1987

Animator: an object-oriented approach

Stephen Kurtz

Rochester Institute of Technology
School of Computer Science and Technology

Animator
An Object-Oriented Approach
to the
Creation and Direction of Computer Animated Actors

by
Stephen Kurtz

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by: _____

Professor Guy Johnson

_____

Professor Peter Anderson

_____

Associate Professor Lawrence Coon

December 21, 1987

Title of Thesis: Animator : An Object-Oriented Approach to the Creation and Direction of Computer Animated Actors.

_____

Stephen Kurtz

1. Preliminary Information

1.1. Abstract

Object-oriented programming is presented as a paradigm for developing interactive systems for computer animation. Object types, evolved conceptually from graphics turtles, are implemented to provide the animator with familiar metaphors for the specification of motion in three-dimensional space. The intention is to create objects that can represent actors, cameras, and decor that the user can direct and animate in a relatively intuitive manner. Vector and turtle objects support the actors, which respond to messages to orient, accelerate and draw themselves on the screen. The MacApp object libraries are used to implement the standard Macintosh user interface and a unit is developed which implements vectors, actors and three-dimensional graphics turtles as objects. The object-oriented approach encourages a conceptual consistency in the external and internal interfaces and is intended to facilitate the development of extensible characters and tools through the cooperative efforts of animators and programmers.

1.2. Key Words and Phrases

Object-oriented programming, animation, three-dimensional, vector analysis, turtle graphics, actors, MacApp, interactive.

1.3. Table of Contents

2. Proposal

Although the real-time creation and display of animated sequences in three-dimensional space remains a problem for the future, a great deal has been done to assist the animator in creating animated films frame by frame. Currently, computers are used in the interactive creation of objects, the generation of successive images to simulate motion or other transformations, and the control of hardware in the production and post-production stages of making a film or video. This thesis will focus on the problems surrounding the specification of motion by the animator.

Many animators are interested in using the computer as a tool, but few are prepared to become programmers in the process. The resistance to the mathematical modeling of objects and motion is in part due to the nature of the work involved, but of equal importance for many animators is the conviction that traditional animation owes its appeal not to the accuracy with which it portrays the physical world, but rather to the expressiveness of the artist/animator. The goal for many is not to simulate the laws of motion but rather to simulate traditional animation. User-oriented paint systems do not as yet provide the animator with the tools to describe satisfying movement in a three-dimensional space.

I am particularly interested in working toward an interactive environment for the animator that is conceptually compatible with an extensible object-oriented programming language. The system would be modal, providing separate environments for creating different classes of objects along with the methods that class of objects might support. My intention is to give the animator direct control of the scene whenever possible, while offering the extensibility of a programming language in other modes. For example, a new class of objects might be created in one mode, while a new actor or instance of that class might be described in a second mode, and a script created for that actor in a third. The modification or creation of the new class might require the use of a computer language by the animator or a programming co-worker, while the other tasks might be accomplished more directly.

I hope to implement a system in order to explore methods of creating objects with parts that move relative to a local co-ordinate system while the entire object accelerates or decelerates in a three-dimensional space. Rendering will be of secondary importance, but

I would like to give the animator the ability to simulate camera movements and perhaps a variable light source. The emphasis will be on providing an intuitive interface and useful feedback for both programmed and interactive work.

I intend to use a version of Pascal with support for object-oriented programming. My current thought is that objects which can behave like three-dimensional graphics turtles can become actors in an animation script. I hope to allow the animator to express changes in position and rates of change relative to feedback about current position, heading and velocity, rather than relating these changes to a global coordinate system. To my knowledge, this is a novel approach to the specification of motion.

I would like to acknowledge my debt to Nadia and Daniel Thalmann, for the guidance and direction supplied by their book, *Computer Animation, Theory and Practice* [Thal 1985] and to Harold Ableson and Andrea diSessa, for their extraordinary text, *Turtle Geometry* [Abel 1981] which motivated this approach .

3. Introduction and Background

3.1. Computer Animation

The creation of animated pictures is a complex and labor intensive process in which the computer can play many roles. Subsequently the term "computer animation" is not well defined. Computers are used in the creation of drawings, the simulation of movement and in various stages of the production and editing process. Although the real-time computation and display of animated graphics is now possibile for simpler images, the emphasis here is on the use of computers to animate complex three-dimensional scenes that require a frame-by-frame approach.

Systems that aid in the drawing stage vary from paint systems that assist the artist in the creation of an image on the screen, to modeling systems which can represent three-dimensional solids as stored data and render stored objects to reflect different points of view, lighting and surface attributes. They also differ markedly in the skills they require of the user, which may vary from studio drawing ability to advanced programming techniques. Different parts of the process are often handled by different systems and assigned to different members of a production team.

In the creation of motion the computer is often used for in-betweening. In this process the animator creates a set of drawings called key frames, which depict an object at various stages during an animated sequence, and the computer creates the drawings that are required in-between the key frames. These systems are modeled after a division of labor that was often used in hand animation. Other systems can generate the frames required to show an image moving along a path described by the animator. Modeled animation systems often allow three-dimensional movements to be programmed by the animator, and then create the required sequence of frames from the stored information. Real-time playback of animated sequences is not yet possible for realistic three-dimensional animation, but many systems now offer real-time previewing or feedback with simplified (wire frame) images.

3.2. Object-Oriented Programming Systems (OOPS)

Computer animators are involved in the modeling of complex objects and the specification of their behavior. Recent work on data abstraction [Guttag 77] has had important implications for computer animation [Thal 83]. Many of the objects modeled by animators (i.e. actors, props, cameras, lights, backgrounds) are specific examples of classes of conceptually similar objects which interact with each other in a prescribed manner to create the scenes recorded on film. Object-oriented programming systems such as SMALLTALK [Gold 84] offer tools extraordinarily well suited to the graphics programmer.

SMALLTALK defines objects as entities consisting of private memory and a set of operations which communicate by sending messages. Each object is an instance of an abstract data type (class), binding a data structure with a set of operations (responses to messages) that constitute the only access to that data. Classes of objects can be customized by defining new classes which refine and extend the inherited interface and data structure of their declared ancestors.

The three concepts that characterize OOPS are: objects, messages and inheritance. Object-oriented programming encourages a design strategy that organizes programs around a hierarchy of abstract data types, rather than the refinement of procedural objectives. An object-oriented system evolves as a collection of objects, customized and refined by descendants, interacting by message sending. A version of Pascal [Appl 87] supporting these concepts was used to create the graphics objects in Animator.

3.3. Actor Based Systems and the Logo Turtle

The term "actor" [Hewitt 73] describes an object that can send and receive messages. Hewitt describes a system in which all elements are actors and the only activity is messages sent between them. Programming in this system consists of defining the responses made by different classes of actors to received messages.

Kenneth Kahn created a system called DIRECTOR [Kahn 76] based on an actor that could be "asked" to do all the things a LOGO turtle can do. LOGO [Papert 70] is a language developed at M.I.T. for use with children. The LOGO turtle responds to the commands: LEFT (n) and RIGHT(n) to change its heading, FORWARD(n) and BACK(n) to move, and PENUP and PENDOWN to determine whether a line should be drawn in its path. All commands are relative to the turtle's current location. Kahn's lisp-like language used the syntax ASK <actor name> (<message>) and built up more complex commands

with procedural definitions. Kahn animated scenes by synchronizing responses to a clock with control structures like AFTER 2 MORE TICKS.

Reynolds created the Actor/Scriptor Animation System (ASAS) at the Architecture Machine Group at M.I.T. [Reynolds 82]. ASAS messages are lisp-like functions which rotate an actor's local three-dimensional coordinate system. These messages extend the LOGO turtle's LEFT and RIGHT commands to include UP, DOWN, CLOCKWISE and COUNTER CLOCKWISE, so that his actors can be animated in three-dimensions. Reynolds describes his "actor" as a "chunk" of code that is executed once each frame.

Also at M.I.T, Abelson and diSessa described a similar three-dimensional turtle in their text, Turtle Geometry [Abel 81]. It was Abelson's description of a three-dimensional turtle and my experiences helping students learn to program with turtle graphics, that inspired Animator.

3.4. Animator

Animator's actors are a class of objects, with private information, that respond to messages and are refined through inheritance very much like the SMALLTALK objects described in section 3.2 above. They were developed before I learned about Hewitt's actors, and should not be confused with them. Animator's actors were designed to be animated or scripted interactively and easily extended through inheritance by a programmer. All actors, graphics turtles, and cameras descend from a common ancestor from which they inherit their general abilities to respond to messages to orient and move in space.

Their inherited data structures include a local coordinate system based on vector objects that respond to messages about vector arithmetic.

4. Vectors

4.1. Vectors: TVector Object Type

The TVector type implements the fundamental vector methods on which all of the graphics objects in Animator are based. The definition of vectors as an object type is more for the purpose of data abstraction than inheritance (the object type is itself childless), but all descendants of the TLCS (local coordinate system) type inherit TVector data fields. The descendants of TLCS include all of Animator's actors, turtles and cameras.

The vector representation of such quantities as position, distance, orientation, velocity and acceleration, is geometrically intuitive and analytically powerful. The fundamental vector methods described in this section support the drawing, moving and projection techniques used to specify and simulate motion in Animator. Where appropriate, a geometric interpretation will be offered. An Object Pascal [Appl 87] implementation of the TVector type is included at the end of this section.

4.2. Vectors: the data fields

A vector in three-dimensional space is an ordered triple of real numbers. They are represented by three data fields, fX, fY and fZ in the TVector object type. Early versions of Animator used MPW Pascal's FIXED type, which represents a real number in 4 bytes, using the first two bytes for the integer and the last two for the fractional part of a real number. Precision is adequate and storage is half that of the Pascal REAL type. Arithmetic with negative values was cumbersome and assignment was inconvenient. As Animator evolved, computational speed proved more valuable than storage space, and the current version benefits in this regard from using the MPW EXTENDED type to represent all real values. This 10 byte representation was chosen, not for the added precision, but because all representations of real numbers in the Standard Apple Numeric Environment are converted to this type before computation. [Appl 85]

4.3. Vectors: methods for assigning values

Vectors are represented as objects rather than as Pascal records to provide data encapsulation. Although Object Pascal permits data fields to be referenced and modified much as a programmer might assign a value to a field of a Pascal record, this is avoided in Animator. Vector data fields are altered only by calling the methods defined as part of the TVector object type. In this respect, TVector is a template for an abstract data type. Direct references to an object's data fields, which take the form aVector.fX, should be thought of as function calls that return the value of the referenced field.

An instance variable of type TVector is more like a Pascal pointer (it is actually a handle or double pointer) than a Pascal record. The NEW statement allocates space for the data fields and the shallow Free method, which all objects inherit from TObject, is sufficient to deallocate or dispose of the allocated memory. For the sake of the following discussion, assume that firstVector and secondVector are instance variables of type TVector for which memory has been allocated. Then the statements or messages:

```
firstVector.Init(20, 22.5,  25);

secondVector.CopyOf(firstVector);
```

would initialize the object firstVector to the triple <20, 22.5, 25> and make secondVector a copy of firstVector. Init and CopyOf are methods of the TVector object type, or messages that all objects of this type respond to. These methods are defined like Pascal procedures, but called in the style of a Pascal record reference, as above. It should be noted that the statement:

```
secondVector : =  firstVector;
```

would make secondVector a second pointer to the object already pointed to by firstVector, rather than a separate copy.

4.4. Vectors: a geometric interpretation

When interpreted as a directed line segment, vectors can be used to describe quantities that have both magnitude and direction, such as displacement, velocity, acceleration, or an edge in a wire frame model.

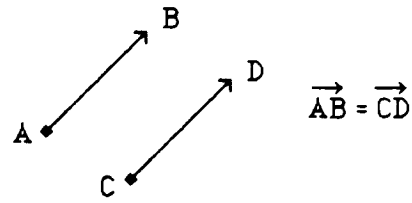


fig. 4.1

Since two vectors having the same direction and magnitude are equal even when their initial and terminal points differ, displacing a vector in space does not change its value as long as the new line segment is parallel to the original (see fig. 4.1). In Animator this property of vectors allows graphics objects and their motion to be described without reference to a particular coordinate system. In order to represent these quantities in the program and draw them on the screen, a correspondence must be established between vectors and the cartesian coordinates of three-dimensional space.

4.5. Vectors: vector coordinates

The cartesian coordinate system associates an ordered triple, (X, Y, Z), of real numbers with each point in three-dimensional space, where X is the directed distance of the point from the YZ-plane, Y is the the directed distance from the XZ-plane, and Z is the directed distance from the XY-plane. We can establish a 1 to 1 correspondence between the set of vector triples, <x, y, z>, and the points of three-dimensional space if we choose the cartesian coordinates (x, y, z) of any point to describe a corresponding vector A, whose magnitude and direction allow us to displace it in space, such that its initial point is at the origin (0, 0, 0), and its terminal point is the point whose cartesian coordinates are (x, y, z). Any vector displaying the required magnitude and direction is by definition equal to A.

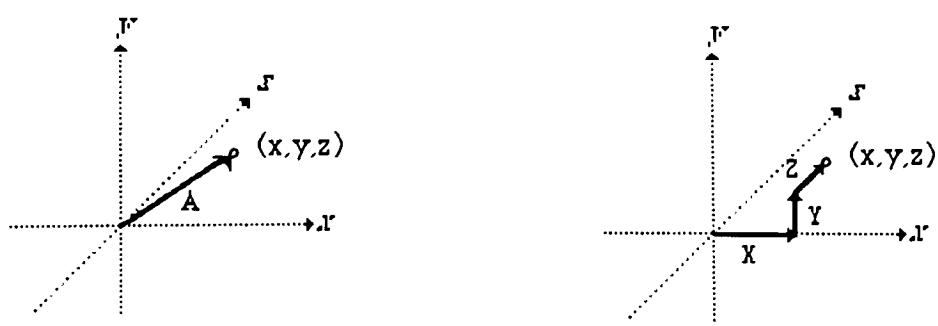


fig. 4.2

4.6. Vectors: the geometry of vector addition

Let A and B be vectors. If B is displaced so that its initial point is the terminal point of A, the vector C from the initial point of A to the terminal point of B is the sum of A and B (fig. 4.3).

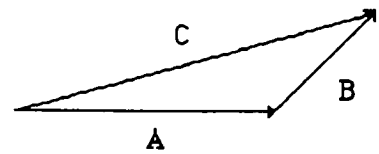


fig. 4.3

In section 4.4 we described each of the cartesian coordinates as directed distances from planes through the origin. Let X, Y and Z be vectors corresponding to these directed distances. For any point (x,y,z), the corresponding vector A is equal to the sum of the vectors X, Y and Z, which when placed end to end, will trace a path from the origin to the point (x,y,z). See fig. 4.2.

4.7. Vectors: the components fX, fY and fZ

Scalar Product geometric definition: Let A be a vector, and c be a scalar constant, then the scalar product cA is a vector with c times the magnitude of A. The direction of cA will be opposite that of A if c is negative and the same as the direction of A otherwise.

Let X, Y and Z be unit vectors (magnitude = 1) whose directions correspond to the positive x, y and z axis respectively. Let A be a position vector initiating from the origin and terminating at the point (x,y,z) as in the previous section. When we say that the vector A is represented by the triple <x,y,z> we mean that x, y and z are scalar coefficients of the three unit vectors above such that xX + yY + zZ = A. In this notation the vectors X, Y and Z are called a basis, and the scalar coefficients x, y and z are called the components of vector A. The data fields fX, fY and fX of TVector are vector components.

4.8. Vectors: implementation of vector addition

Let X, Y and Z be vectors and constitute a basis. Let A and B be vectors such that A = aX + bY +cZ and B = dX + eY + fZ where a,b,c,d,e and f are scalar constants. Then A +

B = aX + bY + cZ + dX + eY +fZ, or collecting like terms: A + B = (a + d)X + (b + e)Y + (c + f)Z.

The TVector object has two vector addition methods: SumOf and AddTo. If aVector, bVector and cVector are of type TVector then the statement : aVector.SumOf(bVector, cVector), assigns the sum of bVector and cVector to aVector.

The statement: aVector.AddTo(bVector), assigns the sum of aVector and bVector to aVector. Note that in either case only the object whose method is called (aVector) is modified.

4.9. Vectors: implementation of the scalar product

Let X, Y and Z be vectors and constitute a basis. Let k be a scalar and A be a vector such that A = aX + bY +cZ where a, b and c are scalar constants. Then kA = kaX + kbY + kcZ.

If theVector is an object of type TVector and k is scalar then the message or method call: theVector.Scale(k), will multiply each component of theVector by k.

Notes on scalar multiplication: If k = 1 then kA = A. If k = -1 then kA will be the additive inverse of A. If the magnitude of A is 1 then the magnitude of kA is k. Choosing X, Y, and Z as unit vectors parallel to their respective axis provides a basis such that the components <x,y,z> of a vector from the origin to a point correspond to the cartesian coordinates for that point.

4.10. Vectors: rotation of a vector in a plane

In a plane we can decompose any vector into a basis consisting of any two non-parallel vectors. Let V be a vector we wish to rotate ANG degrees. Let P be vector, of equal magnitude, perpendicular to V. If we use V and P as a basis for the vector we seek, V-rotated, then the required components are the lengths of the dotted lines in the figure below.

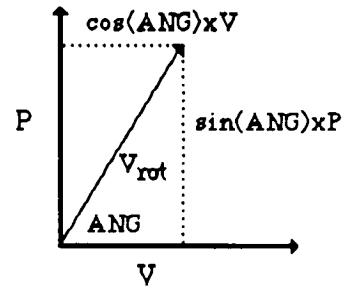


fig. 4.4

$$V_{rot} = (\cos(ANG))\ V + (\sin(ANG))\ P$$

The rotation method is equivalent to the vector sum of two scalar products. TVector objects respond to the method call or message: aVector.Rotate(angle, perpVector), by rotating aVector angle degrees in the plane determined by aVector and perpVector.

4.11. Vectors: the dot product

Let V = aX + bY + cZ and W = dX + eY + fZ, where X, Y, and Z are a basis and a,b,c,d,e, and f are scalar coefficients. Then the dot product of V and W is the scalar obtained by summing the products of their corresponding cefficients (1) V•W = (a x e) + (b x f) + (c x e).

If V and W are placed so that they have the same initial point then it can be shown, by the law of cosines, that the same result is obtained by the formula (2) V•W = |W| |V| cos(A), where A is the angle between V and W. The TVector.Dot function is implemented with formula (1), but the geometric interpretation (the projection of V on W multiplied by the magnitude of W) suggested by the second formula and fig. 4.5a motivates the projection methods used by the graphics turtle.

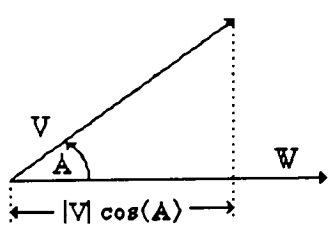


fig. 4.5a

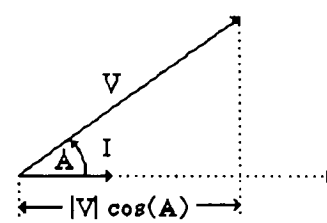


fig. 4.5b

Let W in formula (2) be a unit vector, then the magnitude of W equals 1 and we have: $V \cdot W = |W| |V| \cos(A) = |V| \cos(A)$, which is equal to the scalar projection of V on the extension of the vector W.

This result can be used to implement a change in basis for a vector. Let V be a Vector and I, J and K be non-parallel unit vectors. I, J and K constitute a basis. The coefficient for the I component of V is the magnitude of the projection of V on a line in the direction of I or $V \cdot I$ (see fig. 4.5b). The triple $\langle V \cdot I, V \cdot J, V \cdot K \rangle$ are the components of V in the I, J, K basis. The camera or viewpoint used in animator projections is a basis similar to I, J, K defined in terms of the X, Y, Z basis used for animator's global coordinate system.

4.12. Vectors: Implementation of the TVector Object Type

```
TVector = OBJECT (TObject)
        fX, fY, fZ: EXTENDED; {triple of floating point numbers}

                                {TVector methods}

        PROCEDURE TVector.Init(x, y, z: EXTENDED);
        PROCEDURE TVector.CopyOf(v: TVector);
        PROCEDURE TVector.Sumof(v1, v2: TVector);
        PROCEDURE TVector.AddTo(v: TVector);
        PROCEDURE TVector.Rotate(angle: EXTENDED; perp: TVector);
        PROCEDURE TVector.Scale(k: EXTENDED);
        FUNCTION TVector.DotWith(v: TVector): EXTENDED;
      END; {TVector}

                        {Implementation of Vector Methods}


PROCEDURE TVector.Init(x, y, z: EXTENDED);

{Initialize data fields fX, fY & fZ to x, y & z}

 BEGIN
  fX := x;
  fY := y;
  fZ := z
 END; {TVector.Init}


PROCEDURE TVector.CopyOf(v: TVector);

{Initialize data fields fX, fY & fZ to v.fX, v.fY, v.fZ}

 BEGIN
  fX := v.fX;
  fY := v.fY;
  fZ := v.fZ
 END; {TVector.CopyOf}


PROCEDURE TVector.Sumof(v1, v2: TVector);

{vector is assigned the sum of v1 and v2}

 BEGIN
  fX := v1.fX + v2.fX;
  fY := v1.fY + v2.fY;
  fZ := v1.fZ + v2.fZ
 END; {TVector.Sumof}
```

```
PROCEDURE TVector.AddTo(v: TVector);

{the vector v is added to the object vector}

 BEGIN
  fX := fX + v.fX;
  fY := fY + v.fY;
  fZ := fZ + v.fZ
 END; {TVector.AddTo}


PROCEDURE TVector.Scale(k: EXTENDED);

{vector is multiplied by the scalar k}

 BEGIN
  fX := fX * k;
  fY := fY * k;
  fZ := fZ * k
 END; {TVector.Scale}


PROCEDURE TVector.Rotate(angle: EXTENDED; perp: TVector);

{rotates vector angle degrees in the plane determined by vector
 and perp, a second vector perpendicular to the object vector}

 VAR
  c, s: EXTENDED;

 BEGIN
  angle := angle * gDEGRAD;
  c := Cos(angle);
  s := Sin(angle);
  Init(fX * c + perp.fX * s, fY * c + perp.fY * s, fZ * c + perp.fZ * s)
 END; {TVector.rotate}


FUNCTION TVector.DotWith(v: TVector): EXTENDED;

{return dot product of vector object with v}

 BEGIN
  DotWith := fX * v.fX + fY * v.fY + fZ * v.fZ
 END; {TVector.DotWith}
```

5. Local Coordinate Systems and the TLCS Object Type

5.1. Local Coordinate Systems: the TLCS Object Type

The TLCS object type is the ancestor of all of Animator's actor types as well as the graphics turtle. It provides movement and rotation methods for all of Animator's moving objects.

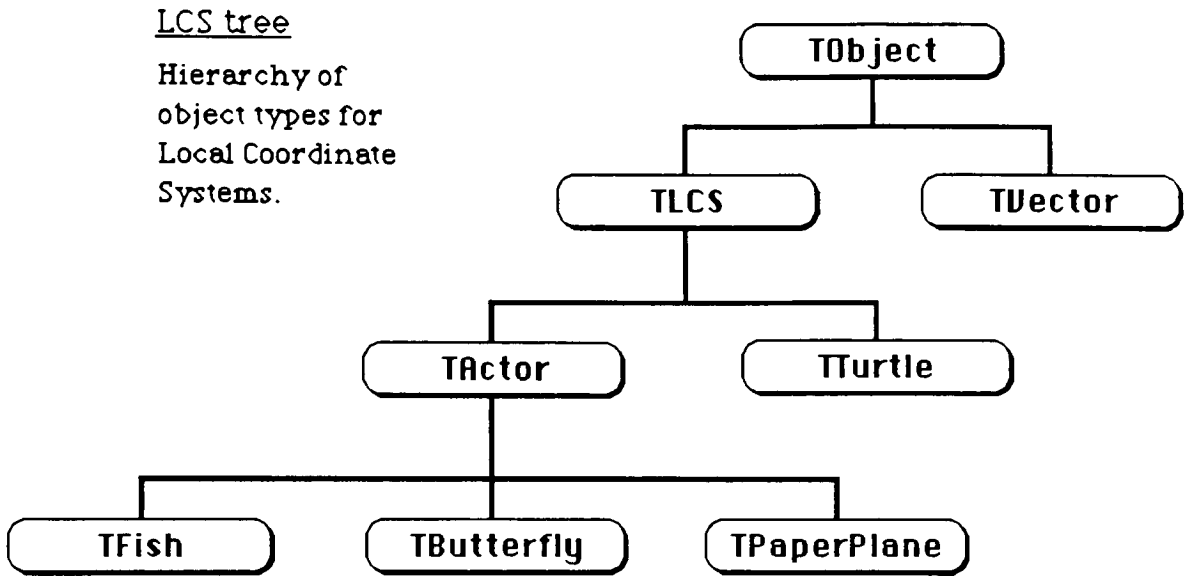


fig. 5.1

TLCS objects contain 4 vector fields: fPos, fFwd, fUp and fLeft, that store data about position, heading and orientation in 3-dimensional space. An Object Pascal declaration for the TLCS object type follows at the end of this section. The Position vector (fPos) locates a LCS in space. The other three fields represent mutually perpendicular unit vectors (see fig. 5.2) that establish a heading (fFwd) and an orientation for the LCS. Each of the vector data fields is represented by an ordered triple, <x,y,z>, relating it to a global basis, but all orientation and movement methods are implemented relative to the local basis determined by fUp, fFwd and fLeft. A fifth vector field, fTemp, facilitates computation.

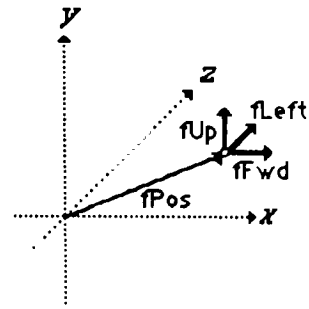


fig. 5.2

5.2. Local Coordinate Systems: fYaw, fPitch and fRoll

The TLCS objects have three REAL or EXTENDED data fields, fYaw, fRoll and fPitch, used to store information about changes in orientation or angular momentum. These values represent rotations about an axis of the local coordinate systems in degrees. The yaw, pitch and roll terminology is often used to describe rotation in three-dimensional space (nautical, aviation, robot motion).

Let yaw be a rotation of the vectors fLeft and fFwd about the axis defined by fUp, pitch be the rotation of fUp and fFwd about fLeft, and roll be the rotation of fLeft and fUp about fFwd. Actors and graphics turtles reference these data fields to update their orientation.

5.3. Local Coordinate Systems: fLens and fItsView

The fLens field is only used by an LCS when it is functioning as a camera or viewpoint for the 2-dimensional projection onto the screen. Earlier versions of Animator implemented a camera object type, but the inclusion of the fLens in all LCS objects allows any actor to be used as the viewpoint. The fLens data field is a REAL or EXTENDED number used to scale the image. Image size on the screen is directly proportional to the value of fLens.

The fItsView field is a handle or double pointer to a MacApp TView object. The pointer allows any LCS object connected to a particular view to reference information in that view. In Animator the view keeps track of the current camera or viewpoint, which is referenced by actors and turtles for drawing. The drawing (one frame at a time) of scripts in which multiple actors and the camera are in motion (not implemented) is anticipated in this design.

5.4. Local Coordinate Systems: allocation and assignment

If CS is declared as an instance variable of type TLCS, then it is a handle (double pointer) whose value is undefined. The NEW command allocates memory to CS. The initialization of its data fields is now acomplished by sending messages to CS in the form of method calls. For example, CS.Init(50, 25, 40), is a message telling CS to initialize its position vector to the triple <50, 25, 40>. The orientation vectors will be assigned default values corresponding to the basis of the global coordinate system, with fFwd directed like the positive x-axis and fUp directed like the positive y-axis, as in fig. 5.2. If the orientation is to be altered, rotation messages will direct CS to compute the new components for the unit vectors fUp, fLeft and fFwd. The EXTENDED fields are initialized to zero and can be reset by messages such as: CS.Yaw(90) and CS.Zoom(300), which tell CS to set the fYaw field to 90 and the fLens field to 300 respectively. Actors override the TLCS.Init method in order to extend it. The inherited method is, however, called by the TActor.Init override.

5.5. Local Coordinate Systems: deep and shallow methods

The Free method de-allocates memory for an object. It is inherited from TObject by all other objects, but must be overridden by those objects whose data fields are themselves dynamically allocated. The inherited method is called a shallow Free method and the override that frees space allocated to each of its pointer type fields is called a deep Free method.

TLCS.Free must free each of its five vector objects. Most descendants of TLCS override this method in order to deal with additional fields for which memory has been allocated. Typically, these deep free methods call on TLCS.Free to finish the job. Pointers such as TLCS.ItsView, which serve as connections to independent objects need not, and in fact should never be used to free the memory that they reference. TLCS.CopyOf is another shallow method (copies fPos, fFwd, fUp and fLeft only) that is overridden and extended by many of its descendants.

5.6. Local Coordinate Systems: methods of rotation

Each of the rotation methods provided for the TLCS object type rotates two of the orientation vectors about an axis along the third. These methods exploit the fact that fUp, fLeft and fFwd (fig. 5.3) are mutually perpendicular unit vectors in two ways: (1) the rotations are always within the plane determined by the rotated pair of vectors, (2) each member of the rotated pair provides a perpendicualr unit vector that facilitates the rotation computation for the other (see section 4.10, rotation of a vector in the plane).

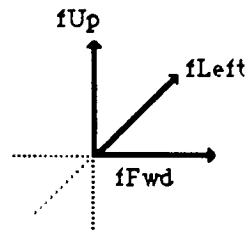


fig 5.3

TurnLeft and TurnRight rotate fLeft and fFwd about an axis along fUp. TiltUp and TiltDown are rotations of fFwd and fUp about fLeft. RollRight and RollLeft rotate fLeft and ffUp about fFwd.

5.7. Local Coordinate Systems: the TiltUp method

Let CS be a TLCS object, then the message CS.TiltUp(30), will rotate CS.fFwd and CS.fUp 30 degrees about the axis determined by CS.Left in the direction from fFwd toward fUp.

fFwd is rotated by the message fFwd.rotate(30, fUp). The second argument to rotate must be a unit vector that is perpendicular to fFwd (90 degrees in the direction of rotation). For fFwd, the vector fUp is the required perpendicular. In order to rotate fUp we require a similar perpendicular which is not directly available, but easily computed. The correct vector would resemble fFwd with its direction reversed. The TiltUp method initializes fTemp to be the negative of fFwd, and sends the message fUp.Rotate(30, fTemp). The other rotations are implemented similarly.

5.8. Local Coordinate Systems: moving

If CS is a TLCS object and D is a scalar distance, then the message CS.Move(D) would move CS, D units, in the direction of CS.fFwd.

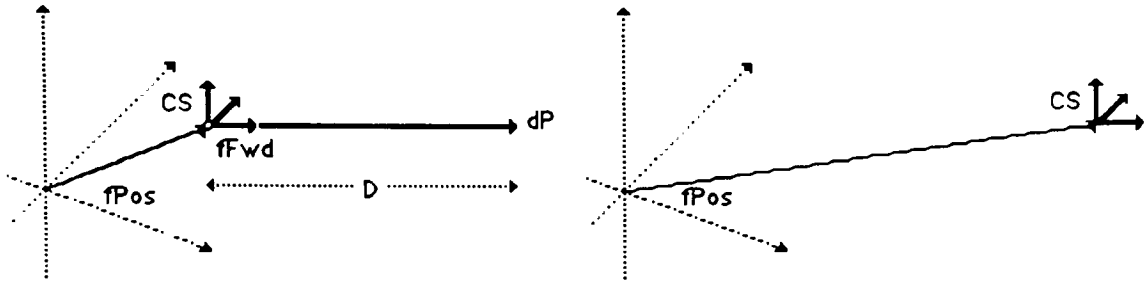


fig. 5.4

Let dP be a vector parallel to CS.fFwd with magnitude D, then dP is the vector represention of the change in position or movement. dP is the product of the vector CS.fFwd and the scalar D. The new position of CS is the vector sum of fPos and dP.

5.9. Local Coordinate Systems: Implementation of TLCS Object Type

```
TLCS = OBJECT (TObject) {Local Coordinate System}
      fPos, {position in global coordinates}
      fUp, {unit vector providing up/down reference}
      fLeft, {unit vector for right/left reference}
      fFwd, {forward indicates the direction of motion}
      fTemp: TVector; {useful during rotations of LCS}
      fLens: EXTENDED; {zoom factor when used as camera}
      fYaw, fPitch, fRoll: EXTENDED; {change in orientation per frame}
      fItsView: TLCSView; {link to MacApp view}


          {Local Coordinate System Methods}

      PROCEDURE TLCS.Init(theView: TLCSView; x, y, z: EXTENDED);
      PROCEDURE TLCS.CopyOf(lcs: TLCS);
      PROCEDURE TLCS.TurnRight(angle: EXTENDED);
      PROCEDURE TLCS.TurnLeft(angle: EXTENDED);
      PROCEDURE TLCS.TiltUp(angle: EXTENDED);
      PROCEDURE TLCS.TiltDown(angle: EXTENDED);
      PROCEDURE TLCS.RollRight(angle: EXTENDED);
      PROCEDURE TLCS.RollLeft(angle: EXTENDED);
      PROCEDURE TLCS.Move(distance: EXTENDED);
      PROCEDURE TLCS.MoveAbs(x, y, z: EXTENDED);
      PROCEDURE TLCS.Zoom(length: EXTENDED);
      PROCEDURE TLCS.Free; OVERRIDE;
      PROCEDURE TLCS.Yaw(delta: EXTENDED);
      PROCEDURE TLCS.Pitch(delta: EXTENDED);
      PROCEDURE TLCS.Roll(delta: EXTENDED);
    END; {TLCS}



{Implementation of Methods for Local Coordinate Systems}


PROCEDURE TLCS.Free; OVERRIDE;

{deep free for local coordinate system}

 BEGIN
  fPos.Free;
  fUp.Free;
  fLeft.Free;
  fFwd.Free;
  fTemp.Free;
  INHERITED Free
 END; {TLCS.Free}
```

```
PROCEDURE TLCS.Init(theView: TLCSView; x, y, z: EXTENDED);

{initialize local coordinate system at x, y, z oriented like global CS}

 VAR
  v1, v2, v3, v4: TVector;

 BEGIN
  fItsView := theView;
  new(v1);
  v1.Init(x, y, z);
  fPos := v1;
  new(v2);
  v2.Init(0, 1, 0);
  fUp := v2;
  new(v2);
  v2.Init(0, 0, 1);
  fLeft := v2;
  new(v3);
  v3.Init(1, 0, 0);
  fFwd := v3;
  new(v4);
  fTemp := v4;
  fYaw := 0;
  fPitch := 0;
  fRoll := 0
 END; {LCS.Init}


PROCEDURE TLCS.CopyOf(lcs: TLCS);

{Make object TLCS a copy of lcs}

 BEGIN
  fPos.CopyOf(lcs.fPos);
  fLeft.CopyOf(lcs.fLeft);
  fUp.CopyOf(lcs.fUp);
  fFwd.CopyOf(lcs.fFwd)
 END; {LCS.CopyOf}


PROCEDURE TLCS.TurnRight(angle: EXTENDED);

{rotate Local Coordinate System angle degrees to the right}

 BEGIN
  fTemp.Init(   fFwd.fX, - fFwd.fY, - fFwd.fZ);
  fFwd.rotate( - angle, fLeft);
  fLeft.rotate(   angle, fTemp)
 END; {LCS.TurnRight}
```

```
PROCEDURE TLCS.TurnLeft(angle: EXTENDED);

{rotate Local Coordinate System angle degrees to the fLeft}

 BEGIN
  fTemp.Init( - fFwd.fX, - fFwd.fY, - fFwd.fZ);
  fFwd.rotate(angle, fLeft);
  fLeft.rotate(angle, fTemp)
 END; {LCS.TurnLeft}


PROCEDURE TLCS.TiltUp(angle: EXTENDED);

{rotate Local Coordinate System angle degrees upward}

 BEGIN
  fTemp.Init(   fFwd.fX, - fFwd.fY, - fFwd.fZ);
  fFwd.rotate(angle, fUp);
  fUp.rotate(angle, fTemp)
 END; {LCS.TiltUp}


PROCEDURE TLCS.TiltDown(angle: EXTENDED);

{rotate Local Coordinate System angle degrees downward}

 BEGIN
  fTemp.Init(   fFwd.fX, - fFwd.fY, - fFwd.fZ);
  fFwd.rotate( - angle, fUp);
  fUp.rotate( - angle, fTemp)
 END; {LCS.TiltDown}


PROCEDURE TLCS.RollRight(angle: EXTENDED);

{roll Local Coordinate System angle degrees clockwise}

 BEGIN
  fTemp.Init(   fLeft.fX, - fLeft.fY, - fLeft.fZ);
  fLeft.rotate(angle, fUp);
  fUp.rotate(angle, fTemp)
 END; {LCS.RollRight}


PROCEDURE TLCS.RollLeft(angle: EXTENDED);
{roll Local Coordinate System angle degrees counter-clockwise}

 BEGIN
  fTemp.Init(   fLeft.fX,   fLeft.fY, - fLeft.fZ);
  fLeft.rotate( - angle, fUp);
  fUp.rotate( - angle, fTemp)
 END; {LCS.RollLeft}
```

```
PROCEDURE TLCS.Move(distance: EXTENDED);
{moves position of object by amount specified as distance in direction
  of the vector component fFwd}

 BEGIN
  fTemp.CopyOf(fFwd);
  fTemp.Scale(distance);
  fPos.AddTo(fTemp);
 END; {LCS.Move}


PROCEDURE TLCS.MoveAbs(x, y, z: EXTENDED);
{set object's position to x, y, z}

 BEGIN
  fPos.Init(x, y, z)
 END; {LCS.MoveAbs}


PROCEDURE TLCS.Zoom(length: EXTENDED);
{set object's lens equal to length}

 BEGIN
  fLens := length
 END; {LCS.Zoom}


PROCEDURE TLCS.Yaw(delta: EXTENDED);
{change the angular velocity about the vertical axis by delta}

 BEGIN
  fYaw := delta
 END; {TLCS.Yaw}


PROCEDURE TLCS.Pitch(delta: EXTENDED);
{change the angular velocity about the horizontal axis by delta}

 BEGIN
  fPitch := delta
 END; {TLCS.Pitch}


PROCEDURE TLCS.Roll(delta: EXTENDED);
{change the angular velocity about the forward axis by delta}

 BEGIN
  fRoll := delta
 END; {TLCS.Roll}
```

6. The Turtle: a Vector-Based Graphics Pen

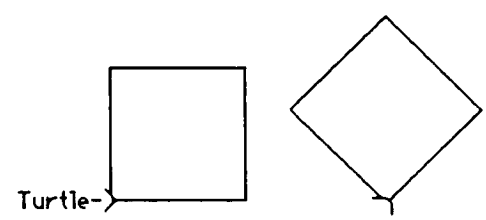
6.1. The Turtle: turtle graphics

The turtle metaphor for scripting the motion of a graphics pen on the screen of a computer demystified computer programming for a generation of school children who were introduced to turtle graphics through the Logo programming language [Papert 1970]. Graphics turtles draw by 'dragging' a graphics pen which marks their path as they obey orders to move and turn. The power of the metaphor seems to come from the immediacy of the commands.

Beginning programmers learn Logo in an environment where each command is executed immediately, avoiding possible confusion about how previous commands have altered the turtle's position or heading. The programmer must assume the turtle's point of view in order to give proper instructions which take the form of commands like "turn left (90 degrees)" or "move forward (10 steps)". Higher level commands are built up from primitive ones as procedures or sub-programs.

The following example of a procedure to draw a square is in a turtle dialect similar to the one defined for the TLCS object type in the Local CS section of this paper.

```
procedure square(distance : extended);

begin
   move(distance);
   turnleft(90);
   move(distance);
   turnleft(90);
   move(distance);
   turnleft(90);
   move(distance);
   turnleft(90);
end;  {square}
```



Note that the squares above are equally likely outcomes of sending the Square message to the turtle. It is, of course, the turtle's initial position and orientation that accounts for the differences.

6.2. The Turtle: drawing an actor

From an analytic point of view, the unusual aspect of specifying a drawing in this manner is the freedom of the descripion from any coordinate system. All lines, points and directions are relative to the turtle's inititial position and heading. Animator exploits these procedural definitions of drawing to implement transformations: rotations and translations are accomplished by altering the turtle's position before the drawing procedure is begun.

In Animator, each actor has a method of drawing itself consisting of a set of commands for a graphics turtle. An actor is moved or rotated by messages modifying the position and orientation data in its vector fields. Similar fields are inherited by all TLCS object types, including turtles. When an actor is called upon to redraw itself, it sends its turtle a message to copy these position and orientation vectors over its own. Since all of the lines drawn by the turtle are relative to its initial position and orientation, the effect is similar to multyiplying by a transformation matrix in a conventional graphics package.

6.3. The Turtle: TTurtle object type - data fields

The TTurtle object type is descendant from the TLCS object type and inherits all of the TLCS data fields and methods described in the Local CS section. These vector based methods allow TTurtle objects to respond to messages to turn and move. The TTurtle object type adds the following data fields, which are specific to its function.

- The boolean field fVisible works with the PenUP and PenDown messages to determine whether the moving turtle will draw a line or simply move to a new position.
- fCenterX and fCenterY locate the center of the screen for the projection method.
- fViewPoint is a handle allowing the turtles projection methods to access the data fields of the current camera.
- fSave provides a place to save the position vector when a drawing is more efficiently completed by moving back to previously visited point.
- fPositionPitch, fPositionYaw and fPositionRoll save degrees of rotation about each of the orientation axes, for situations when an actor's drawing is oriented differently from the local coordinate system that maintains its heading.

Animator's Rotation Menu commands use these fields in conjunction with the TTurtle object's inherited fields fPitch, fYaw and fRoll to maintain and display angular momentum about one or more axis that is independent of the rotation of the actor's local coordinate system. The messages to update these fields and rotate the turtle are sent by the TActor. DrawPoly method, just before drawing is begun.

6.4. The Turtle: projection onto the plane

Assuming its fVisible field is set to true (aTurtle.PenDown), aTurtle issues a command to draw a line from its initial position to its destination. In Animator this line is not drawn immediately on the screen, but is collected in a Macintosh toolbox data structure called a Polygon, which is used like a segment in a display file.

The lines collected in the polygon are expressed in the View's integer coordinate system, an awkward system with the positive y-axis pointing down. TTurtle objects respond to a message like aTurtle.Project by outputting a LineTo(x, y) command, where x and y are View coordinates. The projection of the three-dimensional point associated with its position onto an x, y plane is accomplished by computing a new set of coordinates for the point based on the local coordinate system of the current camera, a TLCS object whose data fields are referenced through the fViewPoint pointer.

6.5. The Turtle : the geometry of TTurtle.Project

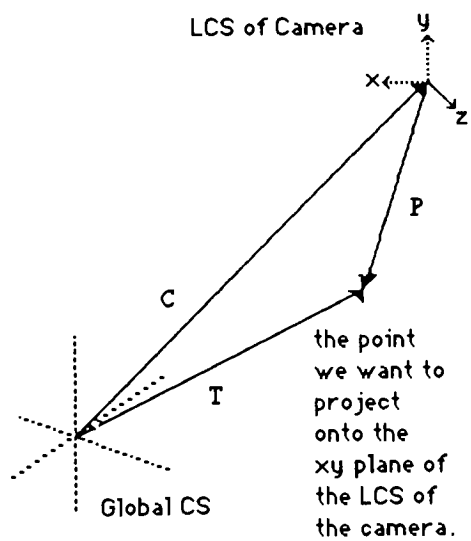


fig. 6.2

Let C be the position vector of the camera and T be the position vector of the turtle, then P, a vector from the tip of C to the tip of T can be computed as the difference between T and C.

P = T-C, can also be viewed as the position vector of the point to be projected from the coordinate system of the camera. The vector P can be resolved into components of the x,y,z basis of the camera LCS by finding the dot product or scalar projection of P on the extensions of each of the unit vectors in the basis of the camera LCS. Drawing lines between points obtained from the x and y components found above would produce a parallel projection (no perspective).

6.6. The Turtle: perspective projection

Animator produces a three-dimensional perspective by multiplying each of the coordinates found in the parallel projection by a ratio of a magnification factor, fViewPoint.fLens, and the orthogonol distance of the projected point from the x,y projection plane. The distance is simply the z component found by TTurtle.Project, and fLens is a value set by TLCS.Zoom. The result is to shrink the projected coordinates of distant objects toward the center of the projection. Increasing fLens magnifies all objects equally.

TTurtle.Project then rounds the result and resolves the differences between the image space coordinate system used by QuickDraw (Macintosh graphics package [Appl 85]) and Animator's world coordinates. If fVisible is true, a line is added to the polygon, otherwise a QuickDraw call is made to MoveTo which moves the graphics pen to (x,y) without drawing the line.

6.7. The Turtle: drawing with the turtle

The posts pictured in fig 6.3 were drawn by the turtle. Except for their z-coordinates they are identical. The perspective and size differences are due to the projection. They are made by repeated calls to the turtle's StackCube method, which draws a cube from the lower, left corner and leaves the turtle at the upper, left corner ready to draw another cube on top of the first.

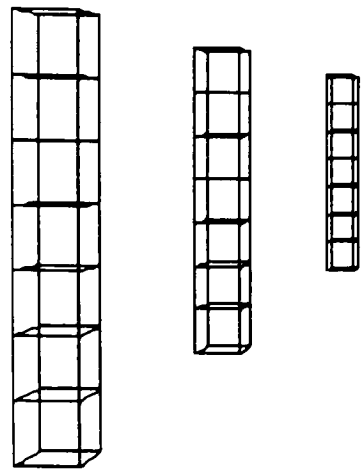


fig. 6.3

The cube is drawn by having the turtle walk around the block with left turns, pausing at each corner to draw a square with tilt-up turns. An important strategy in designing turtle subroutines, such as the tilt-up square, is to return the turtle to its initial position and orientation. The advantage in this example is that the tilt-up squares do not complicate the walk around the block.

6.8. The Turtle: overriding inherited methods

As an immediate descendant of the TLCS object type, TTurtle objects inherit all TLCS methods. Several of these methods are useful but incomplete, in view of the TTurtle object's specialization. A straightforward example is the inherited Move method. It works fine, but unlike any other object of the TLCS type, turtles need to draw lines when they move. The OVERRIDE method simply calls the INHERITED Move method followed by the Project method. The MoveAbs method for transporting an LCS to a position whose coordinates are known is overridden in the same manner.

The Init method is similarly extended to accommodate additional data fields. A more interesting case is the inherited TLCS.CopyOf method which only copies fields common to all descendants of the object type. This is particularly useful when the position and orientation of an actor is copied over the data fields of its turtle. Since the argument to the CopyOf method is of type TLCS, any descendant of this type can be passed. The specialized fields of the object being modified are not affected.

It is, however, sometimes useful to make one turtle an exact copy (all data fields) of another. The TTurtle OVERRIDE of CopyOf will make such a copy, if, and only if, the argument to TTurtle.CopyOf is of type TTurtle.

In the implementation the membership of the argument in TTurtle type is tested and if the argument is of the correct type, type coercion is used to assign a handle of the correct type to point to the data fields. In MPW Pascal, when a type identifier is used as a Pascal function the argument is recast as the designated type and returned by the function. This permits sibling objects to be passed as actual parameters to the same routine, with no loss of information.

6.9. The Turtle: moving parts

In the case of an overidden method the inherited method is also available by prefacing the call with the reserved word INHERITED. In the current version of Animator, actors with parts that move independently are implemented with a single turtle. Increasingly complex actors would be better served by multiple turtles, so that each turtle could maintain static data about its orientation with respect to its immediate predecessor in a hierarchical structure.

In such a design, subordinate turtles would take their initial position and orientation data from other turtles, rather than from the actor. In this case the inherited CopyOf would be appropriate, so as to avoid writing over data fields like fPositionPitch, which must accumulate persistent data.

6.10. Turtle: Implementation of TTurtle object type

```
    TTurtle = OBJECT (TLCS)
            fVisible: boolean; {moving turtle draws line when true}
            fViewpoint: TLCS; {handle for camera }
            fCenterX, fCenterY: EXTENDED;
            fSave: TVector;
            fPositionPitch: EXTENDED;
            fPositionYaw: EXTENDED;
            fPositionRoll: EXTENDED;

            PROCEDURE TTurtle.PenUp;
            PROCEDURE TTurtle.PenDown;
            PROCEDURE TTurtle.CopyOf(lcs: TLCS); OVERRIDE;
            PROCEDURE TTurtle.ChooseCamera(vp: TLCS);
            PROCEDURE TTurtle.Move(distance: EXTENDED); OVERRIDE;
            PROCEDURE TTurtle.MoveAbs(x, y, z: EXTENDED); OVERRIDE;
            PROCEDURE TTurtle.Init(theView: TLCSView; x, y, z: EXTENDED);
OVERRIDE;
            PROCEDURE TTurtle.Project;
            PROCEDURE TTurtle.Square(side: EXTENDED);
            PROCEDURE TTurtle.Cube(side: EXTENDED);
            PROCEDURE TTurtle.StackCube(side: EXTENDED);
            PROCEDURE TTurtle.Triangle(side: EXTENDED);
          END; {TTurtle}

{Methods for graphics turtle}

{initialize turtle using x, y to establish center of CS and z for zoom value}


PROCEDURE TTurtle.Init(theView: TLCSView; x, y, z: EXTENDED); OVERRIDE;

 VAR
  vp: TLCS;
  aVector: TVector;

 BEGIN
  INHERITED Init(theView, x, y, z);
  ChooseCamera(fItsView.fCamera);
  new(aVector);
  fSave := aVector;
  fCenterX := kCenterX;
  fCenterY := kCenterY;
  fPositionYaw := 0;
  fPositionPitch := 0;
  fPositionRoll := 0;
  fVisible := True;
 END; {TTurtle.Init}
```

```
PROCEDURE TTurtle.CopyOf(lcs: TLCS); OVERRIDE;

 VAR  aTurtle: TTurtle;

 BEGIN
  INHERITED CopyOf(lcs);
  IF MEMBER(lcs, TTurtle) THEN
    BEGIN
    aTurtle := TTurtle(lcs);
    fYaw := aTurtle.fYaw;
    fPitch := aTurtle.fPitch;
    fRoll := aTurtle.fRoll;
    fPositionYaw := aTurtle.fPositionYaw;
    fPositionPitch := aTurtle.fPositionPitch;
    fPositionRoll := aTurtle.fPositionRoll;
    fVisible := aTurtle.fVisible;
    fViewPoint := aTurtle.fViewPoint
    END {if}
 END {TTurtle.CPOf} ;


PROCEDURE TTurtle.Project;

{project turtle position with respect to plane of viewing eye}

 VAR
  px, py: EXTENDED; {projection cooordinates for image space}
  x, y, z: EXTENDED; {coordinates in viewpoint's LCS}

 BEGIN
  fTemp.CopyOf(fViewPoint.fPos);
  fTemp.Scale(  1);
  fTemp.AddTo(fPos);
  x := fTemp.DotWith(fViewPoint.fFwd);
  y := fTemp.DotWith(fViewPoint.fUp);
  z := fTemp.DotWith(fViewPoint.fLeft);
  px := x * fViewPoint.fLens / z + fCenterX;
  py := fCenterY - y * fViewPoint.fLens / z;
  IF fVisible THEN
    LineTo(ROUND(px), ROUND(py))
  ELSE
    MoveTo(ROUND(px), ROUND(py));
 END; {project}


PROCEDURE TTurtle.PenDown;
{set visible to true so movement draws lines}

 BEGIN
  fVisible := True
 END; {TTurtle.PenDown}
```

```
PROCEDURE TTurtle.PenUp;
{set visible to false so movement doesn't draw lines}

 BEGIN
  fVisible := FALSE
 END; {TTurtle.PenUp}


PROCEDURE TTurtle.ChooseCamera(vp: TLCS);
{assign vp as camera viewpoint for drawing}

 BEGIN
  fViewPoint := vp
 END;


PROCEDURE TTurtle.Move(distance: EXTENDED); OVERRIDE;
{The inherited TLCS.Move is extended to project and output a 2-D lineto command}

 BEGIN
  INHERITED Move(distance);
  Project
 END; {TTurtle.Move}


PROCEDURE TTurtle.MoveAbs(x, y, z: EXTENDED); OVERRIDE;
{The inherited TLCS.MoveAbs is extended to project and output a 2-D lineto command}

 BEGIN
  fPos.Init(x, y, z);
  Project
 END; {TTurtle.MoveAbs}


PROCEDURE TTurtle.Square(side: EXTENDED);

{draw state transparent square}

 VAR
  i: INTEGER;

 BEGIN
  FOR i := 1 TO 4 DO
   BEGIN
   Move(side);
   TiltUp(90)
   END
 END; {TTurtle.Square}
```

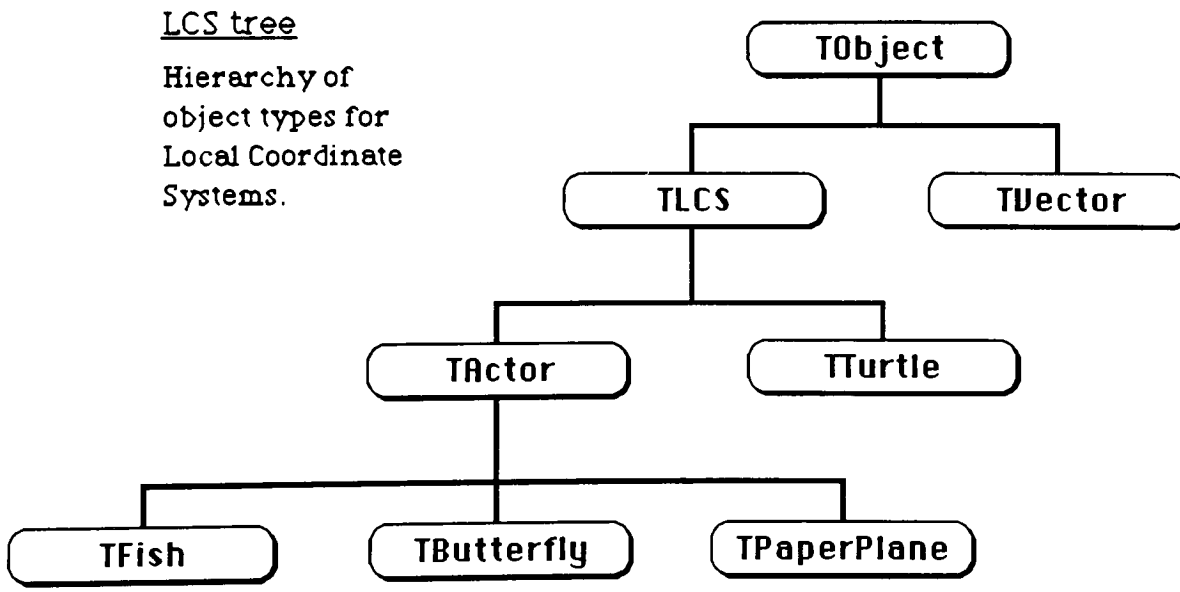
```
PROCEDURE TTurtle.Cube(side: EXTENDED);

{draw state transparent cube}

 VAR
  i: INTEGER;

 BEGIN
  FOR i := 1 TO 4 DO
    BEGIN
    Square(side);
    Move(side);
    TurnLeft(90)
    END;
 END; {TTurtle.Cube}


PROCEDURE TTurtle.StackCube(side: EXTENDED);

{draws cubes one on top of another}

 BEGIN
  Cube(side);
  TiltUp(90);
  Move(side);
  TiltDown(90);
 END;


PROCEDURE TTurtle.Triangle(side: EXTENDED);

{draws state transparent triangle}

 VAR
  i: INTEGER;

 BEGIN
  FOR i := 1 TO 3 DO
    BEGIN
    Move(side);
    TurnLeft(120)
    END; {for}
 END; {TTurtle.Triangle}
```

7. Actors: a Template for Pictures that Move

7.1. Actors: an animated object type

Animator's actors maintain data about their postion in the world coordinate system and how that position is changing (instantaneous velocity, acceleration, angular momentum and acceleration). They are designed to remain in motion (change their position and orientation before each frame is recorded) until they receive a message to alter their data. Animator's menu and mouse controls allow users to stop the actor's clockwork world and send it messages. Actors also respond to messages from objects (TView) that control the image drawn on the monitor. The design anticipates more interesting descendants whose behavior might be modified by messages from other objects in the scene, perhaps even from other actors.

As TActor objects, actors can store information in their data fields and respond to certain messages corresponding to their declared methods. When a new object type is declared to be a descendant of an existing one it can add to or modify its inheritance of data fields and methods. Each of Animator's actor types can trace its ancestry back through TActor to the TLCS object type and TObject. (recall fig. 5.1)



7.2. Actors: TActor data fields

The data for orientation and position is stored in vector fields inherited from the TLCS object type. They are discussed in detail in the sections Local CS and Vectors. The TActor object type declares the following additional data fields in order to maintain and update a display image:

- fPoly is a handle or double pointer to a QuickDraw data structure called a Polygon. Polygons are used to store line drawing commands which can be displayed with a single FramePoly command when the view sends a message to an actor to draw itself.
- fValidPoly is a boolean field which is set to false if an actor or the camera has been moved or rotated, indicating it will be necessary to update the Polygon before display.
- fTurtle is a handle for the graphics turtle which will be used to redraw the actor's Polygon.
- fExtent is a rectangle defining the area of the View effected by erasing the old Polygon and framing the new one.
- fVelocity is a field indicating the distance an actor should be moved in the next frame.
- fAcceleration is an increment or decrement applied to fVelocity before each move.
- fID is an identification number.
- fHiLight indicates that an actor is currently selected and should be shaded on the display.

7.3. Actors: Updating the Polygon

Whenever an actor has received a message to move or rotate it will set fValidPoly to false. The TActor.InvalidatePoly method affords other objects (such as the View) the opportunity of sending an actor a message to invalidate its polygon. The View object broadcasts this message to the entire cast (list of actors) whenever the camera moves or

otherwise alters the viewpoint of projection. Before a view is redrawn an UpDate message is sent to all actors whose fValidPoly field is false.

To prepare for drawing, TActor.UpDatePoly first sends a message to the object pointed to by fCamera to initialize the projection coordinate system (projection onto the plane is described in the Turtle section). Next, the bounding rectangle of the existing polygon is saved (this area of the view will have to be redrawn), and a new one is initialized. Finally, messages are sent passing the actor's local coordinate system to the graphics turtle and instructing the turtle to redraw the polygon.

7.4. Actors: Overriding DrawPoly

TActor is an abstract object type. It serves as a template for all descendant actor types, but is never instantiated. The DrawPoly method inherited by each of TActor's descendants makes some final adjustments to the turtle to account for transformations that are independent of the actor's coordinate system, but does not contain any instructions for drawing. Each descendant of TActor must OVERRIDE the INHERITED DrawPoly method in order to provide an appropriate method for drawing itself. Typically the new method will call the inherited one to take advantage of the general routines it implements. The drawing method consists of messages to move the graphics turtle, which in turn issues QuickDraw commands to create the polygon.

Upon completion of the polygon, control returns to UpDate, and the polygon data structure is closed. The bounding rectangle of the new polygon is then combined with that of the original polygon to provide a description of the portion of the view area that has been altered by the update. Finally, the UpDate method resets fValidPoly to true.

7.5. Actors: moving the picture

In film animation, 30 still images are required for each second of viewing time. The animator must produce a series of discrete images, or frames, depicting the actors as they would appear in each of a series of motion freezing snapshots taken one-thirtieth of a second apart. The fact that the finished images are displayed at 30 frames per second need not dictate the rate of frame production, but a rapid preview or feedback system of some sort is essential to the animator's process. Animator attempts to provide continuous feedback to the user as the movement of an actor is specified, but this feedback is limited to a simulation of the selected actor's motion. The design of Animator anticipates a

production program that will create the frames for scripts involving the simultaneous movement of a cast of actors, but implementation of a frame-by-frame production system was not attempted.

An actor's movement between one frame and the next is represented as a directed distance (vector) taking its direction from the actor's heading and its magnitude from the data field fVelocity. Animator's interface allows the user to select a single actor, in order to preview its progress as the frame counter advances, by double clicking on that actor and holding down the mouse button. Releasing the mouse button will halt the process. If the halted sequence was not satisfactory it can be undone (the actor will revert to its initial position) with a menu command.

The user can stop and start the production of frames in order to change the value of fVelocity. Since these modifications alter the actor's speed abruptly, smooth transitions to new states would require many stops for small modifications. The user may instead choose to modify fAcceleration, specifying an increment or decrement in velocity to be applied before each movement is computed, thus avoiding the frequent keying that would otherwise be required to specify gradual changes.

Since movement will always be directed along the actor's heading, the user is also provided with the option of rotating the actor so as to change the heading. The change in direction can be either immediate or incremental (per frame). An incremental change is achieved by modifying angular momentum. It can be likened to the influence of a rudder on the heading of a boat, or the steering wheel on the heading of a car.

7.6. Actors: Navigation in 3-D space

Rotations in Animator are relative to the actor's heading and orientation rather than to an external reference. The user is asked to assume the point of view of the actor he is directing, or even better, to assume the actor's position in space. Left is the direction pointed to by the extended left arm, and up is directly over the actor's head, even if the actor happens to be upside down. Although this is occasionally counterintuitive, the local frame of reference is invaluable when the actor has moving parts.

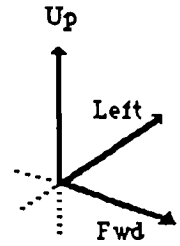


fig. 7.2

When directing Animator's actors, left and right turns are best understood as rotations of fig. 7.2 in which the left and forward axes will rotate together, pivoting around the UP axis. Tilting up or down is a rotation in which UP and FWD move together pivoting around the LEFT axis. Rolling left or right produces a rotation of UP and LEFT about FWD (no change in heading).

The commands: Turn, Tilt, Roll and Acceleration are found in the Control Menu, and operate on the currently selected actor. Rotations that do not affect the movement of actors through space (i.e. spinning) can be specified with the Rotation Menu in a similar manner.

The move command inherited from the TLCS object type takes a DISTANCE as a parameter and moves the object forward along its present heading. The TActor object type overrides this method in order to make changes in orientation and compute an appropriate distance parameter based on fAcceleration and fVelocity. The inherited fields fYaw, fPitch and fRoll, correspond to turn, tilt and roll respectively and are used to record rotations to be made before each move or frame.

7.7. Actors: Descendants of TActor

Creating a new actor type requires some programming in order to describe how the new object will differ from its ancestor. For instance, if the actor is to differ only in appearance, only the TActor.DrawPoly method need be changed (overridden).

fig. 7.3

Currently the most successful actors are of the TFish object type. They are paper thin angel fish made of two triangles. The smaller triangle, or tail, swishes behind the moving fish as it swims. The necessity of redrawing the polygons for each move dictates a simple design, where real time feedback is required. In frame-by-frame production, a descendant of TFish that was visually more interesting, but slower to draw, could be used to produce the final frames. The new fish need differ only in its DrawPoly method.

7.8. Actors: commands as text

In anticpation of an application for the production of Animator scripts each menu or mouse command that alters the appearance or position of an actor could be written as text to another file. The text description should include the selected actor's ID number and the current frame number, as well as the data entered to complete the command. All commands would be written to a text file which can be sorted according to frame number and edited to create scenes.

The production application would draw frames one at a time, using the command file as a script. In order to draw each frame, it would execute all of the commands with that frame number, move and display each actor, and then signal a camera to record the frame. The existing design should support scripts that choreograph multiple actors with a moving camera or viewpoint. Production would be slow, but time increases should remain proportional to increases in scene complexity.

7.9. Actors: Implementation of TActor Object Type

```
    TActor = OBJECT (TLCS)
            fID: INTEGER; {ID number for actor}
            fPoly: PolyHandle; {handle for quickdraw polygon}
            fExtent: RECT; {bounds for invalidated area of view}
            fTurtle: TTurtle; {handle for graphics turtle}
            fAcceleration: EXTENDED; {change in velocity per frame}
            fVelocity: EXTENDED; {change in position along heading per frame}
            fHiLight: boolean; {actor is selected by mouse click in poly extent}
            fValidPoly: boolean; {false when object or camera moves; true when poly is
redrawn}


            {methods for TActor object type}

            PROCEDURE TActor.Init(theView: TLCSView; x, y, z: EXTENDED);
OVERRIDE;
            PROCEDURE TActor.Free; OVERRIDE;
            PROCEDURE TActor.Identify(id: INTEGER);
            PROCEDURE TActor.CopyOf(lcs: TLCS); OVERRIDE;
            FUNCTION TActor.ExtentRect: RECT;
            PROCEDURE TActor.Acceleration(delta: EXTENDED);
            PROCEDURE TActor.Velocity(abs: EXTENDED);
            PROCEDURE TActor.UpDatePoly;
            PROCEDURE TActor.DrawPoly;
            PROCEDURE TActor.Draw;
            PROCEDURE TActor.Highlight;
            PROCEDURE TActor.EnableHiLight(TF: boolean);
            PROCEDURE TActor.InvalidatePoly;
            PROCEDURE TActor.Move(distance: EXTENDED); OVERRIDE;
            PROCEDURE TActor.MoveAbs(x, y, z: EXTENDED); OVERRIDE;
          END; {actor}


    TButterfly = OBJECT (TActor)

          PROCEDURE TButterfly.DrawPoly; OVERRIDE;

        END; {TButterfly}


    TFish = OBJECT (TActor)

          fAngles: ARRAY [0..12] OF EXTENDED;
          fCounter: INTEGER;

          PROCEDURE TFish.Init(theView: TLCSView; x,y,z: EXTENDED);  OVERRIDE;
          PROCEDURE TFish.DrawPoly; OVERRIDE;
```

```
    END; {TFish}

  TPaperPlane = OBJECT (TActor)

        PROCEDURE TPaperPlane.DrawPoly; OVERRIDE;

    END; {TPaperPlane}

  TArch = OBJECT (TActor)

      PROCEDURE TArch.DrawPoly; OVERRIDE;

    END; {TArch}

  TPost = OBJECT (TActor)

      PROCEDURE TPost.DrawPoly; OVERRIDE;

    END; {TPost}

  TCube = OBJECT (TActor)

      PROCEDURE TCube.DrawPoly; OVERRIDE;

    END; {TCube}

{methods for actors & other special LCSs}

PROCEDURE TActor.Free;

{dipose of allocated memory }

 BEGIN
  KillPoly(fPoly);
  INHERITED Free
 END; {TActor.Free}



PROCEDURE TActor.Identify(id: INTEGER);

{provision for assignment of a unique identifier to each actor in scene}

 BEGIN
  fID := id
```

```
 END; {TActor.Identify}


PROCEDURE TActor.InvalidatePoly;

 BEGIN
  fValidPoly := FALSE
 END; {TActor.Invalidate Poly}
PROCEDURE TActor.Init(theView: TLCSView; x, y, z: EXTENDED); OVERRIDE;

{initialize actor at x, y, z oriented like global CS}

 VAR
  aTurtle: TTurtle;

 BEGIN
  fYaw := 0;
  fPitch := 0;
  fRoll := 0;
  fAcceleration := 0;
  fVelocity := 0;
  fHiLight := True;
  fValidPoly := FALSE;
  fPoly := OpenPoly;
  ClosePoly;
  fExtent := gZeroRect;
  new(aTurtle);
  aTurtle.Init(theView, x, y, z);
  fTurtle := aTurtle;
  INHERITED Init(theView, x, y, z)
 END; {Actor.Init}


PROCEDURE TActor.CopyOf(lcs: TLCS); OVERRIDE;

 VAR  aActor: TActor;

 BEGIN
  INHERITED CopyOf(lcs);
  IF MEMBER(lcs, TActor) THEN
   BEGIN
   aActor := TActor(lcs);
   fYaw := aActor.fYaw;
   fPitch := aActor.fPitch;
   fRoll := aActor.fRoll;
   fAcceleration := aActor.fAcceleration;
   fVelocity := aActor.fVelocity;
   fHiLight := aActor.fHiLight;
   fValidPoly := FALSE;
   fPoly := aActor.fPoly;
   fExtent := aActor.fExtent;
   fTurtle.CopyOf(aActor.fTurtle)
```

```
    END {if}
  END {TActor.CopyOf} ;


PROCEDURE TActor.EnableHiLight(TF: BOOLEAN);

  BEGIN
   fHiLight := TF
  END; {enable highlighting}


FUNCTION TActor.ExtentRect: Rect;

  BEGIN
   ExtentRect := fPoly^^.PolyBBox
  END; {ExtentRec}


PROCEDURE TActor.Move(distance: EXTENDED); OVERRIDE;

  BEGIN
   fVelocity := fVelocity + fAcceleration;
   IF fYaw <> 0 THEN TurnLeft(fYaw);
   IF fPitch <> 0 THEN TiltUp(fPitch);
   IF fRoll <> 0 THEN RollRight(fRoll);
   INHERITED Move(fVelocity);
   InvalidatePoly
  END; {Move}


PROCEDURE TActor.MoveAbs(x, y, z: EXTENDED); OVERRIDE;

  BEGIN
   INHERITED MoveAbs(x, y, z);
   InvalidatePoly
  END; {Move}


PROCEDURE TActor.Acceleration(delta: EXTENDED);

{change the velocity or distance the actor moves per frame by delta}

  BEGIN
   fAcceleration := delta
  END; {TActor.Distance}


PROCEDURE TActor.Velocity(abs: EXTENDED);

{change the acceleration or change in velocity per frame by delta}

  BEGIN
```

```
  fVelocity := abs
 END; {TActor.Velocity}


PROCEDURE TActor.UpDatePoly;

{focus camera   save extent of old polygon   initialize new polygon
  initialize turtle - get polygon drawn - calculate extent}

 BEGIN
  IF NOT fValidPoly THEN
    BEGIN
    fItsView.fCamera.TurnRight(90);
    fExtent := fPoly^^.PolyBBox;
    KillPoly(fPoly);
    fPoly := OpenPoly;
    fTurtle.fPos.CopyOf(fPos);
    fTurtle.fFwd.CopyOf(fFwd);
    fTurtle.fUp.CopyOf(fUp);
    fTurtle.fLeft.CopyOf(fLeft);
    fTurtle.PenUp;
    fTurtle.MoveAbs(fPos.fX, fPos.fY, fPos.fZ);
    fTurtle.PenDown;
    DrawPoly;
    ClosePoly;
    fItsView.fCamera.TurnLeft(90);
    UnionRect(fExtent, fPoly^^.PolyBBox, fExtent);
    fValidPoly := True;
    InsetRect(fExtent,   1,   1)
    END {if} ;
  END; {Actor.UpDatePoly}


PROCEDURE TActor.Draw;

{make line drawing of actor's polygon}

 BEGIN
  FramePoly(fPoly)
 END; {TActor.Draw}


PROCEDURE TActor.Highlight;

{special draw method to highlight actor when it is selected}

 BEGIN
  IF fHiLight THEN
    BEGIN
    FillPoly(fPoly, LtGray);
    FramePoly(fPoly)
    END {If}
```

```
 END; {THighlight.Draw}


PROCEDURE TActor.DrawPoly;
 BEGIN
  WITH fTurtle DO
   BEGIN
   fPositionYaw := fPositionYaw + fYaw;
   fPositionRoll := fPositionRoll + fRoll;
   fPositionPitch := fPositionPitch + fPitch;
   IF fPositionYaw > 360 THEN
    fPositionYaw := fPositionYaw - 360
   ELSE IF fPositionYaw < - 360 THEN fPositionYaw := fPositionYaw + 360;
   IF fPositionRoll > 360 THEN
    fPositionRoll := fPositionRoll   360
   ELSE IF fPositionRoll < - 360 THEN fPositionRoll := fPositionRoll + 360;
   IF fPositionPitch > 360 THEN
    fPositionPitch := fPositionPitch   360
   ELSE IF fPositionPitch < - 360 THEN
    fPositionPitch := fPositionPitch + 360;
   fTurtle.TurnLeft(fPositionYaw);
   fTurtle.RollRight(fPositionRoll);
   fTurtle.TiltUp(fPositionPitch)
   END {with}
 END; {TActor.DrawPoly}


PROCEDURE TPaperPlane.DrawPoly; OVERRIDE;
 BEGIN
  INHERITED DrawPoly;
  WITH fTurtle DO
   BEGIN
   RollRight(200);
   TurnRight(90);
   Move(kActorSize);
   TurnLeft(120);
   Move(gDoubleSize);
   TurnRight(30);
   RollLeft(40);
   TurnLeft(150);
   Move(gDoubleSize);
   TurnLeft(120);
   Move(kActorSize);
   TurnLeft(90);
   RollRight(20);
   TiltUp(90);
   Move(kActorSize);
   TiltDown(120);
   Move(gDoubleSize);
   TiltUp(30);
   Move(   2 * gAltitude);
   END {WITH}
```

```
 END; {TPaperPlane.DrawPoly}

PROCEDURE TButterfly.Init(theView: TLCSView; x, y, z: EXTENDED); OVERRIDE;

 BEGIN
  INHERITED Init(theView, x, y, z);
  fAngles[0] := - 24;
  fAngles[1] :=   15;
  fAngles[2] := 0;
  fAngles[3] := 15;
  fAngles[5] := 24;
  fAngles[6] := 30;
  fAngles[7] := 24;
  fAngles[8] := 15;
  fAngles[9] := 0;
  fAngles[10] := - 15;
  fAngles[11] := - 24;
  fAngles[12] := - 30;
  fCounter := 2;
  fflutter := 3;
 END; {TButterfly.Init}

{method to draw a bird}


PROCEDURE TButterfly.DrawPoly; OVERRIDE;

 BEGIN
  INHERITED DrawPoly;
  WITH fTurtle DO
   BEGIN
   {TurnLeft(fFlutter);}
   fSave.CopyOf(fPos);
   RollRight(65 + fAngles[fCounter]);
   TurnLeft(45);
   Move(25);
   TurnLeft(165);
   Move(45);
   MoveAbs(fSave.fX, fSave.fY, fSave.fZ);
   TurnLeft(150);
   RollLeft(130 + 2 * fAngles[fCounter]);
   TurnRight(45);
   Move(25);
   TurnRight(165);
   Move(45);
   MoveAbs(fSave.fX, fSave.fY, fSave.fZ);
   {fFlutter := - fFlutter} ;
   IF fCounter = 12 THEN
    fCounter := 0
   ELSE
    fCounter := fCounter + 1
   END {WITH}
```

```
 END; {TButterfly.DrawPoly}

{initialize a fish object}


PROCEDURE TFish.Init(theView: TLCSView; x, y, z: EXTENDED); OVERRIDE;

 BEGIN
  INHERITED Init(theView, x, y, z);
  fAngles[0] :=  24;
  fAngles[1] := - 15;
  fAngles[2] := 0;
  fAngles[3] := 15;
  fAngles[5] := 24;
  fAngles[6] := 30;
  fAngles[7] := 24;
  fAngles[8] := 15;
  fAngles[9] := 0;
  fAngles[10] := - 15;
  fAngles[11] := - 24;
  fAngles[12] := - 30;
  fCounter := 2;
 END; {TFish.Init}

{method to draw a fish}


PROCEDURE TFish.DrawPoly; OVERRIDE;

  BEGIN
   INHERITED DrawPoly;
   WITH fTurtle DO
    BEGIN
    TiltDown(90);
    Move(gSqrt2);
    TiltUp(135);
    Move(gDoubleSize);
    TiltUp(90);
    Move(gDoubleSize);
    TiltUp(135);
    Move(gSqrt2);
    TiltUp(225);
    TurnLeft(fAngles[fCounter]);
    Move(gActorSize);
    TiltUp(135);
    Move(gSqrt2);
    TiltUp(135);
    Move(gActorSize);
    IF fCounter = 12 THEN
     fCounter := 0
    ELSE
     fCounter := fCounter + 1
```

```
    END {WITH}
  END; {TFish.DrawPoly}

{method to draw a cube}


PROCEDURE TCube.DrawPoly; OVERRIDE;

  BEGIN
    INHERITED DrawPoly;
    fTurtle.Cube(gDoubleSize)
  END; {TCube.DrawPoly}

{method to draw a post}


PROCEDURE TPost.DrawPoly; OVERRIDE;

  VAR
    i: INTEGER;

  BEGIN
    INHERITED DrawPoly;
    FOR i := 1 TO 7 DO fTurtle.StackCube(gDoubleSize)
  END; {TPost.DrawPoly}

{method to draw an arch}


PROCEDURE TArch.DrawPoly; OVERRIDE;

  VAR
    j: INTEGER;

  BEGIN
    INHERITED DrawPoly;
    WITH fTurtle DO
      BEGIN
      FOR j := 1 TO 5 DO StackCube(gDoubleSize);
      Move(gDoubleSize);
      TiltDown(90);
      FOR j := 1 TO 3 DO StackCube(gDoubleSize);
      Move(gDoubleSize);
      TiltDown(90);
      FOR j := 1 TO 4 DO StackCube(gDoubleSize);
      END {with}
  END; {TArch.DrawPoly}
```

8. Animator: a stage for the actors

8.1. Animator: a stage for the actors

The TActor abstract data type and its descendants were designed as objects that could respond to directions about orientation and movement in space in an interactive environment suitable for scripting computer animation. Their implementation as a hierarchy of objects is intended to provide a suitable context for extending the complexity and variety of the actors and their movements by programming. Another design criterion is the suitability of the objects for inclusion in a suite of related application tools.

The Animator application was designed to provide an opportunity to explore the utility of the interactive methods incorporated in these objects and the extensibility of their design.

This section deals with the implementation of Animator as a Macintosh application based on MacApp, a collection of objects supporting the Macintosh user interface.

8.2. Animator: initializing actors - the cast

In order to add a new actor to the cast, the user selects actor from the init menu. Currently there are three different actor types, Butterfly, Fish and Paper Airplane. A dialog allows the user to choose one and prompts for X, Y and Z coordinates in order to locate the actor.

A TFish object is created and inserted into a data structure of type TList referenced by the fCast field of the view. Tlist is an object type supplied by MacApp as a list structure for objects. The element type need not be specified, but the structure is most useful when all of its elements share an immediate ancestor. Animator limits the fCast list to assorted actor types.

The actor will be drawn on the screen and shaded as the current selection. Its default heading is parallel to the X-axis and it is oriented so that up is in the positive Y direction and left is parallel to the positive Z-axis. Its data fields indicate it is an object at rest.

8.3. Animator: drawing the view

In an event driven program, where a view may be obscured or revealed by a response to the user at any time, (for example, opening or closing a window), the view must be prepared to redraw itself at any time, but it is the responsibility of the frame to initiate the process. The exception is when a message changes the way the view should appear. In Animator, this is usually a command that changes an actor's position or orientation. An area of the view corresponding to the selected actor's fExtent rectangle will be invalidated by the command object. This is accomplished by calling the view's ExecuteChange method.

ExecuteChange traverses the fCast list, sending update messages to all actors whose drawings are not current, before notifying the frame of the invalid area. Tlist supports a FOR EACH control structure that allows a procedure to be repeated with each element of the list as its argument. Both the ExecuteChange and Draw methods use it to traverse the fCast list.

8.4. Animator: selecting objects - mouse commands

The selected actor is the target for all Controls Menu and Rotation commands. Selection is acomplished by clicking on the actor with the mouse. Clicking anywhere else will deselect the actor. The selected actor is marked as highlighted (it is distinguished in the view by gray shading) and a pointer in the view is set to reference it. Multiple selection is not implemented for any Animator commands.

The mouse down message is passed to the view by the event manager and a DoMouseCommand method must determine what, if any, reponse is appropriate. DoMouseCommand returns a value signifying no change unless the mouse was clicked within the extent of an actor. DoMouseCommand is passed the location of the click and implements its own function to test each actor. If an actor was selected before the mouse click, DoMouseCommand deselects that actor whether or not the mouse click results in a new selection.

8.5. Animator: the controls menu

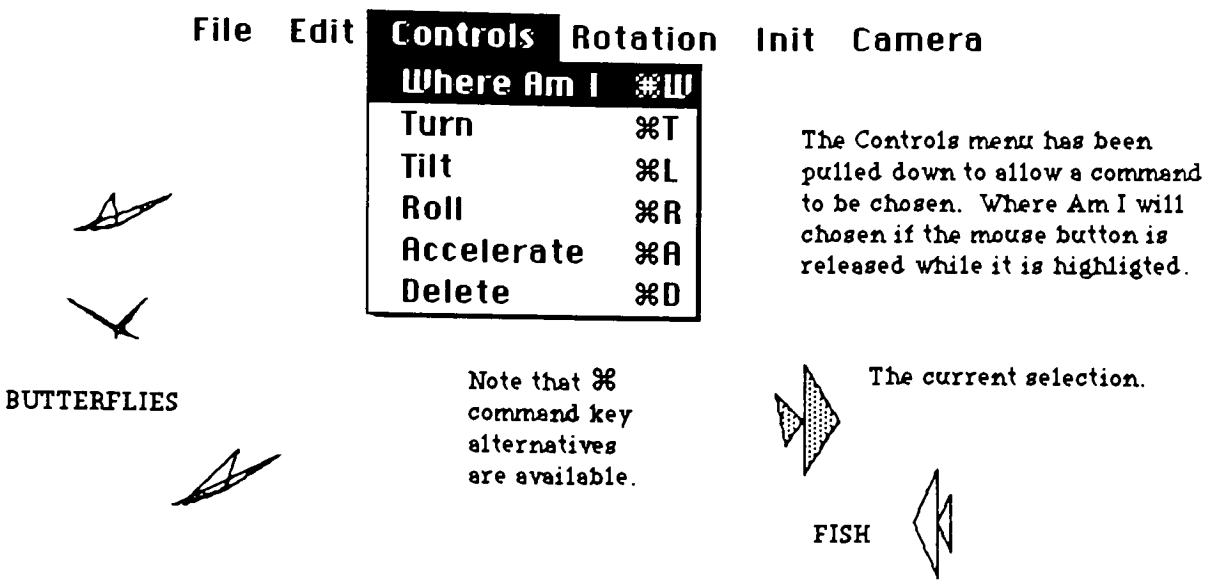


fig. 8.1

8.6. Animator: simulating the move

Double-clicking on the selected actor and holding the mouse button down will initiate a move sequence. The sequence is terminated by releasing the button. Undo from the Edit menu will restore the actor to its original position. This was implemented by overriding the TrackMouse and TrackFeedBack methods typically used for multiple selection or drawing with the mouse.

TrackMouse will be called repeatedly until aTrackPhase = trackRelease. It sends a move message to the selected actor in each cycle. Highlighting is turned off for the duration of the sequence.

The TrackFeedBack method tells the view (ExecuteChange) to invalidate the actor's extent which is the smallest rectangle that includes the actor's polygons before and after moving. Any actor whose extent intersects this rectangle must be redrawn. TrackFeedBack then sends the frame an UpDate message to force immediate redrawing.

9. Conclusions

9.1. Evaluation of Animator's Object Types

The specification of complex movements for Animator's actors takes practice but has some clear advantages over conventional transformations. Where actions can be described as continuous modifications of position, orientation or velocity, without reference to other objects, little or no mathematics or precalculations were required (spirals, circles, oscillation, rolling, acceleration etc.). Where an external point was the focus of the action to be specified (point toward or go to <x,y,z>) specification could become tedious. The difficulties encountered did not seem intrinsic to the design, but rather suggested interesting directions for further development.

For instance a TActor.TurnToward method, taking a point (target) and an angle (change in heading per frame) as arguments, would be very useful. If the point passed as an argument to a command like TurnToward was the position of another object (possibly a moving target) it would greatly simplify the choreography of very complex scenes in which the movements of actors were interdependent (for example: the pursuit of one actor by another actor or a camera).

The creation of actor systems (for example: a school of fish, a flock of birds or a traffic system) by the refinement of the methods of the actors seems to follow naturally from this approach. The user could describe very complex interaction by specifying the behavior (response to messages) of relatively simple individual elements, rather than trying to script each actors response to a large number of dynamic events. This approach seems particularly appropriate to problems of simulation.

The object-oriented design of the system invites extension and refinement. The creation of new actors, benefitting from inheritance proved straight forward. The butterfly actors were added to the system, experimented with and modified three or four times in less than two hours. No new interfacing or recompilation of related modules was required. Separate compilation units defining or refining objects make convenient building blocks for the extension of an application or the design of related systems.

9.2. Future Extensions of Animator's Object Types

A major advantage of the object-oriented design is the extensibility of the objects. The actors and other types are easily refined and extended through inheritance. Actors composed of more elaborate moving subsystems seem like a straightforward extension. These actors could model more complex movements with parts that articulate. A related concept might be the design of objects that respond to messages which change their appearance. Such objects might model deformations or even transformations.

Actors that broadcast messages to the cast list would be very interesting. Such actors should be able to interact to create complex scenes with a minimum of scripting. The creation of animated environments in which no scripting was required is quite feasible with actors that can send each other messages.

A related idea would be the creation of new object types whose function is to send messages through the cast list that modified the data fields of some or all actors. A system evolving along these lines might be useful for creating simulations of real environments or exploring the possibilities inherent in theoretical or fanciful relations between animated objects.

9.3. Extensions to the System

The completion of the scripting function and the design of a frame-by-frame production system would be very satisfying. The object-oriented design seems to support an evolving system very well. It is tempting to think of the objects as computing entities that could be assembled to work together in different ways. The present object types are designed to support a production system that would produce frames to be recorded for future playback at twenty-four or thirty frames per second for video or film. The potential for specifying realistic or aesthetically satisfying movement with the present controls cannot be fully evaluated without building such a system.

Another useful extension would be an interactive turtle graphics environment, for modeling actors. The output of this would be a list of turtle graphics commands suitable for use as a DrawPoly routine.

10. Appendix: Object Pascal

10.1. Object Pascal: object types or classes

In Object Pascal, object type definitions resemble Pascal record declarations. Data fields are referenced with a dot notation or a WITH statement. Field names follow the normal rules for Pascal identifiers, but a convention followed in the language manual of using a lower case 'f' as the first letter of object field identifiers is observed in code appearing in this paper. Identifiers for object types begin with an upper case 'T'.

Vectors play a central role in the implementation of Animator, and are defined by the object type TVector, with the data fields: fX, fY and fZ.

The definition is completed by the declaration of a set of procedures and functions (methods) that operate on these data fields. Method calls follow the same pathname syntax as data fields. The header for a method that multiplies a TVector object by a scalar, k, is: PROCEDURE TVector.Scale (k : EXTENDED).

10.2. Object Pascal: methods or messages

A method call tells an object to execute one of its own methods. The method acts on the data the object stores. Let aVector be an object (object reference variable) of type TVector, and k be a scalar constant, then the method call or message: aVector.Scale(k) would multiply each data field of aVector by k (the TVector.Scale method implements scalar multiplication for the TVector type). The syntax of invoking a method is like a procedure call, where the procedure identifier is preceeded by the object reference variable as a pathname. Statements like: WITH aVector DO Scale(k) are also valid. Within the definition of a method, data fields and method calls of the object type being defined may be referenced without a pathname.

Assigments like aVector.fX : = 8.2; clearly violate the spirit of OOP, but the Object Pascal compiler does not enforce this. References like numval : = aVector.fX; do not violate the data and are thought of as function calls that return the value of the field referenced.

10.3. Object Pascal: inheritance

The Animator application must represent a variety of objects that store data about their orientation and position in three-dimensional space. These objects also need to respond to messages that act on this data. An abstract data type, TLCS (Local Coordinate System), has been defined with the data fields and methods required. The word abstract is used here to indicate that TLCS is intended to serve as a template for the design of more specific object types sharing similar requirements.

Object types can be defined as descendants of existing object types, such that they inherit all of the data fields and methods of their ancestor. The descendant types may specify additional data fields and methods and even re-implement (OVERRIDE) existing methods.

Two important restrictions assure upward compatibility; inherited data fields may not be deleted; and re-implemented methods must have the same parameter list as the inherited method. Consequently, a descendant is always type compatible with its ancestors. This often means that customized object types can be referenced by previously existing code without difficulty.

All of the objects that Animator displays on the screen trace their ancestry through the abstract data type TActor back to TLCS. Because of this, new actors (descendants of TActor), that look or perform differently, can be displayed and moved about by code written before they were even conceived of.

When viewed in terms of their inheritance, objects resemble Frames, a structure used in knowledge representation. If thisFish is of type TFish, and TFish is a descendant of TActor, and TActor is a descendant of TLCS, and TLCS objects can move, then thisFish is a TLCS object, and thisFish can move. Note the 'is a' relation and the inherited attribute, 'can move'. Useful knowledge representation schemes require multiple inheritance (more than one direct ancestor), which is not supported in Object Pascal, but is available in other OOP languages (LOOPS).

10.4. Object Pascal: instances of an object type

The identifiers TVector and TLCS describe classes that an object may belong to. The identifiers aVector and thisFish are referred to in the previous section as object reference variables. Such instances of an object type will, henceforth, be referred to simply as

objects. The identifiers (aVector and thisFish) are actually name handles (double pointers) that reference memory dynamically allocated to store an object's private data. The statement NEW(aVector) allocates memory for aVector's data. This memory is released by the statement aVector.Free. An expanded discussion of this topic including initialization and assignment is found in Vectors 4.3.

Let aFish be an object of type TFish and anActor be an object reference variable of type TActor (an ancestor), then anActor may be assigned to point to the object referenced by aFish. TFish data fields not present in the TActor type may not be referenced through anActor, but are not lost. The recovery of such data is described in Turtle 5.7.

10.5. Object Pascal: MacApp units

The MacApp units declare six major abstract object types (TView, TFrame, TWindow, TDocument, TApplication and TCommand) that support the standard Macintosh interface. Each is a descendant of the abstract object type TEventHandler. Macintosh programs are typically non-model and event driven. Mouse, keyboard and system events constitute messages, which are passed along a linked list of EventHandlers (the target chain). Whether or not any action is taken, each EventHandler passes the message to the next EventHandler on the target chain.

To make use of MacApp, a programmer must define descendants that re-implement each of the EventHandlers to correspond to the data and function of his application. The use of MacApp standardizes the application's data and user interfaces in the same way an object type creates a template for its descendants. When a suite of applications must work together, the benefits, the standardizations of internal and external interfaces, clearly outweigh the limitations imposed when different applications must share the same constraints.

10.6. Object Pascal: TApplication and TDocument

The TApplication object handles the main event loop, dispatching incoming events to appropriate event handlers. Several of its methods must be overidden to perform initialization tasks and create the TDocument object. The TApplication.Run method is called in the main program and loops until the user quits the applicaton.

TDocument objects store the data for an application's documents or files. Applications may have more than one document. The methods DoNeedDiskSpace, DoRead, DoWrite must be overridden in order to save and open documents on disk. Most importantly, methods must be defined to handle commands that add, delete or modify the document's data.

The programmer must also override the methods DoMakeViews and DoMakeWindows to control the display of the document's data on the screen. Documents may have multiple views within a single window as well as multiple windows.

10.7. Object Pascal: TFrame and TWindow

These objects provide substantial support for the display of the visible portion of the view in the familiar Macintosh windows. A frame is a rectangular area displaying either a portion of the view or a collection of subframes. The frame object sends a message to the view to draw, but first it determines the clip region and sets up a grafport origin to permit the view to make QuickDraw calls in its own coordinate system.

The window is a specialized frame, therefore Twindow is a descendant of TFrame. It handles the mouse commands that permit the user to move the window, scroll its contents, make it the active window, or drag it to a different part of the sreen. The style of a window is specified in a special resource file created by the programmer.

Although Animator uses a very simple window to conserve on space, it benefits considerably from the insulation these objects provide with respect to the global coordinate system and the determination of clipping and visibility regions.

10.8. Object Pascal: TView

The TView object has reponsibility for drawing all of the information appearing in the frame (including text). The view has its own coordinate system which may represent an image considerably larger (32,000 x 32,000 pixels) than the Macintosh screen. It responds to messages from other objects to draw some part of itself. The request will include a parameter describing the part of the view that must be drawn. If the request is from a printhandler this parameter will describe the portion of the view visible on the current page.

When the view's data is altered so as to affect its appearance on the screen, it has the responsibility of invalidating the altered area. The next time the view is updated, a parameter consisting of the union of all invalidated areas is passed back to the view. The view also keeps track of highlighting information, and provides feedback when the user moves the mouse. Animator's simulation of motion on the screen is a highly modified mouse tracker.

10.9. Object Pascal: TCommand

When the user selects a menu command, or issues a command with the command key or the mouse, a command object is usually created. The inherited methods for TCommand are DoIt, UnDoIt, ReDoIt, and Commit. These methods constitute a template for the messages which must be responded to by descendants of TCommand.

Whatever the command does in its DoIt method, it must first record enough information to restore the current state, so that it can be undone. The Commit method is called when a new command is issued, allowing the previous command object to do anything necessary to finalize the result. Some commands make apparent changes on the screen without altering the document at all, until the commit method is called.

The TrackMouse and TrackFeedback methods support command objects that are created when the mouse button goes down and active until it is released, such as Animator's Move command.

11. Bibliography

[Abel 81] H.Abelson and A. diSessa, Turtle Geometry, M.I.T. Press, 1981

[Appl 87] Apple Programmmer and Developers Assoc, MacApp, Apple Computer, Cupertino, Ca., 1987

[Appl 85] Apple Computer Inc., Inside Macintosh, Volumes 1, 11, 111, and 1V. Addison-Wesley Publishing Company, Inc., Reading Ma., 1985

[Bobrow 85] Daniel G. Bobrow, et al., "Commonloops: Merging Common Lisp and Object-Oriented Programming," Technical Report ISL 85-8, Xerox Palo Alto Research Center (PARC), Palo Alto, CA, August, 1985

[Cox 84] B. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology, IEEE Software, Vol. 1, Mo. 1, pp. 50-61, January, 1984

[Doyl 86] K. Doyle, B. Haynes, M. Lentczner, L. Rosenstein, "An Object Oriented Approach to Macintosh Application Development," Proceedings of the 3rd Working Session on Object Oriented Languages, Paris, France, January 8-10, 1986

[Fikes 85] Richard Fikes and Tom Kehler, "The Role of Frame-Based Representation in Reasoning," Communications of the Association for Computing Machinery (CACM), Vol. 28, No. 9, pp. 904-920, September, 1985

[Foley 82] J.D. Foley and A. Van Dam, Fundamentals of Interactive Computer Graphics, Addison Wesley Publishing Company, Inc., Reading Ma. 1982

[Garr 86] L. Garret and K. Smith, "Building a timeline Editor from Prefab Parts: The Architecture of an Object-Oriented Application," OOPSLA '86 Proceedings, Portland, Oregon, September, 1986

[Gold 84] A. Goldberg, Smalltalk-80: The Interactive Programming Environment, Addison-Wesley Publishing Company, Inc., Reading, Ma. 1984

[Guttag 77] J. Guttag, "Abstract data types and the development of data structures," Communications of the ACM, 20(6): 396-404, 1977

[Hewitt 77] C. Hewitt and R. Atkinson, "Parallelism and Synchronization in Actor Systems," Proceedings, ACM Symposium on Principles of Programming Languages, 1977

[Hewitt 73] C. Hewitt, P. Bishop, R. Steiger, "A Universal Modular Actor Formalism for Artifical Intelligence," Proceedings, International Joint Conference on Artifical Intelligence, pp 235-245, 1973

[Kahn 76] K.M. Kahn, "An actor-based computer animation language." MIT AI Working Paper No. 120

[Leit 81] L. Leithold, The Calculus with Analytic Geometry, Fourth Edition, Harper and Row, N.Y., 1981

[Meyr 86] N. Meyerwitz, "Intermedia: the Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework," OOPSLA '86 Proceedings, Portland Oregon, September, 1986

[Papert 70] S. Papert, "Teaching Children Thinking," Proceedings, IFIP World Conference on Computer Education, New York, pp I/73-I/78, 1970

[Penna 86] M. Penna and R. Patterson, Projective Geometry and its Applications to Computer Graphics, Prentice-Hall, N.J., 1986

[Reyn 82] C.W. Reynolds, "Computer animation with scripts and actors," Proceedings, SIGGRAPH '82, Computer Graphics, 16(3): 289-296, 1982

[Reyn 78] C.W. Reynolds, "Computer Animation in the world of actors and scripts," (SM Thesis). Architecture Machine Group, MIT 1978

[Scm 86] K. Schmucker, Object-Oriented Programming for the Macintosh, Hayden Book Company, Hasbrouk Heights N.J., 1986

[Simo] J. Simonoff, MacApp Programmers Manual, Apple Computer Inc., Cupertino, CA, 1986

[Thal 85] N. Magnenat-Thalmann and Daniel Thalmann, Computer Animation, Theory and Practice, Springer-Verlag, Tokyo, Japan, 1985

[Thal 83] N. Magnenat-Thalmann and Daniel Thalmann, "Actor and Camera Data Types in Computer Animation," Proceedings, Graphics Interface, pp 203-210, 1983