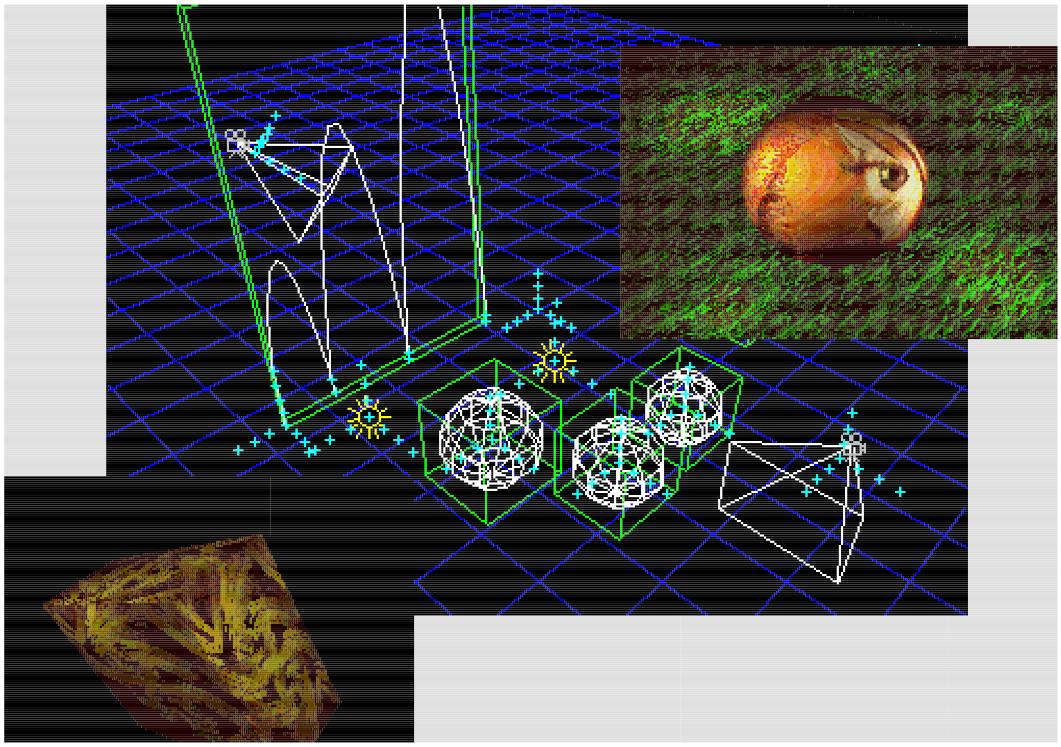
Christoph Kleiner

FOUR DIMENSIONAL DREAMS

Diploma thesis
Institute for Computer Systems
Swiss Federal Institute of Technology Zürich



March 12, 1999

Abstract

This report is the result of four months of work in the field of animated computer graphics. It covers topics from coordinate transformation, space–time positioning (keyframing in particular) up to visualization. In particular, an elegant and powerful space–time positioning algorithm is developed on the base of modules providing complete independence from any kind of coordinate transformation is developed.

As a concrete implementation, the Native Oberon system is extended with an editor that allows the visual composition of dynamic scenes. OO techniques (UML) are used for specification and implementation. The result is an extensible system based on the concept of time–variate variables, additionally providing a scene description language.

Keywords: coordinate system, time variation, motion control system, keyframing, rendering, Oberon, OO, UML

Contents

List of Figures

List of Tables

Preface

A single paragraph can only attempt to describe, what animations are to me. In fact, I have always been fascinated by dynamics and the question how they can be brought into a computational form. It is the opening of a fourth dimension where even mastering three of them is non–trivial. The task of storing an infinite number static scenes can be solved by little means and it is amazing, how simple the recipes behind commercial computer graphics software in fact are at the end.

There is challenge to bring three dimensional objects to behave in the way you imagine and to see behind the mathematical foundations of the topic. The cruical task of implementing a discretized representation of three dimensional objects laid into the computer's memory and concurrently working on their graphical representation is amazing, but hazardous and complex as well.

The tremendous amount of source code is astonishing at first and even at second sight. It is *that* easy to implement the wireframe representation or even a shaded view of a sphere but *that* hard to define a general purpose framework in the field of three dimensional dynamic scenes. What helps, is the deliberation from coordinate transformations. Without this help, I could have been not successful: having impressively looking prototypes is far from building reliable and and useful software – especially in the field of computer graphics. This circumstance may lead to intermediate depressions that can last for quite long periods of time. The fact that adding minor changes to the complex hierarchy of model–view dependencies may blow the whole visualization pipeline stands in opposite to productive and efficient work. The cry for commercial help sometimes lies near.

On the other hand: only fundamental and thus general techniques may serve as solid ground. One cannot claim being able to extend today's technology without deep knowledge of very basic mechanisms. Everything that challenges imagination power entrains the creative potential and opens doors to new and higher stages. This simple insight may free from temporary depressions and induce a strong will to battle through fields of uncertainty.

Chapter 1

Space

"Der aller scharff sinnigst Euclides hat den Grundt der Geometrie zusamen gesetzt."

Albrecht Dürer [DU25]

The base of any artificial three dimensional environment must be a flexible, reliable and – last but not least – *fast* coordinate transformation engine. The above cite is the introduction to Albrecht Dürer's book on basic geometry induced by only circle and ruler [DU25]. Following Euclid, he starts the development of all kinds of graphical primitives with the zero dimensional point, continuing with the straight line ("*Eyn gerade Lini*"), the circle ("*Eyn Zirckel Lini*") and a snake line ("*Eyn Schlangen Lini*"), ending up with a concise introduction to the construction of projectional perspectives.

The following problem motivates the mathematics that dominate behind the scenes of computer graphics: the question about how to uniquely address spacial positions in a closed box can of course be answered by counting and enumerating – and the need for a thick dictionary. A much simpler way to do so is to associate a single point with the respective distance to three adjacent bounding rectangles of the box.

Invariants have turned out to be a good recipe to guarantee robustness of programs. In addition, incrementally extending a closure containing legal data structure states under the preservation of invariants allows – under acceptable assumptions – to proof the correctness of a piece of software. And so I will compose my transformation engine by following mighty Euclid's and Dürer's lead. At the end, there will be an arbitrary deep hierarchy of coordinate systems holding powerful invariants, providing O(n) coordinate transformation.

1.1 Paths, vectors, bases, coordinates

To uniquely identify points in space, a distance measurement is an essential requirement. The intuitive difference between two points in space – the straight directed path between them – turns out to be a fundamental object: a **path**. It can hardly be imagined what the sum of two points in space is. But for two straight directed paths p0 and p1, the answer is simple: the straight directed path obtained by subsequently passing p0 and p1. As a pleasing property, addition seems to be commutative and linearly scaling a path may induce an intuitive scalar multiple.

The question about how spacial positions P may uniquely be addressed is thereby solved as follows: determine some reference point O (**origin**) and identify P by a weighted sum of straight directed paths that end up in P by starting at O. Figure 1.1.iii shows the identification of two spacial positions P0 and P1 by a set B of directed paths and the ordered sets (5,3,2) and (3,2,3) of weights. Of course, this identification does not need to be unique.

These insights are the ingredients of a mathematical model for three dimensional space. The concept of a vectorspace, based on the field of real numbers and with vectors being straight directed paths, satisfies all needs. The fact that straight directed paths (of which each can be interpreted as an infinite number of adjoining spacial positions) form the set of vectors rather than points in space does not please our intuition but must be accepted. Spacial positions P may uniquely be identified by expressing the path from O to P in terms of a fixed set of vectors B. A set B that allows the expression of every vectorspace member v in terms of a weighted sum (**linear combination**)

$$v = w[0]\,B[0] + \ldots + w[n-1]\,B[n-1], \tag{1.1}$$

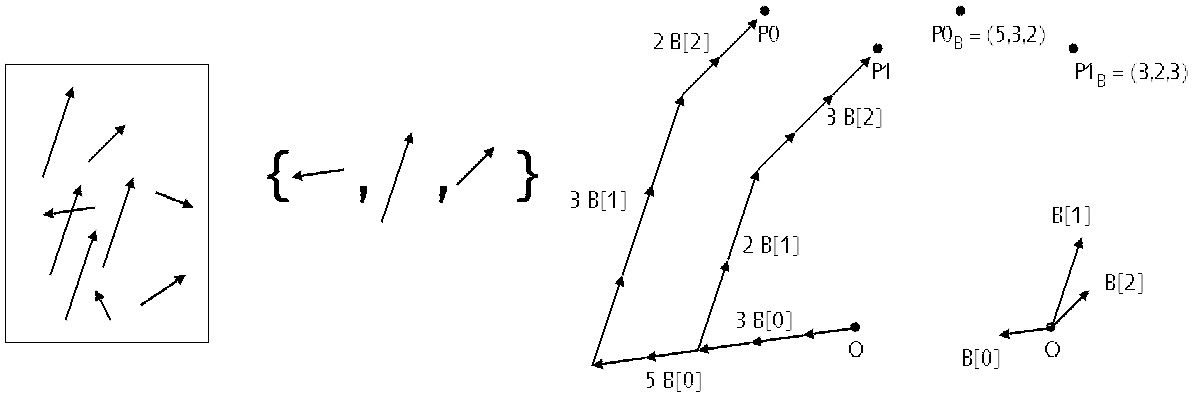
is called a **base** and the **dimension** of a vectorspace is defined as the minimal base cardinality. It is common to designate only bases of minimal cardinality as to be bases but this does not reflect the intuitive meaning of the word – I refer to them as "minimal" bases. Cardinality is minimal if it cannot be decreased, which in turn can be done by expressing one of B's members in terms of the others. So a base containing scalar vector multiples (**colinear vectors**) cannot be minimal. If none of the vectors B[i] can be expressed in terms of others – that is: the equation system

$$0 = w[0]\,B[0] + \ldots + w[n-1]\,B[n-1] \tag{1.2}$$

has no nonzero solution – they are called to be **linearly independent**. The use of minimal bases simplifies imagination and work. The ordered set w of base vector weights identifying a point P in space is called P's **coordinates**. To emphasize identification in terms of a base B, the notation

$$P_B \tag{1.3}$$

shall be used. The common coordinate abstraction is shown by figure 1.1.iv.



i) Vectorspace of straight directed paths

ii) Set B of base vectors

iii) P0 and P1 being identified in terms of B

iv) Coordinate abstraction

Figure 1.1: Spacial positioning

1.2 Base transformations

A base transformation is the process of expressing coordinates P_B in terms of a new base B'. According to (1.1), this is the task to solve

$$P = \sum_i P_B[i]\, B[i] = \sum_i P_{B'}[i]\, B'[i] = P \tag{1.4}$$

for $P_{B'}$. In the case of real–values n–tuples, (1.4) expands to the equation system

$$\begin{aligned} \{ \textstyle\sum_i P_B[i]\, B[i] \}\,[0] &= \{ \textstyle\sum_i P_{B'}[i]\, B'[i] \}\,[0] \\ &\dots \\ \{ \textstyle\sum_i P_B[i]\, B[i] \}\,[n-1] &= \{ \textstyle\sum_i P_{B'}[i]\, B'[i] \}\,[n-1]. \end{aligned} \tag{1.5}$$

Base transformations are a perfect model for seeing the world from different points of view. In terms of matrix multiplications, one gets

$$P = A\, P_{B'} \tag{1.6}$$

with A being given as

$$A = (B[0], \dots, B[n-1]). \tag{1.7}$$

B' may be specified by either explicitly enumerating its members or by coordinates and the implicit assumption that a vector is defined by the straight directed distance between O and the spacial position. If the members of B' are expressed in terms of the base B, the according notation shall be $B'_{B'}$. With $B'_B := ((1,1,0),(-2,3,1),(1,1,4))$, and $P_B := (3,8,2)$, one gets $P_{B'} = (19/4,1,1/4)$, as a simple example. Note that this transformation proves valid for an *any* base B. Figure 1.2 illustrates the objects involved in a base transformation and graphically illustrates coordinate components as parallel lines to the corresponding base vector.

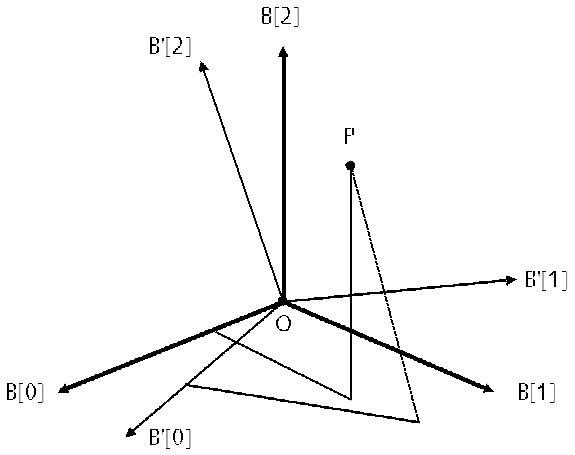


Figure 1.2: Base transformation

1.3 Orthonormal bases

As indicated by figure 1.2, the wish for having "ordered" base vectors rather than the example of figure 1.1.ii is intutionally induced, though nothing prevents from choosing arbitrary bases. Choosing B so that the solution of (1.6) (the computation of A's inverse) may easily be found, is a mathematical motivation. The lengths of base vectors and angles between them will be order criterions that support that task very well.

The length $||\cdot||$ of a coordinate shall be expressed by a real number, thereby implicitly defining the length of the vector (which is a straight directed path) from O to the represented spacial position. A very natural restriction for length measurements is that the length of scalar multiples should scale as the vector:

$$|| av || = a || v ||. \tag{1.8}$$

Note that this is *not* the restriction to linear functions (see below). L–norms defined as

$$|| v ||_L := (a[0]^L + ... + a[n-1]^L)^{1/L}, \tag{1.9}$$

include the popular Euclidean length (L=2) [NISTO94].

Cosine is 1 at 0, 0 at $\pi/2$ and –1 at π. From the fact that the angle between two identical vectors shall be 0 and that vectors with heavily different coordinates shall result in a bigger angle, we have the following definition: the **scalar product** of two vectors a and b shall be defined as

$$a \bullet b := a[0]\, b[0] + ... + a[n-1]\, b[n-1], \tag{1.10}$$

thereby implicitly defining an **angular measurement** between vectors according to the definition

$$COS(\alpha(a,b)) := a \bullet b \, (|| a || \, || b ||)^{-1}, \tag{1.11}$$

which reflects the wish that the length of a and b shall not influence the angular measurement. These definitions allow to set up length and angle criterion that will turn out to simplify the solution of (1.5): the members of an **orthogonal base** have pairwise scalar product 0 (are so–called orthogonal) and for an **orthonormal base**, they additionally have length 1. From the fact that

$$a \bullet b = 0 \qquad (\alpha(a,b) = \pi/2 + i\,\pi) \tag{1.12}$$

$$a \bullet b = 1 \qquad (\alpha(a,b) = i\,2\pi) \tag{1.13}$$

for members of an orthonormal base, the solution A^{-1} of (1.6) is given by its transpose A^T.

1.4 Affine transformations, coordinate systems

The fundamental concept behind spacial positioning is the **affine transformation** (AT). A function f on coordinates P_B is said to be **linear** if

$$f(aP_B+b) = a\, f(P_B) + f(b), \tag{1.14}$$

and **affine** if it can be decomposed into a linear function l plus a constant coordinate c:

$$f(P_B) = l(P_B) + c. \tag{1.15}$$

An orthonormal vectorspace base matrix B (inducing a rotation) followed by a constant translation coordinate t is such an affine transformation (AT) which allows the arbitrary positioning of coordinates in space while preserving their pairwise distance and angular relationship:

$$AT(P_B) := B\, P_B + t, \tag{1.16}$$

whereas its inverse is given by

$$AT^{-1}(P_B) = B^T P_B - B^T t. \tag{1.17}$$

Proof: $AT^{-1}(AT(P_B)) = B^T (B\, P_B + t) - B^T t = P_B$.

An AT is an univariate function with coordinates as domain and image. Nothing prevents from *iteratively* applying AT's and in fact, this is the nature of a **coordinate system** (CS), which is a chain of affine transformations. A CS is an AT with a **parent** AT, which is the successor in the transformation chain and a transformation hierarchy as depicted by figure 1.3 is defined. It is worth mentioning that the rotation in (1.16) precedes the translation, whereas

$$AT(P_B) := B\,(\,P_B + B^T t\,) \tag{1.18}$$

of course gives identical results with a reversed order: *translation* preceding rotation.

The paradoxon why the parent transformation AT.parent must be done *after* the considered AT, though the parent places the coordinate and the AT gives an additional displacement, can be explained as follows: imagine having transformed P_B by all its parent transformations resulting in P'_B. By applying the considered AT, the transformation process transforms P'_B with respect to the vectorspace base, whereas it should simply transform the coordinate with respect to its *parent* CS. In other words: for a rotation around the second base vector being transformed by the whole parent chain, we first of all need to restore the original coordinate P_B, then apply the AT, and must finally apply the whole parent transformation AT.parent again:

$$P''_B = AT.parent(AT(AT.parent^{-1}(P'_B))), \tag{1.19}$$

which of course yields

$$P''_B = AT.parent(AT(P_B))), \tag{1.20}$$

since

$$AT.parent^{-1}(P'_B) = P_B \tag{1.21}$$

which is AT applied on P_B *before* AT.parent. The iterative application of this rule allows hierarchical positioning as shown by figure 1.3: by affine transformations, P is pushed into space by starting at the vector space base and the chosen origin O, whereas all coordinates are at any time given with respect to B.

And *this* fact is completely different in the case of coordinate transformations. Figure 1.4 suggests the question: as in figure 1.3, CS have been graphically presented by the transformed base vectors and affine transformations being straight lines between hierarchies, whereas coordinates are represented as the lengths of the perpendiculars onto transformed base vectors. We now consider the bases C and C' and want to determine $P_{C'}$ from P_C: what are the coordinates of P_C with respect to a different CS C'?

1.5 O(n) coordinate transformation

The general problem of a coordinate transformation is, by extension of (1.4) and written in terms of matrix multiplications, given as the solution $P_{C'}$ of

$$\{ \prod_j AT[j] \} \{ \sum_i P_C[i]\, B[i] \} = \{ \prod_j AT'[j] \} \{ \sum_i P_{C'}[i]\, B[i] \}, \tag{1.22}$$

with AT[j] and AT'[j] being the ordered lists of subsequent parent transformations for C and C' as depicted by figure 1.4. Of course, $P_{C'}$ is given as

$$P_{C'} = \{ \prod_j AT'[j] \}^{-1} \{ \prod_j AT[j] \} P_{C'}. \tag{1.23}$$

Affine transformations of the form (1.16) can, with the aid of homogeneous coordinates [WA92], be written as

$$AT(P_B) = \begin{bmatrix} B & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} P_B \\ 1 \end{bmatrix} \tag{1.24}$$

A necessary precondition for the transformation process is that both CS C and C' have the same last member in their parent list. The fact that the first part of the transformation matrix in (1.23) can be computed by (1.17) in O(1) and that the product of subsequent parent transformations can be cached, allows coordinate transformations to be carried out in O(n), with n being the number of coordinates to be transformed. It is worth mentioning that the application of the concatentation of subsequent parent transformations on any coordinate yields coordinates with respect to the chosen base B:

$$P_B = \{ \prod_j AT[j] \} P_C. \tag{1.25}$$

The advantage of affine transformations is obvious: the matrix in (1.25) can be cached. For the situation of figure 1.5, with the rotational parts of AT'[0] being a rotation of $\pi/4$ and the one of AT[0] being a rotation of $\pi/2$ around B[2] and $P_C := (0,1,0)$, one gets $P_{C'} = (\sqrt{2},0,0)$ by the application of (1.23).

There is a subtle difference with the transformation of what I call directions: (1.23) gives the recipe how to transform points in space. If not the spacial position but the *direction* of the line formed by the straight distance from P to the CS origin shall be subject of the transformation process, the application of (1.23) does not end up in a correct result. In such a situation, it is necessary to transform *both* spacial positions (CS origin and P) must be transformed and thereafter be subtracted. By defining the composed transformation operator T as encountered in (1.23) as

$$T := T0'^{-1} T0 := \{ \prod_j AT'[j] \}^{-1} \{ \prod_j AT[j] \}, \tag{1.26}$$

the transformation of a direction is given by

$$P_{C'} = T P_C - T (0,0,0)_C. \tag{1.27}$$

From the fact that T is only affine but in no way linear, (1.27) cannot be simplified. In fact, it is exactly T's non–linearity that distinguishes the transformation of a direction from (1.23). (1.27) will be of use to define CS orientation with a CS being view target of an object such as a camera.

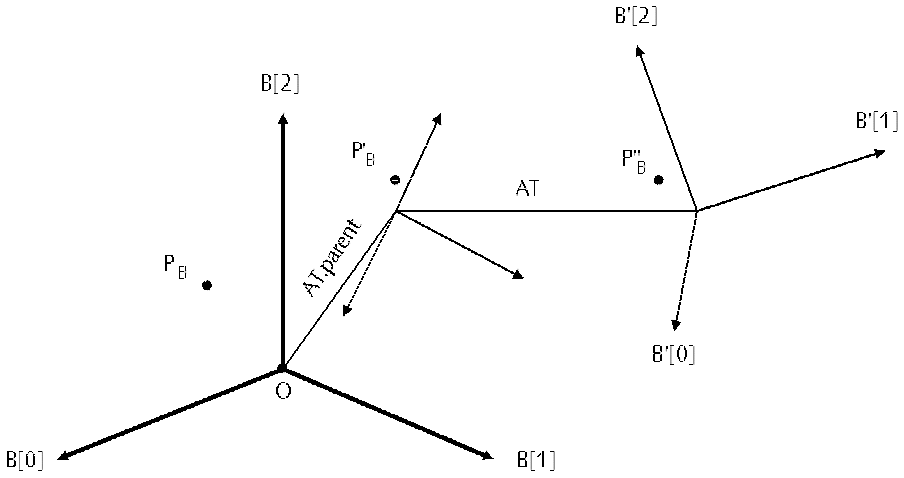


Figure 1.3: AT/AT.parent paradoxon

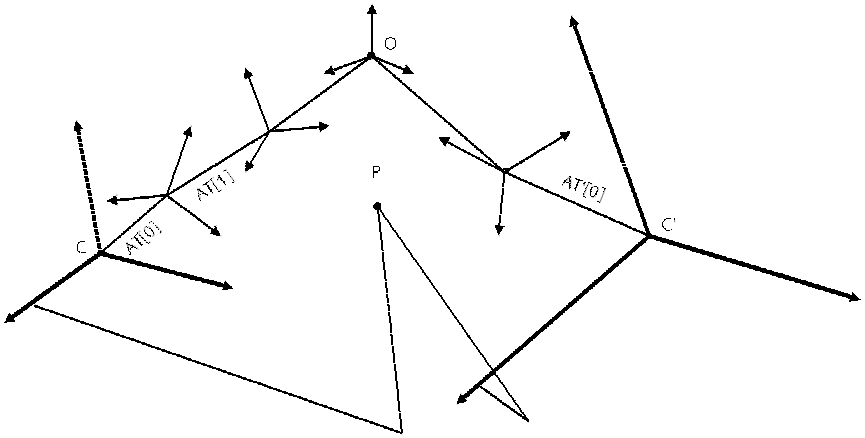


Figure 1.4: Coordinate transformation

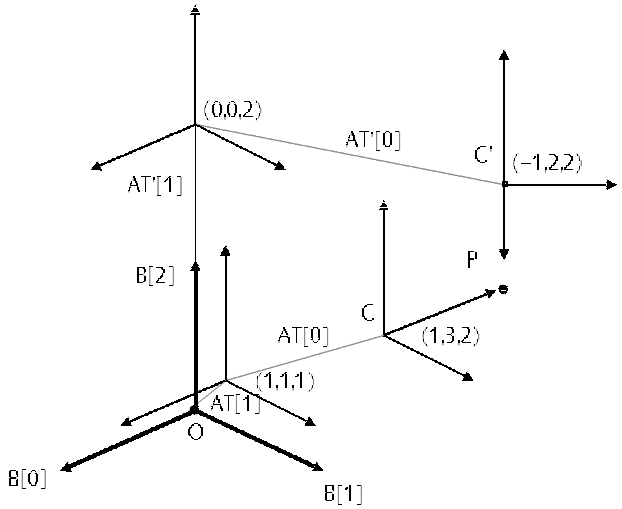


Figure 1.5: Coordinate transformation example

1.6 Implementation

Coordinates as defined in previous sections, consist of the following components: a real–valued triple, the CS they are referring to, and a hint that distinguishes directions from spacial positions to carry out coordinate transformations. Thus, the definitions summarized by table 1.1 are essentially required and lie the base for the implementation of O(n) coordinate transformations.

The definition of a CS induces a forest of trees defined by children pointing to their parent, which is a bottom–up perspective and from a conceptual point of view, it is sufficient. But the invariant computation of T0 suggests an additional view from top to reduce complexity: since a change in a CS' AT causes the necessity to recompute T0 for only its direct and indirect children, a straightforward access to them is required, whereas for a persistency model, there is no need to store this additional access structure. On the other hand, a linear list of CS induces the need to process the entire list to establish the invariance of T0 upon a change of a CS's AT, whereas it is essentially required in association with libraries [FIMA98] to make objects enter the scene upon a load request.

A tree of CS cannot be useful without coordinates referring to particular CS. The decision to add the set of coordinates to the CS descriptor lies near and suits the requirements of the OO approach. There is only a minor addition that must be made in order to provide shared objects and clones. An additional indirection (".c") in form of a reference to the array of attached coordinates allows to customize the CS whose coordinate array shall be used and be interpreted with respect to the current CS.

This concept has another pleasing property: if a second array (".r") of coordinates caches the coordinates with respect to the vectorspace base and a third one (".x") outputs the coordinates in any requested CS, this will not only speed up the transformation process but also allow the arbitrary positioning of cloned objects and their subobjects in particular, while preserving the dependence on the coordinate source, see figure 1.6. It depicts an object S sharing its coordinate array ".c" with C and C', which have different parents and may also be part of different CS hierarchies. The corresponding root coordinates ".r" are computed with the aid of T0.

To determine in O(1) whether a coordinate transformation is possible, a reference to the root of the tree hierarchy can be attached to each CS. The recomputation rules for the following set of invariants are given by table 1.2:

A) Store coordinates with respect to the CS,
B) Cache the last member of the parent recursion,
C) Cache T0,
D) Cache coordinates with respect to the vectorspace base,
E) Cache an object's local extent,
F) Cache an object's compound extent.

The universe of possible data structure states consists – up to now – of an arbitrary number of independent, iteratively created CS. It is obvious, how this closure must be extended: to provide arbitrary deep hierarchies of CS, a method to adjust the parent under preservation of all invariants is required. So far – under general rules concerning information hiding and garbage collection – only legal data structures (by means of the invariants) can be created. The set of potential data structure states is further extended by methods that allow the customization of affine transformations, thus enlarging the restriction to the identity transformation. By setting a CS parent to NIL, a node can be removed from a positioning hierarchy. Module FDDCS forms the spacial heart of FDD and implements the described techniques.

```
Cor = RECORD
    x, y, z : REAL;
    c : CS;
    d : BOOLEAN;
END;
```

```
AT = POINTER TO RECORD
    r00, r01, r02, t0 : REAL;
    r10, r11, r12, t1 : REAL;
    r20, r21, r22, t2 : REAL;
END;
```

```
CS = POINTER TO RECORD (AT)
    parent : CS;
    ...
END;
```

Table 1.1 : CS class definitions

State change	Invariants to be recomputed			
	locally	children	parents	coordinate references
Parent	{B,C,D}	{B,C,D}	{F}	{}
Affine transformation	{C,D}	{C,D}	{F}	{}
Coordinate reference	{D,E,F}	{}	{F}	{}
Coordinates	{A,D,E,F}	{}	{F}	{A,D,E,F}

Table 1.2: Invariant recomputation implications for coordinate system hierarchies

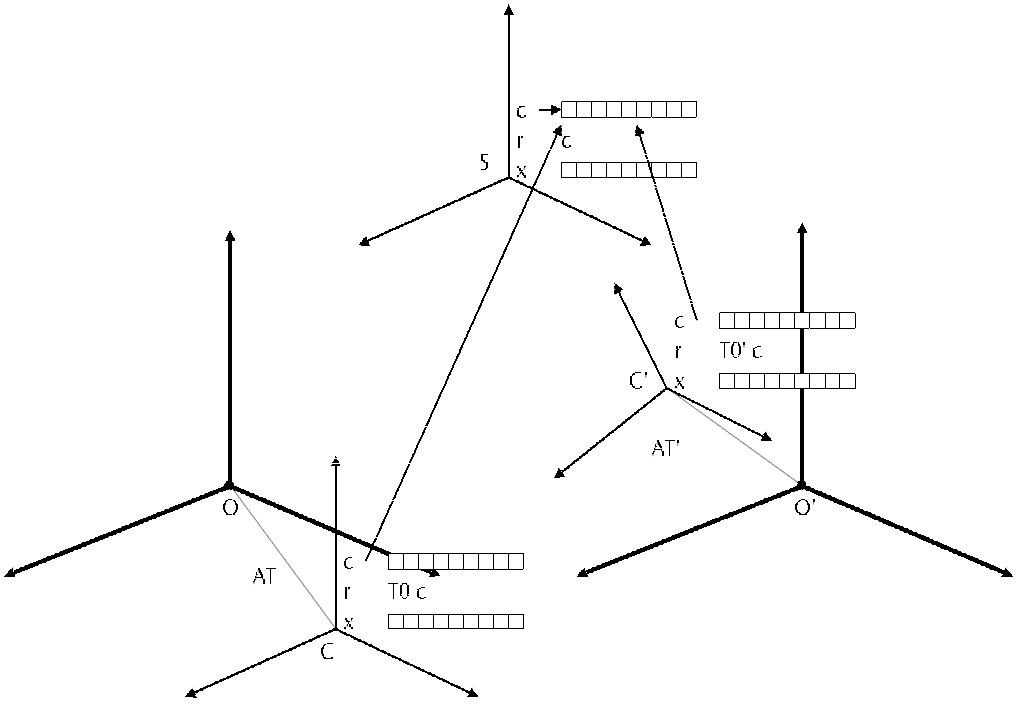


Figure 1.6: Shared coordinates

Chapter 2

Time

"Whereas rendering has mastered enough of the visual complexity of the world to achieve near photorealism, the field of animation has yet far to go in order to master a comparable degree of complexity of the world in motion."

Alan and Mark Watt [WA92]

From three dimensional *static* scenes, there is no trivial generalization to the world of animated – dynamic – scenes. The recognition that this cannot be done by simply adding a vector dimension called "time" motivates a concise formal description: given a static scene S and some scalar time t, a **motion control system** M is a function taking tuples of the form (S,t) as arguments and transforming them into the scene at time t – S(t):

$$M : (S,t) \rightarrow S(t). \tag{2.1}$$

Computationally, this corresponds to the task of storing an infinite number of scenes and it is obvious that efficient techniques to limit the amount of required memory must be developed.

The fact that all objects involved in the discussion of coordinate system hierarchies had a static nature may result in an uncomfortable uncertainty and the advent of solid solutions is accepted with great relief. A first recognition is that the set of all static scenes is a subset of the dynamic scenes: the ones with no animation at all. From the fact that there is no reason to exclude these scenes, the first step is the consideration of a static scene as being dynamic. The whole action of any object then simply is to rest at its place forever.

What also helps is the direct relationship to the traditional way of filming and its primary workflow: a whole film consists of an ordered set of animations glued together by the storyboard. For each animation, a certain amount of requisites such as scenes, actors, costumes, decorations or cameras and their respective dynamics is needed. So to *create* an animation, the first step is to create the whole set of involved objects, then arrange them into scenes and then define their dynamics and the list of camera cuts, which is – technically spoken – a linear list of cameras, each one timestamped with its activation time (figure 2.1). To compose a whole film, a tool to concat an arbitrary number of generated scenes is another necessity.

The central hurdle to overcome is the question about how dynamics in fact are brought in. A simple idea is to make the animation system keep track of a global calendar, containing the total amount of information required to generate the whole set S(t). Following OO concerns of the data structure for static scenes, a "distributed" approach should be chosen: of course, there will be paths such as splines controlling the way of particular objects through space.

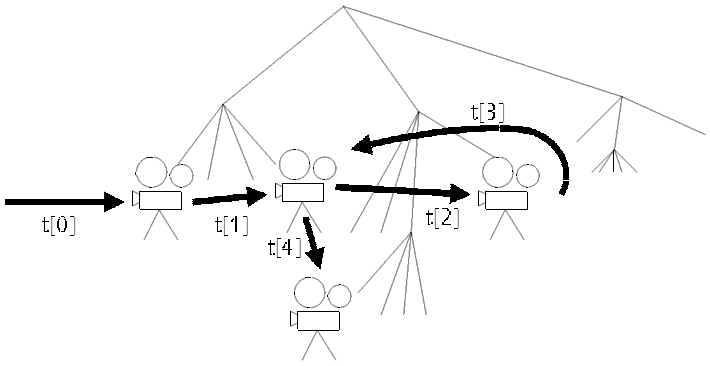


Figure 2.1: Cut list

Consequently, each three–dimensional primitive must contain its private calendar and register its whole set of dynamics. Upon a request at time t, it must place itself to the correct position according to its calendar and a potential controlling path at this time. From the fact that cameras themselves will be dynamic, the whole transformation process needs to be split up into two steps:

1) Set up the position of all objects at time t.
2) Transform all coordinates into camera coordinates.

To resume: by starting with considering static scenes as a subset of dynamic scenes, we have defined a simple basic animation defined by a list of camera cuts in a static scene. Dynamics are brought in by the OO approach of private calendars. So far, no "representational" animation [WA92] has been mentioned, but as calendars allow the definition of arbitrary states, object deformations can be implemented with little effort: a mouth will be opened by adding control points to it and defining their position with respect to time.

2.1 Time–variate variables

The static nature of the data type for affine transformations motivates the development of general techniques about how time–variation can be modeled and, in particular, can be brought into a static (in terms of time–dependency) data structure. The abstraction of **time–variate variables** (TVV) is to define objects whose **value** may vary with respect to a scalar parameter: *time* in the case of FDD. The most general approach is to assume that a TVV's value depends on a potentially arbitrary amount of data and therefore, a TVV T is a bivariate function taking an arbitrary amount of data and a scalar time parameter as input and computing its value upon them:

$$T: (data,t) \rightarrow value. \tag{2.2}$$

In fact, registers are a special case of TVV's with an assignment operator setting the current data d and being defined as

$$h: (d,t) \rightarrow d. \tag{2.3}$$

Such a simple TVV (being constant over all time) allows the description of static scenes by replacing all scene state variables. It corresponds to the generation of an infinite amount of values based on a finite amount of data. Figure 2.2.i shows a real–valued register h(d,t) being associated with the position of a sphere.

To bring in dynamics, the register is now being replaced by a TV h' defined as

$$h': (\{(t_0,d_0),\ldots,(t_{n-1},d_{n-1})\},t) \rightarrow d_i, \tag{2.4}$$

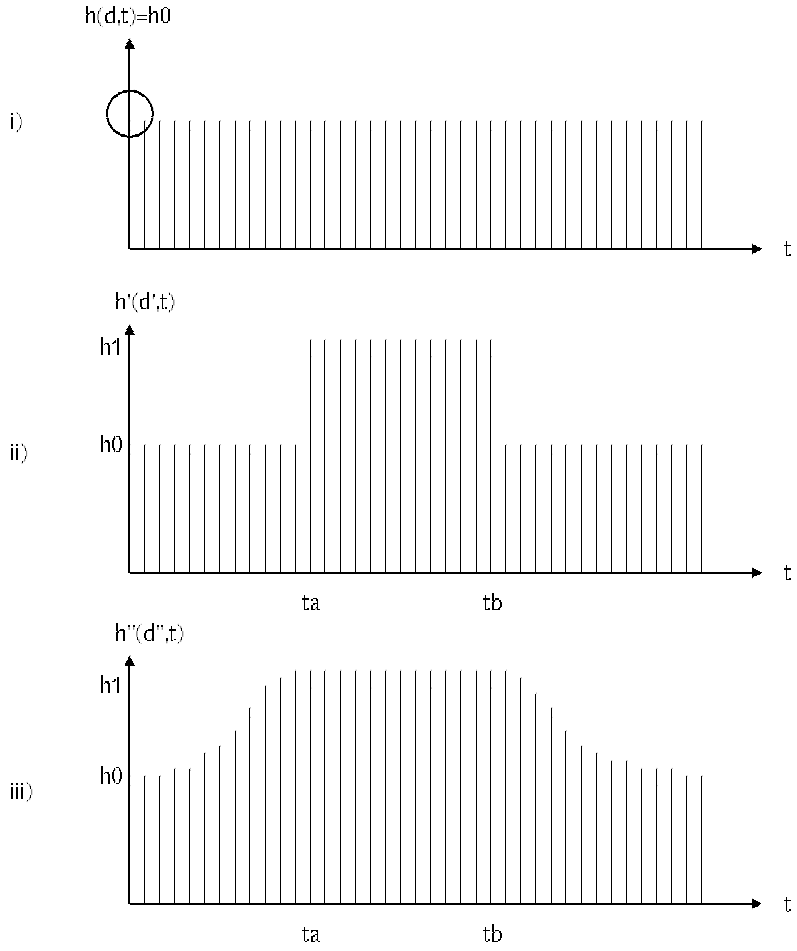


Figure 2.2: Sphere height being controlled by a real–valued TCTVV

where i is the biggest index satisfying $t \leq t_i$. With $d' := \{(0,h0),(ta,h1),(tb,h0)\}$, figure 2.2.iih demonstrates that this definition corresponds to the insertion of an infinite amount of new positions between ta and tb. The data upon which h' computes its value corresponds to three pairs (t_i,d_i) of real numbers, each defining the constant value d_i of h' for the time interval $[t_i,t_{i+1}[$.

This technique has two major drawbacks: first of all, it does not suit smoothness concerns by causing a C^0 sphere height. Additionally, the redundant third data member (tb,h0) should be replaced by more elaborate data allowing the specification of a new TV value for a fixed period of time, whereas for h', the insertion of new data entry (t_i,d_i) causes h' to take on the value d_i from t_i up to t_{i+1}. Thus, h'' has been defined as follows:

$$h'' : (\{(ta_0,tb_0,d_0),\ldots,(ta_{n-1},tb_{n-1},d_{n-1})\},t) \rightarrow V(i,t), \tag{2.5}$$

with

$$V(i,t) := R(d_i,t) \qquad (tb_i \geq t) \tag{2.6}$$
$$V(i,t) := I(d_i,t,d_{i+1}) \qquad (tb_i < t). \tag{2.7}$$

I call h'' a **track–controlled TVV** (TCTVV). R is an arbitrary function computing the TVV's value upon data d_i and I does so by interpolating between d_i and d_{i+1}, whereas the triples (ta_i,tb_i,d_i) shall be called **states**. The case (2.6) occurs if time t hits a track entry and (2.7) takes place whenever t falls between two states. Figure 2.2.iii shows h''(d'',t) with d'' being defined as $d'' := \{(0,0,h0),(ta,tb,h1)\}$ and an assumed C^1 interpolation function.

2.2 Time–variation of affine transformations

2.2.1 Forward versus inverse kinematics

Forward kinematics is the computation f of a (motion) **trajectory** [BA91] X=f(Q) upon a state vector Q, whereas **inverse kinematics** is the computation f^{-1} of Q given X: $Q=f^{-1}(X)$ (note that X can only be specified in terms of forward kinematics). In extension, the **forward dynamics problem** is the computation of the motion trajectory upon a given generalized force vector, whereas it inverse (**IDP**) does so vice versa. [LWP80] gave an elegant and recursive solution to the IDP, whereas [BA91] provides an even more efficient solution based on Cartesian tensors.

In a hierarchy of coordinate systems, the rotational part of affine transformations represents revolute/ball joints and the translatorial part does so for prismatic joints. The state vector Q of an object placed by a coordinate system consists of the whole parent–chain of affine transformations, of which each corresponds to six **degrees of freedom** (**DOF** – the number of independent position variables). The coordinate computation mechanisms of the last chapter inherently implements forward dynamics: with each coordinate system containing a track, featuring states of which each one completely describing an affine transformation, no additional effort must be made at all.

Bones [LO98] are a special kind of space–time constraints and serve as mechanism for modelling live–preserving motions. In general, a **space–time constraint** is a tuple (t,Q–) of a subset Q– of the whole set of potential parameter vectors and some scalar time t where it is active. At time t, the convention is that Q∉Q– and – of course – if a *set* of space–time constraints Q– at a time t is defined, Q may not lie within the intersection of the Q–[i]. The computation process of inverse kinematics is thereby additionally complicated and corresponds to the determination of legal (in terms of space–time constraints) state vectors Q in the state space. [WA92] claims that this corresponds to finding a *path* in its state space, but this picture of smoothness is quite wrong, since the coupling f of trajectory and state vector may be a C^{-1} function. A **bone** is the space–time constraint that the translatorial part of an affine transformation remains constant, thereby reducing the number of DOF to three (orientation), since Euler's theorem claims that each rotation can be expressed as three subsequent rotations around fixed axes. By exclusive use of forward dynamics, bones are inherently implemented by preserving distancial relationships between coordinate systems, which is the case if only the rotational part of affine transformations is modified – by the use of displacements (see below) for example.

Other space–time constraints may be the fixation of a coordinate system at a specified position in space: by an affine transformation with respect to the vectorspace base (forward kinematics). From this point of view, forward kinematics are nothing but special space–time constraints and therefore, a general motion–control system is a inverse kinematics engine trying to satisfy space–time constraints.

2.2.2 Space–time constraints

To provide powerful positioning mechanisms, three techniques are of particular interest:

i) Absolute positioning
ii) Displacements (2.8)
iii) Target objects.

Absolute positioning fixes the position of a CS at a specified point in space, which is – in the general case – done by pulling its origin to the spacial position defined by some spline at time t (spline–driven animation [WA92]). Orientation can be set to the spline's derivative at t. This concept allows to model physical behaviour by the way of defining splines such as gravity parabolas.

Displacements give an AT with respect to the object's parent, thereby directly integrating into the concept of hierarchical positioning. A very general apporach is to use a TV AT defined by a spline. Displacements show pleasant properties concerning the orchestration of multiple objects. Figure 2.3 demonstrates their usefulness in association with absolute positioning: after abolute positioning (by the coordinate systems C0 to C2), it allows multiple objects (such as the spheres S0 to S1) to behave in identical manner while taking a parallel way through space. So to define a ballet of objects of which each one shows the same rotational and translatorial behaviour with a certain spacial offset, displacements turn out to be a perfect instrument by using of some kind spline clones. If the ballet dancers decide to come together and to form a ring, private paths (A, B and C) for each object are required (figure 2.3). The continuous concatenation as required from t=1 to t=1.2 is computed by interpolation techniques.

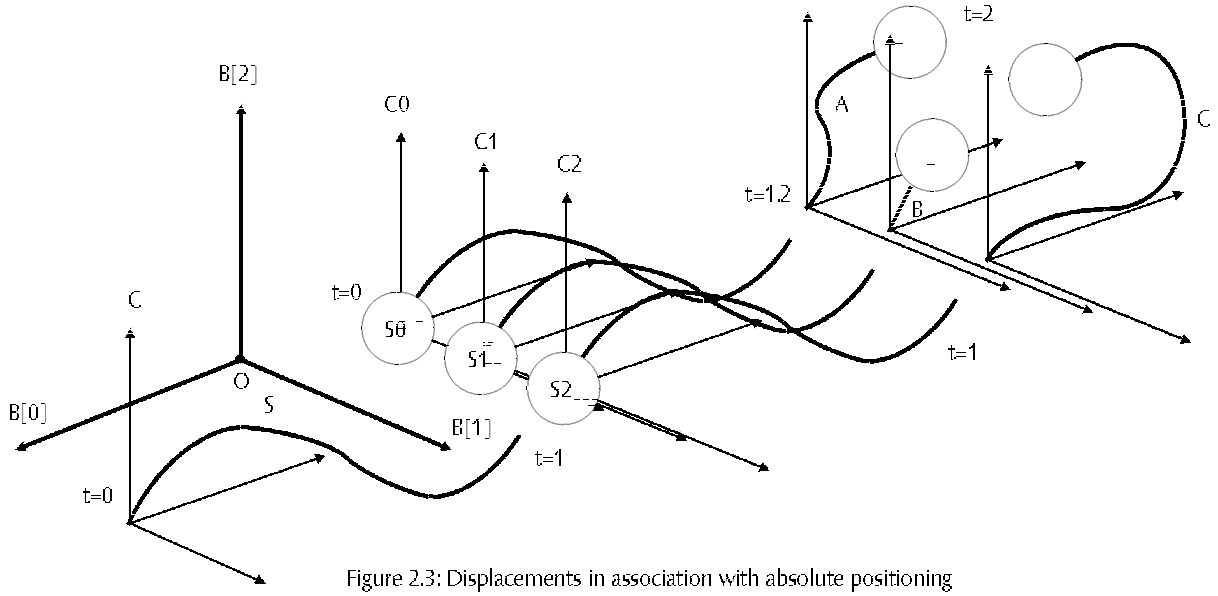


Figure 2.3: Displacements in association with absolute positioning

Target objects fix the orientation of a CS C in such a way that "looks" at another object – by the way of turning $(1,0,0)_C$ into its direction. The two kinematic constraints (2.8.i) and (2.8.iii) and the forward kinematics (2.8.ii) determine the development of the motion control system which implements time–variation of affine transformations.

2.2.3 Causality

The space–time constraints (2.8.i) and (2.8.iii) impose a partial order on the positioning algorithm: to set up an object's position, spline and target must both have set up their *dynamic* position – in all other cases, spline or target will potentially change their position and make the set up position invalid. The central word here is "dynamic": if splines/targets all had a static nature, passing in depth–first manner through the object tree and successively setting up positions according to the track entries would be sufficient.

As introduced by figure 2.3, the techniques (2.8) will heavily be combined and thus, the order in (2.8) is random by no means: the fact that setting orientation towards a target requires an object's position to be set up causes absolute positioning to be at first place. Since absolute positioning is the base for displacements, they have been set at the second place (this choice might be reversed by interchanging two matrix factors, resulting in a reversed AT application order). The graphical respresentation (figure 2.4) of AT states in tracks has also been inspired by the order in (2.8).

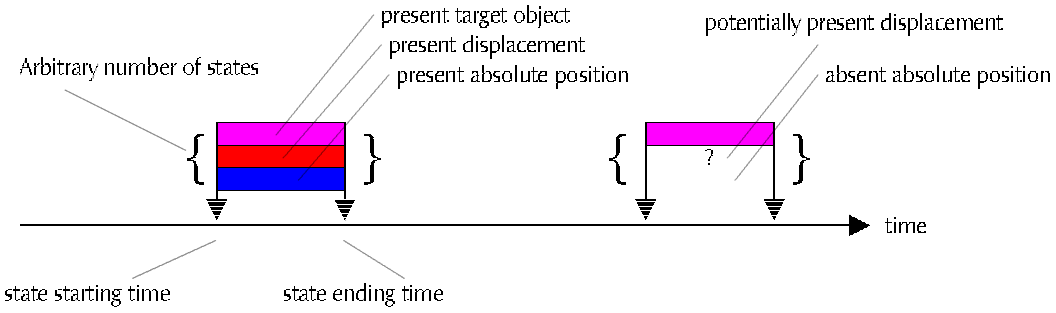


Figure 2.4: Graphical representation of AT states

2.2.4 An algorithm

According to (2.6) and (2.7), a motion control system based on tracks parimarily consists of a restore function R(·) and an interpolation function I(·). In order to scale the interpolation spline from one state to another, the interpolation function I(·) needs to be able to compute the relative time

$$r_i := (t - tb_i)\,(ta_{i+1} - tb_i)^{-1} \tag{2.9}$$

between the two states and thus, its interface must slightly be extended to read

$$V(i,t) := I(d_i, tb_i, t, ta_{i+1}, d_{i+1}) \qquad (tb_i < t). \tag{2.10}$$

The arbitrary combination of the techniques (2.8) imposes a slight complication of both R(·) and I(·) since we cannot rely on the fact that all states will consist of all three constraints. The different cases that may arise in the case of R(·) are summarized by table 2.1. The notation is as follows: aL/aR is the biggest track entry on the left/right of the hit state s with an absolute positioning, whereas s.dis(t) gives the displacement transformation of the state s. Aa target object does influence only the rotational part of an AT and so it does not need to be considered here. It turns out that all regular AT interpolation cases can be covered by

$$I(\ R(sa, sa.tb),\ sa.tb, t, sb.ta,\ R(sb, sb.ta)\ ), \tag{2.11}$$

which calls the restore procedure to determine the affine transformations at sa.tb and at sb.ta and may then compute an arbitrary interpolation. For the interpolation case, the situation is quite more complicated since at the track borders, exceptional cases such as sa or sb being NIL may arise. In certain cases, the restore procedure determines the AT in terms of the interpolation case and vice versa and this imposes the question of the termination of the algorithm. But R(·) calls I(·) only with absolute states (the affine transformations returned by R(aL,aL.tb) and R(aR,aR.ta)) as arguments and vice versa.

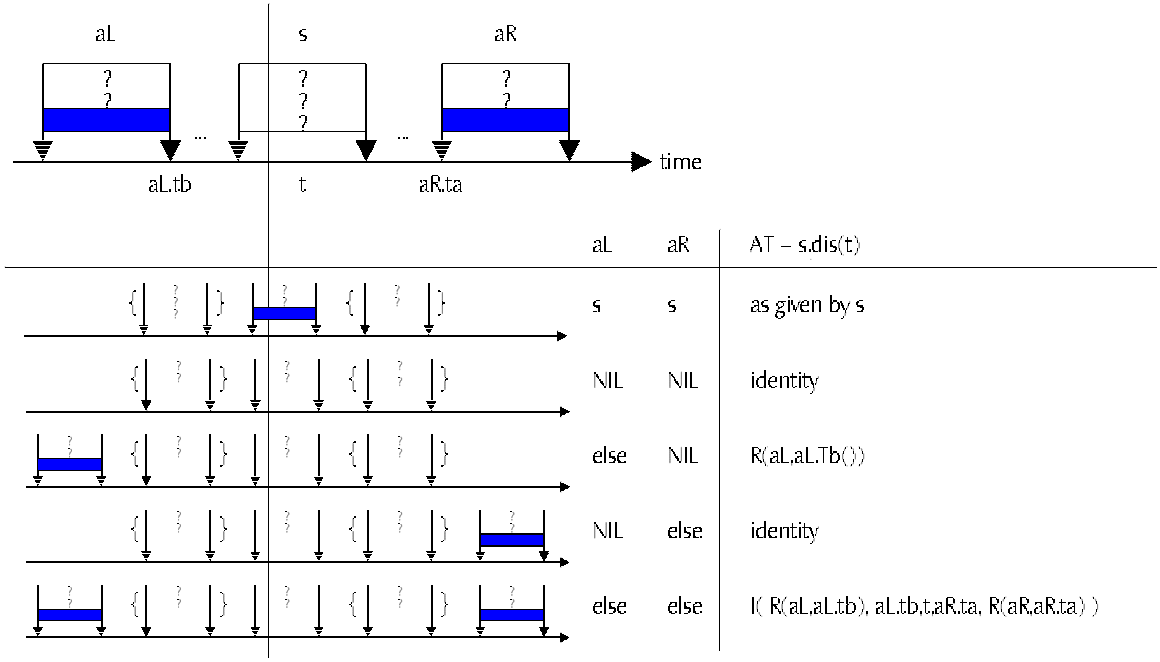


Table 2.1: AT Restore cases

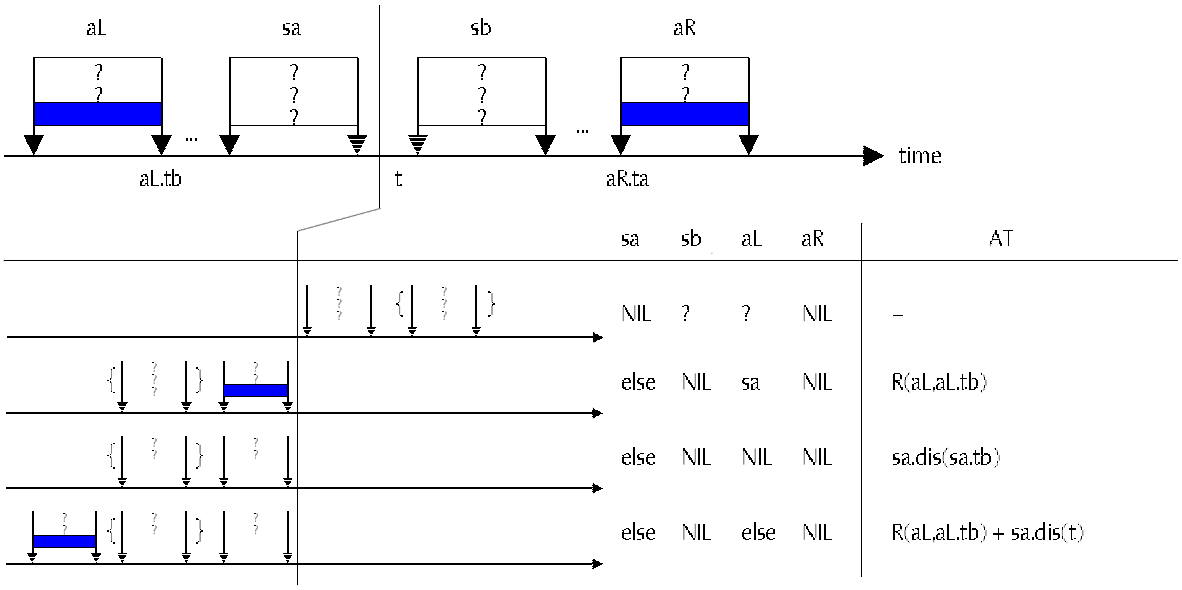


Table 2.2: AT Interpolation border cases

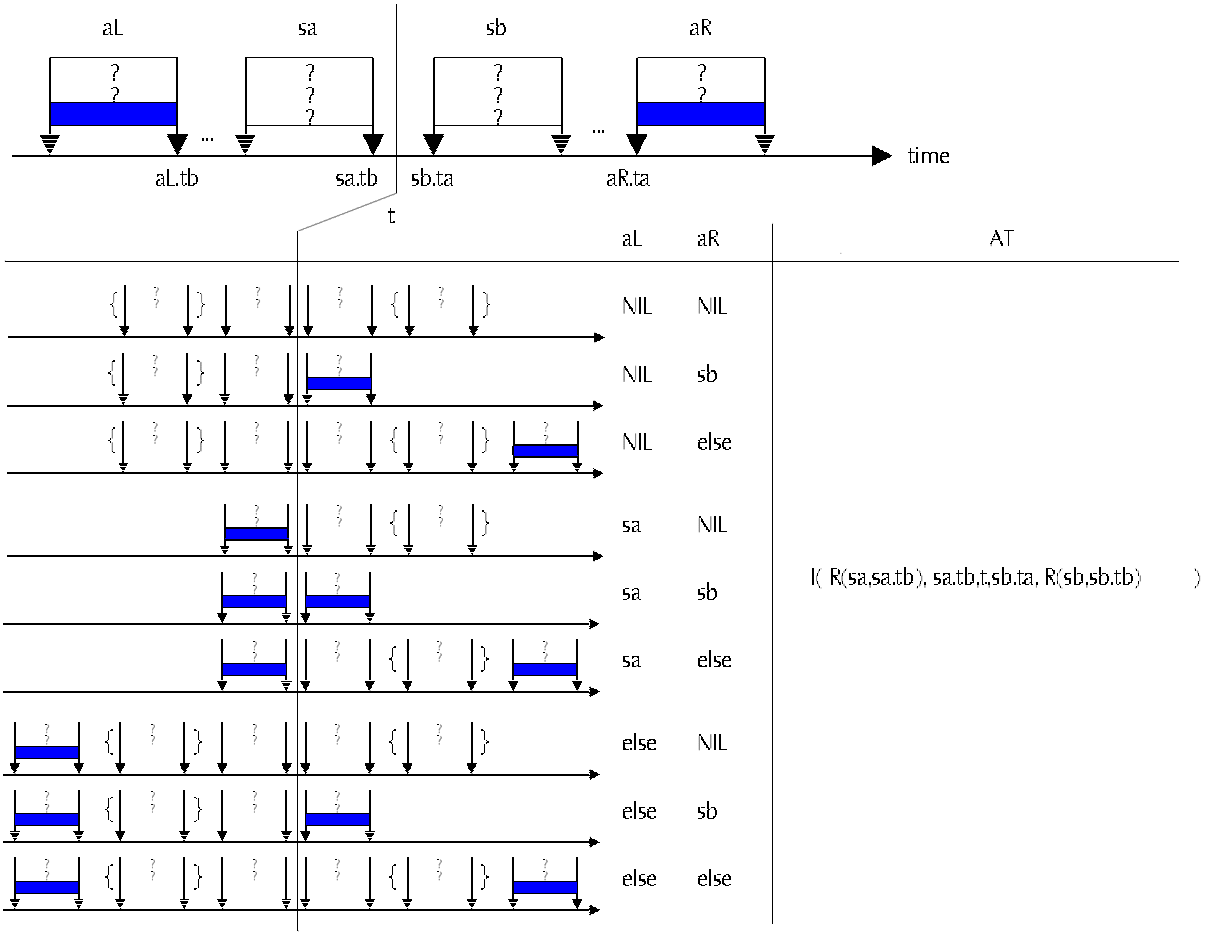


Table 2.3: Regular AT Interpolation cases

What is additionally required in the case of C^n (n>0) interpolation between AT states are derivatives at the state borders. Although the direction computation based on forward dynamics equations [LPW80] and spline derivatives is possible, the first derivative is computed by finite difference quotients for simplicity reasons. Figure 2.5 shows the situation and depicts all relevant terms. The straightforward application of derivative approximations gives the definitions

$$\partial AT/\partial t|_{sa.tb} := [\, AT(sa.tb) - AT(sa.tb - \varepsilon\Delta t_{sa}) \,]\, \varepsilon^{-1} \quad (2.12)$$

$$\partial AT/\partial t|_{sb.ta} := [\, AT(sb.ta + \varepsilon\Delta t_{sb}) - AT(sb.ta) \,]\, \varepsilon^{-1}. \quad (2.13)$$

This requires the time–space model to be additionally evaluated at $sa.tb - \varepsilon\Delta t_{sa}$ and $sb.ta + \varepsilon\Delta t_{sb}$. A special case arises if Δt_{sa} or Δt_{sb} are zero, which is caused by a track entry of length zero. Then, the derivative shall be given by the straight directed distance to the AT at the start of the next track entry:

$$\partial AT/\partial t|_{sa.ta} = \partial AT/\partial t|_{sa.tb} := f\,[\, AT(sb.ta) - AT(sa.tb) \,] \quad (2.14)$$

$$\partial AT/\partial t|_{sb.ta} = \partial AT/\partial t|_{sb.tb} := f\,[\, AT(sb.next.ta) - AT(sb.ta) \,] \quad (sb.next \# NIL) \quad (2.15)$$

$$:= \partial AT/\partial t|_{sa.tb} \quad (else), \quad (2.16)$$

wich an arbitrary scaling factor f.

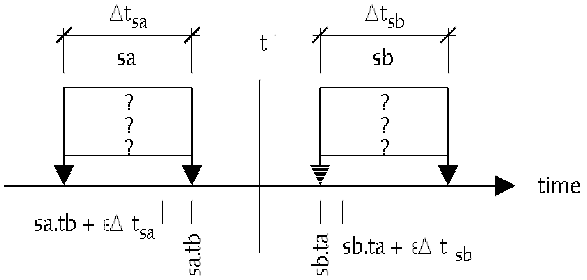


Figure 2.5: AT derivative computation terms

2.2.5 Orientation

The postprocess (2.8.iii) which potentially turns a coordinate system into the direction of a specified object turns out to be a good discussion subject for the parametrization of orientation. Fixing orientation in terms of Euler angles shows major drawbacks such as the well–known Gimbal lock [WA92] caused by a parametrization singularity. Additionally, from the non–communitativity or rotational transformations, interpolating angles does not result in proper results as with the interpolation of the translation offset. Conversely, the parametrization in terms of affine transformations corresponds to the principle of angular displacement [WA92], resolving the drawbacks of Euler angles.

It remains the problem of interpolating orientations, which has successfully been solved by spherical angle interpolation based on quaternions [WA92]. However, this technique also has a major drawback: quaternion interpolation does not prefer any one direction to any other [WA92] and thus the constraint to have a fixed camera up–vector $(0,0,1)_C$ towards the sky becomes an impossibility. From the fact that a major part of potential animations will have exactly such an up–vector, a workaround is as follows: from the fact that having defined an object's position and having $(1,0,0)_C$ pointing towards a specified target leaves one remaining DOF (the rotation around $(1,0,0)_C$), it lies near to bind this DOF to a vertical direction of the up–vector. Consequently, we have

$$B'[2] = B''[2] - P(P''[2],B'[0]), \quad (2.17)$$

as shown by figure 2.6.iii, with P(a,b) being the orthogonal projection of a onto b. Note that (2.17) does not necessarily have length one and so norming B'[2] becomes a necessity. The direction of the second transformed base vector B'[1] can thereafter be computed as the perpendicular on B'[2] and B'[0]:

$$B'[1] = B'[2] \times B'[0]. \quad (2.18)$$

The described technique does also work in the case of figure 2.6.i, where the first base vector is given by the normed spline tangent, but it fails with a vertical B'[0]. A solution to these cases is an additional TVV specifying the undefined up–vector.

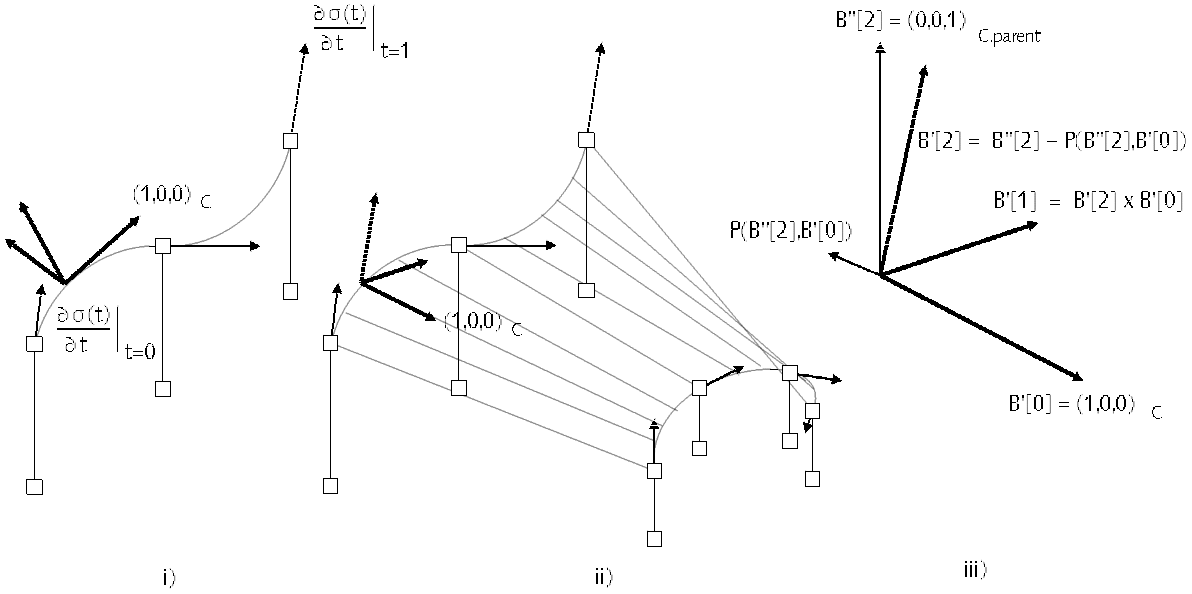


Figure 2.6: Orientation from derivatives and targeting

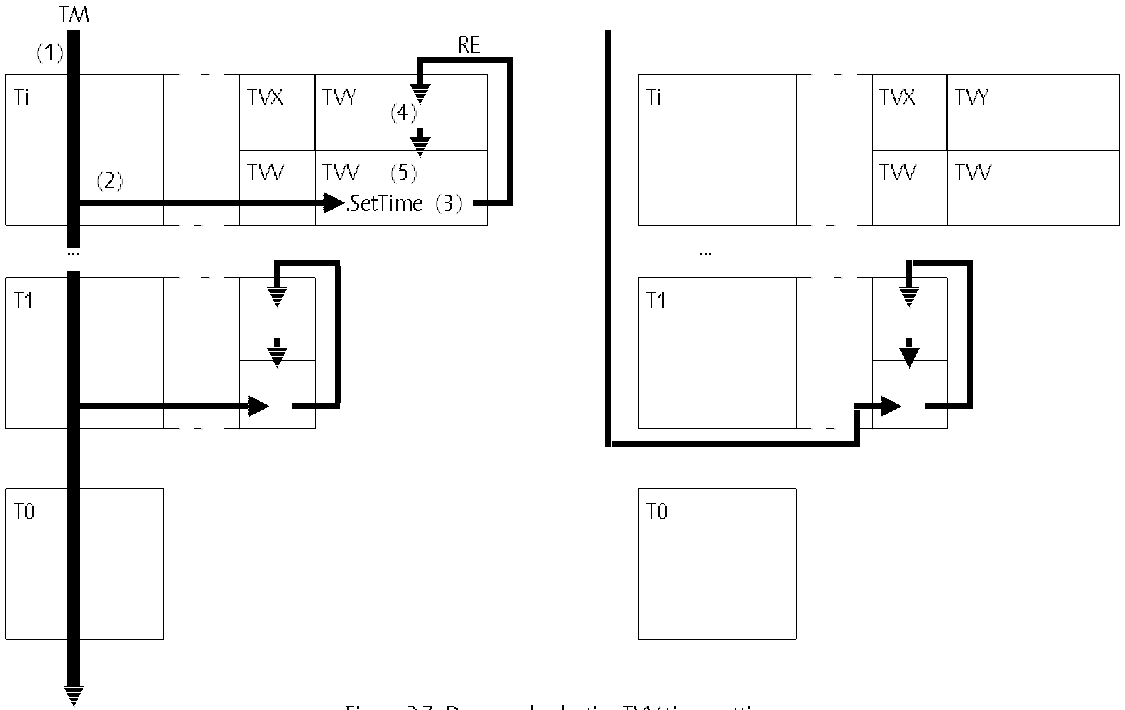


Figure 2.7: Deep and selective TVV time setting

2.2.6 TV type hierarchies, setting time

In general, there are two ways how TV can be introduced: by either deriving a TVV from a TV base class or by extending existing, non–TV types, with TV variables. The definition of CS is non–TV and thus, certainly both techniques will be used. Figure 2.7, whereas the vertical direction represents hierarchies of derived classes and the horizontal direction gives added variables in terms of inheritance, shows the most general case: with the addition of TV variables of type TVX and TVY, a non–TV base type is is successively extended from T0 to Ti.

To set up time, a corresponding message TM is sent to the object (1). The handler of Ti then (2) calls the "SetTime" handler defined by the base class TVV of all TV variables (and of course passes the message to *its* base class). This handler compares the new time to be set with the current time of the TVV and – if it differs from it – sends a restore message RE to itself (3), which is catched by the handler defined by the derived type (TVY in this case). TVY is now enabled to compute its value according to the new time setting (4) and after having done so, it passes the message on to its base type, which is an instance of class TVV in the case of figure 2.7. This TV class sets up its value by assigning the newly set time to a real–valued variable (5). Finally, Ti's base type may proceed in the same manner.

In the case of a track–controlled TVV, even more functionality is provided by the base class TCTVV: upon a call of its "SetTime" handler, it determines upon the new time to be set, whether the value needs to be restored (when a track entry is hit) or to be interpolated (in all other cases) and thereafter sends no recomputation but accordingly either a restore or interpolate message to itself.

The described message–based technique is referred to as **deep time setting**, since all TV parts of type Ti set up their time according to TM. The algorithm implementing TV of affine transformations imposes a computational problem in huge type hierarchies: in order to compute AT derivatives, the time–space model must be evaluated at several points in time. With deep time setting, each time, the values of the total TV amount of an object will be set (and potentially be recomputed), whereas the algorithm simply requires a TV AT to be evaluated. Thus, figure 2.7 demonstrates the principle of **selective time setting**, by which the initiator directly calls the "SetTime" method of the TV to be warped in time.

2.2.7 Hermite interpolation

In principle, any interpolation technique such as Bézier curves or B–Splines can be used to implement (2.7). As a concrete example, the so–called Hermite interpolation [GR97] shall be mentioned, which is a polynom–based approach that allows the fixation of an arbitrary derivative for a single segment at the border, thus allowing the insertion of continuous space curves between arbitrary splines.

Two real–valued function values f(0) and f(1), with f'(0) and f'(1) being the respective derivatives, yield, together with the spline definition

$$s(t) := a\,t^3 + b\,t^2 + c\,t + d, \tag{2.19}$$

the equation system

$$\{\, s(0)=f(0),\ s'(0)=f(0),\ s(1)=f(1),\ s'(1)=f(1) \,\}, \tag{2.20}$$

which is solved by

$$a = -2f(1) + 2f(0) + f'(1) + f'(0),\ b = -2f'(0) - 3f(0) + 3f(1) - f'(1),\ c = f'(0),\ d = f(0). \tag{2.21}$$

By collecting terms with respect to f(0), f'(0), f(1) and f'(1), interpolation coefficients are obtained as the well known Hermite polynoms:

$$s(t) = [\, 2t^3 - 3t^2 + 1 \,]\, f(0) + [\, t^3 - 2t^2 + t \,]\, f'(0) + [\, -2t^3 + 3t^2 \,]\, f(1) + [\, t^3 - t^2 \,]\, f'(1). \tag{2.22}$$

The polynomial coefficients can easily be memorized when being aware that the f(0) coefficient must be 1 for t=0 and 0 for t=1 (on each border with a derivative of 0) – the opposite as for the f(1) coefficient. The f' coefficients fit the derivative by addition and so they must have a derivative of 1/0 at 0 and of 0/1 for the f'(0)/f'(1) coefficient. A space curve suiting vector sized border conditions can be gained by a componentwise Hermite interpolation.

2.3 Time–variate surfaces

As a second application of time–variate variables, time–variate surfaces shall be discussed. From totally three surface types, an implementation should generate smooth transitions from one to another using arbitrary interpolation techniques. TV surfaces can be seen as an example of vertex animation [WA92]: scalar values such as RGB color components are the dynamic sizes in this case.

The three surface types are A) Color, B) Texture and C) Picture, whereas the last is nothing than a special texture. The question about how textures are interpolated can easily be answered as follows: by interpolating colors at corresponding texture coordinates.

The infinite resolution of procedural textures causes a problem in association with bitmaps of finite resolution: when the interplation is done explicitly, the former looses its very pleasing property. From a discretization process, a bitmap of identical resolution must be created and the interpolation result would be given by the pixelwise color interpolation.

There is an elegant workaround: by defining the value (2.5) V(i,t) of a TCTVV TV surface as

$$V(i,t) := (d_i, d_{i+1}, 0) \qquad (tb_i \geq t) \qquad (2.23)$$

$$V(i,t) := (d_i, d_{i+1}, w), \qquad (tb_i < t) \qquad (2.24)$$

the interpolation can be done by the renderer after having determined the texture coordinates – at floating point precision . The value V(i,t) is thereby implicitly given as two surface states and a weight w for the second one (d_{i+1}). With C^0 interpolation, it is then computed as

$$(1-w)\, d_i\, (1-w) + (w)\, d_{i+1}. \qquad (2.25)$$

This technique additionally avoids the creation of a temporary interpolation bitmap. Primitive–specific vertex animation can be added by derivatives of TV CS.

2.4 Implementation

The implementation of the TVV framework (module FDDTVV) is straightforward and the corresponding class definitions may be seen from table 2.4.

The concrete implementation of the outlined motion–control system must in particular provide solutions to (2.8.i) and (2.8.ii). To provide a powerful approach in the latter case, general displacements are implemented as lists, whereas the displacement is given by the concatenation of the AT specified in each node (module FDDDis).

The affine transformations of coordinate systems as defined by module FDDCS are all of a static nature and need to be equipped with an "engine" in order to make them TV. They have been designed to a priori not being TV, since this might not be necessary in many applications such as two dimensional drawing packages, math programs or editors for any static objects. Upon the following preconditions, the reader might try to find the "best" solution to add TV:

(1) Each CS has an AT.
(2) To protect invariant computation, this AT can only be set by methods.
(3) A CS is not TV.

With FDDTime, I give *my* answer in addition with an elegant and simple space–time positioning algorithm. The nature of a TV CS, wich is a special CS, suggests inheritance: a TV CS is defined by table 2.4 and proposed by figure 2.7: A TV part is added to the class descriptor which by itself has a reference to the CS that allows the AT to be modified. Adding a TV variable (featuring some reference to a state variable to be modified) to a descriptor is the only way to bring in TV into a hierarchy of derived non–TV types. If such a non–TV variable of type X is equipped with TV (as the CS' AT is it here), the type of the added TV should be called XTime (if a new TV such as a TV integer is defined, I propose to call its type "TVInt").

```
TVV = RECORD
    time : REAL;
    ...
END;

TVCS = RECORD (CS)
    at: ATTime;
    surface : TVSurface;
    ...
END;
```

```
TCTVV = RECORD (TVV)
    track : List
    ...
END;

ATTime = RECORD (TCTVV)
    cs : TVCS;
    ...
END;

TVSurface* = RECORD (TCTVV)
    val0 : TVSurfaceState;
    val1 : TVSurfaceState
    w : REAL;
    ...
END;
```

```
State = RECORD (Listnode)
    ta, tb : REAL;
    interpolation : INTEGER;
END;

ATState = RECORD (State)
    manner : INTEGER;
    pos, lookAt : TVCS;
    dis : List;
END;

TVSurfaceState* = RECORD (State)
    type : Integer;
    procedure : String;
    picture : String;
    color : Color;
END;
```

Table 2.4: TVV class definitions (simplified)

```
PROCEDURE RestoreAT(s : ATState; time : REAL);
VAR a, dis, atL, atR : AT; aL, aR : State;
BEGIN
    IF s.pos#NIL THEN
        s.pos.at.SetTime(time,FALSE); (*)
        a.t := s.pos.Spline(time);
    ELSE
        (aL,aR) := AbsBounds(s);
        IF (aL#NIL) & (aR#NIL) THEN
            a := InterpolateAT(aL,aL.tb,time,aR.ta,aR,aL.interpolation);
        ELSIF (aL#NIL) & (aR=NIL) THEN
            a := RestoreAT(aL,aL.tb);
        ELSIF (aL=NIL) & (TRUE) THEN
            a := identity;
        END;
    END;
    a := a + s.dis(time);
    IF s.lookAt#NIL THEN
        s.lookAt.at.SetTime(time,FALSE); (*)
        CASE s.manner OF
            DERIVATIVE :
                a.B := d/dt s.lookAt.Spline(time);
            | TARGET :
                a.B[0] := FDDCS.Norm( -a.t + s.lookAt.Spline(time));
                a.B[2] := (0,0,1)cs.parent - FDDCS.ProjectOrthogonal( (0,0,1)cs.parent, a.B[0]);
                a.B[1] := a.B[2] x a.B[0];
        END;
    END;
    cs.SetA(a);
END RestoreAT;
```

The marked lines (*) induce a complication: from the fact that InterpolateAT calls RestoreAT at different points in time, the original AT of the positioning and target object must be backed up and restored, after the AT has been read out. This is the reason that a topological tree traversal is an impossibility. Figures 2.9 and 2.10 show the results obtained from the derived algorithms, which are implemented by module FDDTime. Figure 2.8 depicts three key frames of the Salto morphose animation, which demostrates the application of a displacement stack (two rotations around different axes and with different angular velocities) in combination with a TV surface.

```
PROCEDURE InterpolateAT(sa : ATState; ta, time, tb : REAL; sb : FDDTVV.State; manner : INTEGER);
CONST E = 1/1000; F = 1/5;
VAR aL, aR : State; a0, a1, a2, a3, da0, da1, a, dis : FDDAT.ATDesc; parent : FDDCS.CS; dta, dtb, dt : REAL;
BEGIN
    (aL,aR) := AbsBounds(sa,sb);
    IF (sb=NIL) THEN
        IF (aL#NIL) THEN
            RestoreAT(aL,aL.Tb());
        END;
        cs.Overlay(sa.dis);
    ELSE
        a0 := RestoreAT(sa,sa.Tb());
        a1 := RestoreAT(sb,sb.Key());
        dta := sa.Tb()-sa.Key(); dtb := sb.Tb()-sb.Key();
        IF (dta#0) THEN
            da0 := a0 + 1/(E*dta)*(a0 - RestoreAT(sa,sa.Tb()-E*dta);
        ELSE
            dt := tb-ta;
            IF (dt#0) THEN
                da0 := a0 + F*(a1-a0);
            ELSE
                da0 := a0;
            END;
        END;
        IF (dtb#0) THEN
            da1 := a1 + 1/(E*dtb)*( RestoreAT(sb,sb.Key()+E*dtb) - a1));
        ELSE
            IF (sb.Next()#NIL) THEN
                dt := sb.Next().ta-sb.tb;
                IF (dt#0) THEN
                    da1 := a1 + F*(RestoreAT(sb.Succ(),sb.Next().ta-a1));
                ELSE
                    da1 := a1;
                END;
            ELSE
                da1 := a1 + (da0-a0);
            END;
        END;
        CASE manner OF
          FDDTVV.C0 : a := C0(a0,ta,time,tb,a1);
          | FDDTVV.C1 : a := C1(a0,da0,a1,da1, (time-ta)/(tb-ta));
        END;
    END;
    cs.SetA(a);
END InterpolateAT;
```

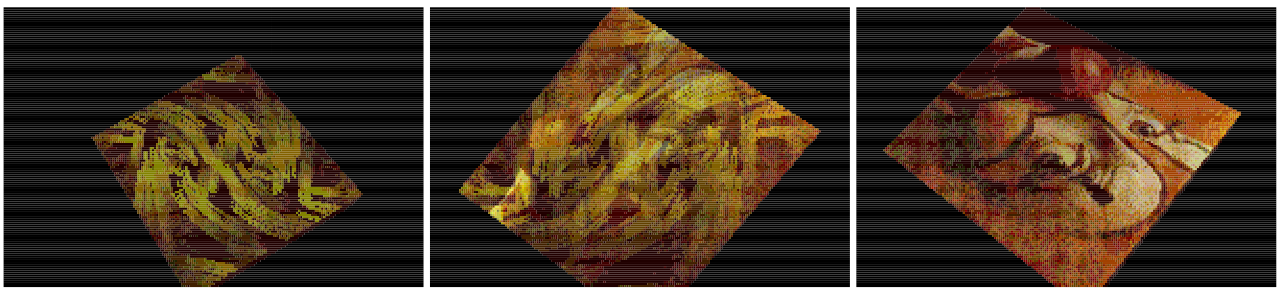


Figure 2.8: Time–variate surface

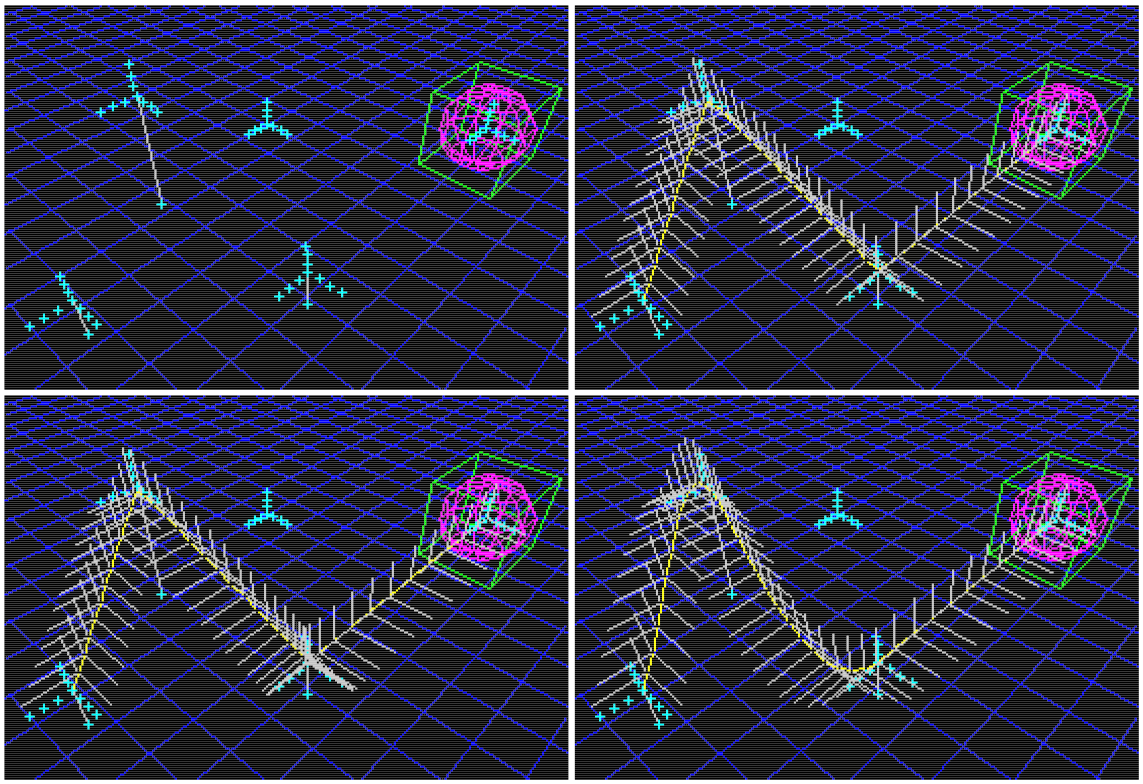


Figure 2.9: Keys, C^0 and C^1 interpolation space paths

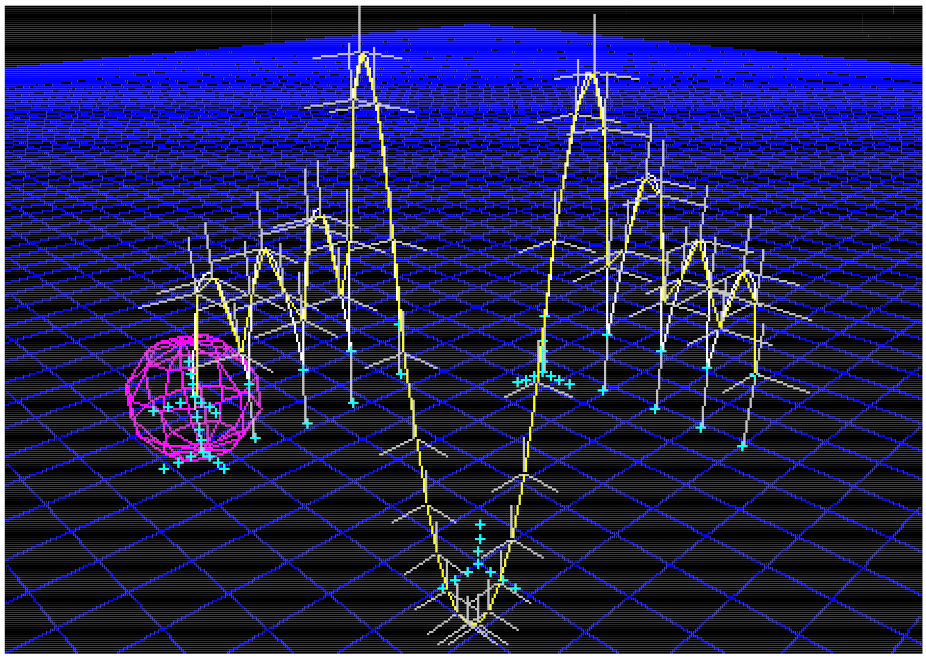


Figure 2.10: C^1 spline concatenation

Chapter 3

Interaction

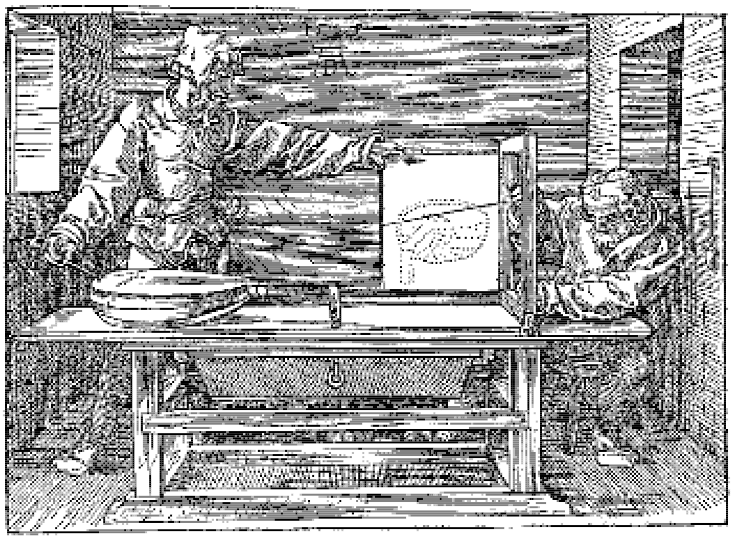


Figure 3.1: "Ein andre meynung"

"... lass deinen Gesellen die Nadel mit dem Faden hinaus strecken auf die nöttigsten Puncte der Lautten, und so oft er auf einem still helt unnd den langen Faden anstreckt so schlag alweg die zwen Feden an der Ram kreuzweys gestractes an den langen Faden und kleb sie zu peden Orten mit einem Wachs an die Ram, und heyss deinen Gesellen seinen langen Faden nachlassen. Darnach schlag die Türlein zu unnd zeychen den selben Puncten da die Feden kreuzweys obereinander gen auf die Tafel, darnach thu das Türlein wider auf und thu mit einem anderen Puncten aber also piss das du die ganzen Lauten gar an die Tafel punctierst, dann zeuch all Puncten die auf der Tafel von der Lauten worden sind mit linien zusame so siechst du was daraus wirt, also magst du ander Ding auch abzeychnen."

To Dürer – the mighty artist –, this technique did not seem to be very honorable and he thus put it under the title "Ein andre meynung" at the very end of his last book in expectation that this very technical method had the potential to become particularly important in mankind's future, what impressively outlines his open mind. The illustration shows rendering in the 16th century.

Interaction is split up into two parts: visualization and modification. The former covers graphical representation whereas the latter is the task of customizing the data structure. The topics are strongly related: intuitive modification is based on graphical representation which on the other hand gives feedback during this customization process. With this chapter, FDD will definitively be recognizable as to be a project in the field of computer graphics.

3.1 Frames

A **frame** is a generic concept to implement interaction. It serves both as data source and sink by forming the view and control part of the model–view–control (**MVC**) paradigm. Its visible part is a rectangular area with width W and height H on the screen. Formally, it can be described as an abstract object F featuring a two–dimensional array state variable of size WxH with each entry being an arbitrary color.

A frame F accepts a TV input (M.X,M.Y,M.K)(t), where (M.X,M.Y) are mouse coordinates with respect to its lower left corner and M.K is the set of pressed mouse buttons. On receiving input events, the frame either changes the graphical representation of the underlying model or modifies it, which also induces the recomputation of the graphical output. In a concrete application, a frame may decide to do one of the following upon receiving a mouse event:

A) Change the view
B) Customize the data structure
  B1) Start a construction process
  B2) Modify upon selection

Point A) induces a customization of the camera, which the current view is based on. The camera itself is an object within the CS hierarchy (figure 3.2) with the observer being placed at $(-d,0,0)_C$, C being the camera CS, a **look–at vector** $(1,0,0)_C$ and $(0,0,1)_C$ being the so–called **up–vector**. In order to provide an universal navigation utility, one may decide to overlay the following parametrized (in ΔM.X and ΔM.Y – the changes in mouse coordinates) affine transformations $AT_{nav[i]}(\cdot)$ to the camera's AT:

$$AT_{nav[0]}(P) := I\,P + \{\,\Delta M.X\,(0,1,0)_C + \Delta M.Y\,(0,0,1)_C\,\} \quad (3.1)$$

$$AT_{nav[1]}(P) := I\,P + \{\,\Delta M.Y\,(1,0,0)_C\,\} \quad (3.2)$$

$$AT_{nav[2]}(P) := R((0,0,1)_{C.parent},\Delta M.Y)\,\{\,R((0,1,0)_C,\Delta M.X)\,P\,\} \quad (3.3)$$

$$AT_{nav[3]}(P) := R((1,0,0)_C,\Delta M.X)\,P, \quad (3.4)$$

where I denotes identity and $R(\alpha,\Delta\varphi)$ is the rotation of $\Delta\varphi$ rad about an axis α. $AT_{nav[0]}(\cdot)$ is a translation in the plane defined by the vectors $(0,1,0)_C$ and $(0,0,1)_C$, $AT_{nav[1]}(\cdot)$ implements a camera dolly, $AT_{nav[2]}(\cdot)$ is the customization of the look–At vector $(1,0,0)_C$, and $AT_{nav[3]}(\cdot)$ simply rolls the camera. (3.1) to (3.3) ensure a vertical (with respect to C's parent) up–vector, whereas (3.4) does not of course.

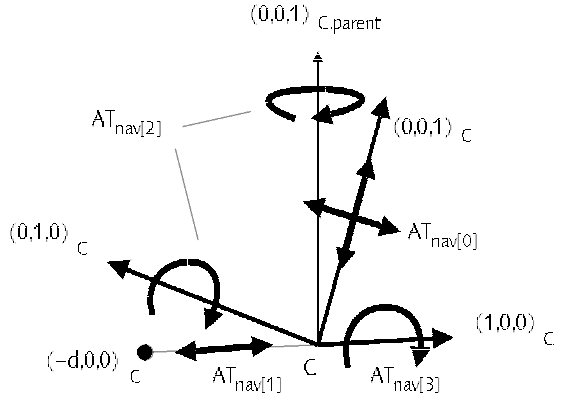


Figure 3.2: Camera navigation

3.2 Wireframe representation

A wireframe represents a three dimensional object with the aid of an arbitrary amount of lines. Despite of the large amount of oblique projection techniques [GR97], projectional perspectives may give the most realistic impression of a scene in terms of lines. On the other hand, they induce numerical problems associated with singularities of the transformation process and open the field for clipping algorithms.

Graphical primitives such as spheres or cylinders consist of an arbitrary amount of coordinates, additional state variables and visualization methods. To provide graphical representations, wireframe coordinates may be gained by an appropriate sampling in the object's parameter space. Table 3.1 summarizes primitive parametrizations, whereas the terms (figure 3.3) for the parabola chain are defined as

$$G := (0,0,-9.81) \tag{3.5}$$
$$t^* := t\left(\sum_i t[i]\right) - \left(\sum_{j\in\{0,..i\}} t[j]\right) \tag{3.6}$$
$$v0 := v_x\, N(H) + v_y\,(0,0,1) \tag{3.7}$$
$$vy := -t[i]\,G[2] - dh\; t[i]^{-1} \tag{3.8}$$
$$H := V(s0[i+1]-s0[i]) \tag{3.9}$$
$$dh := s0[i][2] - s0[i+1], \tag{3.10}$$

where $N(\cdot)$ denotes coordinate normalization and $V(\cdot)$ the vertical of coordinate to the plane defined by the transformed first two base vectors B'[0] and B'[1] of its CS. t^* is the time in $[0,t[i]]$, whereas t[i] is the throw time of the arc defined by s0[i] and s0[i+1]. v_x is an arbitrary value defining the horizontal velocity.

Primitive	Parametrization	Parameter space
Sphere	$(\theta,\varphi) \rightarrow r\,(\cos(\varphi)\cos(\theta),\sin(\varphi)\cos(\theta),\sin(\theta))$	$[-\pi,\pi]\times[0,2\pi]$
Cylinder	$(h,\varphi) \rightarrow r\,(\cos(\varphi),\sin(\varphi),h)$	$[-\infty,\infty]\times[0,2\pi]$
Hermite space curve	$(t) \rightarrow [\,2t^3-3t^2+1\,]\,f(0) + [\,t^3-2t^2+t\,]\,f'(0) + [\,-2t^3+3t^2\,]\,f(1) + [\,t^3-t^2\,]\,f'(1)$	$[0,1]$
Parabola chain	$(t) \rightarrow s0[i] + t^{*2}\,v0 + t^{*2}\,G$	$[0,1]$

Table 3.1: Primitive parametrizations

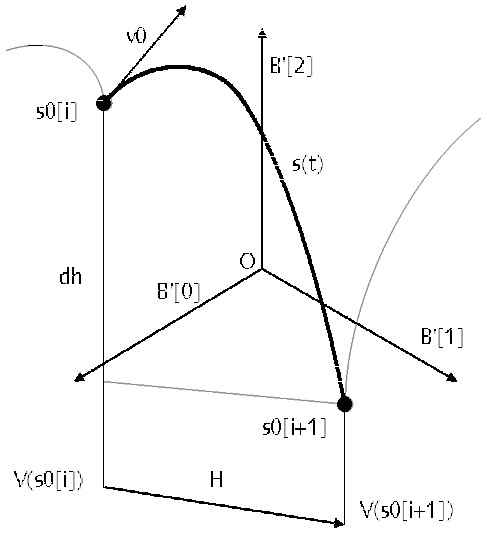


Figure 3.3: Parabola chain arc

3.2.1 Projection process

The projection process is the mapping of spacial positions onto the screen. The reduction from three to two dimensions is done as follows (figure 3.4): from two coordinate systems C and C' with origins O and O', one gets

$$P'_C = (0, P.y\ d\ (P.x+d)^{-1}, P.z\ d\ (P.x+d)^{-1}), \tag{3.11}$$

and from that

$$P'_{C'} = (x0 - P.y\ d\ (P.x+d)^{-1}, y0 + P.z\ d\ (P.x+d)^{-1}, 0). \tag{3.12}$$

C' is the CS with its origin in the lower left corner of the monitor, whereas C has its origin in the center. Thus, the constants x0 and y0 are device–dependent parameters, with 2·x0 being the frame's width and 2·y0 being its height.

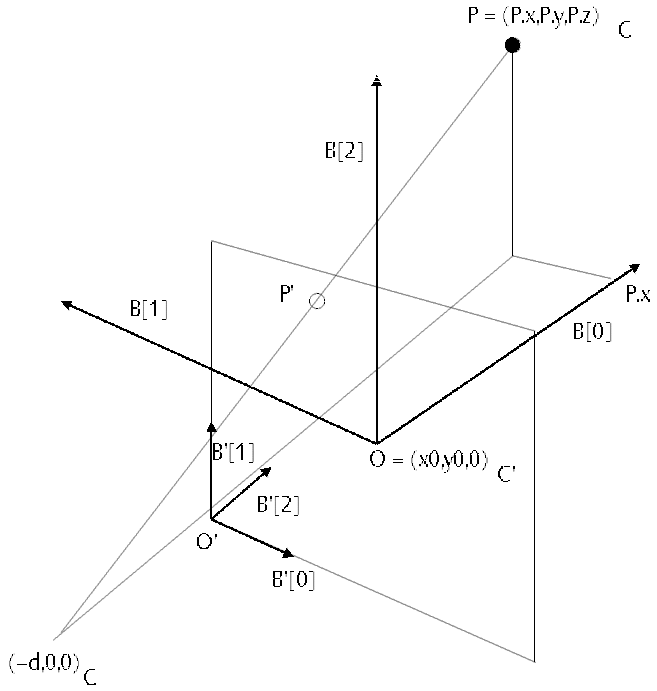


Figure 3.4: Projection process

3.2.2 Parametric clipping in three dimensions

The reason to apply a clipping algorithm in three dimensions rather than on screen coordinates comes from the projection process: it induces screen coordinates that may lie far outside of the visible rectangular area especially for lines that are close and almost parallel to the projection plane and can thus cause overflow traps. The visible volume in a three dimensional scene is an open pyramid with the observer's position at its top and the clipping process is not that simple as for screen coordinates.

A very useful recipe is to transfom the **view frustum** into a block by holding all coordinates on the viewplane at their position and enlarging its peak to the size of the viewplane, as it is shown by figure 3.5. The corresponding transformation $T(c)$, with c being an arbitrary coordinate, can be gained as it is described in the following.

The first recognition is that with the described transformation, c's first coordinate rests at its place there is the temporary result $T(c)=(c.x,?,?)$. Its second and third coordinate $c.y$ and $c.z$ are both scaled with the same factor $\sigma(\cdot)$ as c is moved on a plane perpendicular to the first base vector while being transformed. We may thus set up $\sigma(\cdot)$ as $\sigma(c.x,\delta)$ since it depends on those two variables as follows: we move $c.y$ and $c.z$ far, if $c.x$ is big (because the view frustum is fat there) and little, if $c.x$ is small. On the other hand, if δ is small, we will have to scale with a bigger factor. So $\sigma(c.x,\delta)$ must be small/big if $c.x$ is small/big and big/small if δ is small/big. By checking border conditions of the transformation (such as on the viewplane and for vectors $(n\delta,?,?)$, where n should be chosen in $\{-1,0,1,2,3\}$), one gets the the function values $\sigma(0,\delta)=1$, $\sigma(\delta,\delta)=2^{-1}$ and $\sigma(2\delta,\delta)=3^{-1}$, and by expecting direct and indirect proportions, the scale factor function

$$\sigma(c.x,\delta) = (1 + c.x/\delta)^{-1}, \tag{3.13}$$

is obtained, defining $T(c)$ as

$$T(c) = (c.x,\sigma(c.x,\delta)\cdot c.y,\sigma(c.x,\delta)\cdot c.z). \tag{3.14}$$

As we need to transform all vectors back after having performed the trivial clipping on the block volume, coordinates of the form $(\delta,?,?)$ must be excluded from the transformation process since T is not invertible there. This is no limitation since there is the need to clip off the view frustum's peak which causes all vectors of this form to disappear. This will be done as a clipping preprocess. Although T can be written as a matrix multiplication, it is heavily non–linear since the matrix itself depends on the coordinates to be transformed.

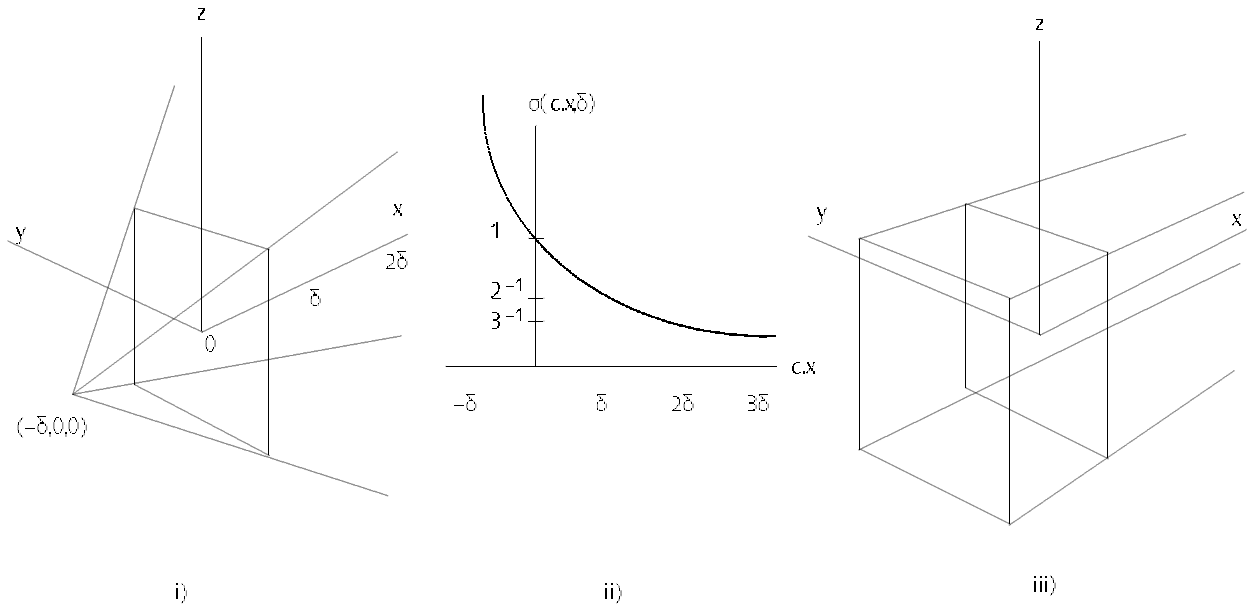


Figure 3.5: From view frustum to block view volume

3.2.3 Keys

There is a fundamental caveat with the wireframe representation of three dimensional TV systems: how can an infinite amount of static scenes be represented on a two–dimensional screen? The obvious solution is to graphically display a single scene S(t), where time t is determined by a visual control element such as a scrollbar. By moving the scollbar, dynamics are brought in by successively adapting the visual representation according to the timebar settings.

At each point in time, a designer may then customize the object's position and the motion control system will generate the positions in between by using interpolation techniques. The problem is obvious: to modify a thereby defined space–time constraint, the timebar must be moved to exactly the corresponding time and the position can then be adjusted.

The provide more efficiency, one might with to have an overview on all defined states within *one* graphical representation to be able to easily customize them. In the same way, states may be modified on a graphical representation of an object's track, a graphical representation of spacial positions has been developed. As figure 3.6 illustrates, it is an additional primitive (such as a sphere or a cylinder) called **key** and its visualization consists of a CS represented by small crosses on the transformed base vectors B'. The vertical from its origin to the plane formed by the first and the second transformed base vector of its parent CS gives a graphical representation of its height.

In the scene depicted by figure 3.6, a cube stays at the position defined by the key k0 for the time interval [sa.ta,sa.tb], then starts its journey towards k1. On this way, a displacement is added from sb.ta to sb.tb, and at time sc.ta, it reaches k1 where its orientation must be turned in direction of k2.

With the aid of keys, the crucial problem of interactively customizing an object's AT at some time t can be solved according to the following rule: if the primitive is absolutely positioned, move the controlling object and if not (interpolation case), insert an absolute positioning based on a key, pulling it to the current position at time t.

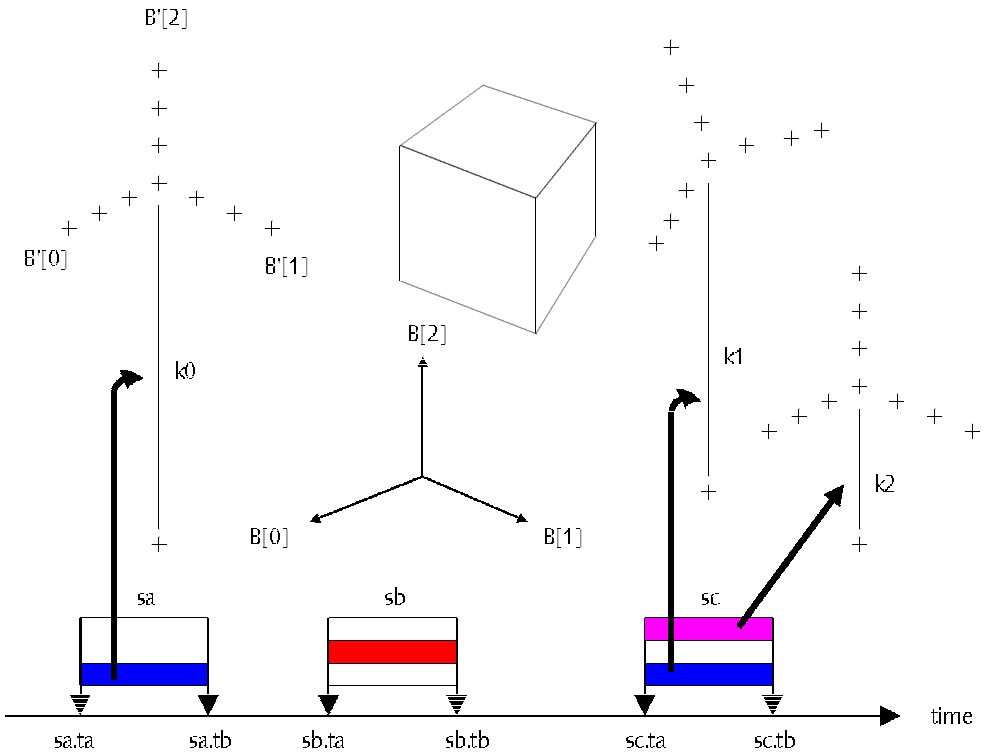


Figure 3.6: AT state visualization

3.3 Rendering

Whereas a wireframe representation shows advantages in a construction process, solid representations are a step towards photorealism. Any solid representation is based on a **lighting model**, which can be described as a directional intensity caused by a coloured object: "*The radiation that emanates in a particular direction from a surface is defined as the sum of the reflected and the emitted intensity.*" [WA92]. Kajiya's **rendering equation** provides a formal description of a rendering process in terms of the more or less accurate approximation of a single equation [WA92,GR97]. Nowadays, popular rendering techniques are **shading**, **raytracing**, **radiosity** and **volume rendering**.

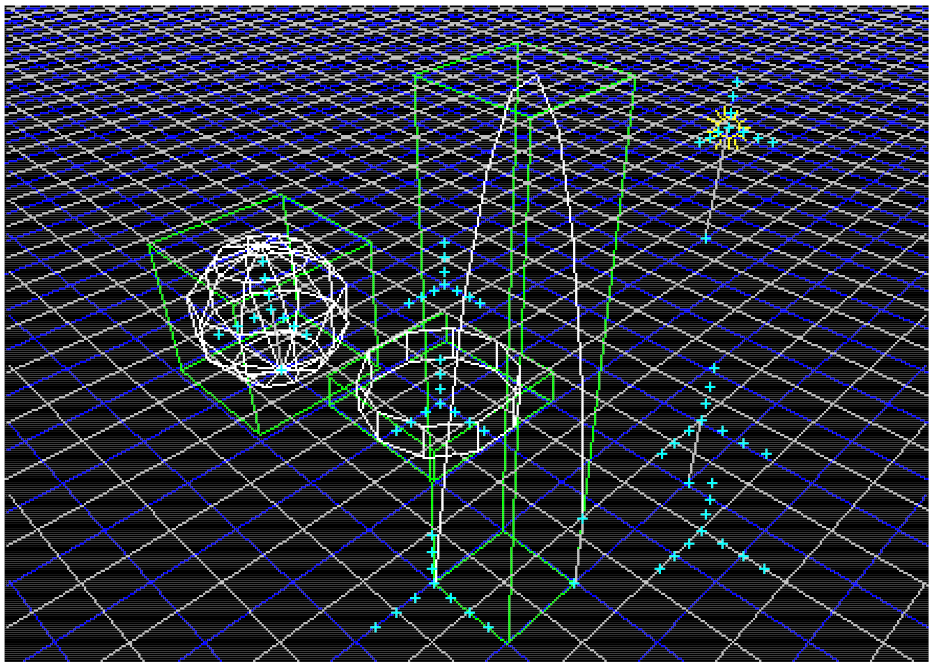
There exists very sophisticated and time–consuming rendering techniques ending in results near photorealism, but simple models may also give sufficiently realistic results. **Phong's reflection model** operates on totally three terms: the ambient, diffuse and specular light part, and defines the **bidirectional spectral reflectivity** [GR97] as

$$k_{diff} + k_{spec} \cos^{n}(\phi), \tag{3.15}$$

where ϕ is the angle between the vector V from intersection point to the observer's eye and the vector R, which is the vector to the light source L mirrored vector at the surface's normal N.

3.4 Textual description

The wish to integrate three dimensional objects into script–generated documents creates the wish for a scene description language. The track concept of time–stamped state desciptors directly leads to a simple syntax of which an example is shown by figure 3.7. A dynamically called generator reads primitive–specific information, whereas the rest of a NEW statement consists of subsequent track states.



```
FDDScript.Demo
NEW "FDDObj.ReadSphere" 1 t ( 0 0 C1 p 3 0 1 ) ( 5 7 C1 d "FDDDis.NewSalto" 1 0 0 1 ) ( 10 10 C1 p 1 5 1 )
NEW "FDDObj.ReadLight" t ( 0 0 C1 p –4 2 2 )
NEW "FDDObj.ReadParabolaChain" 2 ( 0 0 0 ) ( –1 1 1 ) t ( 0 0 C1 p 4 4 0 )
NEW "FDDObj.ReadPlane" t ( 0 0 C1 p 1.5 5.5 0 )
NEW "FDDObj.ReadCylinder" 1 0.5 t ( 0 0 C1 p 2 2 0 )
~
```

Figure 3.7: Scene generation by textual description

3.5 Implementation

3.5.1 Frames

In the Gadgets system, Interaction signals are implemented as messages, whereas an Oberon.InputMsg models mouse actions. An event dispatcher based on the findings of section 3.1 might thus look as follows:

```
WITH M: Oberon.InputMsg DO
    IF (M.id=Oberon.track) & (Gadgets.InActiveArea(self, M)) THEN
        IF (title=TRUE) THEN
            Title(x,y,M);
        END;
        IF (FDDBT.PublicBool(NAVIGATEFLAG)) & (M.keys#{}) THEN
            Navigate(M);
        ELSE
            IF (M.keys = {MIDDLE}) THEN
                Modify(x,y,M);
            ELSIF (M.keys = {LEFT}) THEN
                Create(Q,x,y,w,h,M);
            ELSIF (M.keys = {}) THEN
                Coords(Q,x,y,w,h,M);
            ELSIF (M.keys = {RIGHT}) THEN
                Select(x,y,M,keysum);
            END;
        END;
    ELSE
        baseClass(self,M);
    END;
END;
```

3.5.2 Wireframe representation

Wireframe representations are computed by collecting and visually connecting coordinates computed upon a sampling in the object's parameter space. By the example of a cylinder, one easily recognizes that an equipartitioned sampling in the paramter space may lead to an unnecessary amount of discrete coordinates. Some wireframe representations are depicted by figure 3.8.

A wireframe representation shows an inherent disadvantage: a scene's complexity may overtax the eye if no hidden surface algorithm is applied. For this reason – and to implement the concept of local construction contexts – the concept of focusing, whose effects are depicted by figure 3.9, has been implemented.

Focusing allows the customization of the cylinder by fading out all primitives that are placed lower in the CS hierarchy (such as the spheres and the parabola chain in figure 3.9). That way, a local construction context can be opened and the parent of newly created objects shall either be the focused object (if construction is initiated by a coordinate on it) or its parent (all other cases). The second case requires the generation of floor coordinates (intersection test with the blue focus grid) to place objects perpendicular onto it.

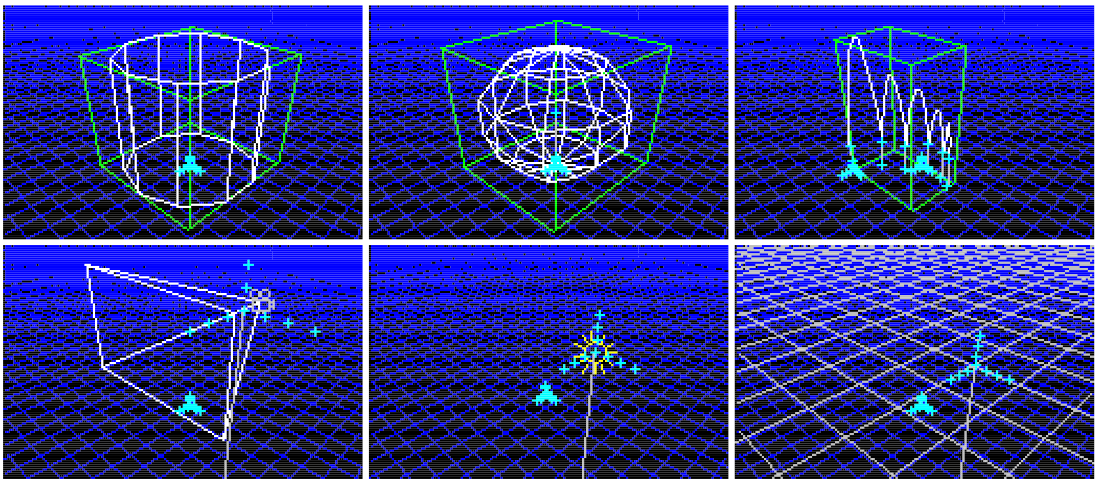


Figure 3.8: Wireframe representation

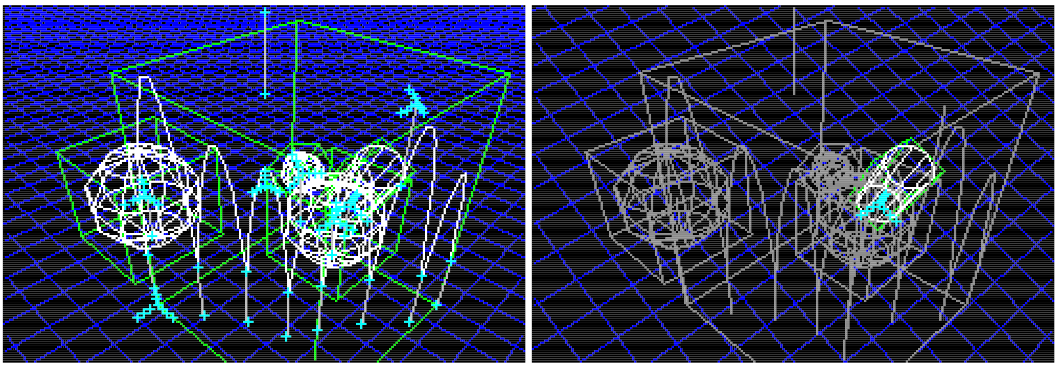


Figure 3.9: Focusing

```
PROCEDURE Clip3D*( VAR p0, p1 : FDDSpace.Cor; delta, s1, s2 : REAL) : BOOLEAN;
CONST HORIZON = 1000;
VAR sigma : REAL;

    PROCEDURE Clip( VAR x0, y0, z0, x1, y1, z1 : REAL; vx0, vy0, vx1, vy1 : REAL) : BOOLEAN;
    VAR tin, tout, xdir, ydir, zdir : REAL;
    BEGIN
        IF (x0 < vx0) & (x1 < vx0) THEN RETURN FALSE; END; (*on the left*)
        IF (x0 > vx1) & (x1 > vx1) THEN RETURN FALSE; END; (*on the right*)
        IF (y0 < vy0) & (y1 < vy0) THEN RETURN FALSE; END; (*below*)
        IF (y0 > vy1) & (y1 > vy1) THEN RETURN FALSE; END; (*above*)
        tin := 0; tout := 1;
        xdir := x1-x0;
        IF xdir > 0 THEN (*left: entry, right : leaving*)
            tin := FDDBT.max( tin, (vx0-x0)/xdir); tout := FDDBT.min( tout, (vx1-x0)/xdir)
        ELSIF xdir < 0 THEN (*left: leaving, right : entry*)
            tout := FDDBT.min( tout, (vx0-x0)/xdir); tin := FDDBT.max( tin, (vx1-x0)/xdir);
        END;
        ydir := y1-y0;
        IF ydir > 0 THEN (*bottom: entry, top : leaving*)
            tin := FDDBT.max( tin, (vy0-y0)/ydir); tout := FDDBT.min( tout, (vy1-y0)/ydir)
        ELSIF ydir < 0 THEN (*bottom: leaving, top : entry*)
            tout := FDDBT.min( tout, (vy0-y0)/ydir); tin := FDDBT.max( tin, (vy1-y0)/ydir);
        END;
        zdir := z1-z0;
        IF tin>tout THEN RETURN FALSE; END;
        x1 := x0+(tout*xdir); x0 := x0+(tin*xdir);
        y1 := y0+(tout*ydir); y0 := y0+(tin*ydir); (*ATTENTION: THE ORDER IS ESSENTIAL!*)
        z1 := z0+(tout*zdir); z0 := z0+(tin*zdir);
        RETURN TRUE;
    END Clip;

    PROCEDURE Transform( VAR me : FDDSpace.Cor; sigma : REAL);
    BEGIN
        me.y := me.y*sigma;
        me.z := me.z*sigma;
    END Transform;

BEGIN
    (*Parametric clipping in x (assert invertibility of view frustum transformation):*)
    IF ~Clip(p0.x,p0.z,p0.y,p1.x,p1.z,p1.y, -9/10*delta,-HORIZON, HORIZON,HORIZON) THEN RETURN FALSE; END;
    sigma := 1/(1 + p0.x/delta); Transform(p0,sigma); (*Transform view frustum to view block*)
    sigma := 1/(1 + p1.x/delta); Transform(p1,sigma); (*Transform view frustum to view block*)
    IF ~Clip(p0.y,p0.z,p0.x,p1.y,p1.z,p1.x, -s1,-s2,s1,s2) THEN RETURN FALSE; END; (*Parametric clipping in y and z*)
    sigma := (1 + p0.x/delta); Transform(p0,sigma); (*Transform view block into view frustum*)
    sigma := (1 + p1.x/delta); Transform(p1,sigma); (*Transform view block back into the view frustum*)
    RETURN TRUE;
END Clip3D;
```

Appendix A

Language use

There are some remarks that must be made concerning the programming language and its use in FDD. It turns out that Oberon features all important OO concepts at a level of simplicity that is not reached by any other programming language and establishes its fame of being a general purpose language. The following sections are the result of the intensive programming process and are a collection of ideas and advice.

A.1 Oberon

```
MyClass = POINTER TO RECORD (BaseClass)
  attribute0 : Type0;
  attribute1 : Type1;
  ...
  (*------------*)
  PROCEDURE Method0(); BEGIN ... END Method0;
  (*------------*)
  PROCEDURE Method1(); BEGIN ... END Method1;
END;(*MyClass*)
```

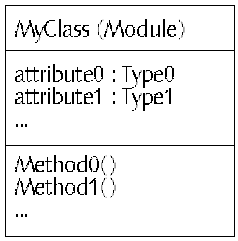


Figure A.1: UML class diagram versus type–centered source text

There can be no question: FDD's source text structure *must* reflect the OO approach. In particular, it is strongly associated with **UML** class diagrams. Figure A.1 shows the obvious similiarity carrying specification seamless to implementation. Various UML terms have simple implementations in the Oberon world: 1:1 associations and 1:n associations can be modeled by pointers or pointer to reference lists. Libraries then form a perfect persistency model, if the list is bound and Gadgets.Write/ReadRef is used to store/load it. Assertions are a convenient way of expressing pre– and postconditions, whereas the former delegates responsibility to the module client and the latter is an obligation to the thereby concluded code fragment [FOKE98]. Generalization/Inheritance is implemented as type extension.

A bad inheritance hierarchy can easily be detected when **initializer** parameters are passed from a derived type down to the base type. With a derived type, one can decide to take all the initalizer parameters of the base type into the new initalizer interface or to leave some out. The latter case is signaled when a field of the base type may somehow identify the derived type – such as a textual description – and will remain constant during its lifetime. Good examples are compound types (FDDObj.BasicPrimitives as an example) that unify a set of similar objects in order to remove source text redundancy: the numerous generators do not need to take a textual description as parameter since with calling the generator, the type is already specified and its desciption is only type– but not instance–specific. So an initializer must only have parameters that are essentially required to put the newly created instance into a legal state s0. This may bound the speed of degeneration into more and more complicated interfaces of derived types.

The definition of a proper **namespace** is a main key in the intention of creating complex and transparent modules as well. In this context, the naming of attributes plays a major role. A first subtle difference must be made between attributes that serve only as grouping coverage for subattributes. In this case, the general rule that a variable should not be called as its type can be discarded since the grouping attribute defines a hierarchy and must by its name denote the part of the object its subattributes represent. So when a pointer to a surface descriptor (featuring numerous parameters as subattributes) is attached to an object, it should be called "surface" to state that its subattributes make part of a surface. Calling it "s" is not advisable, since the pointer will not change its value over the object's lifetime and additionally represents a 1:1 association. By naming the surface pointer "s", variability over time is pronounced which is not desirable in this case.

From the mentioned example, there comes another insight concerning the namespace: if the surface descriptor is called "s" and one of its subattributes is a scalar value with name "s", a confusing ".s.s" reference will denote an access to it. Therefore, no attribute name should be similar to the names of its subattributes or even be a prefix. Furthermore, by using the type–centered approach of methods, displeasing calls such as "FDDDS.InsertNodeIntoList(me : Node; l : List);" can be replaced by "l.Insert(me);", since the variable type dermines the called procedure. Inheritance hierarchies and the absence of a scope resolution operator cause the need for naming initializers "<Module>Init(...)" for base types.

Generally, **methods** should be procedures that either allow A) supervised modification of an object's state or B) perform a computation on an object's state. Computations in which no object can be distinguished as to be the primary operation target (such as the addition of two vectors), should be defined as module–centered procedures. Additionally, if the type is extensively used in time–critical processes, the descriptor should be kept as small as possible. For this reason, operations in association with coordinates (module FDDSpace) rely on module–centered procedures. Procedure variables may implement shared methods and since they cannot be stored explicitly, they must be assigned within the object's initializer. If a method of a derived class does not make use of newly added (by type extension) record fields, it can be moved to the base class.

The question whether to use **messages** or direct method calls oftenly arises with the definition of large, nonhomogeneous systems. In order to get bestmost results, the recognition that they both do not provide the same advantages must cause a careful comparison between both implementation techniques with respect to the considered problem. The advantage of directly calling handlers is certainly their efficiency in time–critical processes such as for display or visualization operations. On the other hand, the lack of a predefined base class hierarchy handling and a scope resolution operator in particular leads to unorder in the namespace, whereas the definition of a sole message handler heals this disease. The question about where (in the source text) message classes should be defined, can be answered as follows: if it will primarily be broadcast, it should be placed with the primary initiator. But if its nature primarily is to inform a certain class about a special event, it should be defined with the recipient to allow insight about what the message affects.

A.2 Source code conventions

How source code looks essentially determines – in association with the language syntax – readability, and from that i) reusability, ii) accessability and iii) extensibility for further development. Thus, a major effort must be to provide well–structured program sources and a primary point of doing so is to establish general source code conventions. The OO approach explains the preference of the type–centered style. FDD source code obeys the fundamental rules as described in the following.

A.2.1 Formatting: programming art

Formatting is the science of presentation and must thus not be done in intiuitive manner. A first use of formatting is *grouping* and an obvious rule must thus be:

R1 *Group, what belongs together.*

This statement has many implications. It forbids the looking of the following fragments, since they suggest that "A = 1;" is somehow closer to the keyword "CONST" than "B = 2;":

```
CONST                    CONST A = 1; B = 2;
   A = 1;
   B = 2;
```

The latter is incorrect for a second reason: the equipartitioned space between the terms does not reflect the statement structure. It does not signal that "1;" belongs to "A =" rather than to "B =". The following solutions suit the needs of the eye and must thus be solutions of choice:

```
CONST                    CONST  A = 1;   B = 2;

   A = 1;  B = 2;


VAR

   v1 : INTEGER;
```

The second version clearly determines the two constants as being subordinary elements of the "CONST" statement by highlighting it in a title–like fasion. It is worth mentioning that the use of non–proportional fonts does prevent from fluent reading and should completely be avoided. A second use of formatting is *selection*:

R2 *Use colors for selective recognition.*

```
FDDFrame = POINTER TO RECORD (FrameBuffer)
   time* : REAL; (*view time*)
   title* : BOOLEAN; (*FDD banner on startup*)
   show* : Show; (* determine lookings*)
   l : Objects.Library;
   scene* : FDD.Scene;
   re* : FDDDS.List;
   p : FDDFilms.Playtask;
   x0, y0 : INTEGER; (*help variables set by ".Draw"*)
   xPPM, yPPM : REAL; (*help variables set by ".Draw"*)
   s1, s2 : REAL; (*help variables set by ".Draw"*)

   PROCEDURE Attributes (VAR M: Objects.AttrMsg);
   BEGIN
      (...)
   END Attributes;

END;
```

```
FDDFrame = POINTER TO RECORD (FrameBuffer)
   (*0*) time* : REAL; (*view time*)
   (*1*) title* : BOOLEAN; (*FDD banner on startup*)
   (*2*) show* : Show; (*determine looking*)
   (*3*) l : Objects.Library;
   (*4*) scene* : FDD.Scene;
   re* : FDDDS.List;
   p : FDDFilms.Playtask;
   x0, y0 : INTEGER; (*help variables set by ".Draw"*)
   xPPM, yPPM : REAL; (*help variables set by ".Draw"*)
   s1, s2 : REAL; (*elp variables set by ".Draw"*
   (*————————————————————*)
   PROCEDURE Init( me : FDDFrame; ...);
   BEGIN
      (...)
   END Init;
   (*————————————————————*)
END; (*FDDFrame*
```

The above code fragments point out its meaning: the reader may selectively determine the aim of its view: comments or definitions. Instead of having a big gap between state variables and methods suggesting a broken object, horizontal lines identify the beginning of a new part but also glue the record definition together. Of course, not everybody lays that much importance on lookings, but I am sure that D. E. Knuth would agree: the key to readable source code is formatting.

A.2.2 Placement: programmed methodology

The last example of section A.2.1 already showed the important concepts concerning statement placement in large record definitions, which are as follows:

R3 *Place non–redundant state variables on top.*
R4 *Place pointer variables after non–pointer variables.*
R5 *Place prior elements on top and in lifecycle sequence.*
R6 *Enumerate record fields* (*1*), (*2*), ... *and mark the ones to be computed upon others as to be redundant* (*r*).

As an effect of the stated rules, a copy method will always be very similar to the record definition, whereas the variable enumeration makes it easy to verify its completeness, as well as for file storing/loading routines. R5 induces a natural order for method declarations: after having read the state variables of the record definition, a programmer wants to be able to instantiate and to use the object and it is thus advisable to place the initializer on top and a potential destructor at the bottom. For large projects, it is also acceptable to put such standard code to the end to straightforward emphasize amazing or essential methods. The key to complete and ordered source code is placement.

A.2.3 Information hiding: reliability and evidence

What is the primary purpose of information hiding? Steadily growing software projects require abstract interfaces and hidden components in order to remain maintainable. State variables being part of invariants must be protected against uncontrolled modification and must thus be hidden. An access mechanism can only be provided by methods respecting the invariants. For the purpose of abstraction, a client module does not make any sense without information hiding: the key to reliable and useful client modules is information hiding.

A.3 Propositions

A couple of ideas to improve the programming environment have come up during the development of FDD and I want to open them for a discussion.

- Assignment vectors: a := (x,y,z);
- Source text comments automatically fall into the background for the purpose of selective recognition
- In the source text, method source text should either be visible (open) or hidden (closed)
- Read only attributes (as defined in Oberon–2) make read–methods for attributes involved in invariantes unnecessary
- With the concise definition of Gadget copy methods, the compiler can provide them to avoid redundant source text

Appendix B

Module reference

This appendix gives a functional and implementation–specific description of the modules that form FDD . The order roughly follows a bottom–up manner beginning with very basic modules that are the foundation of the complex data structure. Table B.1 lists the modules, whereas the category shall merely serve as navigation help than as a disjoint partition of the source text. As a convention, no initializers [FIMA98] provided by the extended language of the Native Oberon compiler have been used. To ensure a fluent walk–through, repetitive details such as self–pointers, copy methods, message handlers or initializers are skipped.

Category	Module	Contents
Space	FDDBT	Basic types (color), corresponding methods and conversion routines
	FDDDS	Generic data type list and a list of object references as a concrete instance
	FDDSplines	B–Spline base function, knot generator, Hermite interpolation
	FDDAT	Affine transformations (AT), AT view Gadget
	FDDSpace	Coordinate systems (CS)
Time	FDDTVV	Time–variate (TV) variables (TVV) featuring track–controlled TVV (TCTVV) as application
	FDDDis	TV displacement stacks
	FDDTime	TV CS and TV surface
	FDDScript	Scene generation by textual desciption: a script language
Interaction	FDDFrameBuffers	Generic view gadget featuring framebuffer display output
	FDDFilms	Background play–tasks and a film view gadget
	FDDGFX	Low–level graphics (framebuffer drawing, frame look, 2D clipping)
	FDDInteraction	Scene frame
	FDDContainers	MMI Container Gadget featuring modes Editor, Director and Player
	FDDTextures	Texture module
	FDDRender	Task–based film rendering
	FDDObj	Construction primitives (sphere, parabola chain...), default scene

Table B.1: Module contents

B.1 Space

B.1.1 FDDDS

Significant types: List, Node
Significant procedures: (List) Head/Tail, Empty, Insert/Delete, Find
(Node) Key, Prev/Next
Significant variables: –

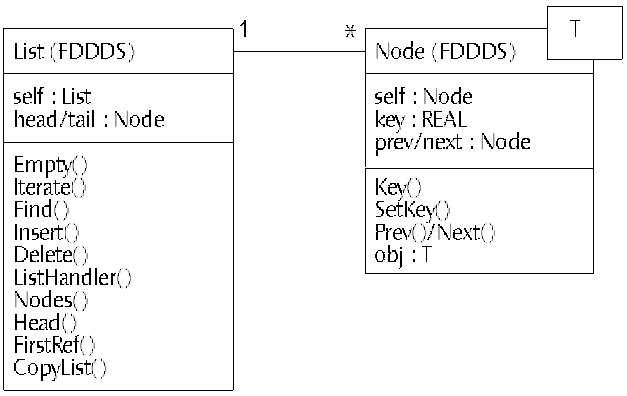


Figure B.1: FDDDS UML class diagram

```
List = POINTER TO RECORD (Gadgets.ObjDesc)
    PROCEDURE Delete (me: Node);
    PROCEDURE Empty (): BOOLEAN;
    PROCEDURE Find (key: REAL): Node;
    PROCEDURE FirstRef (): Ref;
    PROCEDURE Head (): Node;
    PROCEDURE Insert (n: Node; where: INTEGER);
    PROCEDURE Iterate (upcall: Listupcall; closure: Closure);
    PROCEDURE Nodes (): INTEGER;
    PROCEDURE Tail (): Node;
END;.
```

FDDDS provides a convenient implementation of (optionally) ordered lists. "Find" returns the node with the biggest key smaller than the scalar parameter. The "Node" features a reference to any class derived from Objects.Object. With the aid of type casts, lists of any type can be build and accessed. The C++ concept of templates [PO94] is not required in this case. With aid of FDDDS, a tree iterator can easily be implemented and might look as follows:

```
MyTree = POINTER TO RECORD (FDDDS.Node)
    (*data*)
    children : FDDDS.List;
    (*--------------------*)
    PROCEDURE Iterate();
    VAR c : FDDDS.Ref;
    BEGIN
        (*action*)
        c := t.children.FirstRef(); WHILE (c#NIL) DO c.Iterate(); c := c.NextRef(); END;
    END Iterate;
END;
```

B.1.2 FDDSplines

Significant types: –
Significant procedures: N, C1Coeffs, C1, C1iterate, UEIK
Significant variables: u

```
PROCEDURE N*( i, n : INTEGER; t : REAL) : REAL;
BEGIN
  IF (t < u[i]) OR (t > u[i+n+1]) THEN RETURN 0; END; (*check support interval*)
  IF n = 0 THEN
    IF (( t >= u[i]) & ( t < u[i+1])) OR (t = 1) THEN RETURN 1 ELSE RETURN 0; END;
  ELSE
    IF u[i+n]-u[i]#0 THEN
      IF u[i+n+1]-u[i+1]#0 THEN RETURN
(t-u[i])*N(i,n-1,t)/(u[i+n]-u[i])+(u[i+n+1]-t)*N(i+1,n-1,t)/(u[i+n+1]-u[i+1]);
      ELSE RETURN (t-u[i])*N(i,n-1,t)/(u[i+n]-u[i]); END;
    ELSE
      IF u[i+n+1]-u[i+1]#0 THEN RETURN (u[i+n+1]-t)*N(i+1,n-1,t)/(u[i+n+1]-u[i+1]) ELSE RETURN 0; END;
    END;
  END;
END N;
```

... is the B–Spline base [GR97] of degree n with support interval [u[i],u[i+n+1]], whereas a uniform, endpoint–interpolating knot vector u can be computed using the

```
PROCEDURE UEIK*( n, controls : INTEGER);
```

with "n" being the number of knots and "controls" the number of control points. Finally, it provides C1 Hermite space curve interpolation with the aid of the following set of procedures:

```
PROCEDURE C1Coeffs (VAR a, b, c, d: FDDCS.Cor; p0, p1, p2, p3: FDDCS.Cor);
PROCEDURE C1 (VAR pos: FDDCS.Cor; a, b, c, d: FDDCS.Cor; t: REAL);
PROCEDURE C1iterate (upcall: PROCEDURE (pos: FDDCS.Cor; t: REAL); a, b, c, d: FDDCS.Cor; stepwidth: REAL);
```

B.1.3 FDDAT

Significant types: AT, ATFrame
Significant procedures: (AT) Define, Set, Invert, Neg, Rot, Scale, Add/Sub
Significant variables: –

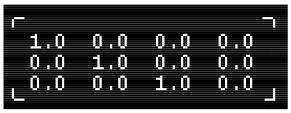


Figure B.2: AT visualization

```
AT = POINTER TO ATDesc;
ATDesc = RECORD (Gadgets.ObjDesc)
    r00, r01, r02, t0, r10, r11, r12, t1, r20, r21, r22, t2: REAL;
    PROCEDURE Concat (a1: ATDesc; VAR a0: ATDesc);
    PROCEDURE Define (nr00, nr10, nr20, nr01, nr11, nr21, nr02, nr12, nr22, nt0, nt1, nt2: REAL);
    PROCEDURE Invert;
    PROCEDURE LinInt (a: ATDesc; ta, time, tb: REAL; b: ATDesc);
    PROCEDURE Rot (ax, ay, az, phi: REAL);
    PROCEDURE Scan;
    PROCEDURE SetDesc (a: ATDesc);
END;
```

This model–view pair gives a general implementation of affine transformations. The type of the pointer–based record AT is not kept anonymous: to keep coordinate system transformations local and fast, they are carried out on the stack and do not use dynamically created heap objects. What may be crucial, is the parameter order in the "Define" method: to simplify imagination, the columns of the transformation matrix are kept together.

B.1.4 FDDSpace

Significant types: Cor, Corarray, Box, Extent, CS, CSMsg, UpdateMsg
Significant procedures: (Cor) DefineCor, Add/Sub, Dot, Cross, Apply, ProjectOrthogonal/Perpendicular, ExpressCor
(Corarray) CreateCorarray, ExpressCorarray
(Box) Define
(Extent) V0, V1, V0l, V1l, CorExtent, Bound
(CS) SetA/Parent/Obj/C, Rl, Overlay, FirstChildRef, n, A, Parent, C, Express, Is, LocalExtent
(CSMsg) Broadcast
Significant variables: insertion

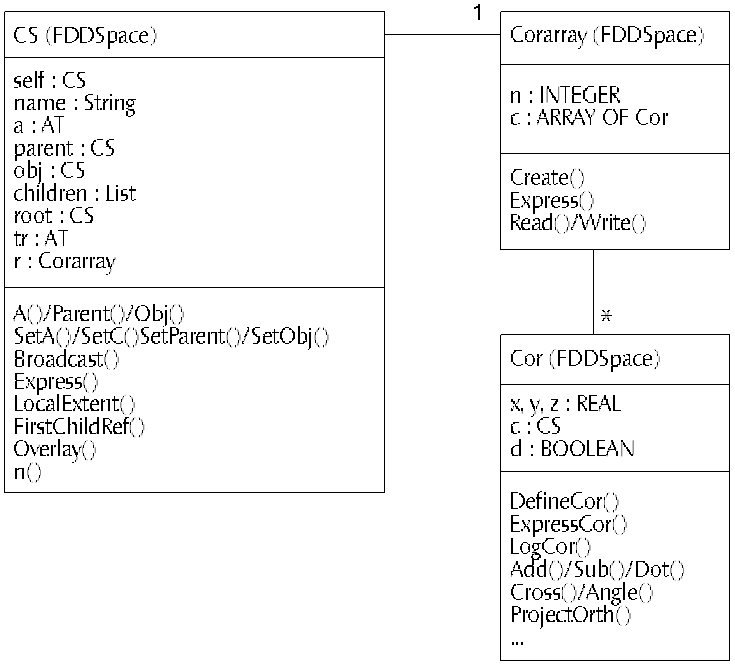


Figure B.3: FDDSpace UML class diagram

B.1.4.1 Coordinates

```
Cor* = RECORD
    x*, y*, z* : REAL;
    c* : CS;
    d* : BOOLEAN;
END;.
```

FDDSpace is the spacial heart of FDD and may prove its usefulness in many other applications from the general purpose approach. Based on affine transformations, it defines arbitrary hierarchies of coordinate systems (CS) together with coordinates that may be expressed in any CS. It is not usual that coordinates are implemented in such a strict manner: one oftenly relies on the three scalars. But as we will see, this closed mathematical definition greatly simplifies all high–level routines since the low–level machinerie cares about all kinds of coordinate mappings with a lazy evaluation scheme as in

```
PROCEDURE Add*( VAR v : Cor; a, b : Cor);
BEGIN
    IF a.c#b.c THEN ExpressCor( a, b.c); END;
    defineCor( v, a.x + b.x, a.y + b.y, a.z + b.z, b.c, b, b.d);
END add;,
```

and at the end of a computation process (such as object intersection), we can just request a vector being expressed in a particular CS with the aid of

```
PROCEDURE ExpressCor*( VAR me : Cor; in : CS);
VAR inv : FDDAT.ATDesc;
BEGIN
    ASSERT(me.c.root=in.root); (*else: transformation not possible!*)
    IF me.c#in THEN
        Apply( me, me.c.tr);
        inv.SetDesc(in.tr);
        inv.Invert();
        Apply( me, inv);
        me.c := in;
    END;
END ExpressCor;.
```

Based on the orthogonal projection procedure,

```
PROCEDURE Vertical*( VAR v : Cor; c : Cor);
VAR f : REAL; e2 : Cor;
BEGIN
    DefineCor(e2, 0,0,1, c.c, TRUE);
    ProjectOrthogonal(c,e2,f);
    Copy(c,v);
END Vertical;
```

computes the vertical of a coordinate onto the plane defined by the transformed first two base vectors of its CS. The following code fragment shows appropriate client code and demonstrates the application of the coordinate transformation facility at the artificial example shown by figure 1.5.

```
TYPE CS = POINTER TO RECORD (FDDSpace.CS) END;

PROCEDURE Run*();
VAR ab, a1, a2, b1, b2 : CS; cor : FDDSpace.CorarrayDesc; S : REAL;  a : FDDAT.ATDesc;
BEGIN
    S := Math.sqrt(2)/2;
    FDDSpace.CreateCorarray(cor,0,NIL);
    NEW(ab); ab.InitCS(ab, NIL,ab,cor); a.I(); ab.SetA(a);
    NEW(b1); b1.InitCS(b1, ab,b1,cor); a.Define(1,0,0, 0,1,0, 0,0,1, 0,0,2); b1.SetA(a);
    NEW(b2); b2.InitCS(b2, b1,b2,cor); a.Define(S,S,0, -S,S,0, 0,0,1, -1,2,0); b2.SetA(a);
    NEW(a1); a1.InitCS(a1, ab,a1,cor); a.Define(1,0,0, 0,1,0, 0,0,1, 1,1,1); a1.SetA(a);
    NEW(a2);
    FDDSpace.CreateCorarray(cor,1,a2); FDDSpace.DefineCor( cor.c[0], 0,1,0,a2,FALSE);
    a2.InitCS(a2, a1,a2,cor); a.Define(0,1,0, -1,0,0, 0,0,1, 0,2,1); a2.SetA(a);
    a2.Express(b2);
    FDDSpace.LogCor(a2.x.c[0]);
END Run;
```

Having in mind that it does not even make sense to create coordinates without specifying the corresponding CS, one cannot imagine doing it differently. The "d" field of the coordinatedescriptor Cor may be used to define line directions (CS origin together with a single coordinate) and to express it in any CS – as usual with ExpressCor – which is a slightly different transformation process as known from (1.27). The application of affine transformations on coordinates is covered by

```
PROCEDURE Apply*( VAR v : Cor; a : FDDAT.ATDesc);.
```

```
PROCEDURE Clip3D*( VAR p0, p1 : Cor; delta, s1, s2 : REAL) : BOOLEAN;
CONST HORIZON = 1000; VAR sigma : REAL;

    PROCEDURE Clip( VAR x0, y0, z0, x1, y1, z1 : REAL; vx0, vy0, vx1, vy1 : REAL) : BOOLEAN;
    VAR tin, tout, xdir, ydir, zdir : REAL;
    BEGIN
        IF (x0 < vx0) & (x1 < vx0) THEN RETURN FALSE; END; (*on the left*)
        IF (x0 > vx1) & (x1 > vx1) THEN RETURN FALSE; END; (*on the right*)
        IF (y0 < vy0) & (y1 < vy0) THEN RETURN FALSE; END; (*below*)
        IF (y0 > vy1) & (y1 > vy1) THEN RETURN FALSE; END; (*above*)
        tin := 0; tout := 1;
        xdir := x1-x0;
        IF xdir > 0 THEN (*left: entry, right : leaving*)
            tin := FDDBT.max( tin, (vx0-x0)/xdir); tout := FDDBT.min( tout, (vx1-x0)/xdir)
        ELSIF xdir < 0 THEN (*left: leaving, right : entry*)
            tout := FDDBT.min( tout, (vx0-x0)/xdir); tin := FDDBT.max( tin, (vx1-x0)/xdir);
        END;
        ydir := y1-y0;
        IF ydir > 0 THEN (*bottom: entry, top : leaving*)
            tin := FDDBT.max( tin, (vy0-y0)/ydir); tout := FDDBT.min( tout, (vy1-y0)/ydir)
        ELSIF ydir < 0 THEN (*bottom: leaving, top : entry*)
            tout := FDDBT.min( tout, (vy0-y0)/ydir); tin := FDDBT.max( tin, (vy1-y0)/ydir);
        END;
        zdir := z1-z0;
        IF tin>tout THEN RETURN FALSE; END;
        x1 := x0+(tout*xdir); x0 := x0+(tin*xdir);
        y1 := y0+(tout*ydir); y0 := y0+(tin*ydir); (*ATTENTION: THE ORDER IS ESSENTIAL!*)
        z1 := z0+(tout*zdir); z0 := z0+(tin*zdir);
        RETURN TRUE;
    END Clip;

    PROCEDURE Transform( VAR me : Cor; sigma : REAL);
    BEGIN
        me.y := me.y*sigma; me.z := me.z*sigma;
    END Transform;

BEGIN
    IF ~Clip(p0.x,p0.z,p0.y,p1.x,p1.z,p1.y, -9/10*delta,-HORIZON, HORIZON,HORIZON) THEN RETURN FALSE; END;
    sigma := 1/(1 + p0.x/delta); Transform(p0,sigma);
    sigma := 1/(1 + p1.x/delta); Transform(p1,sigma);
    IF ~Clip(p0.y,p0.z,p0.x,p1.y,p1.z,p1.x, -s1,-s2,s1,s2) THEN RETURN FALSE; END;
    sigma := (1 + p0.x/delta); Transform(p0,sigma);
    sigma := (1 + p1.x/delta); Transform(p1,sigma);
    RETURN TRUE;
END Clip3D;
```

implements 3D clipping in object space of the line from p0 to p1 based on (3.14). The observer at (-delta,0,0) sits on top of the view frustum with the x-axis being its symmetry axis and the dimension 2*s1 x 2*s2 at x=0. Clip3D by itself uses two dimensional parametric clipping.

B.1.4.2 Coordinate systems

```
CS = POINTER TO RECORD (Gadgets.ObjDesc)
    x: CorarrayDesc;
    children : FDDDS.List;
    PROCEDURE Parent (): CS;
    PROCEDURE A (VAR a0: FDDAT.ATDesc);
    PROCEDURE SetParent (p: CS; manner: INTEGER);
    PROCEDURE SetA (a0: FDDAT.ATDesc);
    PROCEDURE SetC (c0: CorarrayDesc);
    PROCEDURE SetObj (o: CS);
    PROCEDURE Express (in: CS);
END;
```

The key to hierarchies of coordinate systems is the "SetParent" method: a CS is an affine transformation with respect to another affine transformation (its parent). This recursive definition – together with the restriction, that such a chain must be terminated at some time – allows the construction of arbitrary depth CS hierarchies. The global variable "insertion" (being either FDDDS.HEAD or FDDDS.TAIL) specifies the insertion method into the parent's "children" list. With its aid, FDDSpace implements the top view of the CS hierarchy using a binary tree representation [REI98] with nodes of varying degree. The CS has not been derived from the AT to ensure supervised access to the AT by use of the "A" and "SetA" methods. The "x" field allows access to the coordinates, which can be expressed in any CS by using the "Express" method.

A CS can only be useful if there is a set of coordinates referring to it. They can be installed by use of the "SetC" method. Behind the scenes, the system establishes the following invariants:

0) Coordinates are internally always given with respect to the CS.
1) A field "root" of type CS contains the last member of the "parent" recursion (whose parent is NIL).
2) An AT "tr" gives the total AT of the CS with respect to "root".
3) An array "r" contains the coordinates with respect to "root".
4) The field "extent" (see below) contains an object's local and compound extent.

The invariants are restored upon every call of "SetA", "SetC", "SetObj" and "SetParent" by the

PROCEDURE RI(s : SET);,

whereby the parameter is set according to the recomputation implications shown by table 1.2. The last implication induces a slight complication: since the CS may be target of external ".obj" pointers, a change in the coordinate array ".c" must be broadcast to all CS referring to it. It is therefore necessary that a CS must maintain a list (".crefs") of all those connections.

B.1.4.3 Boundingboxes

What has been a crucial task in FDD's predecessor – the implementation of bounding boxes – has now a simple and compact solution. Since a bounding box can be described by two single spacial coordinates, and from the need of having the "l"ocal and the compound (including its children) extent, the definition

```
Extent* = RECORD
    v0, v1: Cor;
    v0l, v1l : Cor;
END;
```

becomes a necessity. Luckily, there is an elegant algorithm to set up v0, v1, v0l and v1l based on the previously defined procedures:

```
LocalExtent(extent);
copyCor(extent.v0l,extent.v0);
copyCor(extent.v1l,extent.v1);
re := children.Head();
WHILE (re#NIL) DO
    Define(b, re.obj(CS).extent.v0,re.obj(CS).extent.v1); (**)
    FOR i := 0 TO 7 DO
        ExpressCor(b[i],self);
        Bound(extent.v0,b[i],extent.v1);
    END;
    re := re.Next();
END;
```

It is based on a data type "Box" which is an array of eight coordinates representing the corners of a box, and can be set up by calling "Define" with the two defining vertices as parameters (**). FDDSpace defines a "LocalExtent" method that computes the object's extent in local coordinates and can be overridden by derived types. As default, it computes the extent based on the installed coordinates. Thus, adding a record field of type "Extent" to the CS descriptor solved the problem of bounding boxes (within two hours).

B.2 Time

The parameter time is brought into static scenes by a general purpose approach called "time–variate variables" (TVV). It allows an extensible design so that adding primitive–specific TVV to FDD is a task that can be done with little effort. The system is based the concept of "track controlled" TVV (TCTVV) with a generic view model that allows visualization and editing of any TCTVV in association with custom state editors.

B.2.1 FDDTVV

Significant types: TVV, TimeMsg, TCTVV, State, CollectMsg, TCTVVFrame
Significant procedures: (TVV) SetTime, Time
(TCTVV) SetKey/Delete, Find, GenerateState, SetTime,
(State) Pred/Succ, PropagateUpdate, Restore, SetInterpolation, SetT, Tb
(TCTVVFrame) SetTVV
Significant variables: c, sel

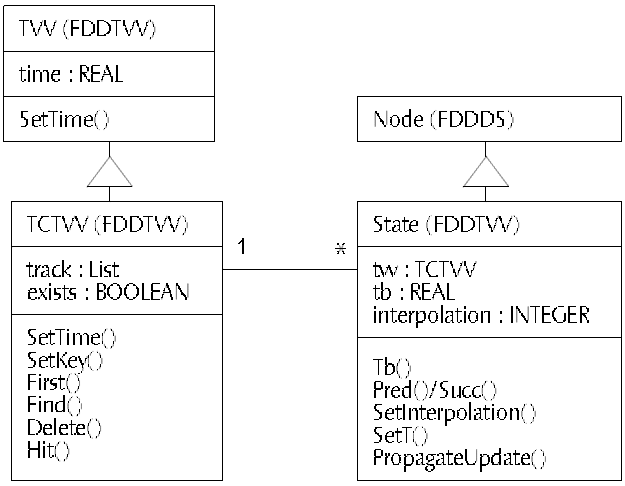


Figure B.4: TVV UML class diagram

B.2.1.1 Concept

The principal idea of TVV is to replace variables such as an integer "i" by objects whose state (which is an entire number in the case of "i") may vary with respect to a scalar parameter: time in the case of FDD. Since TVV's state potentially depends on an arbitrary amount of data we may define

```
TVV* = POINTER TO RECORD (Gadgets.ObjDesc)
   time : REAL;
   (*--------------------*)
   PROCEDURE SetTime(newTime: REAL; force: BOOLEAN);
   VAR r : RecomputeMsg;
   BEGIN
      IF (newTime#time) OR force THEN
         r.time := newTime;
         handle(self,r); (*activate handlers of derived types: send itself a message*)
      END;
   END SetTime;
   (*--------------------*)
   PROCEDURE TVVHandler*(VAR M: Objects.ObjMsg)
   BEGIN
      IF M IS RecomputeMsg THEN
         time := M(RecomputeMsg).time;
      ELSE baseClass(self,M); END;
   END TVVHandler;
END;
```

whereby an example of a simple TVV based on the built–in data "time" is given by

```
Log* = POINTER TO RECORD (Gadgets.ObjDesc)
    val : REAL;
    (*––––––––––––––––––––––*)
    PROCEDURE Handler*(VAR M: Objects.ObjMsg)
    BEGIN
        IF M IS RecomputeMsg THEN
            val := Math.ln(M(RecomputeMsg).time);
            TVVHandler(M);
        ELSE TVVHandler(M); END;
    END Handler;
END;,
```

and one sees that the instantiation of a TVV does only pay well if the value of the TV size depends on other data than time, since the above example simply rewrites the procedure Math.ln in a complicated fashion. Additional data will be set or read by methods that can be added to the TVV descriptor or by adapting the message handler.

B.2.1.2 Application (TCTVV)

The application of TVV's in FDD is straightforward: there is a need for A) TV AT's, B) TV surfaces and C) primitive–specific TVV's. The most simple data–based TVV could be defined as follows: we want to specify the TVV's value at several points in time and the TVV shall be able to reproduce these states. The question about what value is returned in between is again specific to the TVV any can be answered in its handler upon the reception of a RecomputeMsg. These reasonings have led to the concept of "track–controlled time–variate variables" (TCTVV) which FDD is built on. A TCTVV primarily consists of an ordered list (its "track") of states, featuring "SetKey" and "Delete" methods to insert/remove states, which will – in this context – also be referred as keys.

```
TCTVV = POINTER TO RECORD (TVV)
    exists: BOOLEAN;
    PROCEDURE Delete (me: State);
    PROCEDURE Find (time: REAL): State;
    PROCEDURE Generate (ta, tb: REAL): State;
    PROCEDURE Broadcast*(M : FDDSpace.CSMsg);
    PROCEDURE SetKey (s: State; VAR res: INTEGER);
    PROCEDURE SetTime(newTime: REAL; force: BOOLEAN);
    ...
END;
```

Remarks: 1.) The BOOLEAN "exists" is set to FALSE whenever a time before the first track entry is set. In all other cases, the TVV may customized it in its handler. 2.) "Find" returns the state with the biggest key smaller than the parameter "time". 3.) With the definition of the "virtual" procedure "Generate", TCTVV prepares itself for the collaboration with the track editor "TCTVVFrame" (see below): to insert new states, a state generator is required.

The basic type of a state features a collection of methods that will prove their usefulness especially in the implementation of the TV ATTime.

```
State = POINTER TO RECORD (FDDDS.Node)
    PROCEDURE Pred/Succ (): State;
    PROCEDURE PropagateUpdate;
    PROCEDURE Restore (Q: Display3.Mask; buf: Pictures.Picture; x, y, w, h: INTEGER; sel: BOOLEAN);
    PROCEDURE Interpolation() : INTEGER;
    PROCEDURE SetInterpolation (i: INTEGER);
    PROCEDURE SetT (a, b: REAL);
    PROCEDURE Tb (): REAL;
END;
```

Remarks: 1.) "PropagateUpdate" should be called on each state modification. It determines, whether the change has any impact on the TVV's value (if the current time is in the direct neighbourhood of the state) and if so, it calls "SetTime" which forces the TVV to update its value. Note that this recomputation rule may not be valid for derived TCTVV's (if a state influences the TVV's value further than in the direct neighbourhood – such as B–Spline bases): this may cause the necessity to override "PropagateUpdate".

In this context, one must mention that TCTVV overrides the "SetTime" method defined by TVV in such a way that a TCTVV will never receive a "RecomputeMsg". The reason to do so is as follows: there are two different cases in the evaluation of a TCTVV's value: the time to be set A) hits a track entry or B) falls between two states. TCTVV determines the case and then sends either a "RestoreMsg" or a "InterpolateMsg" to itself. Furthermore, we have the possibility to define sets of states by specifying a state's start and end (with respect to time) by using the "SetT" method.

B.2.1.3 Editing

To be able to define and edit arbitrary tracks, a representation in the form of a timebar lies near, with events being represented by small triangles pointing at a certain position in time. The implementation with Oberon's "Gadgets" framework is also obvious: it suffices to define editing facilities such as the generation of states in time, moving and editing them. "TCTVVFrame" a simple frame Gadget with the capability to add, remove, retime and to select states. The current (single) selection is hold in the global variable "sel".

```
CollectMsg* = RECORD (Objects.ObjMsg)
    id : INTEGER;
    res : INTEGER;
    name : FDDBT.Name;
    Enum : PROCEDURE (name : ARRAY OF CHAR);
    obj : TVV;
END;
```

To assign a particular TVV to the track editor, a "CollectMsg" is broadcast to the application whose TVV are subject of editing. The application then may decide, to whom the message is passed on and this will usually be – as it is the case with FDD – the selected object. Using the enumerator of the "CollectMsg", it tells FDDTVV, which of its variables are TV. The track editor must then display a list gadget with all enumerated strings (which are the names of TVV's) to give the user the possibility to select a particular one. If done so, the list Gadget executes the command "TCTVVFrame.SetTVV <s>~" with <s> being the name of the selected TVV name. TCTVVFrame.SetTVV then again sends a "CollectMsg", but this time with its id parameter set to Objects.get and receives in turn a pointer ("obj") to the selected TVV, the track editor thereafter operates on.

B.2.2 FDDDis

Significant types: Displacement, BasicDisplacement
Significant procedures: (Displacement) Get, NewSalto, NewTranslation, NewWheel
Stack
Significant variables: newDisplacement

In order to implement displacements and absolute positioning, this module cares about the first option. To provide a flexible approache, it implements a "Displacement" as a list node with a "virtual" procedure "Get" to determine the AT represented by the instance. The definition as list node makes it possible to define arbitrary depth displacement stacks, whereby the total AT defined by such a linear list is computed by use of the

PROCEDURE Stack*(p : FDDDS.List; ta, time, tb : REAL; VAR exists : BOOLEAN; VAR a : FDDAT.ATDesc);

which concats the AT returned by calls of the method

PROCEDURE Get (ta, time, tb: REAL; VAR exists: BOOLEAN; VAR a: FDDAT.ATDesc; VAR v: FDDSpace.Cor);

in each list node. From the fact that displacements are – as primitives – dynamically linked to the runtime system, generator procedures (they attach the newly created object at "newDisplacement") must be provided, which are called by "FDDTime.AddDis".

As examples for displacements, the compound type "BasicDisplacement" in association with the generator procedures "NewSalto", "NewTranslation" and "NewWheel" has been implemented. An integer variable serves to distinguish between the different displacements, whereas if may take on the following values: DUMMY, WHEEL, SALTO and TRANSLATION. The first one serves as initial dummy displacement whenever a new "ATTimeState" (see below) is created by the track editor.

B.2.3 FDDTime

Significant types: State, ATTime, CS, Scene, TVSurface
Significant procedures: (State) Add, Restore, SetFixpoints, SetVelocities
(ATTime) Generate, Overlay, Set, SetATKey, RestoreAT, InterpolateAT
(CS) Spline, InfluencedBy
(TVSurface) Generate, Restore, Interpolate
(Scene) SetCamera, Source, Delete, Express
AddDis
Significant variables: keyconstructor

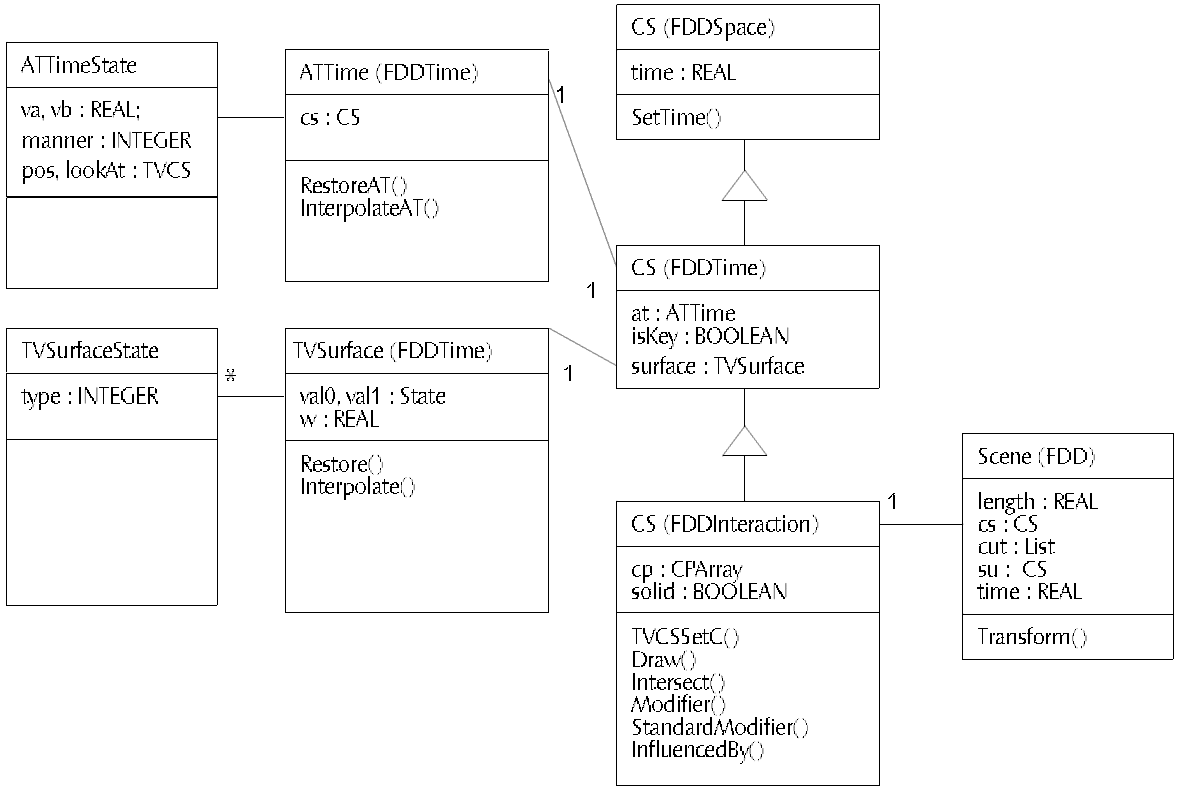


Figure B.5: CS UML class diagram

With FDDATTime, we have reached another fundamental part of FDD: TV of AT. Remember: the AT in the trees of CS (as defined by FDDSpace) are all of a static nature and need to be equipped with some kind of an "engine" that makes them TV. CS have been designed to a priori not being TV, since this might not be necessary in many applications such as two dimensional drawing packages, math programs or editors for static three dimensional objects. The situation before FDDATTime looks as follows:

1.) Each CS has an AT.
2.) This AT can only be accessed by the SetA method.
3.) A CS is not derived from FDDTVV.TVV.

Here, the reader might try to find the "best" solution.

B.2.3.1 Adding time varation

With FDDATTime, I give *my* answer in addition with an elegant and simple space–time positioning algorithm split up into two parts "RestoreAT" and "InterpolateAT" as proposed by the concept of TCTVV. First of all, one recognizes that a TV CS is a special CS and the concept of inheritance lies thus near: a TV CS has been defined as

```
CS = POINTER TO RECORD (FDDSpace.CS)
        isKey: BOOLEAN;
        at: ATTime;
        PROCEDURE Spline (ta, time, tb: REAL; VAR exists: BOOLEAN; VAR a: ATCS; VAR v: FDDSpace.Cor);
    END;,
```

whereby the "Spline" method will be discussed with the dissection of the time–space positioning algorithm. The important definition made here is the newly introduced field ".at": adding a TV variable to a descriptor is the only way to bring in TV into a hierarchy of derived non–TV types. When a non–TVV of type X is equipped with TV (as the CS' AT is it here), the type of the added TV should be called XTime (if a new TV such as a TV integer is defined, its type should be called "TVInt"). It becomes obvious, how time in such a TV CS will be set: by using the SetTime method of the TCTVV ".at". To be able to modify the AT, an instance of type ATTime must know the CS derivative it is attached to, which motivates the following definition:

```
ATTime* = POINTER TO RECORD (FDDTVV.TCTVV)
    cs : CS;
    (*--------------------*)
    PROCEDURE ATTimeHandler(self : ATTime; VAR M: Objects.ObjMsg);
    VAR obj : ATTime; o : Objects.Object;
    BEGIN
        IF M IS FDDTVV.InterpolateMsg THEN
            WITH M : FDDTVV.InterpolateMsg DO
                InterpolateAT(M.sa,M.ta,M.time,M.tb,M.sb,M.manner);
            END;
            TCTVVHandler(M);
        ELSIF M IS FDDTVV.RestoreMsg THEN
            WITH M : FDDTVV.RestoreMsg DO
                RestoreAT(M.state,M.time);
            END;
            TCTVVHandler(M);
        ELSIF ...
    END ATTimeHandler;
    (*--------------------*)
    PROCEDURE RestoreAT(s : FDDTVV.State; time : REAL);
    BEGIN
        ...
    END RestoreAT;
    (*--------------------*)
    PROCEDURE InterpolateAT(sa : FDDTVV.State; ta, time, tb : REAL; sb : FDDTVV.State; manner : INTEGER);
    BEGIN
        ...
    END InterpolateAT;
END;.
```

The following sections define the nature of an AT state and then derive the implementation of both methods "RestoreAT" and "InterpolateAT".

B.2.3.2 AT states

A basic ingredient of an AT state has already been mentioned: a stack of displacements in form of a list, whereas for *absolute* positioning, a reference to a CS is required. These two record fields (.dis, .pos) are in fact the fundamental substance of an

```
ATTimeState = POINTER TO RECORD (FDDTVV.State)
    pos : CS;
    dis : FDDDS.List;
    (*--------------------*)
    PROCEDURE Add (me: FDDDis.Displacement);
END;
```

consists of. As usual, an "ATTimeState" defines its visual representation for FDDTrackView, which is basically the string of all displacements in this case. The handle "Add" allows to overlay the current displacement stack an additional member, which thus of course must be inserted at the *head* of the list (1.20).

There may soon be additional requirements: the current definition of an "ATTimeState", does not allow to rotate an object at a certain place in direction of another – potentially dynamic – object, as it is essentially required for camera perspectives. A similar feature is the restriction that an object should follow a spline and thus setting its position to coordinates on the spline and adjust its rotational position to its derivative. These two cases are satisfied by the additional record fields

```
manner: INTEGER; (*DERIVATIVE/TARGET*)
lookAt: CS;,
```

and the method

```
PROCEDURE SetFixpoints (m0: INTEGER; p0, l0: CS);
```

allows both coordinate references (.pos, .lookAt) including the manner, the .lookAt reference is interpreted, to be set by a single call. To be able to influence the interpolation of AT States, might come the wish to specify velocities with which a state is approached/left, which completes the definition of an "ATTimeState":

```
va, vb: REAL;
```

B.2.3.3 Restoring an AT

As a necessity from TCTVV, the definition of a procedure that restores a state upon receiving a "RestoreMsg" is required. One might have been wondering, why FDDTVV equips that message with an additional time parameter, since for a time in [s.Key(),s.Tb()], the value of the TV is assumed to be constant, according to the ideas behind TCTVV. The reason is as follows: there exist A) explicit and B) non–explicit states. The first type determines the value of the TVV completely from the local state data, whereas the latter requires information from other objects, to which it refers by the help of pointers. One recognizes that an "ATTimeState" is a typical representant of case B) since it relies on the correct spacial position of the ".pos", as well as of the ".lookAT" CS and must therefore be able to set their time accordingly.

Lines 0 to 2 (see next page) establish preconditions that allow a TV AT to be set up. If the "ATTimeState" has a valid CS .pos reference, its position is set up and the object is moved to the spline position defined by the "Spline" handler (Line 10: the computed translation is stored in a0.t). If .pos is NIL, there are different cases that must be individually treated. First of all, line 12 determines the biggest (with respect to time) predecessor and the smallest successor of the state s to be restored. Table 2.1 summarizes the different outcomings and the current local track situation together with the corresponding AT to be set. The implementation of the four possible cases is given by the lines 13 to 20.

Thereafter, the AT defined by the (potentially empty) displacement stack is computed and applied to the AT to be set (line 22). Before setting the AT definitively (line 38), lines 24 to 33 turn the CS into the direction of the referenced object by defining the rotational part of the AT if the .lookAt reference is valid.

```
    PROCEDURE RestoreAT(s : FDDTVV.State; time : REAL);
    VAR abs, rel, atL, atR : FDDAT.ATDesc; a, a0 : ATCS; v : FDDSpace.Cor; aL, aR : State; temp : REAL; parent : FDDSpace.CS;
    BEGIN
0       IF cs.Parent()#NIL THEN
1           parent := cs.Parent(); parent(CS).at.SetTime(time,FALSE); (*establish precondition: a set up parent*)
2           IF parent(CS).at.exists THEN
3               WITH s : State DO
4                   abs.I(); DefineATCS(a0,abs,cs.Parent());
5                   IF (s.pos#NIL) THEN (**)
6                       temp := s.pos.at.Time();
7                       s.pos.at.SetTime(time,FALSE); IF ~s.pos.at.exists THEN exists := FALSE; RETURN; END;
8                       s.pos.Spline(s.Key(),time,s.Tb(),exists, a,v);
9                       s.pos.at.SetTime(temp,FALSE); (*restore time*)
10                      FDDSpace.Copy(a.t,a0.t);
11                  ELSE
12                      AbsBounds(aL,s,s,aR); (*postcondition: (aL#s) & (aR#s) from (**) *)
13                      IF (aL#NIL) & (aR#NIL) THEN
14                          InterpolateAT(aL,aL.Tb(),time,aR.Key(),aR,aL.Interpolation());
15                      ELSIF (aL#NIL) & (aR=NIL) THEN
16                          RestoreAT(aL,aL.Tb());
17                      ELSIF (aL=NIL) & (TRUE) THEN
18                          abs.I(); cs.SetA(abs);
19                      END;
20                      cs.A(abs); DefineATCS(a0,abs,cs.Parent());
21                  END;
22                  FDDDis.Stack(s.dis, s.Key(), time, s.Tb(),exists,dis); abs.Concat(abs,dis);
23                  IF (s.lookAt#NIL) THEN
24                      s.lookAt.at.SetTime(time,FALSE);
25                      CASE s.manner OF
26                          DERIVATIVE : (*the view direction as rotational part ot the spline's AT:*)
27                              s.lookAt.Spline(s.Key(),time,s.Tb(),exists,a0,v);
28                          | TARGET : (*compute the POINT IN SPACE e0 where "me" is looking at:*)
29                              s.lookAt.Spline(s.Key(),time,s.Tb(),exists,a0,v);
30                              FDDSpace.Sub(a0.t, a0.t,a.t);
31                              getCSUponE0(a0, a0.t);
32                      END;
33                      FDDSpace.Copy(a.t, a0.t); (*add the translatorial part*)
34                  END;
35                  ExpressATCS(a0,cs.Parent());
36                  ExtractAT(abs, a0);
37                  cs.SetA(abs);
38              END;
39          ELSE
40              exists := FALSE;
41          END;
42      END;
    END RestoreAT;
```

B.2.3.4 AT state interpolation

One might be afraid that interpolating AT states could turnout to be more complex, but – luckily – it can elegantly be expressed in terms of RestoreAT. The procedure interface gives the previous ("sa") and the next state ("sb"), the corresponding time window [ta,tb], the therein contained position where the TVV value need to be recomputed (time) and the interpolation technique to be used (manner). What comes first is business as usual: lines 0 to 2 establish necessary preconditions. Then, from 5 to 13, the exceptional case of "sb" being NIL is treated. As precondition, we can assume "sa" not to be NIL (TCTVV). Table 2.2 summarizes all other cases with sb=NIL and gives the AT that must be set: with all these cases, no interpolation is necessary at all. What remains, is to consider all cases with sb#NIL: they are summarized by table 2.3 and one concludes that the AT to be set can be written as simple AT interpolation between the AT set by Restore(sa,sa.Tb()) and Restore(sb,sb.Key()) (Lines 15, 16), which is done in 34 or 35. Module TVV sends the "InterpolateMsg" with the interpolation technique selector being set to the one defined by "sa". Lines 17–31 compute the AT derivatives, whereas 32 is a C1 interpolation in time.

```
    PROCEDURE InterpolateAT(sa : FDDTVV.State; ta, time, tb : REAL; sb : FDDTVV.State; manner : INTEGER);
    CONST E = 1/1000; F = 1/5;
    VAR aL, aR : State; a0, a1, a2, a3, da0, da1, a, rel : FDDAT.ATDesc; parent : FDDSpace.CS; dta, dtb : REAL;
    BEGIN (*precondition: sa#NIL (from TCTVV)*)
0       IF cs.Parent()#NIL THEN
1           parent := cs.Parent(); parent(TVCS).at.SetTime(time,FALSE); (*establish precondition: a set up parent*)
2           IF parent(TVCS).at.exists THEN
3               WITH sa : State DO WITH sb : State DO
4                   AbsBounds(aL,sa,sb,aR);
5                   IF (sb=NIL) THEN (*so: we also have aR=NIL*)
6                       IF (aL=sa) THEN
7                           RestoreAT(aL,aL.Tb());
8                       ELSE
9                           IF (aL#NIL) THEN
10                              RestoreAT(aL,aL.Tb());
11                          END;
12                          FDDPos.Stack(sa.rel,sa.Key(),sa.Tb(),sa.Tb(),exists,rel); cs.Overlay(rel);
13                      END; (*shortened!*)
14                  ELSE
15                      RestoreAT(sa,sa.Tb()); cs.A(a0);
16                      RestoreAT(sb,sb.Key()); cs.A(a1);
17                      dta := sa.Tb()–sa.Key(); dtb := sb.Tb()–sb.Key();
18                      IF (dta#0) THEN
19                          RestoreAT(sa,sa.Tb()–E*dta); cs.A(da0); da0.Neg(); da0.Add(a0); da0.Scale(1/(E*dta)); da0.Add(a0);
20                      ELSE
21                          da0.Set(a1); da0.Sub(a0); da0.Scale(F); da0.Add(a0); (*F–scaled direction to at1*)
22                      END;
23                      IF (dtb#0) THEN
24                          RestoreAT(sb,sb.Key()+E*dtb); cs.A(da1); da1.Sub(a1); da1.Scale(1/(E*dtb)); da1.Add(a1);
25                      ELSE
26                          IF (sb.Next()#NIL) THEN
27                              RestoreAT(ATCast(sb.Succ()),FDDDS.Key(sb.Next())); cs.A(da1); da0.Scale(F);
28                          ELSE
29                              da1.Set(a0); da1.Sub(a1); da1.Neg(); da1.Scale(F); da1.Add(a1);
30                          END;
31                      END;
32                      time := SHORT(FDDTVV.InterpolateScalar(sa.vb,sa.Tb(),sa.Tb(),time,sb.Key(),sb.Key(),sb.va,C1));
33                      CASE manner OF
34                        FDDTVV.C0 : a.LinInt(a0,sa.Tb(),time,sb.Key(),a1);
35                          | FDDTVV.C1 : ATC1(a, a0,da0,a1,da1, (time–sa.Tb()/(sb.Key()–sa.Tb())));
36                      END;
37                  END;
38                  cs.SetA(a);
39              END; END;
40          ELSE
41              exists := FALSE; (*parent does not exist.*)
42          END;
43      END;
    END InterpolateAT;
```

B.2.3.5 TV surfaces

As a second implementation of TCTVV, TV surfaces have been implemented and the CS are equipped with this facility. The technique of interpolating surfaces might be of particular interest, whereby a surface state is defined as follows:

```
State = POINTER TO RECORD( FDDTVV.State )
    type: BasicGadgets.Integer;
    procedure: BasicGadgets.String;
    picture: BasicGadgets.String;
    diffuse: FDDBT.Color;
    specular: FDDBT.Color;
    emission: FDDBT.Color;
    specularExp: BasicGadgets.Integer;
END;
```

It features totally three surface types: A) procedure–based, B) bitmap–based and C) colored. From the fact that upon a request to interpolate two such surface states, one is faced with the fact that by carrying out this process, the infinite resolution of surfaces based on procedures would be lost by the application of a discretization process and that this should be avoided. This problem is solved by delegation of the task to the renderer which in turn must determine the colour of an intersected object by interpolating between the two values "val0" and "val1" of the TCTVV

```
TVSurface = POINTER TO RECORD (FDDTVV.TCTVV)
    val0*: State;
    val1*: State;
    w*: REAL;
    (...)
END;
```

whereas the scalar "w" in [0,1] determines the weight of the second state (val1). From that, both necessary procedures to recompute the TCTVV value are trivially given by

```
PROCEDURE Restore(state : FDDTVV.State; time : REAL);
VAR c : Objects.CopyMsg;
BEGIN
    c.id := Objects.deep; Objects.Stamp(c); state.handle(state,c); val := c.obj(State);
    w := 0.0; (*full weight on val0*)
END Restore;

PROCEDURE Interpolate(sa : FDDTVV.State; ta, time, tb : REAL; sb : FDDTVV.State; manner : INTEGER);
CONST D = 1;
VAR c : Objects.CopyMsg;
BEGIN
    WITH sa : State DO WITH sb : State DO
        IF (sa.type.val=COLOR) & (sb.type.val=COLOR) THEN
            InterpolateColor(val0.diffuse, sa.diffuse,ta,time,tb,sb.diffuse, manner);
            InterpolateColor(val0.specular, sa.specular,ta,time,tb,sb.specular, manner);
            InterpolateColor(val0.emission, sa.emission,ta,time,tb,sb.emission, manner);
            val0.specularExp.val :=
                ENTIER(FDDTVV.InterpolateScalar(D,sa.specularExp.val,ta,time,tb,sb.specularExp.val,D, manner));
            w := 0.0; (*full weight on val0*)
        ELSE
            c.id := Objects.deep; Objects.Stamp(c); sa.handle(sa,c); val0 := c.obj(State);
            c.id := Objects.deep; Objects.Stamp(c); sb.handle(sb,c); val1 := c.obj(State);
            w := (1–(time–ta)/(tb–ta));
        END;
    END; END;
END Interpolate;
```

whereas the interpolation of two color–based surfaces is carried out explicitly for efficiency reasons.

B.2.3.6 Scenes

The data type "Scene" unifies methods concerning an entire CS hierarchy: it maintains the cut list, which is the ordered list of timestamped cameras. To activate a camera at a specified time, the method "SetCamera" may be called. It is derived from FDDTVV.TVV.

Based on the CS method "InfluencedBy", which determines whether an object is influenced by another one by scanning its track for absolute positions and targets, "Source" determines whether an objects determines any other one's position and returns FALSE if not, which is the case if it is a source [SI98] in the graph of "A is positioned by B" relations. Based on Source, "Delete" determines, whether an object is bound or not and accordingly deletes it. Note that if new dependencies beside positioning between coordinate systems are introduced by inheritance, the method "InfluencedBy" must be redefined.

"SetCS" assigns a CS hierarchy to the scene, whereas "Express" expresses the entire scene in a specified CS. To find objects according to their name, the "Find" method may be used.

B.2.4 FDDScript

Significant types: –
Significant procedures: Generate
Significant variables: F, newObj, s

In order to open the facilities of FDD to script–generated documents, a language to textually specify dynamic three dimensional scenes has been defined. Its syntax is widley determined by FDD's internal data structure and can thus be easily understood. The syntax in EBNF is given by table B.2 and the primitive–specific parser syntax shown by table B.3.

The internal scene generation process shall now be discussed. Upon a call of

PROCEDURE Generate*(VAR from : Texts.Scanner);

FDDScript starts the generation of a new scene. After having consumed a "NEW" statement, FDDScript calls the subsequently following handler which reads the primitive–specific data and must attach the newly generated object at the global variable "FDDScript.newObj". The primitive parser must use "FDDScript.s" as AT state parameter upon object initialization, since this is the first track entry of the TV AT. The keyword "t" is used to signal the beginning of the track description, whose syntax can be read from table B.2. If the end of the scene description is reached, "FDDScript.F" contains the generated frame which is now at the disposition of the module client.

Scene	=	{"NEW" Primitive} "~".
Primitive	=	parser parameters "t" Track.
Track	=	{ "(" State ")" }.
State	=	"(" ta tb Interpolation ["p" Position] ["l" LookAt] { "d" Displacement } ")".
Position	=	x y z.
LookAt	=	? manner.
Displacement	=	{ generator parameters }.
Interpolation	=	C0 | C1.

Table B.2: FDDScript syntax in EBNF

Parser	Syntax		
FDDObj.ReadSphere	Sphere	=	radius.
FDDObj.ReadCylinder	Cylinder	=	radius height.
FDDObj.ReadLightLight	Light	=	.
FDDObj.ReadCamera	Camera	=	distance magnification.
FDDObj.ReadParabolaChain	ParabolaChain	=	tips "(" x y z ")" { "(" x y z ")" }.
FDDObj.ReadPlane	Plane	=	.

Table B.3: Primitive parser syntax in EBNF

B.3 Interaction

B.3.1 FDDFrameBuffers

Significant types: FrameBuffer
Significant procedures: InitFrameBuffer, MakePict, Clear
Significant variables: (FrameBuffer) self, buf

FDDFrameBuffers is the base type of the FDD views FDDFrames.FDDFrame, FDDFilms.Film and FDDContainers.FDDContainer. Frame buffering is based on the picture "buf", which has been made accessible to type derivatives. Upon resizing, a frame buffer of new size is created and the old contents are copied. "Clear" erases the frame buffer contents with a specified color. The command "MakePict *" inserts a Rembrandt frame of the marked FrameBuffer at the caret. The initializer determines the initial frame buffer color and whether its contents are made persistent on storing:

```
PROCEDURE InitFrameBuffer (me: FrameBuffer; p0: BOOLEAN; col: INTEGER);
```

B.3.2 FDDFilms

Significant types: Film, PlayTask, PlayMsg, Adjustment, FrameAdjustment
Significant procedures: (Film) Height, Width, Play/Stop, Suspend/Reactivate, Add
(PlayTask) Suspend/Reactivate, RemoveThis, Remove
(FrameAdjustment) Add, Compute
Convert
Significant variables: –

The data type "Film" is a simple implementation of frame–based animations. A "Film" primarily features a list of frame descriptions that allow to subsequently display appropriate frames. The term description is by no means arbitrary: as a brute force implementation, a first description format is a list of pictures implemented by a FDDDS.List. The method "Add" allows a new frame to be added, whereas its key specifies the associated time. An assertion ensures that only pictures of identical size can be inserted.

There are methods "Play", "Suspend", "Reactivate" and "Stop" to control its visual activity. They operate on a "PlayTask" which implements the background refresh signals and which is installed/removed on a call of "Play"/"Stop". In its initializer, a "PlayTask" takes a list of frames as parameter, to which it subsequently broadcasts instances of the type "PlayMsg" which are the refresh signals. The addressed frame – after having performed the visual update – may then specify the delay to the next play signal by the aid of the delay field of the "PlayMsg", which allows frame rates that adapt to current workload.

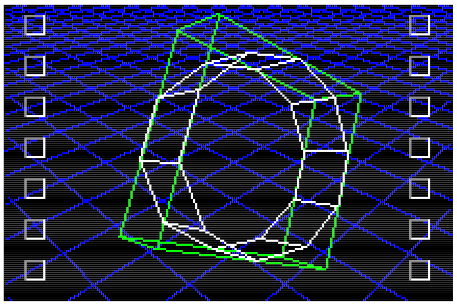
As a second frame description, delta compression, computing a frame upon the change with respect to its predecessor, has been implemented. Each such "FrameAdjustment" consists of an arbitrary large array of pixel adjustments and is computed by calling its "Compute" method. "Add" allows its application on an arbitrary picture, which will as usual be a frame buffer. From the fact that a pixel adjustment's coordinates are stored by a single integer value, only films of small dimensions can be converted into the delta compression format so far, whereas the global command "Convert" performs the conversion.

Table B.4 summarizes the findings of the comparison between both frame desciption techniques. For the wireframe representation A) of a rotating cylinder, the required data amount can be reduced by a factor of two, whereas the break–even point is reached at only 4 frames. B) shows a rotating texture–mapped sphere and the power of runlength encoding in association with large homogeneous display areas. The performance ratio drops to 1.3 and with only 10 frames, the required amount of data is larger than with runlength encoding. This is caused by the fact that the first frame requires totally 30816 adjustments (the whole picture), which could greatly be reduced by storing the first frame using runlength encoding. Since each adjustment requires totally 3 bytes, the data amount could be reduced from 199KB down to

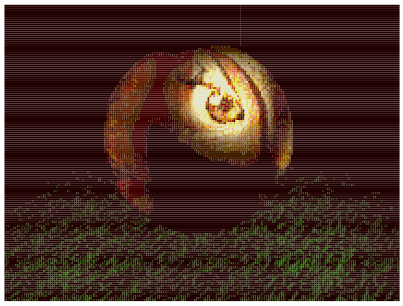
$$199\text{KB} - 3*30.8\text{KB} + (137\text{KB}/10) = 120.3\text{KB}, \qquad \text{(B.1)}$$

thus already compressing the data amount.

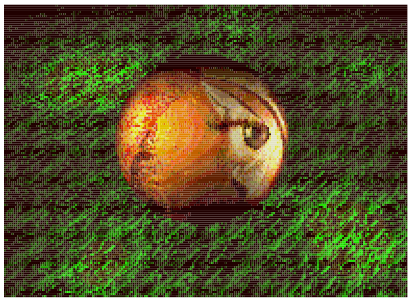
The disadvantages of runlength encoding are pointed out by scene C) which gives the delta compression a performance ratio of 2.4 by exploiting the fact that again only the sphere has a dynamic nature. In conclusion, the second description technique could ideally be improved by runlength encoding of frame adjustments. In the case of dynamic cameras, both compression techniques will prove uneffective.



A) n=100 frames @ 216 x 146



B) n=10 frames @ 214x144



C) n=10 frames @ 210x141

i)		1119			137			289
ii)		571			199			209
iii)		6329			30816			29610
iv)	~	1867		~	3533		~	4600
v)	~	11.2n		~	13.7n			28.9n
vi)	~	19 + 5.6n		~	93 + 10.7n		~	89 + 12.0n
vii)	~	11200		~	13700		~	28900
viii)	~	5619		~	10793		~	12089
ix)	n=	4		n=	31		n=	6
x)		2.0		~	1.3		~	2.4

i) File size (runlength picture encoding, KB)
ii) File size (delta compression, KB)
iii) Adjustment amount for first frame
iv) Average adjustment amount for following frames
v) Runlength compression complexity (KB)
vi) Delta compression complexity (KB)
vii) Estimated file size (runlength picture encoding, 1000 frames, KB)
viii) Estimated file size (delta compression, 1000 frames, KB)
ix) Break even point
x) Delta compression performance ratio (frame average)

Table B.4: Delta versus runlength compression

B.3.3 FDDGFX

Significant types: Viewport
Significant procedures: Clip, ClippingOn/Off, Line, PicFrame, Left/Right/Lower/UpperBorder
Significant variables: f, xPPM, yPPM

There are two main purposes of this module: A) provide graphical primitives such as lines, rectangles, ... being drawn into frame buffers, B) give 2D clipping algorithms and C) define the look of FDD frames. The first part is covered by methods that all feature an interface only slightly modified from the Gadgets display module Display3:

```
PROCEDURE Arrow (p: Pictures.Picture; c, x0, y0, x1, y1: INTEGER);
PROCEDURE Ellipse (p: Pictures.Picture; col, X, Y, a, b: INTEGER; style: SET; mode: INTEGER);
PROCEDURE Line (p: Pictures.Picture; c, x0, y0, x1, y1, mode: INTEGER);
PROCEDURE Rectangle (p: Pictures.Picture; c, X, Y, W, H, mode: INTEGER);
```

Parametric 2D clipping [GR97] for lines can be toggled on/off using the pair

```
PROCEDURE ClippingOn;
PROCEDURE ClippingOff;
```

of procedures, with

```
PROCEDURE SetViewport*( x0, y0, x1, y1 : INTEGER);
```

defining the viewport. Explicitly, it can be done by using

```
PROCEDURE Clip (VAR x0, y0, x1, y1: INTEGER; v: Viewport): BOOLEAN;
```

whereby a return value FALSE signals, that the line lies completely outside the specified rectangular clipping area and does thus not need to be painted.

The third point C) is based on the following set of methods:

```
PROCEDURE AddFrame (VAR x, y, w, h: INTEGER);
PROCEDURE SubtractFrame (VAR x, y, w, h: INTEGER);
PROCEDURE Frame (Q: Display3.Mask; x, y, w, h: INTEGER; str: ARRAY OF CHAR);
PROCEDURE LeftBorder (): INTEGER;
PROCEDURE LowerBorder (): INTEGER;
PROCEDURE RightBorder (): INTEGER;
PROCEDURE UpperBorder (): INTEGER;
```

The add/subtract pair allows the conversion of border (frame on the screen) into client coordinates (visible screen area inside the frame), whereas the "Border" methods explicitly give the frame width.

B.3.4 FDDInteraction

Significant types:	FDDFrame, Reference, ConstructionParameters, Camera, CS
Significant procedures:	(FDDFrame) Draw, Line3D, Clip3D, ToScreen, Track, Navigate, Select, Modify, Coords (CS) Draw, Intersect, Modifier, AddAT, SetPosition, Modifier Constructor
Significant variables:	firstCol, secondCol, selCol, selection, cpar

B.3.4.1 Visualization

```
CS = POINTER TO RECORD (FDDTime.CS)
  cp: CPArray;
  solid: BOOLEAN;
  PROCEDURE Compute;
  PROCEDURE Draw (F: Display.Frame; col: INTEGER);
  PROCEDURE FDDInteractionCopy (VAR M: Objects.CopyMsg; from, to: CS);
  PROCEDURE FDDInteractionHandler (VAR M: Objects.ObjMsg);
  PROCEDURE FDDInteractionSetC (v: FDDSpace.CorarrayDesc);
  PROCEDURE Intersect (p0, dir: FDDSpace.Cor; VAR lambda: REAL; VAR n: FDDSpace.Cor; VAR tx, ty: REAL): BOOLEAN;
  PROCEDURE Modifier (cp: INTEGER; time: REAL; x, y: INTEGER; F: Display.Frame; VAR res: INTEGER);
  PROCEDURE SetPosition (a: FDDAT.ATDesc; time: REAL; VAR res: INTEGER);
  PROCEDURE StandardModifier (cp: INTEGER; time: REAL; VAR res: INTEGER);
END;

FDDFrame = POINTER TO RECORD (FDDFrameBuffers.FrameBuffer)
  time: REAL;
  scene: FDDTime.Scene;
  drawTree, focus: CS;
  camera: Camera;
  PROCEDURE Clip3D (p0, p1: FDDSpace.Cor): BOOLEAN;
  PROCEDURE Line3D (col: INTEGER; p0, p1: FDDSpace.Cor);
  PROCEDURE ToScreen (v: FDDSpace.Cor; VAR sx, sy: INTEGER);
  PROCEDURE Draw;
  PROCEDURE Intersect (me: FDDTime.CS; recursive: BOOLEAN; p0, dir: FDDSpace.Cor;
      VAR t: CS; VAR p, n: FDDSpace.Cor; VAR lambda, tx, ty: REAL): BOOLEAN;
  PROCEDURE Play/Stop;
  PROCEDURE Track (x, y, MX, MY: INTEGER; VAR l: FDDDS.List);
END;
```

FDDInteraction with its CS class specifies handlers that must be customized by derived primitives to define specifiy behaviour. Remarks: (1) The Boolean variable "solid" determines, whether the object is treated by the renderer. A primitive with a solid field being FALSE can thus only be accessed by control points. (2) The "cp" array determines, which coordinates are active control points. (3) The "Modifier" calls the "StandardModifier" when not being redefined.

Primary visualization actor beside the draw handler is the primitive called "Camera" which defines itself – as all primitives – as a CS derivative featuring two additional state variables: the distance to the viewplane (d) and a linear magnification factor (m). Recall that the observer sits at (–d,0,0). An FDDFrame refers to the camera whose view it depicts and posesses a scalar variable for its time. Based on the frame's size, the method "Draw" depicts the attached scene to the framebuffer. For wireframe animation previews, the "Play"/"Stop" method installs/removes a play task. The potentially recursive (with respect to the CS hierarchy) "Intersect" method is called by the renderer and "Track" determines a list of object references according to the mouse coordinate (MX,MY).

B.3.4.2 Modification

```
WITH M: Oberon.InputMsg DO
    IF (M.id=Oberon.track) & (Gadgets.InActiveArea(self, M)) THEN
        IF (title=TRUE) THEN
            Title(x,y,M);
        END;
        IF (FDDBT.PublicBool(NAVIGATEFLAG)) & (M.keys#{}) THEN
            Navigate(M);
        ELSE
            IF (M.keys = {MIDDLE}) THEN
                Modify(x,y,M);
            ELSIF (M.keys = {LEFT}) THEN
                Create(Q,x,y,w,h,M);
            ELSIF (M.keys = {}) THEN
                Coords(Q,x,y,w,h,M);
            ELSIF (M.keys = {RIGHT}) THEN
                Select(x,y,M,keysum);
            END;
        END;
    ELSE
        baseClass(self,M);
    END;
END;
```

Upon receiving a mouse event, a framehandler may – according to section 3.1 – either A) change the view or B) customize the data structure, whereas customization means either starting a construction process or modifying the data structure according to a selection. The framehandler input message dispatcher has thus been defined as shown above: "Navigate" implements the navigation transformations (3.1) to (3.4). "Create" starts the primitive construction by calling the generator procedure set by "Constructor" (the terms involved in construction are discussed with the dissection of module FDDObj). "Select" sets the "selection" according to the nearest touched object and deletes it when left interclicking occurs (and the object is not bound), whereas "Coords" displays the intersected object's name and the touched coordinates in the view.

"Modify" starts the customization of the data structure based on the touched control point. The fundamental actor behind the scenes is

```
PROCEDURE AddAT*( at : FDDAT.ATDesc; VAR to : CS; VAR u : Upcall; time : REAL; VAR res : INTEGER);
```

which is called whenever affine transformations of the data structure are customization subjects. It tries to move/rotate an object o at time t according to the following rule, which does *not* allow the definition of animated keys:

```
IF o IS Key THEN
    Add AT to the SimplePos to o's track entry
ELSIF we have no track entry at time t THEN
    Create a key k as o's absolute position at time t;
    Add AT to k
END;.
```

The definition thus is that a key has a sole track entry with a displacement giving the affine transformation with respect to its parent. Note that in principle, dynamic keys are possible, but this causes major problems with the graphical customization of affine transformations represented by them.

B.3.5 FDDContainers

Significant types: –
Significant procedures: (FDDContainer) NXNY
GetFilm, MakePict, NewDoc, Render, New, ResetCuts
Significant variables: –

"FDDContainer" forms FDD's user interface in association with custom editor sections as described in the user guide. To get films from the depository into documents, the command "GetFilm *" inserts the star–marked film at the caret. "MakePict *" inserts a Rembrandt frame of the marked FDDContainer at the caret., whereas the "Render" command is called by the R! button of the visual frame representation. "NewDoc" creates a new text document and inserts an FDDContainer at the first position of the text stream.

```
FDDContainer = POINTER TO RECORD (FDDFrameBuffers.FrameBuffer)
   mode       : BasicGadgets.Integer;
   eB, dB, pB : BasicGadgets.Button;
   timebar    : Scrollbars.Frame;
   playing    : BasicGadgets.Boolean;
   playB      : BasicGadgets.Button;
   renderB    : BasicGadgets.Button;
   model      : FDDInteraction.FDDFrame;
   films      : FDDDS.List;
   time       : BasicGadgets.Integer;
END;
```

From an implementational point of view, an FDDContainer refers to a model, which is an FDDFrame being displayed in the editor mode. In director mode, an FDDFrame for each camera is fitted into the container extent, whereas in the player mode, the same procedure occurs with the list of films, of which each of course is also a frame. In the director mode, parental control [FIMA98] is executed by catching left mouse button clicks and assigning the camera of the hit FDDFrame to the scene at the current time, thereby defining the cut list. Though the cut list can be reset by easy means, an implementation as TCTVV would allow its editing.

Resizing the container in the editor mode fixes the aspect ratio, whereas in other modes, it can be freely customized. The constant aspect ratio of films and multiple subviews require a positioning algorithm which computes the number nx/ny of frame in horizontal/vertical direction and the respective tile width/height tw/th based on the total subframe number n and the frame width/height w/h as follows:

```
PROCEDURE NXNY( n, w, h : INTEGER; VAR nx, ny, tw, th : INTEGER);
BEGIN
   nx := 0; ny := 0;
   WHILE nx*ny < n DO
      IF (w DIV (nx+1)) > (ENTIER((h DIV (ny+1))/Display.Height*Display.Width)) THEN
         INC(nx);
         tw := w DIV nx;
         th := SHORT( ENTIER( (w DIV nx)/Display.Width*Display.Height));
         ny := h DIV th;
      ELSE
         INC(ny);
         tw := SHORT( ENTIER( (h DIV ny)/Display.Height*Display.Width));
         th := h DIV ny;
         nx := w DIV tw;
      END;
   END;
END NXNY;
```

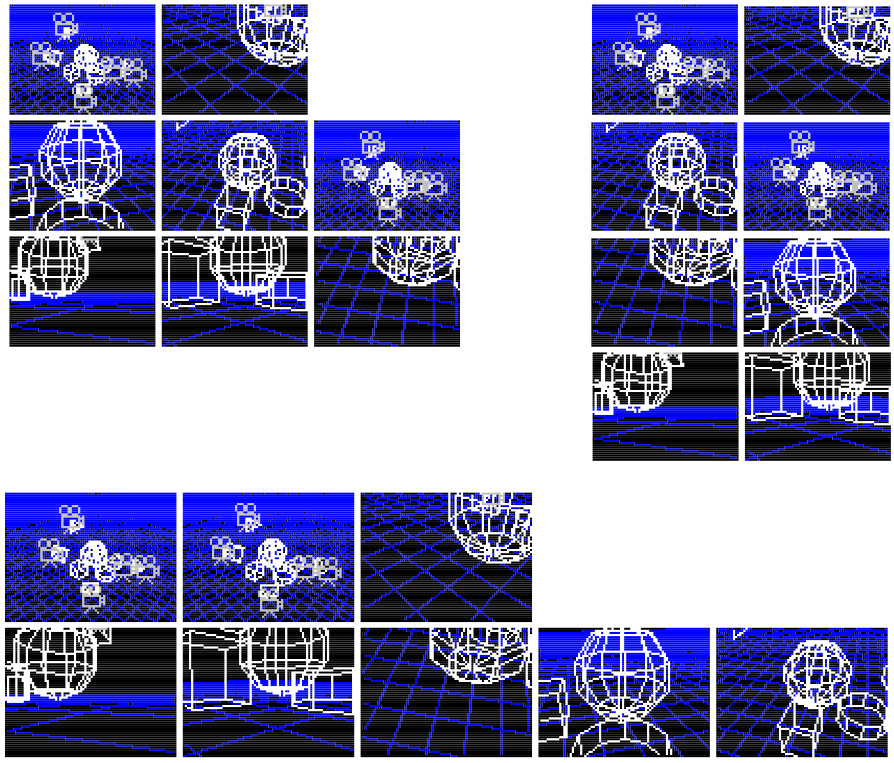


Figure B.6: Subframe positioning algorithm

B.3.6 FDDTextures

Significant types: Texture
Significant procedures: Chessboard, Chessboard3D, Mandelbrot, Rainbow, Rubik, Waves, YingYang
Significant variables: (x,y), (u,v,w), color

A small step towards photorealism provides the simple module FDDTextures. Upon 2D (x,y) or 3D texture coordinates (u,v,w), arbitrary textures returning a computed "color" may be specified. Table D.2 gives an overview on the implemented textures.

B.3.7 FDDRender

Significant types: Rendertask, Light
Significant procedures: (Rendertask) Init
Significant variables: NewLight

Realistic visualization is done by a simple rendering module implementing Phong's shading model [WA92]. It carries out the surface state interpolation and sends rendered film frames to the film object specified at initialization. To provide rendering previews, wireframe representations can be concatenated in the same manner. The constant AMBIENTLIGHT defines the name of the ambient light object inserted by the default scene generator in FDDObj. The primitive "Light" is defined as a CS derivative to bring in brightness.

B.3.8 FDDObj

Significant types: ModifierPars, BasicPrimitives
Significant procedures: –
Significant variables: m

B.3.8.1 Primitives as derived types

FDDObj is a collection of three dimensional primitives that can be inserted into scenes by using FDD's graphical editor. Each such object is derived from FDDInteraction.CS, such as a

```
Sphere = POINTER TO RECORD (FDDInteraction.CS)
   radius : REAL;
   ...;
```

which adds a single primitive–specific record field to the CS descriptor. To simplify the definition of new primitives, the compound data structure "BasicPrimitives" implementing spheres, cylinder, planes, parabola chains and keys in one type has been implemented. As Oberon allows the definition of virtual methods [PO94] as

```
MyType = RECORD
   PROCEDURE MyVirtualProc(); BEGIN HALT(99); END MyVirtualProc;
END;
```

FDDInteraction defines the following procedures which every primitive in turn must provide:

```
PROCEDURE Draw*(F : Display.Frame ; col : INTEGER);
PROCEDURE Intersect*( p0, dir : FDDSpace.Cor; VAR lambda : REAL; VAR n : FDDSpace.Cor; VAR tx, ty : REAL) : BOOLEAN;
PROCEDURE Modifier*( cp : INTEGER; time : REAL; x, y : INTEGER; F : Display.Frame; VAR res : INTEGER);
```

whereas the last executes a standard handler (StandardModifier) when not being redefined. All other specific behaviour such as storing/loading is done in Gadget–like manner.

B.3.8.2 Construction

Objects are dynamically linked to the runtime system and thus, a construction procedure as it is

```
PROCEDURE ConstructSphere*();
VAR step : REAL; b : BasicPrimitives;
BEGIN
   NEW(b); b.Init(b, FDDInteraction.cpar.re.o, FDDInteraction.cpar.s, SPHERE, 0.0, 0.0); b.Compute();
   m.keysum := {2};
   Unkey(m);
   m.u.Send(m.u,NIL,b);
   REPEAT
      Wait( m.xOld,m.yOld,m.x,m.y,m.keys,m.keysum);
      step := 2*((m.x–m.xOld)*FDDInteraction.cpar.xStep + (m.y–m.yOld)*FDDInteraction.cpar.yStep);
      b.radius := FDDBT.max(0.0,b.radius + step); b.Compute();
      m.a.I(); m.a.Trans(0,0,step);
      FDDInteraction.AddAT( m.a, FDDInteraction.cpar.k,FDDInteraction.cpar.sender.time,m.res);
      FDDInteraction.cpar.k.Express(FDDInteraction.cpar.sender.camera);
      b.at.SetTime(FDDInteraction.cpar.sender.time, TRUE);
      b.Express(FDDInteraction.cpar.sender.camera);
      m.u.Send(m.u,NIL,b);
   UNTIL (0 IN m.keysum);
   Postfix(m.keysum,FDDInteraction.cpar.k,b);
END ConstructSphere;
```

in the case of a sphere must be provided. The construction process is – as a major difference to FDD's predecessor TDD – fully under control of the generator. The construction parameters are – as it is necessary for dynamic binding – given by the global variable "FDDInteraction.cpar", which is an instance of

```
    ConstructorParameters* = POINTER TO RECORD
        sender* : FDDFrame; (*initiator*)
        xStep* : REAL; (*to use while defining new objects*)
        yStep* : REAL;  (*to use while defining new objects*)
        x0*, y0* : INTEGER; (*absolute frame position, where a "create" command took place*)
        re* : Reference;
        k* : FDD.TVCS; (*the key object*)
        p* : FDDATTime.State;
    END;
```

B.3.8.3 Drawing

Upon a framework call of its draw handler, a primitive has the following possibilities: A) Sketch lines between coordinates in three dimensional space, B) draw arbitrary objects based on coordinates and C) do arbitrary painting based on graphical primitives offered by FDDGFX. Of course, nothing prevents from combining the options in order to achieve the bestmost visual impression. FDD cares about coordinate transformations, clipping, the mapping to the screen and the viewport settings. The draw handlers of the sphere and the light (FDDRender) shall serve as respective examples for different draw styles:

```
PROCEDURE Draw(F : Display.Frame; col : INTEGER);
CONST M = PHISTEPS*(THETASTEPS+1);
VAR phiStep, thetaStep : INTEGER;
BEGIN
    WITH F : FDDFrames.FDDFrame) DO
        IF (self=FDD.selection) THEN col := FDDFrames.selCol ELSE col := FDDFrames.firstCol; END;
        FOR phiStep := 0 TO PHISTEPS-1 DO
            FOR thetaStep := 0 TO THETASTEPS-1 DO
                F.line3D(col,
                    x.c[CO+phiStep*(THETASTEPS+1)+thetaStep],
                    x.c[CO+phiStep*(THETASTEPS+1)+thetaStep+1]);
                F.line3D(col,
                    x.c[CO+phiStep*(THETASTEPS+1)+thetaStep],
                    x.c[CO+((phiStep*(THETASTEPS+1)+thetaStep+THETASTEPS+1) MOD M)]);
            END;
        END;
    END;
END Draw;

PROCEDURE Draw*(F : Display.Frame; col : INTEGER);
CONST S = 5; RAYS = 12;
VAR i : INTEGER; sx, sy : INTEGER; s, c : REAL;
BEGIN
    WITH F : FDDFrames.FDDFrame DO
        IF (self=FDD.selection) THEN col := FDDFrames.selCol ELSE col := FDDGFX.YELLOW; END;
        F.ToScreen(x.c[0],sx,sy);
        FDDGFX.ClippingOn();
        FDDGFX.Ellipse(F.buf, col, sx,sy, S, S, {}, Display.replace);
        FOR i := 0 TO RAYS-1 DO
            s := S*Math.sin( 2*Math.pi/RAYS*i); c := S*Math.cos( 2*Math.pi/RAYS*i);
            FDDGFX.Line( F.buf, col,
                sx+1*SHORT(ENTIER(c)), sy+1*SHORT(ENTIER(s)),
                sx+2*SHORT(ENTIER(c)), sy+2*SHORT(ENTIER(s)), Display.replace);
        END;
    END;
END Draw;
```

Remarks: (1) The primitives may choose the draw color themselves but should follow the standard FDD coloring, with respect to selection in particular. FDDFrames defines the following default colors:

firstCol (*white*), secondCol (*grey*), selCol (*pink*) : INTEGER;

(2) Coordinates are available in the "x.c" array, which is a "Corarray" as defined in FDDSpace. (3) 3D clipping is enabled as default, whereas 2D clipping is not but can be toggeled using FDDGFX.ClippingOn/Off(). (4) To draw lines, ellipses or rectangles that are based on coordinates, the method

PROCEDURE ToScreen*(v : FDDSpace.Cor; VAR sx, sy : INTEGER);

of the passed frame parameter F computes screen coordinates. As an example, the light source draw handler maps the center of the coordinate system (which is x.c[0]) to the screen, thereby determining, where circle and rays must be painted, of which its graphical representation consists.

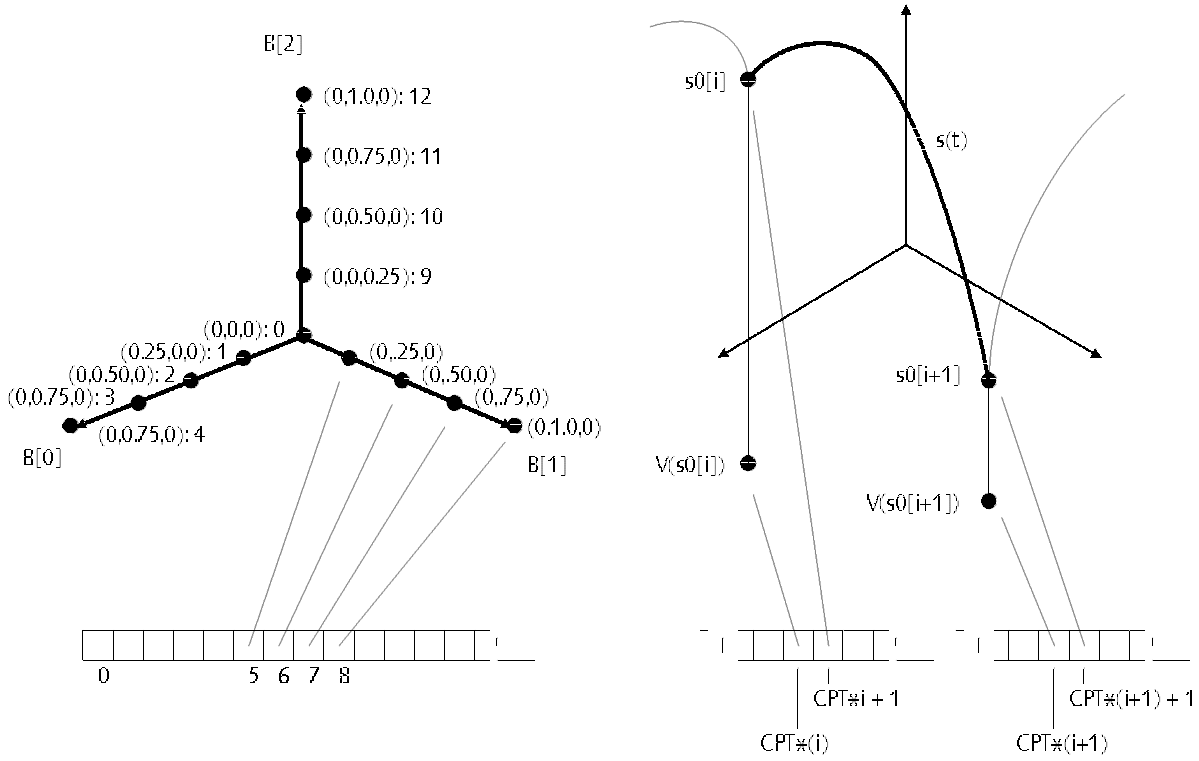


Figure B.7: Parabola chain coordinate array

B.4 Summary

Table A.3 gives an overview on the discussed modules and the important respective classes, methods and variables. The summary is not complete since many implementation–specific details that do not serve a fluent walk–through have been skipped. For further details, the reader is referred to the source text of FDD.

Module	Variables	Classes	Methods
FDDDS	–	List	Head/Tail, Empty, Insert/Delete, Find –
		Node	Key, Prev/Next
FDDSplines	u	–	N, C1Coeffs, C1, C1iterate, UEIK
FDDAT	–	AT	Define, Set, Invert, Neg, Rot, Scale, Add/Sub
		ATFrame	NewFrame
FDDSpace	insertion	Cor	DefineCor, Add/Sub, Dot, Cross, Apply, Vertical ProjectOrthogonal/Perpendicular, ExpressCor
		Corarray	CreateCorarray, ExpressCorarray
		Box	Define
		Extent	V0, V1, V0l, V1l, CorExtent, Bound
		CS	SetA/Parent/Obj/C, RI, Overlay, FirstChildRef, n, A, Parent, C, Express, Is, LocalExtent
		Clip3D	

Table B.5: Module summary (space)

Module	Variables	Classes	Methods
FDDTVV	–	TVV	SetTime, Time
		TimeMsg	
		TCTVV	SetKey/Delete, Find, Generate, SetTime
		State	Pred/Succ, PropagateUpdate, Restore, SetInterpolation, SetT, Tb
FDDDis	newDisplacement	Displacement	Get
		BasicDisplacements	
			Stack, Generate, NewSalto, NewTranslation, NewWheel
FDDTime	keyconstructor	ATState	Add, Restore, SetFixpoints, SetVelocities
		ATTime	Generate, Overlay, Set, SetATKey, RestoreAT, InterpolateAT
		CS	Spline
		TVSurfaceState	
		TVSurface	Generate, Restore
		UpdateMsg	
			AddPos
FDDScript	F, newObj, s	–	Generate

Table B.6: Module summary (time)

Module	Variables	Classes	Methods
FDDFrameBuffers	–	FrameBuffer	InitFrameBuffer MakePict
FDDFilms	–	Film	Height, Width, Play/Stop, Suspend/Reactivate, Add
		PlayTask	Suspend/Reactivate, RemoveThis, Remove
		PlayMsg	
		FrameAdjustment	
FDDGFX	f, xPPM, yPPM	Viewport	Clip, ClippingOn/Off, Left/Right/Lower/UpperBorder
		Line, PicFrame	
FDDInteraction	selection, cpar, first/second/selCol		
		Scene	Express
		CS	Draw, Intersect, Modifier, AddAT, SetPosition, InfluencedBy
		FDDFrame	Line3D, Clip3D, ToScreen, Intersect, Track, Draw, Select, Navigate, Modify, Create
		Reference	
		ConstructionParameters	
		Camera	
FDDContainers	–	FDDContainers	NXNY GetFilm, MakePict, NewDoc, Render, New, ResetCuts
FDDTextures	(x,y), (u,v,w), color	Texture	Chessboard(3D), Mandelbrot, Rainbow, Rubik, Waves, YingYang
FDDRender	–	Rendertask	Init
		Light	NewLight
FDDObj	–	BasicPrimitives	
		ModifierPars	
			DefaultScene

Table B.7: Module summary (interaction)

Appendix C

Overview on commercial systems

Not surprisingly, there are many players in the field of computer graphics and image synthesis has been one of the most progressive disciplines in the last years. [LO98] provides a brief summary of a couple of programs running on all common operating systems and offering both animation and rendering facilities as FDD does. Of course, they all feature most of FDD's little possibilities – and much more, since this kind of software is usually maintained by a big development team and all of the presented programs have at least experienced five revisions. Thus, a general comparison with respect to functionality does not make much sense.

What in fact distinguishes FDD from commercial rendering software is the hardware necessary and in particular, the required amount of memory. Where it is advisable to equip the system with at least 64MB of RAM even for small programs presented by [LO98], almost nothing is – in comparison – needed to run FDD. Of course, this circumstance is primarily caused by the very limited set of construction and animation facilities of FDD.

Anyway, there are some subtle facts to be discovered. By comparing programs presented by [LO98] with respect to some of FDD's properties, one might collect the findings of table C.1. The fact that FDD has significant properties that commercial software at a price of $2000 does not provide, documents the hard task of integrating additional fundamental functionality into large existing projects and shows that, with the implemented time–space positioning algorithm and the animation framework, a solid ground has been laid. But the real highlights of FDD are an elegant and transparent implementation and the solid foundation for further development it provides by the concept of TVV which makes it simple to add new time–variate sizes.

Facility	FDD	Imagine 3D (Impulse)	Shade III 1.30 (Expression Tools)	Real 3D (Realsoft)	Cinema 4D XL (Maxon)	Lightwave 3D (Newtek)
Fly along paths	+	+		+		+ +
Objects as view target	+			+		+ +
Show paths	+		+		+	+
Vertex animation	+					
Price ($)	–	600	1200	1200	1800	2000

Table C.1: Comparison of commercial sytems

Requirements for today's stand–of–the–art modelling, animation and rendering software are [LO98] metaballs, morphing, particle systems/physical simulation, environment maps, inverse kinematics, bump mapping, Boolean operations, artificial lens effects, bones and NURBS (non–uniform rational B–Splines [GR97]), procedural textures, shadow–maps, volume rendering and volumetric light in particular, of which FDD provides only the inherent bone concept (if forward kinematics such as displacements are exclusively used), since the primary focus of the project has been the development of animation facilities – the fourth dimension. Note that most of the uncovered topics to not concern the time–space model, apart from inverse kinematics and particle systems/physical simulation and are merely impressive render–specific special effects which have not been addressed with the project. The fact that a famous animation tool such as Softimage (Microsoft) does provide metaballs only in its "Extreme" version at $18000, might give a hint for the complexity and the financial intensity of the topic.

There are also systems at $0, such as "POV–Ray" (POV–Ray Team at www.povray.org, [CA96]), which provide extremely powerful rendering facilities – but based on textual scene description without any graphical user interface. For an overview on modeling and animation software below $250, see [HI98]. As a summary, [HI98] distills the fact that they all do not provide complex modelling tools such as NURBS or metaballs as optionally provided by excessively expensive professional tools.

Appendix D

User guide

FDD adopts to the MMI definitions [FIMA98] of the Oberon system. This appendix will thus explain only FDD–specific commands, whereas the official documentation may introduce the novice to the general rules. The full report gives a solid background about FDD's foundings. It is advisable to run FDD at a resolution of at least 800x600 pixel. Click here to open the **central command tool**. It shows on top a row of buttons of which each enters an **editor section**, whose usage will be explained by the construction of a small example scene. Being a little bit experienced with the Gadgets system, FDD's user interface may be customized.

D.1 Global section

D.1.1 Creating a scene

An FDD window is a Gadget and may thus be inserted in any document accepting Frames as contents. The most simple way to create a document containing a three dimensional FDD scene is to use the **New** command of the central command tool you have just called. It opens an unnamed default text document containing a scene as shown by figure D.1. Alternatively, you may wish to insert a new scene into an existing document – then place the caret wherever you want and use **Insert scene**.

FDD presents itself by a **central coordinate system** in the middle of the frame and a floor, being represented by a rectangular grid. Upon tracking the rectangular frame area, the title text dispairs and FDD is ready to accept your input. You may customize the number of grid quads per edge by setting the **Gridquads** constant found in the wireframe subsection of the global editor section. The grid spacing is one meter in both directions and so if you specify an even number of grid quads, the lines will cross in the coordinate system's center.

The three buttons at the frame bottom switch between the three applications **Editor**, **Director** and **Player**, whereas the first is a modelling tool, the second allows the composition of arbitrary camera cuts to a film and the last is a depository for rendered films. Note: rendered films are not stored by the FDD frame. They must be exported by using the **Get *** command of the **Film...** iconizer Additionally, you my use the **play button** or the **timebar** to either either see the defined dynamics in realtime or to set an arbitrary time for the depicted scene. For the following steps, switch to the editor and move the timebar to the very left.

FDD operates in two **modes** which can be distinguished by the **Navigate** button: if pressed, you can move the current view camera in space and if not, the data structure may be customized. That is: primitives will be inserted or can be modified. But first of all, it is necessary to become familiar with the navigation facility: press the Navigate button. By moving the mouse into the FDD frame and pressing mouse buttons (and combinations), their effects on camera position orientation may be explored. The assignments are as follows: the left button moves the camera parallel to the viewplane, the middle one implements camera dollying whereas the right turns it into an arbitrary direction. Pressing all of them simultaneously rolls the camera around the view direction.

By releasing either the Navigate button or by selecting a construction primitive (press the corresponding picture in the central command tool), the **construction mode** is entered. Press the left mouse button to start the creation of the selected primitive. Construction processes are terminated by pressing the right mouse button, whereas by interclicking with the middle one cancels. The **deletion** of objects happens Oberon–fashioned: selection with the right mouse button and interclicking with the left one. Solid objects such as spheres or cylinder may be selected by a right mouse click on their wireframe representation, whereas the sensitive point of transparent objects such as cameras is the center of their coordinate system. The middle mouse button allows object modification when a **control point** (a small cyan cross) is tracked. You may experiment with the various control points on a primitive coordinate system. Table D.1 summarizes the default editor mouse commands. From FDD's extensibility, this may of course differ for newly programmed primitives.

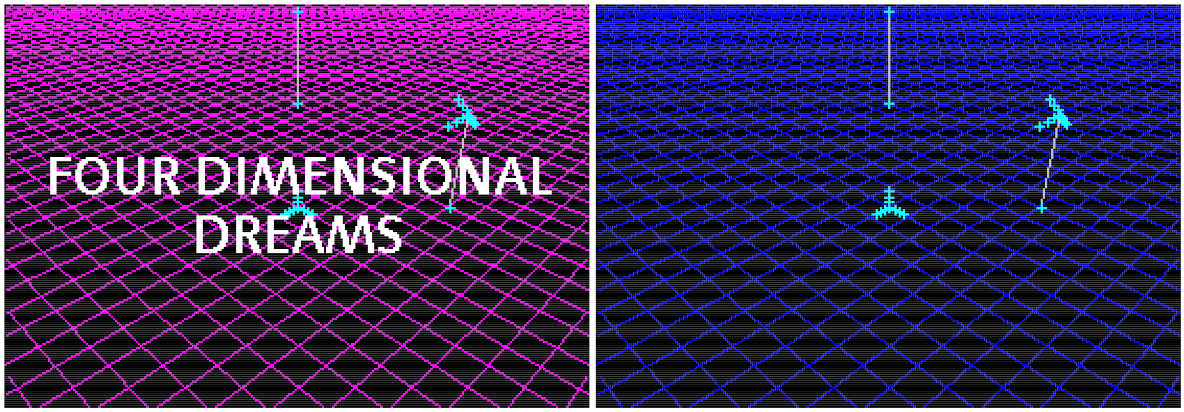


Figure D.1: Opening an FDD session

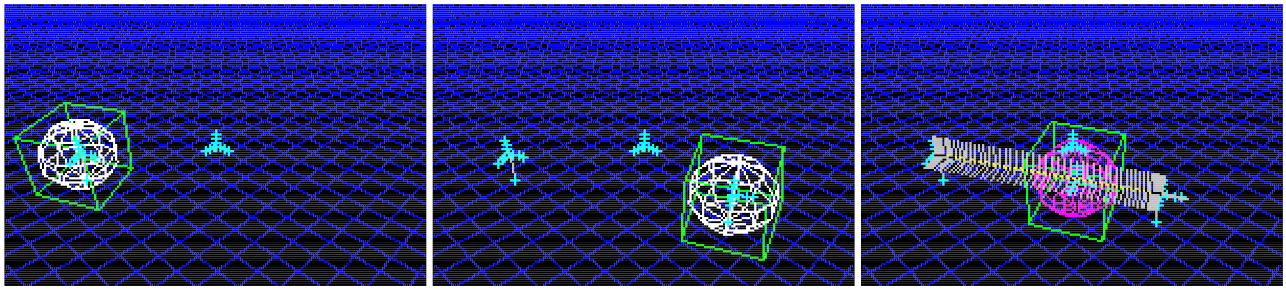
To follow the next steps, you must pull the timebar to the very left – which corresponds to the film time 0.0 – and create a scene consisting of a single sphere (figure D2.i). The rightmost position corresponds to the total scene length, which can be customized by marking an FDD frame and calling Columbus.

Pull the timebar somewhat to the right and move the sphere by using one of the outermost control points of its coordinate system (CS): the first CS will remain at its place (representing the first position at time 0.0), while a newly created (representing the position at the new time) one will follow the movement. Both CS represent now various positions of the sphere in time and are thus referred as keys. The scene should look like figure D2.ii. By moving the timebar, the nature of the thereby defined dynamics can be observed: the sphere moves from its original place to the new position. The space path can be visualized by selecting the primitive, marking the FDD frame and using the **trace** * command (figure D.2.iii). Note that it requires the whole time–space model to be evaluated and may thus take some time. So to go on, switch of the trace. The view can additionally be customized by the **Floor**, **Contours**, **Keys**, **CP**, **Bounds** and the **CS** commands or the **Toggle** * iconizer.

The dynamics defined so far are all absolute. That is: they directly specify an object's position in space. To overlay additional dynamics, displacements (see track section) may be used. If you put a camera into the scene, you may switch it on by selecting it, marking the FDD frame and using the **Set cam** * command. Now place a second sphere on top of the first one and select it. By using the **Focus** * command, the grid moves its origin to it and thereby allows to easily place new objects on the plane it visualizes. **Unfocus** * reverts the process. By pressing **Insert key** *, you may insert a new key when there is none at the current time. For the swich **Coordinate bar** may be used to toggle a coordinate status bar in FDD frames.

Mode	Left	Middle	Right	Action
Navigation	x			Move camera
		x		Dolly camera
			x	Turn camera
Construction	x			Start construction process (right terminates)
		x		Modify objects
			x	Select and delete objects

Table D.1: Editor mouse commands



i) ii) iii)

Figure D.2: Defining dynamics

D.1.2 Defining the cut list

Assigning a camera to an FDD view does not correspond to assign a film a set of subsequently activated cameras. This task is done by the **Director**, which is actiated by pressing the **D** button on an FDD frame. The scene vanishes and is being replaced by a set of frames of which each one shows the scene from a scene contained in it. A red recording mark shows the camera which is active at the current time.

In order to customize the cut list, you can specify a new camera by moving the timebar to the corresponding time and activating it by a click with the left mouse button into the frame depicting its view. By moving the timebar, the film can be inspected by following the recording mark. The cut list can be reset by clicking with the right and interclicking with the left mouse button.

D.1.3 Rendering

A rendering process essentially requires light. Thus, **light sources** must be placed into the scene in order to get proper results. With having defined the scene, FDD needs the time interval and the number of frames to be rendered. This can be specified by adapting **Time interval** and **Frames** in the **Render setup**. The **Cyclic** flag must be activated if a seamless transition from the last frame of the film to its beginning must be created. This corresponds to the sampling

$$t_i := t_a + i\,(t_b - t_a)\,(\text{frames})^{-1}, \qquad (i \in \{0,\ldots,\text{frames}-1\}) \qquad \text{(D.1)}$$

with t_b/t_a being the film's start/end time, whereas the frames of a a non–cylic film is of course sampled as

$$t_i := t_a + i\,(t_b - t_a)\,(\text{frames}-1)^{-1}. \qquad (i \in \{0,\ldots,\text{frames}-1\}) \qquad \text{(D.2)}$$

To generate previews of a defined scene, the renderer may be run to generate wireframe representations (**Color model**). By selecting the **Phong** shading model, rendering time will explode but generate smoothly shaded surfaces. The renderer is started by pressing the **R!** button.

If the renderer has finished its work (check the log), a new film is inserted into the player film depository, which can be called by pressing the **P** button. To play, select an arbitrary film with the right mouse button any press the play button. The FDD frame will adapt to the film's size and thereafter display the rendererd sequence of pictures. Films can be deleted from the depository in the usual manner and can be inserted into documents by consume operations.

D.1.4 CS tree

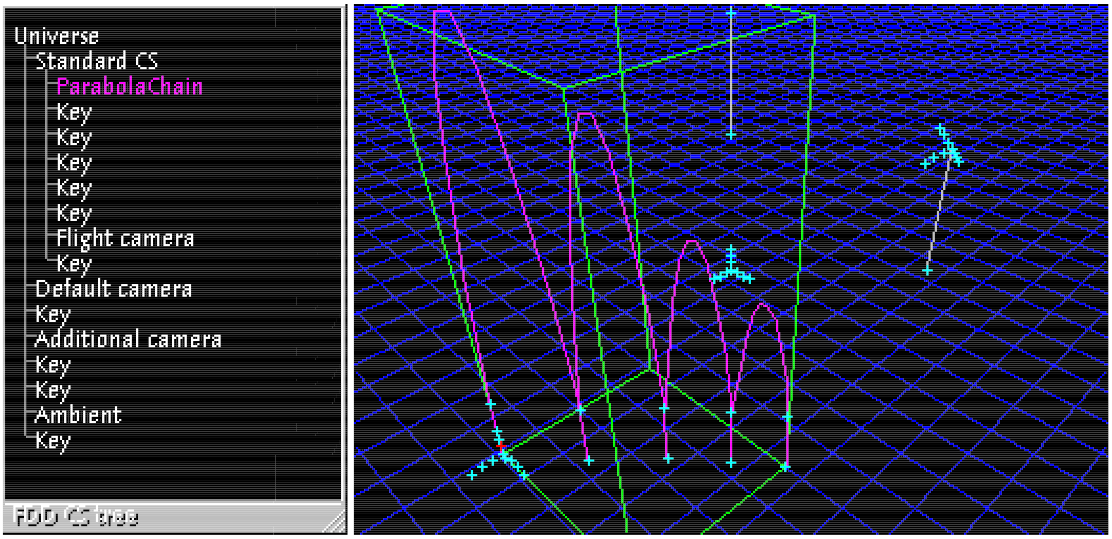


Figure D.3: FDD CS hierarchy

To read a scene into the hierarchy view **FDD CS tree**, mark an FDD frame and then execute the presented **Get** * command. Figure D.3 depicts an example scene with its **CS hierarchy**. In the CS tree hierarchy, objects may be selected and deleted as with an FDD frame and the one–to–one correspondence between the two model views should be obvious. Note that only object can be deleted that do not position other objects. So to delete a key k, first of all the objects that is controls must be deleted.

The **Ambient** light object models the ambient term of Phong's reflectivity model and may only be selected in the CS hierarchy, since it is not visible in the modelling view. It can be seen that even in static scenes, each object is controlled by a key which determines its initial position at time 0.0.

The **Default camera** is activated each time a new session is opened. You may switch to other camera views by selecting them here any by using the Set cam * command in the centrol command tool. The **Flight camera** is a dynamic camera based on a set of keys. As default, new objects are inserted into the **Standard CS**. Recall that this may be changed by focusing another object.

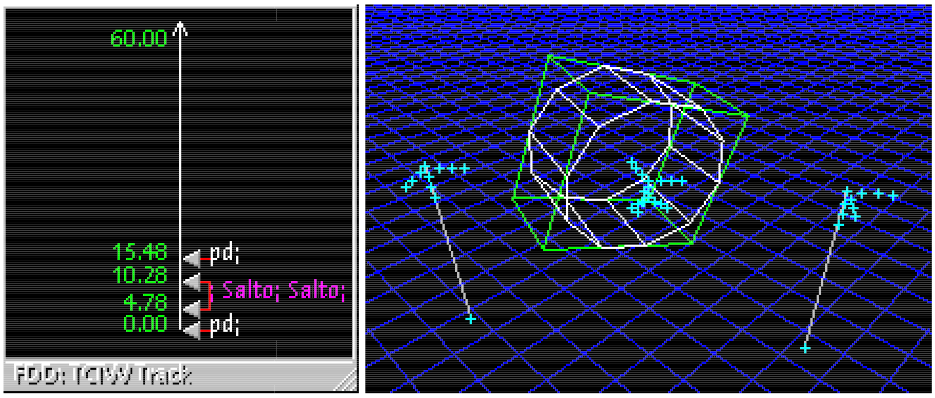


Figure D.4 : Track editor

D.1.5 Track editing

An object's **track** is the total list of all dynamics. The **Get** command shows all time–variate sizes of the selected FDD object. Use the middle mouse key to select an entry of the list whose track is then being displayed by the track view (see figure D.4 for an example). The term **AT** designates the object's affine transformation (that is: its spacial position), whereas **Surface** shows the track of the time–variate surface.

Each track entry can be **select**ed (right mouse button), **move**d (left) or **delete**d (select with left interclick). New track entries are **create**d by clicking with the left mouse button at some time where there is none yet. Track entries concerning an object's affine transformation can be edited in the AT section (accordingly, surface track entried are customized in the surface section). Their graphical representation consists of a list of keywords of which each corresponds to a positioning action. Figure D.4 shows the track of a cylinder with two positioning keys and an additional displacement (see AT section) between them: Saltos.

At the first position, absolute positionings are listed, whereby a **p** signals a positioning object such as a key, a **d** denotes an object fixing orientation according to a spline derivative and **t** indicates that a target object specifying its orientation has been defined.

A track entry's displacement is given by the concatenation of a list of affine transformations. A simple example of such a list entry is the so–called Wheel: its affine transformation is a rotation about an arbitrary axis. The command **Add↑** in the displacement iconizer (**Dis**) interprets the first passed parameter as the displacement generator, which by itself may scan additional parameters. As an example, the generator "FDDDisNewWheel" additionally needs axis and rotation frequency to be specified. The call

FDDTime.AddDis FDDDis.NewWheel 0 0 1 2 (D.3)

thus adds a rotation around (0,0,1) at a frequency of two times per second to the selected track entry. By repetitively adding displacement, compound dynamics may be defined. Note that positioning is done according to the order presented by the track view: from the non–commutativity of affine transformations, overlaid displacements are inserted at the beginning of the displacement list.

D.1.6 Surface editing

If a surface track entry has been selected, the surface iconizer allows its modification. Again, the **Get** command serves to load a surface state. On the left, the surface type may be selected as either being colored or texture–mapped. Either procedural or bitmap–based textures may be chosen, where for the former case, the textures shown by table D.2 have been defined. To select a bitmap texture, the corresponding textfield must be filled by a name shown by the file list. An object's color can be specified by customizing **diffuse** and **specular** part and its **emission**, all by defining RGB components. The fourth specular slider defined the specular reflectivity exponent [GR97].

i) ii) iii) iv) v) vi)

i) FDDTextures.Rubik	iii) FDDTextures.Rainbow	v) FDDTextures.Chessboard
ii) FDDTextures.Mandelbrot	iv) FDDTTextures.Waves	vi) FDDTextures.YingYang

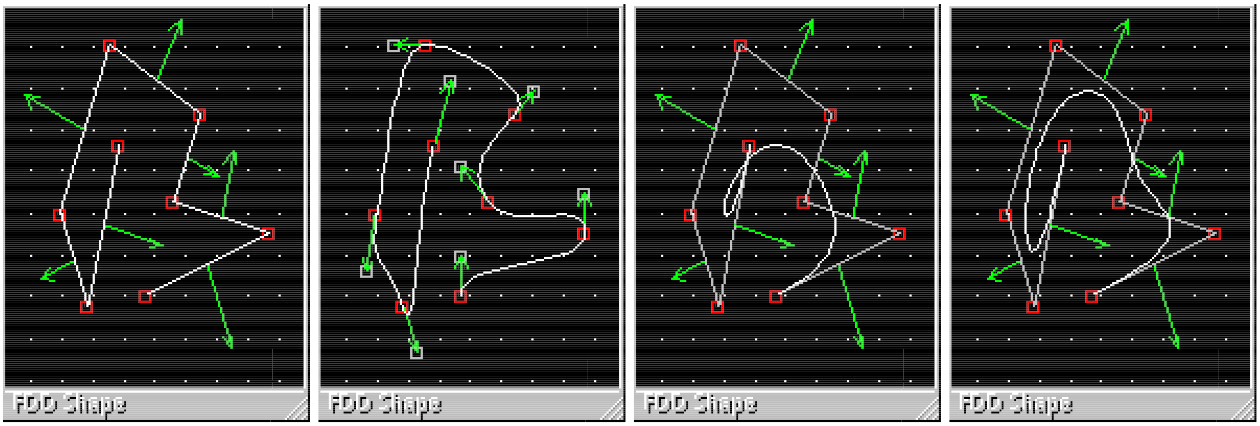
Table D.2: Textures

D.3 Shape section

The shape section allows the definition of two dimensional shapes. A new view is opened with the **New view!** command. The following **operation principle** is essential: any operation works in **two steps**. By clicking with the left mouse button into the view, one advances from one step (or state) to the next and gives a so–called **reference** to the editor.

Red squares mark the **vertices** of the shape. They are connected by white segments. In the middle of each such segment, a perpendicular green arrow is shown: the corresponding normal. The normals give the shape a **direction**: they point to the *right*, when walking along the path in direction of the parametrization.

The shape class can be chosen in the **Class** subsection. A Hermite interpolation additionally allows the definition of a spline derivative at vertices: each spline vertex has a **direction arrow** attached to it, which specifies the direction of the curve in that actual vertex. By giving the editor pairs of references, all available operations summarized by table D.3 can be induced. Each time an operation has been executed, the editor is set back to the idle state and the process starts again.



i) Polygon ii) Hermite interpolation iii) Bézier curve iv) B–Spline

Figure D.5: Shape classes

First reference	Second reference	Operation
view background	view background	creation of a shape
view background	vertex	insertion of a new after the referenced vertex
view background	direction arrow	insertion of a new vertex after the reference
view background	shape	none
vertex	view background	move the vertex accordingly
vertex0	vertex1	vertex0 and vertex1 are subsequent: delete vertex0
		vertex0 and vertex1 are *not* subsequent: delete vertices between the two references
vertex	direction arrow	delete vertices between the two references
vertex	shape	move the shape accordingly
shape	any	move the shape accordingly
direction arrow	any	adapt the direction arrow accordingly

Table D.3: Shape editor commands

Appendix E

File index

Directory	File	Contents
\	README.TXT	CD information, shortened file index
Demonstration	FDD.D.Panel	Salto methamorphose, Eyeball
	FDD.D.Space.Mod	Coordinate transformation example
Interface	FDD.I.Dev.Tool	Developer environment
	FDD.I.Shape.Tool	Shape editor
	FDD.I.Tool	Central command tool
Data	FDD.Lib	program variables, logo animation
Report	FDD.R.Abstract.Text	Hyperlink report: Abstract
	FDD.R.Appendix.Conclude.Text	Hyperlink report: Concluding remarks
	FDD.R.Appendix.Files.Text	Hyperlink report: File index
	FDD.R.Appendix.ModRef.Text	Hyperlink report: Module reference
	FDD.R.Appendix.Overview.Text	Hyperlink report: Overview on commercial systems
	FDD.R.Appendix.Source.Text	Hyperlink report: Language use
	FDD.R.Appendix.UserGuide.Text	Hyperlink report: User guide
	FDD.R.Bib.Text	Hyperlink report: Bibliography
	FDD.R.Interaction.Text	Hyperlink report: Chapter interaction
	FDD.R.LOF.Text	Hyperlink report: List of figures
	FDD.R.LOT.Text	Hyperlink report: List of tables
	FDD.R.Preface.Text	Hyperlink report: Preface
	FDD.R.Space.Text	Hyperlink report: Chapter space
	FDD.R.TOC.Text	Hyperlink report: Table of contents
	FDD.R.Time.Text	Hyperlink report: Chapter time
	FDD.R.Title.Text	Hyperlink report: title page
	FDD.R.TeX	"TeX" report file list
	FDD.R.Text	Report (single document)
	FDD.R.ps	Report Postscript dump
	FDD.R.ps.zip	Report Postscript dump (zipped)

Source	FDD.S.AT.Mod	Module FDDAT
	FDD.S.BT.Mod	Module FDDBT
	FDD.S.Complex.Mod	Module FDDComples
	FDD.S.Containers.Mod	Module FDDContainers
	FDD.S.DS.Mod	Module FDDDis
	FDD.S.Dis.Mod	Module FDDFilms
	FDD.S.Films.Mod	Module FDDFrameBuffers
	FDD.S.FrameBuffers.Mod	Module FDDGFX
	FDD.S.GFX.Mod	Module FDDInteraction
	FDD.S.Interaction.Mod	Module FDDListview
	FDD.S.Listview.Mod	Module FDDObj
	FDD.S.Obj.Mod	Module FDDRender
	FDD.S.Render.Mod	Module FDDRoots
	FDD.S.Roots.Mod	Module FDDScript
	FDD.S.Script.Mod	Module FDDSelect
	FDD.S.Select.Mod	Module FDDShapeFrames
	FDD.S.ShapeFrames.Mod	Module FDDShapes
	FDD.S.Shapes.Mod	Module FDDSpace
	FDD.S.Space.Mod	Module FDDSplines
	FDD.S.Splines.Mod	Module FDDSplines
	FDD.S.TVV.Mod	Module FDDTVV
	FDD.S.Targa.Mod	Module FDDTextures
	FDD.S.Textures.Mod	Module FDDTextures
	FDD.S.Time.Mod	Module FDDTime
Executables	*.Sym, *.Obj	Symbol and object files
Various	FrameGadgets.Mod	Extended language frame Gadget skeleton
	ModelGadgets.Mod	Extended language model Gadget skeleton
	Kill.Mod	Task killer
	TeX.Mod	Concatenate text documents
	TeXTemplate.Text	"TeX" document template
	Greek14.Pr3.Fnt	Fixed Greek font (vertical height := 2B)
	Greek12.Pr3.Fnt	Fixed Greek font (vertical height := 25)
	Greek10.Pr3.Fnt	Fixed Greek font (vertical height := 25)
System	...	Intel Native Oberon (release 2.3.0)
Archive	FDD.Arc	Distribution version (no source text)
	FDD.D.Arc	Demonstration files

Appendix F

Concluding remarks

FDD has significantly determined the last four months of my life and gave me the possibility to gain further programming experience in the field of computer graphics – of time–variant scenes in particular. It is the concrete expression of a long childhood's dream and I am proud of having pursued this aim. What this work has led to can be seen from this report.

I first of all would like to thank Prof. Dr. Jürg Gutknecht for giving me the possibility to implement the four dimensional dreams. His group maintains one of the most modern and effective standalone operating systems that may really be people's servant – and not the other way around, as one might oftenly have the impression about today's technology. I want to thank him and Pieter Muller for this wonderful system called Intel Native Oberon and also dedicate FDD to all the people, on whose work or personal support my thesis relies. It has been a great honour to develop an application in such an excellent environment. The system's efficiency allowed the development to be completely carried out on a 60MHz Pentium processor. Additionally, my personal assistant Erich Oswald deserves nothing but respect for being so patiently with me. I am probably not the easiest person to work with. The spirit at the Institute for Computer systems and the spirit of Oberon in particular influenced the work especially in the sense of Einstein's famous quote about simplicity.

With this probable last chapter of my studies here in Switzerland I also want to thank the ETH for giving me so much and for being such a motivating companion. It has been challenge, motivation and satisfaction to learn from personal idols such as Niklaus Wirth, Christian Blatter or James Massey. A very special thank goes to the department secretary Hanni Hilgarth for helping me to get through administrative trouble and paperwork, which definitively is *none* of my favourites. Markus Gross has given me the confidence that computer graphics is a challenging scientific discipline, that may rule a whole life and I will always remember his beautiful and amazing lectures in the last hours before weekend in the late Friday's afternoon. Their intensive and fascinating spirit have carried me through several terms and have been of major influence: that's what I have in fact been studying for.

At the end, thanks go to my dear friends Oskar Muheim and Marco Baldelli for helping me through difficult times in my life and for listening to the bunch of problems, that I sometimes might have emptied over their heads during the last months. And I will never forget, what my parents all have done for me. But the biggest help of course has always been my true love: I love you – Ines – and I hope, that you will always be by my side.

March 1999,

Christoph Kleiner
Rubacherweg 49

6472 Erstfeld
Switzerland

041/880'18'68

Bibliography

[DU25] A. Dürer:
"**Unterweisung der Messung mit dem Zirkel und Richtscheit**",
library Swiss Federal Institute of Technology Zürich (1525)

[GR97] M. Gross:
"**Graphische Datenverarbeitung I/II**",
lecture script ETH Zürich (1997)

[BA91] C. A. Balafoutis:
"**Dynamic analysis of robot manipulators: a Cartesian tensor approach**",
Kluwer academic Publishers (1991)

[RGBC96] C. Rose, B. Guenther, B. Bodenheimer, M. Cohen:
"**Efficient generation of motion transitions using spacetime constraints**",
proceedings of siggraph (1996)

[FIMA98] A. Fischer and H. Marais:
"**The Oberon Companion**",
vdf Hochschulverlag Zürich (1998)

[SI98] K. Simon:
"**Algorithmische Graphentheorie**",
lecture script ETH Zürich (1998)

[RE198] M. Reiser:
"**Parallele und verteilte Simulation**",
lecture script ETH Zürich (1998)

[FOKE98] M. Fowler with K. Scott:
"**UML Distilled – applying the standard object modeling languague**",
Addison Wesley Reading (1998)

[PO94] I. Pohl:
"**C++ for Pascal programmers**",
Benjamin/Cummings Publishing Company (1994)

[LO98] J. Loviscach:
"**Visionen in 3D: Software für Animation und Rendering**",
c't Magazin für Computertechnik 2/98, pp. 120, Heise Verlag (1998)

[CA96] C. Caroli:
"**Strahlenforscher: 3D gratis in Perfektion**",
c't Magazin für Computertechnik 9/96, pp. 300, Heise Verlag (1996)

[HI98] G. Himmelein:
"**Virtuelle Modellbausätze**",
c't Magazin für Computertechnik 16/98, pp. 92, Heise Verlag (1998)

[MA98] J. L. Massey:
"**Applied digital information theory**",
lecture script ETH Zürich (1998)

[WA92] A. and M. Watt:
"**Advanced animation and rendering techniques**",
Addison–Wesley (1992)

[UP90] S. Upstill:
"**The RenderMan Companion**",
Addison–Wesley (1990)

[NISTO94] K. Nipp and D. Stoffer:
"**Lineare Algebra**",
Verlag der Fachvereine (1994)

[LWP80] J. Y. S. Luh, M. W. Walker and R.P. Paul:
"**On–Line computational scheme for mechanical manipulators**",
ASME J. Dyn. Syst. Meas. And Contr. Vol 102, pp. 69–79 (1980)