifTrue: [ values:= alterValues at:(name asSymbol).
        sum:= values at: 1.
        2 to: (values size) do: [ :i |
            sum:= sum + (values at: i)].
        sum := sum / (values size).
        self instVarAt: count put: sum]].!

resolveAndResetAlterValues
    "Resolve the alter values to get the new values, then
    reset the alter values.
    This version simply sums all the alter values
    and averages them"

| sum values count |

count:= 0.
(self class allInstVarNames) do: [ :name |
    count:= count + 1.
    (alterValues includesKey: (name asSymbol))
    ifTrue: [ values:= alterValues at:(name asSymbol).
        sum:= values at: 1.
        2 to: (values size) do: [ :i |
            sum:= sum + (values at: i)].
        sum := sum / (values size).
        self instVarAt: count put: sum]]
    self resetAlterValues!

right: distance
    "Move my origin right by distance"

    self origin: (origin + (rightVector * distance))!

rightVector

^rightVector!

rightVector: newValue

(alterValues at: #rightVector ifAbsent: [
    alterValues at: #rightVector
    put: OrderedCollection new])
    addLast: newValue!

```

roll: angle

"Roll my coordinate system by angle (in degrees).

Rotates the y and z axis"

! temp !

temp := upVector rotateWith: normalVector and: angle.

normalVector := normalVector rotateWith: (upVector negated) and: angle.

upVector := temp!

set: aSymbol to: newValue

"Add newValue to the instance variable aSymbol's
alter value"

(alterValues at: aSymbol ifAbsent: [
alterValues at: aSymbol
put: OrderedCollection new])
addLast: newValue!

setClock: aClock

"Set the clock that controls me. If another already
controls me, remove me from his dependent objects"

clock isNil

ifFalse:[clock removeDependentObject: self].

clock:= aClock.

aClock addDependentObject: self!

setOrigin: aVector

"Set the origin to aVector"

origin:= aVector!

setOriginInWorld: aVector

"Set the origin to aVector, defined in world coordinates"

origin:= relativeTo coordinateFor: aVector!

setRelativeTo: aThreeDObjectOrNil

"Set the object that my coordinate system is
relative to. This can be aThreeDObject, or nil"

((aThreeDObjectOrNil isKindOf: ThreeDObject)

! (aThreeDObjectOrNil == nil))

ifFalse:[^self error:

"Can only be defined relative to a ThreeDObject or nil"].

relativeTo:= aThreeDObjectOrNil!

setVelocity: newValue

"Immediately set my velocity"

```
velocity:= newValue!  
  
setWorldOrigin: aVector  
    "Set the origin to aVector, defined in world coordinates"  
  
origin:= relativeTo coordinateFor: aVector!  
  
setWorldVelocity: newValue  
    "Immediately set my velocity to my equivalent for  
    the world coordinate newvalue"  
  
velocity:= self coordinateFor: newValue!  
  
up: distance  
    "Move myself up by distance"  
  
self origin: (origin + (upVector * distance))!  
  
update  
    "Update myself by performing all my behaviour laws, and  
    then resolving and resetting all the alter values for my  
    attributes"  
  
self performAllLaws; resolveAndResetAlterValues!  
  
update: aParameter  
    "Something has changed - update myself  
    accordingly - inform my dependents"  
  
self performAllLaws.  
self changed: #position!  
  
upVector  
  
^upVector!  
  
upVector: newValue  
  
(alterValues at: #upVector ifAbsent: [  
    alterValues at: #upVector  
    put: OrderedCollection new])  
    addLast: newValue!  
  
upVector: aVector rightVector: bVector normalVector: cVector  
    "Set my upVector, rightVector, and normalVector"
```

```

upVector := aVector.
rightVector := bVector.
normalVector := cVector.

```

usesLaw: anAnimationLaw

"Checks if the receiver uses the given animation law.
Starts by checking the laws of the receiver, then
going up through the class heirarchy, checking all
the laws that are applicable, until it is found. If
not found answers false"

```

| currentClass laws nonLaws |

```

```

currentClass := self class.
nonLaws := OrderedCollection new.
nonLaws addAllLast: minusLaws.

```

```

(plusesLaws includes: anAnimationLaw)
ifTrue: [^true].

```

```

[currentClass = Object]
whileFalse: [
    laws := InheritedLaws at: currentClass ifAbsent: [nil].
    (laws isNil)
    ifFalse: [
        (laws includes: anAnimationLaw)
        ifTrue: [
            (nonLaws includes: anAnimationLaw)
            ifFalse: [^true]].
        nonLaws addAllLast:
            (NonInheritedLaws at: currentClass
             ifAbsent: [OrderedCollection new]).
        currentClass := currentClass superclass].
    ^false!

```

```

velocity
    "Answer my velocity"

```

```

    ^velocity!

```

```

velocity: newValue.

```

```

(alterValues at: #velocity ifAbsent: [
    alterValues at: #velocity
    put: OrderedCollection new])
addLast: newValue!

```

worldCoordinateFor: aVector

"Answer the world coordinate for aVector defined in my
coordinate system. Uses a tail recursive method to step
through coordinate systems until the world system is

reached"

```
^relativeTo worldCoordinateFor:
  (self nextLevelCoordinateFor: aVector)!
```

worldDisplacementFor: aVector

"Return a Vector displacement in the world coordinate system
by using tail recursion to get down to the world system"

```
^relativeTo worldDisplacementFor:
  (self nextLevelDisplacementFor: aVector)!
```

worldDistanceFrom: aVector

"Return the distance I am from aVector. This
method returns the distance my origin is from
aVector. More accurate distances may be obtained
from subclass implementation. Assumes aVector is in
world coordinates"

```
(aVector isKindOf: Vector)
  ifFalse: [self error: 'parameter must be a Vector'].
^((relativeTo worldCoordinateFor: origin) distanceFrom: aVector)!
```

worldDistanceFromObject: aThreeDObject

"Return the distance I am from aThreeDObject. This
method returns the distance my origin is from
aThreeDObject's origin. More accurate distances may
be obtained from subclass implementation. This method
transforms coordinates to world coordinates before
computing the distance"

```
(aThreeDObject isKindOf: ThreeDObject)
  ifFalse: [self error:
    'Parameter must be a ThreeDObject']
^((relativeTo worldCoordinateFor: origin)
  distanceFrom:
    (aThreeDObject relativeTo worldCoordinateFor:
      (aThreeDObject origin))))!
```

worldOrigin: newValue

"Set the origin to be newValue, which is a world coordinate"

```
! temp !
```

```
temp := relativeTo coordinateFor: newValue.
```

```
(alterValues at: #origin ifAbsent: [
  alterValues at: #origin
  put: OrderedCollection new])
```

```
addLast: temp!

worldVelocity: newValue
    "set my velocity to my equivalent for the world
    coordinate newValue"

(alterValues at: #velocity ifAbsent: [
    alterValues at: #velocity
    put: OrderedCollection new])
addLast: (self coordinateFor: newValue)!

yaw: angle
    "Yaw my coordinate system by angle (in degrees).
    Rotates the x and y axis"

    | temp |

    temp := rightVector rotateWith: upVector and: angle.
    upVector := upVector rotateWith: (rightVector negated) and: angle.
    rightVector := temp! !
```

```

ThreeDObject subclass: #CameraPen
instanceVariableNames:
    'active focalLength twoDPen penPosition penHeading cotViewingAngleOverTwo tZero tOne
normalForm scaledForm scaleFactor name'
classVariableNames: "
poolDictionaries: "
comment:
    'This class represents a virtual camera, which has
    a three dimensional pen that objects can use to draw
    themselves. The pen accepts world coordinates, transforms
    them into its view coordinates, and clips lines before
    actually drawing them. As well, the camerapen has a
    representation similar to an animated meta object, and will display
    itself if it is not active (if it is active, we are viewing
    through it, so would not be able to see it)' !

```

```

!CameraPen class methods !

```

```

initialize

```

```

    self addAccessingMethods!

```

```

new

```

```

    "Answer a new CameraPen initialized -
    the initialize method is called by
    ThreeDObject"

```

```

    ^super new! !

```

```

!CameraPen methods !

```

```

active

```

```

    ^active!

```

```

clippedLineFrom: startPoint to: endPoint

```

```

    "Draw a clipped line from startPoint to endPoint -
    assumes the points are vectors defined in my right
    coordinate system"

```

```

    penPosition:= startPoint.

```

```

    self clipProjectAndDrawTo: endPoint.

```

```

    penPosition:= endPoint.!

```

```

clipProjectAndDrawTo: newPosition

```

```

    "PRIVATE - draw line from old pen position to
    newPosition. Provides clipping on the three dimensional
    line segment and displays the two dimensional

```

projection if it is visible - assume old pen position
and newPosition are in my right viewing pyramid coordinates"

| deltaX deltaY deltaZ startPoint endPoint |

tZero := 0.

tOne := 1.

deltaX := (newPosition x) - (penPosition x).

deltaZ := (newPosition z) - (penPosition z).

(self clipT: (deltaX negated - deltaZ)

with: ((penPosition x) + (penPosition z)))

ifTrue:

[(self clipT: (deltaX - deltaZ)

with: ((penPosition z) - (penPosition x)))

ifTrue:

[deltaY := (newPosition y) - (penPosition y).

(self clipT: ((deltaY negated) - deltaZ)

with: ((penPosition y) + (penPosition z)))

ifTrue:

[(self clipT: (deltaY - deltaZ)

with: ((penPosition z) - (penPosition y)))

ifTrue:

[(tOne < 1)

ifTrue:

[endPoint := ((penPosition x) +

(tOne * deltaX))

with: ((penPosition y) +

(tOne * deltaY))

withz: ((penPosition z) +

(tOne * deltaZ))

ifFalse: [endPoint := newPosition].

(tZero > 0)

ifTrue:

[startPoint := ((penPosition x) +

(tZero * deltaX))

with: ((penPosition y) +

(tZero * deltaY))

withz: ((penPosition z) +

(tZero * deltaZ))

ifFalse: [startPoint := penPosition].

twoDPen lineFrom:

((self project: startPoint) rounded) to:

((self project: endPoint) rounded).

]]]]!

clipT: p with: q

"PRIVATE - Test used in three dimensional
clipping - see Liang and Barskey 3D clipping"

| r accept |


```

accept:= true.
(p < 0)
ifTrue:
  [r:= q/p.
   (r > tOne)
   ifTrue:[accept:= false]
   ifFalse:
     [(r > tZero)
      ifTrue:[ tZero:= r].
     ]
  ]
ifFalse:
  [(p > 0)
   ifTrue:
     [r:= q/p.
      (r < tZero)
      ifTrue:[accept:=false]
      ifFalse:
        [(r < tOne)
         ifTrue:[tOne:= r].
        ]
      ]
   ifFalse:
     [(q < 0)
      ifTrue:[accept:=false].
      ]
  ].
^accept!

cotViewingAngleOverTwo

^cotViewingAngleOverTwo!

cotViewingAngleOverTwo: newValue

(alterValues at: #cotViewingAngleOverTwo ifAbsent: [
  alterValues at: #cotViewingAngleOverTwo
  put: OrderedCollection new])
addLast: newValue!

displacePenBy: aVector
  "Displace pen position by a vector. Since
  aVector is in world coordinates it must
  first be transformed to my right viewing coordinate system"

  | newPosition |

  newPosition:= penPosition +
    (self rightViewingDisplacementFor: aVector).
  self clipProjectAndDrawTo: newPosition.

```

```
penPosition:= newPosition!
```

```
displayOn: aForm
```

```
"Make my twoDPen's destForm be aForm"
```

```
twoDPen initPen: aForm;
```

```
combinationRule: (Form over).
```

```
self updateCotViewingAngle!
```

```
displayWith: aCameraPen
```

```
"Display myself with aCameraPen. My representation
```

```
is in form, so I must shrink it to the proper size
```

```
depending on depth, if the scaleFactor has changed.
```

```
NOTE: do not display myself if aCameraPen is myself"
```

```
| scale transformedPos viewPosition worldOrigin|
```

```
(aCameraPen == self)
```

```
ifTrue:[^nil].
```

```
worldOrigin:= relativeTo worldCoordinateFor: origin.
```

```
transformedPos:= aCameraPen coordinateFor: worldOrigin.
```

```
"If I'm behind the camera, don't display myself"
```

```
((transformedPos z) <= 0.0)
```

```
ifTrue:[^nil].
```

```
scale:= (1 / ((aCameraPen focalLength) / (transformedPos z))  
asFloat).
```

```
(scaleFactor = scale)
```

```
ifFalse:[ scaleFactor:=scale.
```

```
scaledForm:= normalForm shrink: (0@0 extent: normalForm extent)
```

```
by: scaleFactor @ scaleFactor].
```

```
viewPosition:= aCameraPen
```

```
rightViewingCoordinateFor: worldOrigin.
```

```
viewPosition:= (aCameraPen project: viewPosition) rounded.
```

```
viewPosition:=(aCameraPen twoDPen destForm center) +
```

```
(viewPosition x @ (viewPosition y negated)).
```

```
viewPosition:= viewPosition -
```

```
((scaledForm extent x // 2)@(scaledForm extent y // 3)).
```

```
((aCameraPen twoDPen mask) == (Form white))
```

```
ifTrue:[
```

```
scaledForm reverse.
```

```
aCameraPen twoDPen destForm
```

```
copy: (scaledForm boundingBox)
```

```

    from: scaledForm
    to: viewPosition
    rule: Form orRule,
    scaledForm reverse}
  ifFalse:[
    aCameraPen twoDPen destForm
    copy: (scaledForm boundingBox)
    from: scaledForm
    to: viewPosition
    rule: Form andRule].!

drawLineFrom: aVector to: bVector
  "Draw a line from aVector to bVector. Assume
  aVector and bVector are defined in world coordinates,
  so first transform them to my right viewing coordinate
  system, then clip the line to be drawn"

  self clippedLineFrom: (self rightViewingCoordinateFor: aVector)
    to: (self rightViewingCoordinateFor: bVector)!

drawPlaneAtDepth: aZValue
  "Draw the representation of the xy-plane at the
  given z coordinate"

  | box xOffset yOffset |

  xOffset:= twoDPen extent x.
  yOffset:= twoDPen extent y.
  box:= Box origin: ((origin x - xOffset)
    withy: (origin y - yOffset)
    withz: (origin z + focalLength))
    extent: ((xOffset * 2) withy: (yOffset * 2)
    withz: (aZValue - focalLength - (origin z))).
  box normalVector: normalVector;
  upVector: upVector;
  rightVector: rightVector.
  box displayWith: self!

focalLength

?focalLength!

focalLength: newValue

(alterValues at: #focalLength ifAbsent: [
  alterValues at: #focalLength
  put: OrderedCollection new])
  addLast: newValue!

form

```

```
"Answer my normal form"

^normalForm!

initialize
  "Initialize myself with focalLength 100"

  super initialize.
  twoDPen := NormalPen new.
  focalLength:= 100.0.
  penPosition := 0 withy:0 withz:0.
  penHeading:= rightVector
  twoDPen up; goto: 0@0; down.
  self off.
  self updateCoiewingAngle.
  self updateForm.!

name

^name!

name: aString

name:= aString.
self updateForm.!

normalForm

^normalForm!

off
  "Turn myself off, i.e. make myself inactive"

  active:= false!

on
  "Turn myself on, i.e. make myself active"

  active:= true!

penDown
  "Lower my pen"

  twoDPen down!

penGoTo: aVector
  "Move pen position to a vector - first transform
  aVector to my right viewing pyramid coordinates"

  ! newPosition !
```

```

newPosiuon:= self rightViewingCoordinateFor: aVector.
self clipProjectAndDrawTo: newPosition
penPosiuon:= newPosiuon!

```

```

penHeading

```

```

^penHeading!

```

```

penHeading: newValue

```

```

(alterValues at: #penHeading ifAbsent: [
  alterValues at: #penHeading
  put: OrderedCollection new])
addLast: newValue!

```

```

penPosiuon

```

```

^penPosiuon!

```

```

penPosiuon: newValue

```

```

(alterValues at: #penPosiuon ifAbsent: [
  alterValues at: #penPosiuon
  put: OrderedCollection new])
addLast: newValue!

```

```

penUp

```

```

"Raise my pen"

```

```

twoDPen up!

```

```

project: aVector

```

```

"Project the vector aVector and
return the 2D point - Assumes aVector is
in my right viewing pyramid coordinates"

```

```

(aVector = Vector nil)
ifFalse:[
  ^ (((aVector x) @ (aVector y))
    * (((twoDPen extent x) / (aVector z)) asFloat)))
ifTrue: {^ 0@0)!

```

```

quickProject: aVector

```

```

"Project the vector aVector and
return the 2D point - Assumes aVector is
in my coordinate system - not my right viewing
coordinate system"

```

```

(aVector = Vector nil)
ifFalse:[
  ^(((aVector x) @ (aVector y))
    * ((focalLength / (aVector z)) asFloat))
  ifTrue: [^ 0@0]!

rightViewingCoordinateFor: aVector
  "Return aVector as it would be defined in my
  right viewing coordinate system. Since it is in world
  coordinates, first transform it into the coordinate system
  that I'm relative to, then transform it to my right
  viewing coordinate system"

  |temp newPosition|

  temp:= relativeTo coordinateFor: aVector.
  temp:= temp - origin.
  newPosition :=(((temp dotProduct: rightVector) * cotViewingAngleOverTwo)
    withy:
      ((temp dotProduct: upVector) * cotViewingAngleOverTwo)
    withz:
      (temp dotProduct: normalVector)).
  ^newPosition!

rightViewingDisplacementFor: aVector
  "Return aVector as it would be defined in my
  right viewing coordinate system. Note: it is a
  displacement. First transform it to the coordinate
  system I am relative to, then transform it into my
  right viewing coordinate system"

  |temp newPosition|

  temp:= relativeTo displacementFor: aVector
  newPosition :=(((temp dotProduct: rightVector) * cotViewingAngleOverTwo)
    withy:
      ((temp dotProduct: upVector) * cotViewingAngleOverTwo)
    withz:
      (temp dotProduct: normalVector)).
  ^newPosition!

scaledForm

  ^scaledForm!

scaleFactor

  ^scaleFactor!

scaleFactor: newValue

```

```

(alterValues at: #scaleFactor ifAbsent: [
    alterValues at: #scaleFactor
    put: OrderedCollection new])
addLast: newValue!

showDepthPlanes
    "Draw the depth planes"

    | depth |

    depth := origin z + focalLength.
    twoDPen gray.
    (30 to: 300 by: 30) do: [:i |
        self drawPlaneAtDepth: (depth + i)].
    twoDPen black.!

tOne

    ^tOne!

twoDPen

    ^twoDPen!

twoDPen: newValue

    (alterValues at: #twoDPen ifAbsent: [
        alterValues at: #twoDPen
        put: OrderedCollection new])
    addLast: newValue!

tZero

    ^tZero!

updateCotViewingAngle
    "Update the instance variable which provides the
    transformation to a right viewing pyramid - Must
    be called whenever the focal length is changed - or the
    destination form of my twoDPen"

    cotViewingAngleOverTwo := 1.0 /
        ((twoDPen extent x)
         / focalLength).!

updateForm
    "Update my form from the global, StandardCameraPenForm.
    Add my name. Since this changes my form, I'll set
    scaleFactor to nil to ensure the change is immediate, i.e.
    the scaledForm will be updated on the next display"

```

```

| scanner |

normalForm:= Form new extent: 50@50.
normalForm copy: (0@0 extent: 50@50)
  from: StandardCameraPenForm
  to: 0@0
  rule: Form over.
scanner:= CharacterScanner new
  initialize: (0@35 extent: 50@15 )
  font: Font eightLine
  dest: normalForm.
name isNil
  ifTrue:[name:= '?'].
scanner display: name
  at: (25 - ((Font eightLine stringWidth: name)
    // 2))@0.
scaleFactor:=nil!

worldCoordinateForViewingCoordinate: aVector
  "Answer aVector, which is defined in my right viewing
  coordinate system,
  as it would be defined in the world coordinate system"

  ^relativeTo worldCoordinateFor:
    ((( rightVector * (aVector xComponent) / cotViewingAngleOverTwo)
      + (upVector * (aVector yComponent) / cotViewingAngleOverTwo)
      + (normalVector * (aVector zComponent))) + origin)!

zoom: aNumber
  "Increment my focalLength by aNumber. i.e. the distance between my eye
  and the view plane - this has a zooming effect"

  focalLength:= focalLength + aNumber.
  self updateCotViewingAngle! !

ThreeDObject subclass: #Box
  instanceVariableNames:
    'extent'
  classVariableNames: "
  poolDictionaries: "
  comment:
    'A wireframe cube that can be animated. Described by origin and extent, so it can be resized simply by
    altering the extent.' !

  'Box class methods !

  initialize

  self addAccessingMethods!

```


new

"Answer a Box initialized and given a default extent of
<50,50,50."

^super new setExtent: (50 withy: 50 withz: 50)!

origin: aVector extent: bVector

"Answer a Box that has aVector as
its origin and extent bVector defining its width, height
and depth"

^super new origin: aVector extent: bVector!

withx: aNumber withy: bNumber withz: cNumber

"Answer a Box that has width aNumber,
height bNumber, and depth cNumber"

^super new initialize: origin: (0 withy: 0 withz: 0)
extent: (aNumber withy: bNumber withz: cNumber)! !

!Box methods !

asVectors

"Answer the collection of displacement vectors that
describe myself, i.e. that can be used to display myself"

| theVectors |

theVectors := OrderedCollection new.

theVectors add: (0 withy: (extent yComponent) withz: 0);
add: ((extent xComponent) withy: 0 withz: 0);
add: (0 withy: (extent yComponent negated) withz: 0);
add: ((extent xComponent negated) withy: 0 withz: 0);
add: (0 withy: 0 withz: (extent zComponent));
add: (0 withy: (extent yComponent) withz: 0);
add: ((extent xComponent) withy: 0 withz: 0);
add: (0 withy: (extent yComponent negated) withz: 0);
add: ((extent xComponent negated) withy: 0 withz: 0);
add: ((extent xComponent) withy: 0 withz: 0);
add: (0 withy: 0 withz: (extent zComponent negated));
add: (0 withy: (extent yComponent) withz: 0);
add: (0 withy: 0 withz: (extent zComponent));
add: ((extent xComponent negated) withy: 0 withz: 0);
add: (0 withy: 0 withz: (extent zComponent negated)).

^theVectors!

displayWith: aCameraPen

"Display myself using aCameraPen :

First I must create the collection of displacement
vectors that describe myself, then transform each

to a world displacement vector, and finally ask
the CameraPen to displace its pen by each"

```
aCameraPen
  penUp;
  penGoTo: (relativeTo worldCoordinateFor: origin);
  penDown.
  (self asVectors) do: [ :each | aCameraPen displacePenBy:
    (self worldDisplacementFor: each)].
```

extent

```
^extent!
```

extent: newValue

```
(alterValues at: #extent ifAbsent: {
  alterValues at: #extent
  put: OrderedCollection new})
  addLast: newValue!
```

origin: aVector extent: bVector
"Initialize my vector representation"

```
origin := aVector.
extent := bVector.!
```

resize: aVector
"Resize myself using aVector. My new extent is
aVector in my coordinate system"

```
extent := self coordinateFor: aVector!
```

setExtent: newValue

```
extent := newValue! !
```

```
ThreeDObject subclass: #Sphere
  instanceVariableNames:
    'radius'
  classVariableNames: "
  poolDictionaries: "
  comment:
    'A basic sphere with a radius. Drawn simply as a
    circle' !
```

```
!Sphere class methods !
```

```
initialize
```

```

self addAccessingMethods!

new
    "Answer a sphere initialized and given a default
    radius of 50"

    ^super new setRadius: 50!

radius: aNumber origin: aVector
    "Create a sphere with radius aNumber and
    center at aVector"

    ^super new radius: aNumber; setOrigin: aVector! !

!Sphere methods !

displayWith: aCameraPen
    "Display myself with aCameraPen"

    |transformedOrigin|

    transformedOrigin:= aCameraPen coordinateFor:
        (relativeTo worldCoordinateFor: origin).

    aCameraPen penUp;

    penGoTo: (relativeTo worldCoordinateFor: origin);
    penDown.
    (aCameraPen twoDPen)
    ellipse:
        (((radius * ((aCameraPen focalLength) / (transformedOrigin z)))
          asFloat) rounded)
    aspect: 11/13

    !

distanceFrom: aVector
    "Answer the distance I am from the point described by the
    vector aVector. Since I am a sphere, this is the distance
    my center is from the point minus my radius. Assume aVector
    is defined in the coordinate system I am defined relative to"

    |distance|

    (aVector isKindOf: Vector)
    ifFalse:[^self error: 'Parameter must be a Vector'].
    distance:= (origin distanceFrom: aVector) - radius.
    (distance < 0)
    ifTrue:[^0].

```

```
^distance!  
  
radius  
  
^radius!  
  
radius: newValue  
  
(alterValues at: #radius ifAbsent: [  
    alterValues at: #radius  
    put: OrderedCollection new])  
    addLast: newValue!  
  
resize: aVector  
    "Resize myself according to aVector. Done by  
    finding the distance aVector is from my origin  
    and setting my radius to that distance"  
  
    radius:= aVector distanceFrom:  
        (relativeTo worldCoordinateFor: origin)!  
  
setRadius: aNumber  
    "Set my radius to aNumber"  
  
    radius:= aNumber!  
  
worldDistanceFrom: aVector  
    "Answer the distance I am from the point described by the  
    vector aVector. Since I am a sphere, this is the distance  
    my center is from the point minus my radius. Assume aVector  
    is defined in the world coordinate system"  
  
    | distance |  
  
    (aVector isKindOf: Vector)  
        ifFalse:[^self error: 'Parameter must be a Vector'.  
        distance:= ((relativeTo worldCoordinateFor: origin)  
            distanceFrom: aVector) - radius.  
        (distance < 0)  
            ifTrue:[^0].  
        ^distance! !
```

END

22-06-88

FIN