UGA: A Unified Graphics Architecture

Philip M. Hubbard, Matthias M. Wloka, and
Robert C. Zeleznik

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-91-30
June 1991

UGA: A Unified Graphics Architecture[1]

Philip M. Hubbard   Matthias M. Wloka   Robert C. Zeleznik

November 11, 1991

[1] This work was sponsored in part by NSF, DARPA, IBM, Sun Microsystems and NCR. Hardware contributions by Hewlett Packard and Digital Equipment Corporation and software contributions by Bitstream Inc., Pixar and Visual Edge Software Ltd. are also gratefully acknowledged.

Abstract

UGA is a system that supports research into the integration of modeling and animation paradigms. The heart of the system is a database of objects whose attributes can change over time. The database allows a variety of techniques to be used to modify and display the attributes of objects. Emphasis has been placed on maintaining the flexibility of the system without sacrificing the speed necessary for interactive applications.

This document focuses on the implementation of the UGA database. The important database algorithms are described, and the software packages that implement these algorithms are summarized.

Contents

Chapter 1

Introduction

1.1 About this Document

This document describes the design and implementation of the UGA graphics and animation system. The focus is on the software packages that form the heart of UGA. These packages are described in detail, i.e., specifics of data structures and routines are given.

Some of the software packages of UGA have been described in other documents [HUAN91a, WLOK90]. The main difference between those other documents and this document is in scope. While those other documents concentrate on particular details of individual packages, this document attempts to give an overall picture of how the parts of the UGA system fit together.

1.2 Audience

This document was written with the following audience in mind:

- students who need to learn about UGA to maintain and extend the existing code;
- people who are interested in building a system similar to UGA from scratch.

This document specifically will not be:

- a high-level description of UGA (this description has already been given by Zeleznik *et al.* [ZELE91]);
- a collection of UNIX `man` pages that describe every routine in the system;
- a programmers "how-to" guide that gives step-by-step instructions for exploiting the extensibility built into the system (a few such guides are mentioned in the bibliography);
- a description of how to produce animations with the UGA system.

1.3 Prerequisites

Readers of this document are expected to be familiar with the general capabilities of the UGA system. The following documents will provide the necessary background knowledge:

- "An Object-Oriented Framework for the Integration of Interactive Animation Techniques" by Robert C. Zeleznik et al [ZELE91] presents a high-level description of the system's goals and architecture.
- "Description of the FLESH Language" by Cindy Grimm and Mitch Henrion [GRIM91] describes the scripting language that instructs UGA to produce an animation. An understanding of how a user specifies an animation is essential in order to understand how the UGA system produces that animation.

This document was written so that someone who has read the aforementioned documents should be able to read it from start to finish without difficulty.

1.4 Document Structure

The document is organized as follows:

- Chapter 2 gives a broad overview of UGA. It should give the reader a glimpse at the capabilities of the system. It also introduces all the packages that will be described in detail in subsequent chapters.
- Chapter 3 begins the detailed description of the main UGA software packages. This chapter starts with low-level packages and works up to higher-level packages.
- Chapter 4 completes the description of the main packages. This chapter fills in details that are important but not essential for an understanding of the main packages. These details are presented in this chapter rather than Chapter 3 to keep the earlier chapter as easy to understand as possible.
- Chapters 5 and 6 discuss some software packages that use the main UGA packages to achieve particular effects. The concepts described in these chapters will make sense after the main UGA packages have been presented.

The sections of this document contain many forward and backward references to other sections. We apologize to readers who find it burdensome to follow these references, but we feel the references are essential for a description of UGA. The UGA system is complex enough that there are many subtle relationships between the main software packages. A description of any one package thus requires at least some knowledge of many of the other packages.

1.5 Conventions

Several typesetting conventions are used throughout this document. New terms are typeset in *italics* when first mentioned. Software package names are typeset as PACKAGE. Function names, data structure names and FLESH language fragments are typeset as `item_name`.

A simple naming convention is used for all function and data structure names [STRA91]. The name of the function or data structure is always preceded by the capitalized name of the package that owns it. If the function or data structure is private to that package (i.e., declared `static`) then there is an underscore "_" between the package name and the function/structure name. For example, a public function named "public" that is part of the package named "package" will be named `PACKAGEpublic()`, and a private function named "private" belonging to the same package will be named `PACKAGE_private()`.

As mentioned in the previous section, there are many forward and backward references in the text. When referencing sections we give the section number and the section name, as well as the page on which the section begins. Even when referring to very specific concepts, we use the same referencing convention. Consequently, the referenced concept might not actually be on the page mentioned in the reference; it will, however, be in the section that starts on that page.

1.6 Nobody is Perfect

We would like to know about any mistakes or confusing passages in this document. Please send any comments by e-mail to `mmw@cs.brown.edu`. Please set the subject line to "UGA Arch Doc Bug."

1.7 Acknowledgements

This document would not have been possible without the help of Daniel G. Aliaga, Brook Conner, Pete Granger, Nathan Huang, Henry Kaufman, Brian Knep and Chris Nuzum.

Chapter 2

Overview

This chapter is an overview of how UGA works. The purpose is to introduce the software packages of UGA and the relationship between these packages, not to describe them in complete detail. Chapter 3 (The Kernel of the Database, page 23) will describe the packages in more depth.

2.1 Basic Concepts

The UGA system supports the construction and use of *time-parameterized models*. These models are groups of entities whose structure and properties are allowed to change over time, usually for the purpose of creating a visual effect. The basic design of the UGA system has already been described by Zeleznik *et al.* [ZELE91]; this section will summarize that description.

The entities in time-parameterized models are *objects*. There are several broad categories of objects. *Geometric objects* represent entities with a physical appearance, e.g., a ball, a building, a human figure. *Controllers* are entities that affect how other objects change over time, e.g., a controller that acts like a muscle and controls the movement of a set of geometric objects that represent an arm. *Interaction objects* allow their properties to be modified directly by a user, e.g., an object that acts like a finite-state machine for detecting mouse events and triggering responses. Note that an object can belong to all of the above categories simultaneously; the categories are not necessarily disjoint.

The characteristics of an object are defined by the *messages* that it receives. Messages allow objects to be translated, rotated, colored, deformed, etc. For historical reasons, messages are also known as *change operators* (*chops*). Since the term "chop" occurs throughout the source code for UGA, that term will be used instead of "message" throughout the remainder of this document.

A chop is a function of time[1]. The particular nature of the function is determined by a series of time-value pairs called *control points* (*ctpts*). We use the term *value-time* to refer to the time associated with a value in a ctpt. The usefulness of this term will become apparent in Section 3.5 (The IMP Package, page 38) when interpolation between ctpts is described.

Chops are intentionally abstract. They direct *what* an object should do, not *how* it should do it. How a chop affects an object is determined by the object itself, based on the *class* to which the object currently belongs. The fact that an object can change classes arbitrarily is an important feature of UGA. A chop can cause an object to change classes, so the semantics of an object's chops may be determined by other chops associated with that object.

A time-parameterized model can be represented completely by an ordinary text file. This file contains statements in the FLESH scripting language [GRIM91]. These statements name the objects in the model, and describe the chops associated with each of those objects.

2.1.1 The Database

The UGA system is designed to assist application programs in the construction and display of time-parameterized models. An application program tells UGA how a model changes over time, and asks UGA to display the model or return details of its status at a particular time. Thus, UGA functions as a database management system that is tailored for graphics and animation.

We use the term "database" for both the UGA database management system and the data it stores. This usage makes sense because we use an object-oriented design, in which the main components combine data and the operations that can be performed on that data.

[1]Whenever "time" is mentioned in connection with a chop, it means the internal time scale associated with a model (as opposed to "real time" or "clock time")

We use the term *client* for any agent that makes use of the UGA database. Clients may be application programs, and they may also be entities in the database itself. An example of the latter is a controller object. In order to determine the properties of other objects and make changes to them, a controller accesses the database as if it were an external application program. It is often possible to achieve a specific goal with either of the preceding types of clients. For example, a renderer could be an application program that uses the database to provide it with the description of the scene being rendered. It would query the database about the scene and write out a bitmap file when done (the traditional rendering process). Alternatively, a renderer could be a controller that has as one of its properties the image that it rendered. A client could display the image by querying it from the controller.

Clients access the database by calling a set of routines, currently written in the C programming language. The database interface (*DBI*) provided by these routines is layered to allow different degrees of control during database access. The goal is to provide simplicity for clients that access the database in standard ways while still allowing flexibility for clients with specialized needs.

The database utilizes several efficiency mechanisms to speed the creation and use of time-parameterized models. These mechanisms are hidden from the clients as much as possible. An example of an efficiency mechanism that has been implemented is caching; an example of a mechanism that is planned is the distribution of database computations across a network of computers.

2.1.2 The Kernel

A traditional graphics package is a black box that performs graphical services. The only access it provides to its services is through a set of interface functions. When a programmer wants to use the graphics package, she writes an external application program that calls the interface functions of the package.

This model does not strictly apply to UGA. An application may be implemented not only by writing an external program, but also by adding a new controller object that is internal to UGA, as described in the previous section. It is therefore not always meaningful to talk about some functionality being "external" or "internal" to UGA. It is sometimes useful, however, to talk about the *kernel* of the database. This term is used to refer to the software modules that directly support the database operations. A module that supports a particular operation on a particular type of object is considered to be outside the kernel of the database.

The remainder of this chapter and all of Chapters 3 (The Kernel of the Database, page 23) and 4 (More Details of the Kernel, page 75) will describe the kernel of the database. Chapters 5 (Examples of Interesting Object Classes, page 89) and 6 (Beyond the Kernel of the Database, page 97) will describe the parts of the database that are important but which are not considered central to the kernel.

2.2 Database Overview

This section sketches how clients interact with the database. The purpose of this section is to provide a glimpse at the operation of the database and how the functionality is distributed over the different software packages. The high-level discussion in this section should prepare the reader to understand the more detailed discussion of individual database packages in the next chapter. The packages that will be mentioned in this section are shown in Figure 2.1 (page 13), grouped according to their general role in the UGA system.

As mentioned in Section 2.1 (Basic Concepts, page 9), time-parameterized models are constructed from objects and chops. The main functions provided by the database are thus:

- adding an object to the time-parameterized model in the database;
- adding a chop to an object in the database;
- determining an attribute of an object (*inquiring data* from an object);
- changing a chop that is already in the database; and
- performing a "high-level" editing operation that can change several chops at once.

Before discussing how the database supports these functions through its DBI, consider a concrete example of how a client might use these functions.

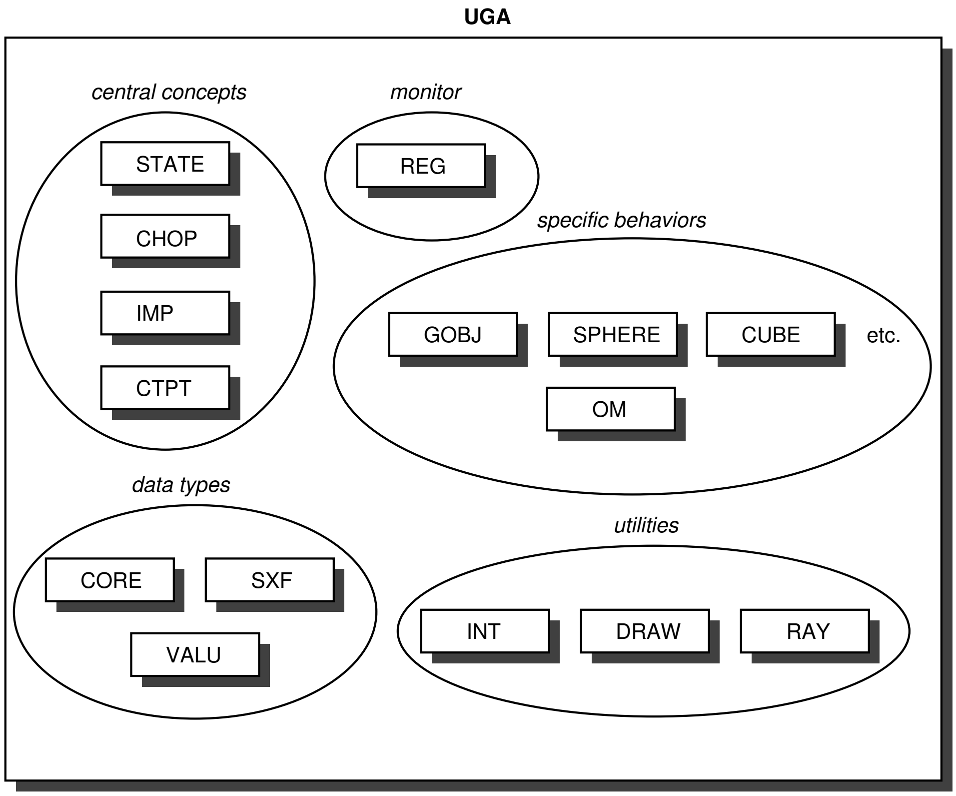

Figure 2.1: The software packages of UGA, grouped according to their general role in the system.

Assume the client is an interactive animation editor. A session involving a person using the client might progress as follows.

The user tells the client to read in a FLESH script, such as the script depicted in Figure 2.2 (page 14). The client reads the script from a text file, and needs to enter into the database the animated model described in the script. The client uses DBI functions to add objects and chops to the database.

To visualize the model described by the script, the user asks the client to animate the model. The client must thus render a series of frames. At each frame time the client uses DBI functions to get the visual attributes of

```
zelda   : rep       0  = torus
          color     0  = [ 0,  1,  0]  /* green */
          translate 0  = [-1, -2, -3]
                    10 = [ 1,  2,  5]
          bend      5  = [ /* parameters to specify
                                no bending */ ]
                    10 = [ /* parameters to specify
                                a lot of bending */ ]
;
```

Figure 2.2: A sample FLESH script.

each object in the model. In our example of Figure 2.2 (page 14), the client would inquire the translated polyhedral boundary representation (*b-rep*) of the described object `zelda`. The client can then render the frames with, say, a hardware z-buffer algorithm

Not satisfied with the model read from the script, the user decides to modify it. She tells the client to alter some of the chops, e.g., to change the color of `zelda` to brown. The client, in response, uses DBI functions to change the color-setting chop.

To make a more elaborate change to the model, the client might have to change multiple chops in subtle ways. It would be preferable if the client did not have to worry about the details of these chops. A client should be able to specify an editing operation in abstract, high-level terms and have the database map that operation into changes to specific chops. High-level editing operations are not yet fully incorporated into UGA, so their details will not be presented in this document. The only place they will be mentioned is in Section 3.8 (The GOBJ Package and Object Classes in General, page 66), where the rudimentary mechanisms that support them will be discussed.

2.2.1 Adding an Object

The most important aspect of an object is its ability to be changed by chops. The list of chops that are changing an object is called the object's

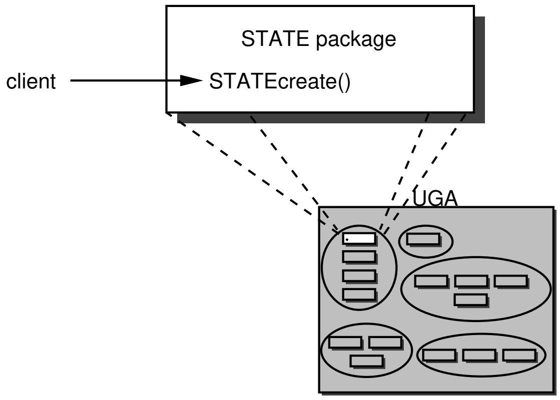

Figure 2.3: Adding an object.

state. The STATE package manages the state of each object, maintaining the data structures that represent a state and coordinating the operations on the chops in a state. Operations on the individual chops are handled by the CHOP package.

To add an object to a model in the database, a client calls the STATE package routine `STATEnew()`. The name of the new object is given as a parameter. The routine makes an association between this name and a new `STATEstate` data structure. The `STATEstate` is then returned to the client because all subsequent operations on the object will use this data structure. The operation of adding an object is summarized graphically in Figure 2.3 (page 15). Details of how the STATE package operates on a `STATEstate` are given in Section 3.7 (The STATE Package, page 52).

When an object is created, it belongs to the "gobj" ("*g*eneral *obj*ect") class. The GOBJ package contains the routines associated with this class of object, as described in Section 3.8 (The GOBJ Package and Object Classes in General, page 66).

2.2.2 Adding a Chop

Now an object has been created. This object is a generic object that does not change over time, since no chops have been applied to it yet.

Chops are needed to change the appearance of this object. An important chop is the `rep` chop, which changes the representation of an object. In Figure 2.2 (page 14), a `rep` chop makes the object `zelda` a member of the torus class at time 0. The object will have all the characteristics of a torus from time 0 on.

The CHOP package defines a `CHOPchop` data structure, which represents a chop. A `CHOPchop` stores all the chop's ctpts and interpolation methods (*imps*). Ctpts are handled by the CTPT package and imps are handled by the IMP package.

In Figure 2.2 (page 14) the first two chops have one ctpt each and the last two chops have two ctpts each. None of them explicitly specify interpolation methods. Internally, however, they do store the default interpolation methods. The default interpolation method for all chops is discrete.

The following operations are executed when a chop is added to an object: First the necessary memory for the `CHOPchop` data structure is allocated by calling `CHOPcreate()`. The returned chop does not yet contain any ctpts. They are added by calls to the `CHOPset()` routine, which creates a new ctpt and links it into the given chop. The process of adding a chop is illustrated graphically in Figure 2.4 (page 17).

Every chop is assigned a *priority*. This value corresponds roughly to the line number of the chop in a FLESH script. Chop priorities define the order in which chops are evaluated. This evaluation order is important because the operations represented by chops are not always commutative; an example will be given in Section 3.7 (The STATE Package, page 52). Chops near the beginning of the script are called *low-priority chops* and chops near the end are called *high-priority chops*. Note that these terms do not indicate the importance of the chops; saying a chop is "low-priority" means merely that it has a low priority value (i.e., a low line number), not that it has limited importance.

2.2.3 Inquiring Data

A client gets information from the database by inquiring a piece of data from an object. The chops associated with an object specify how data values change over time, so the inquiring client must specify the time at which to compute the data. In Figure 2.2 (page 14) for example, inquiring

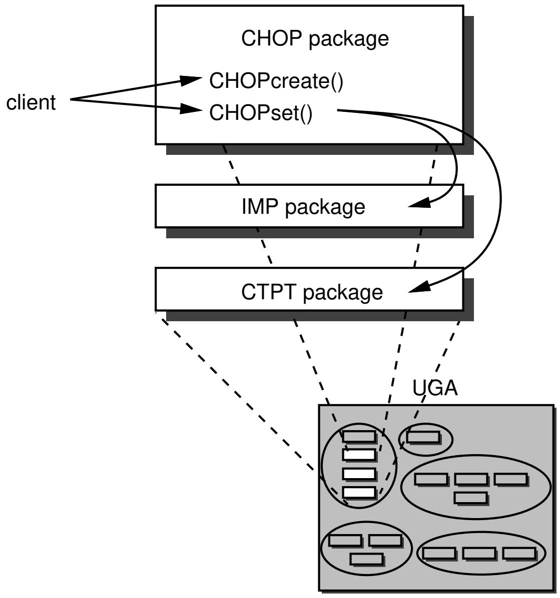

Figure 2.4: Adding a chop.

the current transformation matrix (CTM) for `zelda` at time 5 would return a matrix that translates `zelda` to the position [-1, -2, -3] (since discrete interpolation is used by default).

To inquire some data of an object at a particular time, the client calls an *inquiry method* associated with the object's class. In each object class there is a method for each kind of data that can be inquired. The color of `zelda`, for example, is determined by calling the "color" method of the torus object class.

The methods of an object class distinguish it from any other object class. The most obvious difference between a torus and a sphere, for example, is the shape of their polyhedral b-reps, represented as `TRIPobjects`[2]. The sphere class and the torus class each have a method that calls TRIP, but the two methods ask for different polygons, each creating the shape appropriate for its own class.

[2] The TRIP package implements an abstract data type for polyhedra whose faces are triangles. A `TRIPobject` is an instance of this data type. The details of this package are not important for this document and are given elsewhere [HUAN91a].

The STATE package keeps track of the inquiry methods. A client obtains the method used to inquire a particular kind of data from a particular object by calling the `STATEsetup_method()` routine. This routine will make the object's `STATEstate` point to a data structure, a `STATEbehavior`, in which the client can find a pointer to the appropriate inquiry method. As just mentioned, the particular method pointer returned will depend on the class of the object.

When the client calls the inquiry method, the method computes the appropriate data at the specified time. Computing this data involves stepping through the chops in the object's state and determining how each chop affects this kind of data. This iterative process is called a *state traversal*. The STATE package provides a routine, `STATEtraverse()` that traverses a state to compute a piece of data from an object.

At each chop visited during the traversal, `STATEtraverse()` calls a *chop method* with the inquiry time as a parameter. The chop method determines the effect of the current chop on the data being inquired. The chop method takes as input the data that has already been computed by the earlier chops in the state; the method then adds the effect of the current chop onto this data. An example of this kind of "additive" effect occurs when an object's position is being computed in the form of a CTM. Each translation chop specifies a relative displacement from the current position of the object, so each translation chop method modifies an existing CTM to include another relative displacement. Not all chop methods are additive, however. A color-specification chop, for example, sets the color of an object to be a particular value, ignoring whatever color the object currently has.

Chop methods, like inquiry methods, are dependent on an object's class because a chop may be interpreted differently by objects belonging to different classes. The chop methods associated with a class are part of the package that implements that class. The GOBJ package, for example, implements the "gobj" class and contains its chop methods.

To determine the effects of a chop, a chop method calls several packages. The CHOP package is called to determine how the sequence of ctpts associated with the chop cooperate to determine the chop's value at the inquiry time. Determining this value involves interpolating between the ctpts, so CHOP calls the interpolation routines of the IMP package. The IMP package needs to know the values of the ctpts, so it calls the CTPT package to evaluate each ctpt independently. Notice that the CHOP package knows how to

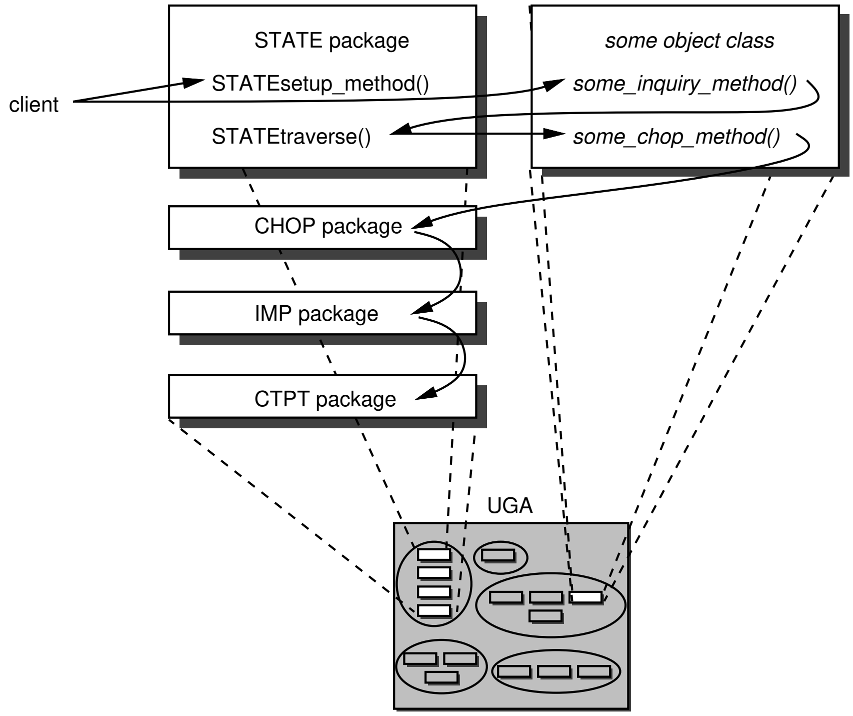

Figure 2.5: Inquiring data from an object.

process sequences of ctpts, while the CTPT package knows how to process only individual ctpts.

A state traversal can be an expensive operation. To compensate, UGA tries to avoid making a full state traversal whenever possible. A *lazy evaluation* scheme is used, i.e., no data is computed until it is actually inquired by some client. Whenever a piece of data is inquired, it can be cached so that it can be reused during a later inquiry. If the chops that affected the inquired data do not change their values during a period of time, the data itself will not change during that period. A *validity interval* is associated with a cache to indicate the time period over which the cached data will not change. A subsequent inquiry at a time within the validity interval can be satisfied by retrieving the cached data. If the chops that affected the cached data are modified, the validity interval must be updated so that it indicates the new period during which the data remains unchanged.

```
depardieu : rep       0  = cube
            translate 0  = [-1, -2, -3]
                      10 = [ 1,  2,  3]
          ;

/* The camera class is called "leika". */
fellini : rep    0 = leika
          orient 5 = [[10, 10, 10],
                      depardieu.translate, [0, 1, 0]]
        ;
```

Figure 2.6: A FLESH script involving a reference. As mentioned in the FLESH manual [GRIM91], the `translate` attribute of an object can be considered to be the same as its "position" (in an intuitive sense). The technical differences are not important for this example.

The process of inquiring data is illustrated graphically in Figure 2.5 (page 19). The specific details of state traversal are quite subtle, and will be discussed at length in Sections 3.7 (The STATE Package, page 52) and 4.6 (More Details of the STATE Package, page 84).

2.2.4 Changing a Chop

There are three mechanisms to modify a chop. The user is able to add a new ctpt, change the value[3] of an existing ctpt or delete an existing ctpt. A new ctpt is added by the client calling `CHOPset()`. An existing ctpt can be changed by calling `CHOPset_ctpt()`. The routine `CHOPdelete_ctpt()` is used for deleting ctpts.

As mentioned in the previous subsection, UGA exploits caches for efficiency. Some of these caches may no longer be valid after a user changes a ctpt. The above-mentioned functions `CHOPset()` and `CHOPset_ctpt()` will automaticly invoke mechanisms to invalidate all caches dependent on the changed data. This process is known as *invalidation*. Dependent caches

[3]Remember that a ctpt is simply a time-value pair. Therefore "changing the value" means either changing the time part or changing the value part of the time-value pair.

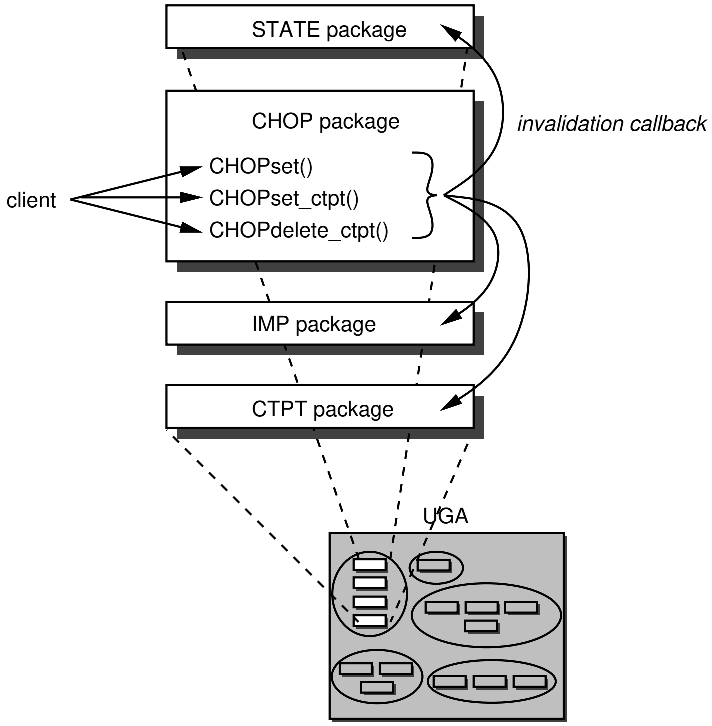

Figure 2.7: Changing a chop.

occur when one part of the model references another part. See Figure 2.6 (page 20) for an example. When `fellini`'s position is inquired, a cache will be created. This cache will be dependent on `depardieu`'s position. Invalidation is an important part of lazy evaluation. The idea of lazy evaluation is that it is cheaper to invalidate a cache than to update the cache's contents. The cost of updating the cache will only be incurred if a future inquiry actually needs the contents of that cache.

Invalidations are implemented through callbacks from CHOP to STATE. The CHOP package calls the states of all objects dependent on that particular chop. The STATE package then invokes a state traversal to walk through the state and to invalidate all dependent caches. The packages that interact when a chop is changed are summarized in Figure 2.7 (page 21).

Chapter 3

The Kernel of the Database

This chapter starts with low-level packages and works up to higher-level ones, filling in more details of the database operations sketched in the previous chapter. The progression is from low-level to high-level packages because advanced concepts like state traversal cannot be described in detail if basic concepts like validity intervals and chop caches are not understood. Certain details will be postponed until Chapter 4 (More Details of the Kernel, page 75), e.g., any data structures or fields in data structures that are not essential to understanding the main purpose of each package.

3.1 The VALU Package

The VALU package manages data that is structured primarily in *syntactic* rather than *semantic* terms. The main use of VALU data structures occurs during the parsing of FLESH scripts.

When a script is parsed, VALU data structures are built to represent the arguments of the parsed chops. Most of the interpretation of these data structures is deferred until the chops are actually evaluated. For example, a `translate` chop might take a list of three numbers as an operand. The list represents a 3D vector in this context. A chop that specifies the parameters of a superquadric surface also takes a list of three numbers (intuitively, there is one parameter each for the "shape" of the longitude curves, the "shape" of the latitude curves, and the radius of the central hole [BARR81]. This list does not represent a vector, but instead three individual values. In both situations, however, the VALU structure built is just a list of three numbers; the appropriate interpretation is performed later.

3.1.1 Basic Data Types

The basic data structure provided by VALU is the `VALUitem`. The main part of `VALUitem` is a `union` of four different subtypes:

1. A `VALU_atom` can be an integer, a floating-point number or a character string.

2. A `VALU_pointer` is a pointer to a higher-level data type that VALU is not expected to understand. The manipulations of this pointer that VALU can make are provided via callback functions in a `VALUuser_type` structure. This structure contains pointers to routines to copy the higher-level data type, print it, free it, pack it (into a single block of contiguous memory locations) and unpack it. The last two operations are used when a `VALUitem` needs to be transmitted as part of a network packet; see Section 6.3 (Networking, page 101). The maintenance of `VALUuser_type` structures is described in Section 3.1.2 (User Types, page 26).

   A `VALU_pointer` is used is when a chop in a FLESH script contains a reference to an object, e.g., so a camera can orient itself to point at the position of a cube in the scene. A FLESH fragment that illustrates this example is given in Figure 2.6 (page 20). The reference to the cube will be represented by a `STATEref` structure, described in Section 3.7 (The STATE Package, page 52). The chop gets to this `STATEref` via a `VALU_pointer`, and manipulates it through the STATE callback routines in the `VALU_pointer`'s `VALUuser_type` structure.

3. A `VALU_expr` is an expression involving `VALUitems`. The expression can be a binary operation on two `VALUitems`, a unary operation on one `VALUitem`, or a function taking one or more `VALUitem` as arguments. The operations and functions supported are those needed for the values of chops in FLESH scripts (see the FLESH manual [GRIM91] for details).

4. A `VALU_list` is a list of `VALUitems`. This list is not given any semantic meaning ("type") by VALU; this meaning will be added later by the

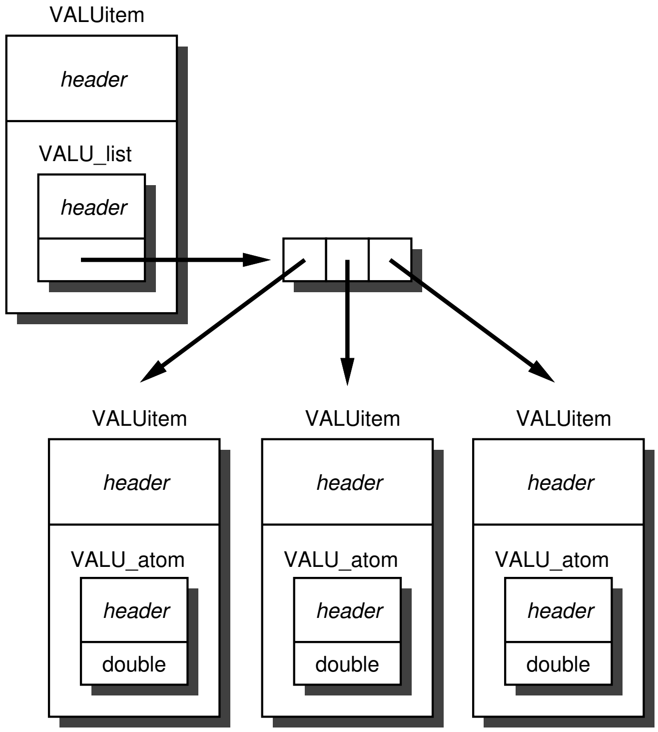

Figure 3.1: A VALU tree that represents a three-element vector.

CORE and SXF package, as described in Section 3.2 (The CORE and SXF Packages, page 27).

The `VALU_expr` and `VALU_list` subtypes allow hierarchical data structures to be built. These hierarchies are known as VALU *trees*. The VALU tree that represents a three-element vector is depicted in Figure 3.1 (page 25).

A VALU tree represents a parse tree for the value of some chop's ctpt. When that chop needs to be evaluated to determine its effect on an object, the VALU tree needs to be evaluated. Evaluation of a `VALUitem` is performed by the `VALUeval()` routine. The rules of evaluation are as follows:

1. A `VALU_atom` evaluates to itself.

2. A VALU_pointer is evaluated by calling the evaluation routine in the associated VALUuser_type structure. This routine produces as its result a VALUitem that needs no further evaluation.

3. A VALU_expr is evaluated by recursively evaluating the one or more operand VALUitems and then applying the appropriate operation or function.

4. A VALU_list is evaluated by recursively evaluating all its elements.

A VALUitem is called *elementary* if it cannot be evaluated any further. A VALU_atom is elementary by definition. A VALU_list is elementary if all of its elements are elementary.

The evaluated form of a VALU tree is frequently cached. The cached form is maintained by the SXF package, which is described in Section 3.2 (The CORE and SXF Packages, page 27). To build the cached form from the tree, access to each node of the tree is needed. This access is provided by the VALUget_subtree() routine. This routine takes as parameters a VALUitem (assumed to be a tree) and a path to a particular node in that tree. The further details of this routine are not important at this point, and more discussion of this routine is saved for Section 4.2 (More Details of the VALU Package, page 77).

The VALU package provides a routine VALUprint() to print a VALUitem in a meaningful way. This routine is used when a script built by an interactive modeler needs to be displayed.

3.1.2 User Types

As already mentioned, a VALUuser_type structure contains pointers to the callback routines that allow VALU to manipulate a VALU_pointer without actually knowing what it represents. Whenever a higher-level package wants to tell VALU about a new type, it calls VALUadd_user_type(). This routine registers the new type by building a VALUuser_type structure and putting it in an array indexed by the new type's identifier. Every VALUitem contains a type_index field that can hold a type identifier. The VALUuser_type appropriate for a VALUitem whose union is storing a VALU_pointer can thus be found in the array.

The array of `VALUuser_type` structures is stored in the `VALUid` structure. This structure is a repository for general house-keeping data needed by VALU. The `VALUid` is built once, and every `VALUitem` stores a pointer to it. The other, less important fields of the `VALUid` structure are described in Section 4.2 (More Details of the VALU Package, page 77).

3.2 The CORE and SXF Packages

The CORE and SXF[1] packages together provide a standardized way of handling data that has more semantic meaning than a `VALUitem`. This standardized data handling is most important when the value of a chop at a particular time is being computed. Different chops can have different types of values, but the routines that evaluate chops need a common data structure for all types. A `VALUitem` does not serve this purpose efficiently because its structure is based on syntax rather than semantics.

Recall, for example, the VALU tree for a three-element vector, as depicted in Figure 3.1 (page 25). This tree requires four complete `VALUitems` and their associated overhead. It seems reasonable that a script parser might produce such a representation, but it also seems appropriate to convert to a more compact representation for further processing. The CORE and SXF packages provide such a representation.

3.2.1 Core Types Versus Form Types

The common data structure that is used throughout chop evaluation is the `SXFform`, defined in the SXF package. It represents an instance of a *form type* (often called a "form"). Form types are defined in terms of *core types* (often called "cores"). Instances of a core type are represented by `COREcore` data structures, which are supported by the CORE package.

The distinction between core types and form types is best understood by considering the history of the UGA system. When the system was being designed, all applications that might use the system could not be anticipated. Each of these applications might need to add new data types, instances of which could be produced by evaluating new kinds of chops. It seemed likely

[1] This acronym is pronounced "sex eff." It stands for "*s*tandard *xf*orms," which in turn stands for "standard transformations."

that many of these new data types would just be collections of basic data types like integers, real numbers, vectors, etc. So UGA was designed to incorporate a small set of atomic data types, a small set of basic operations on these atomic data types, and a mechanism for building aggregates of these atomic data types. The atomic data types are the core types, and the aggregates are the form types.

The basic operations that can be performed on core types are the following:

- allocating a `COREcore` to be an instance of that core type;
- freeing an instance;
- copying an instance;
- printing an instance in textual form (for debugging);
- initializing an allocated instance (to a value appropriate for that core type);
- packing and unpacking an instance for network transmission; and
- adding, subtracting and multiplying two instances.

The arithmetic operations just mentioned are used when a `COREcore` needs to be interpolated from two existing `COREcores`. Interpolation is a fundamental part of chop evaluation, and will be described further in Section 3.5 (The IMP Package, page 38).

The `SXFform` is the common data structure used throughout chop evaluation, so packages need to be able to perform the above operations on instances of form types, too. Since a form type is simply a collection of core types, an operation on a form type is defined to be that operation performed on each of the core types included in the form type. As a consequence of this definition, a form type acts like "a sum of its parts." In other words, an operation on a form cannot add any meaning beyond the meaning of that operation performed on the cores in the form.

A 3D vector is an example of a data type that is implemented as a core and not a form. At one level, such a vector is a collection of three numbers—its Euclidean coordinates. During chop evaluation, however, it

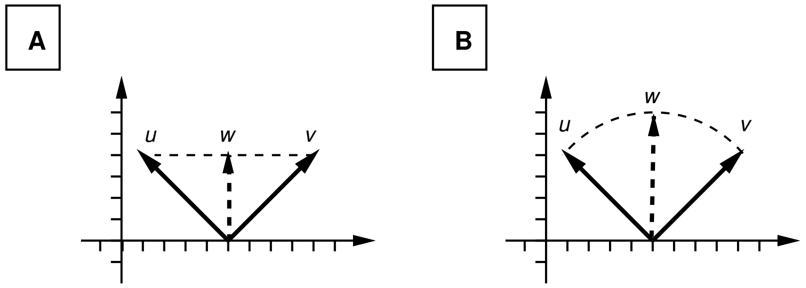

Figure 3.2: Interpolating the vector w halfway between vectors u and v in two different ways: Figure A shows linear interpolation of the coordinates independently; Figure B shows slerp.

generally makes sense to treat a vector as an atomic piece of data. Interpolation of one vector from two others, for example, should not be performed by interpolating the coordinates individually. A better approach is to use "slerp" ("*s*pherical *l*inear int*erp*olation"). Slerp is like linear interpolation on the surface of a sphere, where a line corresponds to an arc. The difference between linear interpolation of the individual components and slerp is exemplified in Figure 3.2 (page 29). To be able to use slerp, a vector must be defined as a core type and not a form type.

3.2.2 Representing Cores

The `COREcore` structure acts as a handle that can refer to any core type. The main component of a `COREcore` is a union of pointers to each of the data structures that represents a single core type (e.g., an `int`, a `double`, a `MAT3vec`[2], etc).

The `COREcreate()`, `COREdestroy()`, `COREcopy()` and `COREprint()` routines are used to allocate, free, copy and print a `COREcore`, respectively. Each of these routines takes as a parameter the core type to operate on. This parameter is of type `COREtype`, an enumerated type with an element

[2] The MAT3 package represents 3D vectors and transformation matrices in UGA. It provides the expected operations, and its details are omitted.

for each core type known to the system. Each of the routines contains a switch statement with a case for each of the possible core types.

To handle arithmetic operations on cores, the CORE package contains a separate routine for each operation on each core type. A COREcore_def structure is allocated for each core type, and pointers to the three routines for the core type are stored in that structure. The global array CORE_core_defs[] holds the COREcore_def structures for all the core types, indexed by their COREtype elements.

3.2.3 Representing Forms

The SXFform structure can be used to represent an instance of any form type. This structure contains a type field that indicates the type of form being represented. The value of this field is an element from the enumerated type SXFform_type. The main part of a SXFform is the core field, an array of pointers to COREcore structures. This array has one element for each of the core types used in the definition of this form type.

The particular core types that are aggregated in a particular form type are described by a SXFform_def structure. Each form type has its own such structure. One field of this structure indicates the number of elements in the core array mentioned above, and the core type in each such element; the particular fields of a SXFform_def structure are described in Section 4.3 (More Details of the CORE and SXF Packages, page 78). The SXFform_def structure for each form type known to UGA is stored in a global array, SXF_form_defs[], indexed by the SXFform_type element corresponding to the form type.

This array is used by the SXFcreate() routine when it is called to allocate a new instance of a form type. This routine allocates a piece of memory large enough to hold the complete SXFform, including the core array of pointers to all the form's cores. Once the SXFform itself has been allocated, COREcreate() is called once for each core included in the form. The routines SXFdestroy(), SXFcopy() and SXFprint() operate in a similar manner. They use a SXFform_def structure to determine the cores that are aggregated in the form, and call CORE routines to manipulate each of these cores.

The SXFform_def structure also defines how the corresponding SXFform can be built from an evaluated VALU tree. The `SXFval_to_form()` routine performs this operation. The routine takes the data for each of the core types in the `SXFform`'s `core` array from one node of the evaluated VALU tree. As already described in Section 3.1 (The VALU Package, page 23), the VALU package provides the `VALUget_subtree()` routine for accessing any of those nodes. All that is needed is the path through the tree to the particular node. A specification of each such path is stored in the `SXFform_def` structure. The `SXFform_def` structure will be discussed further in Section 4.3 (More Details of the SXF Package, page 78).

3.2.4 High-Level Cores

Some core types are data structures maintained by high-level packages. These high-level packages are not visible to the CORE package (i.e., they are not `#included` by CORE). Thus, the `COREcore` structure must use a generic pointer[3] to refer to these data structures. The allocating, copying and destroying of these data structures must be handled by callbacks to the high-level package. The `COREregister()` routine allows callbacks to be associated with these core types. Note that a distinct core type is needed for each of these high-level data structures (so a different set of callbacks can be associated with each of them).

An example of such a core type is the "aggstuff" core. This core is used by the AGG package, which allows objects in a time-parameterized model to be grouped together into aggregate objects. The AGG package maintains private data structures that describe how the grouping of objects changes over time. These data structures are simpler to process if they can be stored as the values of special chops on an "agg" object. The value associated with each of these special chops is an "aggstuff" form that has an aggstuff core as its only component. The aggstuff core points to an instance of the private AGG data structure. Aggregate objects are described further in Section 5.1 (Aggregate Objects, page 89).

Notice in the previous example that there is a one-to-one relationship between the aggstuff core and the aggstuff form. In other words, an aggstuff form always contains only one aggstuff core, and an aggstuff core never exists except as the only element of an aggstuff form. In this situation,

[3] The most suitable generic pointer type depends on the operating system. The `UGAptr` type is defined to be a machine-independent generic pointer.

it seems unnecessary to have both a core and a form; it would be simpler if the aggstuff form could point directly to the private data structure of the AGG package. The majority of form types, however, do not correspond directly to a single high-level core type. For these form types, the decision to require that each form type contain only pointers to core types (as opposed to pointers to arbitrary structures) makes sense.

3.3 The INT Package

It is essential for any animation system to exploit interframe coherency in order to reduce computation time. The UGA system uses the INT package to associate all object data with a time interval over which the data is valid or invalid.

Validity intervals are associated with the caches used during state traversals; see Section 3.6 (The CHOP Package, page 43) and Section 3.7 (The STATE Package, page 52) for more details. Values interpolated by the IMP package are tagged with a validity interval; see Section 3.5 (The IMP Package, page 38). To be able to support aggregate objects, each chop must be associated with an interval during which it is active; this subtle use of intervals is described in Section 4.5 (More Details of the CHOP Package, page 82) and in Section 5.1 (Aggregate Objects, page 89).

The INT package contains support for time intervals. This includes the creation of intervals as well as various operations on intervals. Support is also provided for functions mapping intervals to other intervals.

3.3.1 Interval Types

Intervals are either *simple* or *complex*. There are four types of simple intervals.

- *Range* intervals occupy a finite, continuous range of time, and are defined by a start time and an end time. For example, $[5.0, 10.0)$ is the interval starting at time 5.0 and ending at time 10.0. A range interval is assumed to be closed on the left and open on the right. Therefore, the interval $[5.0, 10.0)$ contains the time 5.0, but does not include the time 10.0. This convention has been adapted so that intervals can be easily connected to form continuous larger intervals.

- *Point* intervals occupy a single moment in time. An example is $[5.0]$.
- *Infinite* intervals occupy an infinite amount of time, although they may have well-defined (not ∞) start or end times. Infinite intervals are closed on the left when the left time is well-defined, and open on the right when the right time is well-defined. The following are examples of infinite intervals: $(-\infty, 10.0), [30.0, +\infty)$, and $(-\infty, +\infty)$. The first example does not include the time 10.0, but the second example does include the time 30.0.
- *Empty* intervals occupy no time.

A *complex* interval is made up of two or more simple intervals. Complex intervals can only be made by using INT operations on simple or complex intervals. In this document, complex intervals are syntacticly written as a series of comma-separated simple intervals enclosed in curly braces. An example is the complex interval $\{[0.0, 10.0), [15.0], [25.0, \infty)\}$, which is made up of a range interval, a point interval, and an infinite interval.

3.3.2 Operations

The INT Package supports the following operations on intervals.

- The *union* of two intervals A and B is a new interval containing all time occupied by either A or B. For example, the union of the two simple intervals $[0.0, 5.0)$ and $[10.0, 15.0)$ is the complex interval $\{[0.0, 5.0), [10.0, 15.0)\}$. The union of the two complex intervals $\{[0.0, 10.0), [15.0, 25.0)\}$ and $\{[5.0, 20.0), [25.0, 30.0)\}$ is the simple interval $[0.0, 30.0)$. The union of the simple interval $[0.0, 5.0)$ and the point interval $[2.0]$ is the simple interval $[0.0, 5.0)$.
- The *intersection* of two intervals A and B is a new interval containing only time occupied by both A and B. For example, the intersection of the two simple intervals $[0.0, 10.0)$ and $[5.0, 15.0)$ is the simple interval $[5.0, 10.0)$. The intersection of two the simple intervals $[0.0, 5.0)$ and $[10.0, 20.0)$ is an empty interval.

Intersection is used extensively in data inquiry. In the course of a single data inquiry, many other pieces of data may be gathered, each with its own validity interval. The validity interval of the data inquiry as a whole is the intersection of all the validity intervals of the individual pieces of data.

- The *subtraction* of an interval A from an interval B is all the time in interval B except for any time in the intersection of A and B. For example, the subtraction of the simple interval $[5.0, 10.0)$ from the simple interval $[0.0, 20.0)$ is the complex interval $\{[0.0, 5.0), [10.0, 20.0)\}$. The subtraction of the simple interval $[0.0, 5.0)$ from the infinite interval representing all time, $(-\infty, +\infty)$, results in the complex interval $\{(-\infty, 0.0), [5.0, +\infty)\}$. In general, any interval A subtracted from the infinite interval $(-\infty, +\infty)$ gives the negation of A, that is, all times not in A.

3.3.3 Mapping Functions

The INT package provides support for the declaration of a linear function between two intervals. This function is called a *mapping function*. Given intervals A and B, $(A \rightarrow B)$ denotes the mapping function from A to B. The mapping function from A to B is only defined, if and only if both A and B are simple range intervals. This mapping function may be used in two ways:

1. A time t in interval A may be mapped to a corresponding time in interval B. That is, the time whose relative position in interval B corresponds to time t's relative position in A is returned. For example, suppose interval A is $[0.0, 10.0)$ and interval B is $[20.0, 40.0)$, and time t is 5.0. Then executing the mapping function $(A \rightarrow B)$ with time t returns the time 30.0, because t's relative position in interval A is halfway between A's start and end times, and 30.0 is similarly halfway between interval B's start and end times.

2. A time t may be mapped through an *inverse mapping function*, which finds the time whose relative position in interval A corresponds to time t's relative position in B. Using the same definitions of A and B as in the previous example, a time t of 30.0 may be inversely mapped through $(A \rightarrow B)$ and return 5.0.

The following is an example for the application of the mapping mechanism. The example explains how mapping functions are useful in retaining cache values, even though the current version of UGA does not make use of this functionality. The reasons for not using mappings will be explained in Section 3.6 (The CHOP Package, page 43). Although the example is therefore hypothetical, it is still useful in understanding mapping functions.

Suppose the value-time of a ctpt in a chop is changed. If the interpolation method of the chop is discrete, it is not necessary to invalidate caches based on that chop. The validity intervals of those caches can instead be updated using a mapping.

For example, let us call the ctpt whose value-time changes c_{change}. The ctpt directly preceding c_{change} is named c_{prec} and the ctpt directly succeeding c_{change} is called c_{succ}. The interval defined by the value-times of c_{prec} and c_{change} is called A. More specifically A_{before} is the interval before the value-time of c_{change} is changed and A_{after} is the interval after the value-time of c_{change} has been changed. Analogously, we define B_{before} and B_{after} to be the intervals defined by the ctpts c_{change} and c_{succ}.

All caches dependent on the values defined by c_{prec} and c_{change} have validity intervals that are either included in A_{before} or in B_{before}. Defining s and e to be, respectively, the start- and end-times of the validity interval of the cache, one of the following conditions is true:

$$(1)\ \text{value-time}(c_{prec}) \leq s \leq e \leq \text{value-time}(c_{change})$$

$$(2)\ \text{value-time}(c_{change}) \leq s \leq e \leq \text{value-time}(c_{succ})$$

In general a cache validity interval is shorter than A_{before} or B_{before}. The reason for this is that the validity interval is computed to be the intersection of the validity intervals of all the chops the cache depends on.

The caches for which condition 1 holds true can be retained by mapping s and e through $(A_{before} \rightarrow A_{after})$. The caches that fulfill condition 2 are retained by mapping their start- and end-times s and e through $(B_{before} \rightarrow B_{after})$.

3.4 The CTPT Package

The CTPT package manages control points (ctpts). A ctpt is a pairing of a time t and a value v; this pairing can be denoted $\langle t, v \rangle$. A ctpt is associated with a chop. Each chop defines a function of some set of parameters; the ctpts of a chop define how the values of these parameters change over time. As mentioned in Section 2.1 (Basic Concepts, page 9), the time component of a ctpt is known as the *value-time*. This phrase will be used to clarify the discussion in Section 3.5 (The IMP Package, page 38).

3.4.1 Data Structures

Each ctpt is represented by a `CTPTpoint` structure. One field contains the value-time component of the ctpt. Another field holds the ctpt's value. The value is a `CTPTval` structure. This structure has a union that points to a `VALUitem` or a `SXFform`.

To understand why a ctpt's value could be a `VALUitem`, recall Section 3.1 (The VALU Package, page 23). This section mentioned that the parsing of a FLESH script causes `VALUitems` to be constructed for the parameters of the chops. Since these parameters are really ctpts, a ctpt must be able to store a `VALUitem` as its value. We do not evaluate these `VALUitems` to `SXFforms`, since we would violate the lazy evaluation paradigm. In addition, by evaluating a `VALUitem` important information might be lost. For example, evaluating a `VALUitem` that contains a reference to another object would result in a constant `SXFform` that has no knowledge about the dependency.

To see why the value could also be a `SXFform`, recall Section 2.2.3 (Inquiring Data, page 16). This section mentioned that caches are created when data is inquired from an object. The mechanism that stores a piece of cached data acts as a function of time: it returns the cached data at any time during the cache's validity interval. Chops are the general mechanism for representing functions of time in UGA, so caches can be represented as chops. A "cache chop" stores the cached data in its ctpts. Since inquired data is always expressed as a `SXFform`, a ctpt must be able to store a `SXFform`. The use of chops as caches will be described in more detail in Section 3.6 (The CHOP Package, page 43).

The `CTPTpoint` structure also contains a type field. This field indicates how the ctpt is being used. There are three possible values:

1. `CTPT_TYPE_VALID` means that this ctpt and the one following it define a range over which interpolation can occur. (The ctpts of a chop are stored in ascending chronological order). The value of the chop's

parameters over this interval will be interpolated using some function of the ctpts at the ends of the interval. The topic of interpolation is discussed in detail in Sections 3.5 (The IMP Package, page 38) and 3.6 (The CHOP Package, page 43).

2. `CTPT_TYPE_PVALID` means that this ctpt defines a "point interval," i.e., a single instant of time. In other words, no interpolation can be performed in the region between this ctpt and the following one. This kind of ctpt is useful for some kinds of caches. By convention, the data in a cache is assumed to be valid during an interval of time. But a cache may store a value that is valid only at the instant the value was inquired (this situation occurs if any interpolation technique other than discrete interpolation was used to compute the value). The validity interval for this cache must thus be a point interval. It is more efficient to represent the point interval with a single `CTPT_TYPE_PVALID` ctpt than with two `CTPT_TYPE_VALID` ctpts that have the same value-time.

3.4.2 Evaluating a Ctpt

The main operation supported by the CTPT package is ctpt evaluation. The routine `CTPTeval()` takes a `CTPTpoint` and a data inquiry time t that has been passed from STATE down through CHOP and IMP. The routine returns a `COREcore` containing the ctpt's value at time t. It also returns the interval of time over which this value is valid.

If the ctpt being evaluated stores a `SXFform`, then `CTPTeval()` returns one of the `COREcores` contained in that `SXFform`. The particular `COREcore` to select is identified by a parameter to `CTPTeval()`. Selecting part of a ctpt's data in this manner is essential for the support of *tracks*. This concept is explained in more detail in Section 3.5 (The IMP Package, page 38). Note that the inquiry time t is not used in this case because all `SXFforms` are constant functions of time.

Evaluating a ctpt that stores a `VALUitem` requires evaluation of the `VALUitem` at the inquiry time t. The evaluation is performed by `VALUeval()`, as described in Section 3.1 (The VALU Package, page 23). The inquiry time t is important because the `VALUitem` may vary over time. Recall, for example, Figure 2.6 (page 20). In this FLESH script, the object `fellini`'s `orient` chop contains a ctpt that references the "translate" (position) of the object `depardieu`. This position will change over time, so the ctpt must be evaluated as of a particular time t. The actual mechanism with which `VALUeval()` can be told to evaluate a `VALUitem` at a particular time is somewhat subtle; the details are not important at this point, and are saved for Section 4.4 (More Details of the CTPT Package, page 80). Once the `VALUitem` is evaluated, it is converted to a `SXFform` with a call to `SXFval_to_form()`. A particular `COREcore` is then extracted as described in the preceding paragraph.

The `COREcore` produced by `CTPTeval()` is used by the IMP package to interpolate a chop's parameter values as of an inquiry time. As will be described in Section 3.5 (The IMP Package, page 38), this interpolation usually requires evaluating several ctpts. The `CTPTeval()` routine is oblivious to this interpolation; it just evaluates a single ctpt.

3.5 The IMP Package

The IMP package is used to interpolate ctpts. An `IMPobj` is the basic data type of the IMP package. The IMP package defines a set of interpolation type constants, each corresponding to an interpolation method. Each `IMPobj` has an IMP interpolation type and a set of ctpts. Since specific interpolation methods can be further specified by a number of additional parameters, these additional parameters are also stored in the `IMPobj`.

The IMP package currently supports four basic interpolation methods: Discrete, Linear, Integrate and Quaternion. For more information on the semantics of these methods, see the FLESH manual [GRIM91]. Note the absence of spline interpolation methods. The IMP package currently does not support spline interpolation as this kind of interpolation can be achieved by using path objects (see Section 5.2, Paths, page 91).

The CHOP package calls `IMPinterpolate()` to compute interpolated values and their validity intervals for given chops. The IMP package inquires and evaluates values from ctpts via the `CTPTeval()` function.

In order to interpolate between ctpts, it is necessary to be able to find the value-time (see Section 3.4, The CTPT Package, page 35) of the ctpt and its associated value. The value-time of a ctpt can be found through the `CTPT_TIME()` macro. The value is computed by the `CTPTeval()` function. Further information about these functions can be found in Section 3.4 (The CTPT Package, page 35). The data type returned by `CTPTeval()` is a

COREcore.

The choice to interpolate COREcores instead of VALUitems is based on the fact that COREcore data has a more efficient format that is semantically richer than a VALUitem. It is more efficient because COREcore data is always fully evaluated in contrast to a VALUitem. A VALUitem can store an expression tree as its value, which would have to be evaluated every time we need to use this VALUitem in an interpolation operation.

The COREcore data structure is semantically richer than a VALUitem, because a COREcore knows what kind of data is stored within it. This semantic richness is used when computing the interpolation value. For example, it is necessary to know that the data being interpolated is a quaternion to be able to apply quaternion addition and multiplication (see the next paragraph). A VALUitem would store the axis and the angle of a quaternion as a list of a list of three doubles and a double. This representation is not sufficient as it encodes too little semantic information about the data type.

Since each type of COREcore is essentially a "subclass" of a general COREcore "class", each interpolation method is implemented as abstract operations on members of the COREcore class. Each particular COREcore type overrides, as appropriate, the operations defined in the base COREcore class. For example, the operator "+" used in linear interpolation is executed as double, vector or quaternion addition depending on the used COREcore type.

3.5.1 Interpolation Mechanism

Interpolation methods are implemented in the IMP package. However, since ctpts not only store constants, but also time-varying expressions including references to other objects as their values, we had to extend our notion of how to interpolate between ctpts.

Let us review how interpolation normally works via a simple example. Given two ctpts, a and b, we can define a linear interpolation between these two points. If a is defined at time 0 and b is defined at time 10, then the points lying on the line between a and b are said to linearly interpolate a and b. See Figure 3.3 for a picture of this example. Put into our context, a and b would be defined as ctpts with value-times 0 and 10, respectively. The points on the line between a and b could be computed by inquiring IMP at

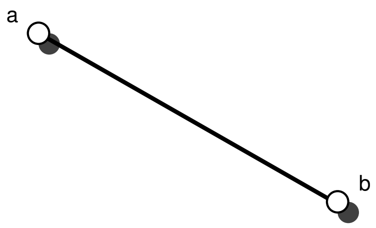

Figure 3.3: Linear interpolation between non-time-varying ctpts.

various inquiry times between 0 and 10. Note the difference between inquiry time and value-time.

The above example is a special case in IMP, since the ctpts are not time-varying, in the sense that they do not change their values over time. Let us therefore expand the example. Now the ctpts a and b, instead of being constant value expressions, are defined to be the positions of moving objects. Their respective value-times are still 0 and 10. What is the result of a linear interpolation between a and b?

Before answering this question, let us explain how this example can occur in animations. If an artist wants to have the camera focused on one object and then wants to shift focus to another object, she would realize the shift of focus by employing a linear (or other) interpolation method between two ctpts, each of them having as its value a dependency on the position of an object in the scene. Control point a depends on the object on which she wants to focus first and ctpt b depends on the object to which she wants to shift the focus. How far apart the value-times of the ctpts are, determines how fast the focus shift will occur. In our example it will take 11 frames (from time 0 to 10), assuming that we render one frame per time unit.

Time-varying ctpts are interpolated by first evaluating their value expression to a constant value. The evaluation is done by calling `CTPTeval()` with the current ctpt and the inquiry time of the interpolation. Note that the evaluation occurs at the inquiry time, not the value-time, since using the value-time would result in the same behavior as for non-time-varying ctpts. Once we obtained the constant values for the ctpts for the current inquiry time, we apply the interpolation formula in the normal way to get an in-between point. Figure 3.4 was computed by inquiring all times between

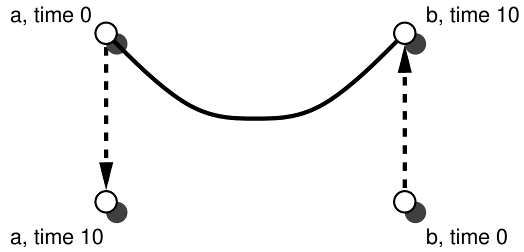

Figure 3.4: Linear interpolation between time-varying ctpts.

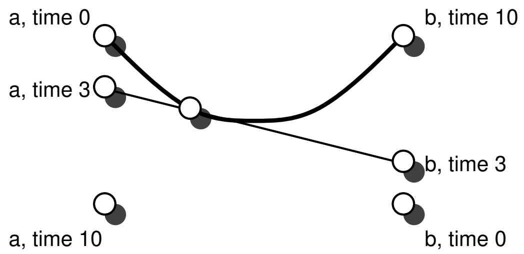

Figure 3.5: How to interpolate time-varying ctpts.

0 and 10.

Looking at Figure 3.5 we can explain the result of Figure 3.4 graphically. When interpolating between a and b we first find out the positions of a and b at inquiry time, say time 3. The line between those two now constant points corresponds to the line previously shown in Figure 3.3 (page 40). Since we are interpolating at time 3 we choose the point that lies

$$\frac{3}{10} = \frac{\text{inquiry time}}{(\text{value-time}(b) - \text{value-time}(a))}$$

on the way from point a to point b.

3.5.2 Extended Interpolation Methods

As mentioned above, IMP currently supports the following basic interpolation methods: Discrete, Linear, Integrate and Quaternion. These basic interpolation methods can be extended by supplying additional parameters to each

```
translate 0 = [1,1,1] : linear(ease(3,3))
          5 = [2,2,2] ;
;
```

Figure 3.6: Specifying an extended interpolation method.

method. The linear interpolation method can, for example, be extended to include the usual accelerate, decelerate and ease interpolation methods. In FLESH such an extended interpolation is specified as shown in Figure 3.6 (page 42). The following discussion, which will explain how this extension mechanism is derived, only mentions linear interpolation; the concepts can be generalized to the other interpolation methods as well.

Linear interpolation is defined as follows. Given two ctpts $a = \langle t_a, v_a \rangle$ and $b = \langle t_b, v_b \rangle$, we can interpolate in between them for any given inquiry time t_{inq} assuming that $t_a < t_b$ and $t_a \leq t_{inq} \leq t_b$. The interpolated value is defined to be

$$v_{inq} = v_a + t'_{inq}(v_b - v_a)$$

with

$$t'_{inq} = \frac{t_{inq} - t_a}{t_b - t_a}$$

The definition of t'_{inq} is a simplification of the general linear interpolation problem to the linear interpolation problem between the two ctpts $a = \langle 0, v_a \rangle$ and $b = \langle 1, v_b \rangle$. In other words, before we interpolate, we project t_{inq} into the interval $[0, 1]$ and call it t'_{inq}. The projection used is linear. This means that v_{inq} will assume all values between v_a and v_b as t_{inq} is continuously varied between t_a and t_b. In addition v_{inq} is directly proportional to t_{inq}, since it is linearly dependent on it.

We are free to change the projection used for t'_{inq}. For example, we could define t'_{inq} to be

$$t'_{inq} = (\frac{t_{inq} - t_a}{t_b - t_a})^2$$

Now, we still project t_{inq} into the interval $[0, 1]$, however the projection formula is quadratic now. As before v_{inq} will assume all values between v_a and v_b as t_{inq} is continuously varied between t_a and t_b. However, in contrast to before, v_{inq} is quadraticly dependent on t_{inq}. It looks as if v_{inq} is accelerating towards v_b as t_{inq} gets closer to t_b.

Therefore interpolation methods can be extended by changing the projection formula for t_{inq}. In the above example we described the acceleration extension to the linear interpolation method. The deceleration and ease extensions are very similar and are therefore not described in this document. Note that the ease extension for linear interpolation can be further parameterized to control how fast to ease-in and ease-out.

3.5.3 Validity Intervals

Finally, IMP computes not only interpolated values, but also the time intervals over which the interpolated values are valid. This validity interval is used by the CHOP package when it determines how long an evaluated datum is valid.

In general, the interpolated values will be valid for only the point interval at the time of the interpolation request. The exception to this rule is discrete interpolation, since in discrete interpolation a value is constant until the next ctpt. Consequently, the interpolated value is valid from the time of the ctpt that precedes the inquiry time to the time of the ctpt after the inquiry time. For an illustration of this fact see Figure 3.7 (page 44).

Inquiring a datum after the last ctpt — let it be $\langle t_{last}, v_{last} \rangle$ — will result in a validity interval of $[t_{last}, +\infty)$. The returned value will be v_{last}. See Figure 3.7 (page 44) for an example.

3.6 The CHOP Package

The CHOP package was created to support the concept of chops. Chops are central to UGA. The reader should already have a good intuitive understanding of chops as they were introduced early on in Section 2.2 (Database Overview, page 12).

In addition to supporting chops, the CHOP package defines and operates on the following data structures: `CHOPscope` and `CHOPinvalids`. `CHOPscopes` are used to segment the list of chops in the database into semantically different sections. `CHOPinvalids` are used to store invalidation

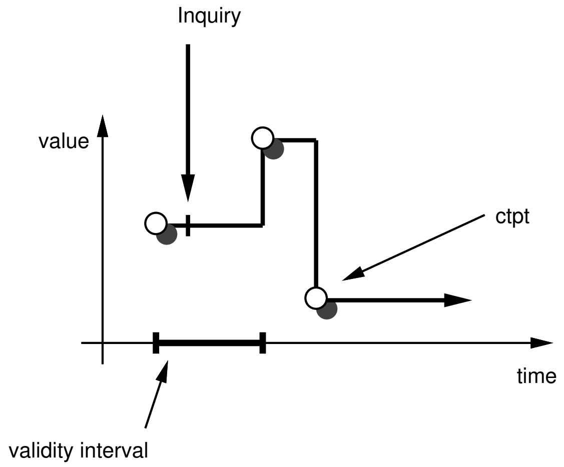

Figure 3.7: Validity interval for discrete interpolation.

intervals.

3.6.1 The CHOPchop Data Structure

The essence of a chop is defined by its `CHOP_guts` field. The guts of a chop point to all its ctpts and interpolation methods. The reason the guts are separate from the CHOPchop is that the same guts can be shared by different CHOPchops. A more detailed explanation of why this functionality is essential can be found in Section 4.5 (More Details of the CHOP Package, page 82).

The Chop Type

The type of a chop determines the type of the FLESH change operator it represents. For example, a chop can be of type `translate`, `scale`, `rotate`, `color`, etc. The type also determines whether the chop is a regular chop or a cache chop. A regular chop is what is normally referred to as a "chop" and resembles the change operator concept in a FLESH script.

A cache chop implements the caches already mentioned in Section 2.2.3 (Inquiring Data, page 16). Caches will be discussed later in this section. For now, note that the only difference between a regular chop and a cache chop is that the ctpt values of the former are stored as `VALUitems`, while the later stores these as `SXFforms`. Regular chops store parsed values (from a script), while cache chops store evaluated expressions for maximum speed.

If a `ctpt` stores a `VALUitem` as its data, it is necessary to transform this `VALUitem` into a `SXFform` of the proper `SXFform_type` before returning it to an inquiry. There is a one-to-one correspondence between `SXFform_types` and chop types. The type field of a chop thus consists of the `SXFform_type` combined with a flag denoting whether the chop is a regular chop or a cache chop.

The Chop Priority

The `CHOP_guts` data structure stores a priority as explained in Section 2.2 (Database Overview, page 12). The priority of a chop corresponds to the chop's position within a sequential ordering of all chops. It is possible for more than one chop to share the same priority. This case often arises when caches are adjacent, because the relative ordering of adjacent caches does not matter.

The Chop Callbacks

The `CHOP_guts` structure maintains a callback list. This callback list is used to store pointers to all data structures that need to be notified whenever the `CHOP_guts` are modified. These pointers implement a data dependency network. Since UGA uses lazy evaluation with caching, it is important to be able to invalidate all caches that are dependent on other chops. The data dependency network implemented by the callback lists enables us to handle data invalidation in an efficient manner.

The data structures stored in the callback list are either `CHOPchops` or `STATEstates`. Examples of callbacks to each of the two types of data structures are given in Section 3.7 (The STATE Package, page 52) because most of the work of invalidation is performed by STATE.

In addition to managing who to call back, CHOP also computes and relays more specific information about the callback. For example, CHOP passes the chop that caused the callback to `STATEchop_changed()`, so STATE knows where to begin the invalidation.

The CHOP package also computes the interval of time that is affected by modifying a chop. This invalidation interval is computed by calling `IMPcalc_inval()`, since IMP knows about how the interpolation methods influence the invalidation intervals. The invalidation interval is stored in an `CHOPinvalids` data structure and passed as part of the callback. Only the cached values within the invalidation interval need to be discarded. Section 3.7.4 (Invalidation and Revalidation, page 59) explains the invalidation concepts in more detail. The `CHOPinvalids` data structure is described in detail in Section 3.6.3 (The `CHOPinvalids` Data Structure, page 48).

The Chop Tracks

Another important abstraction is the *track*. Tracks are needed to be able to interpolate different parts of a `SXFform`. It is desirable to use different interpolation methods for different parts of a `SXFform` for added flexibility and functionality. For example, it should be possible to interpolate two rotation ctpts by using spline interpolation for the point around which to rotate, discrete interpolation for the rotation axis and linear interpolation for the rotation angle. Since all these values (point around which to rotate, rotation axis, rotation angle) are specified and stored in one `SXFform`, we must be able to split this information up and perform the interpolation independently. This splitting is supported by the `CHOPtrack` data structure. Each `CHOPtrack` stores a list of `IMPobjs`, defining which interpolation method is to be used for each part of the `SXFform`. Each chop has as many `CHOPtracks` as there are `COREcores` to a `SXFform`. Since the `SXFform` used in a chop depends on the type of the chop, the number of tracks used in each chop depends on the type of the chop. The `CHOP_ctpts` data structure, which is part of the `CHOP_guts`, is used to store all the `CHOPtracks` for the given chop.

The `CHOP_ctpts` data structure also stores an array of pointers to all the ctpts of its chop. Notice that the concept of tracks allows us to keep the ctpts as single entities, i.e. we do not need to split the ctpts up into their `COREcores` to be able to perform different interpolation methods on the `COREcores` of a ctpt. See also Figure 3.8 on page 47 for an illustration of a sample `CHOP_ctpts` data structure.

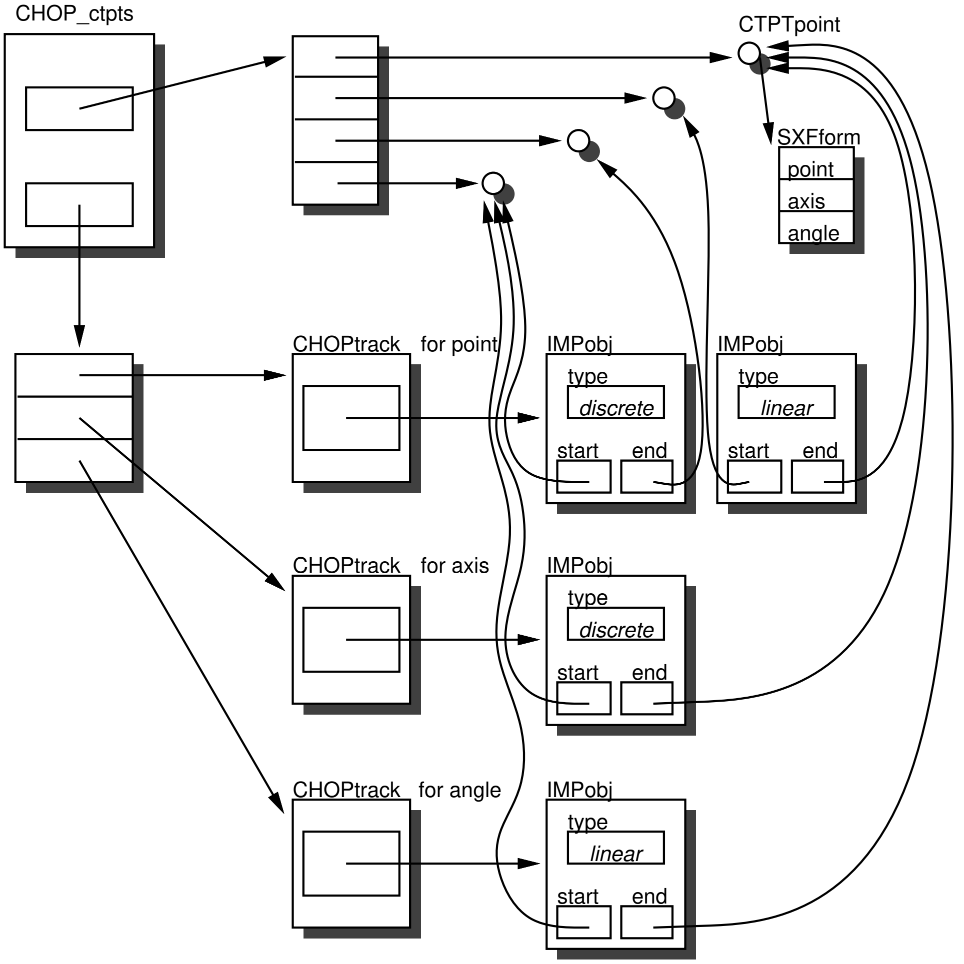

Figure 3.8: The `CHOP_ctpts` data structure.

3.6.2 The `CHOPscope` Data Structure

The UGA system supports the notion of *scopes*. A scope is a semantically-related group of chops in the database. For example, scopes are used to group chops that specify Newtonian forces applied to an object. These chops must be grouped because they require a common concept of *history*. More specifically, their effects depend on the velocity and acceleration of the object, quantities that must be integrated over time. This use of scopes will be discussed extensively in Section 3.7.5 (Traversal for Dynamic Simulation, page 62). Scopes could be used for other purposes, too. A controller, for example, might add chops to a controlled object and need at a later point to find only the chops it added (as opposed to chops added by other sources). The controller could put its chops into a scope, and assume that no other sources add chops to that scope.

Each chop in a script belongs to a certain scope. The scope information is stored in the `CHOPscope` data structure. Via the `CHOPscope` data structure it is possible to identify all chops belonging to this particular scope, regardless of which object the chops belong to.

A scope contains the head and tail of a linked list of chops, i.e., the first and last chop of the scope. This linked list is ordered from lowest priority to highest priority within the scope. In addition the scope is flagged with a type. Currently UGA supports two types of scopes, `CHOP_S_DYN` and `CHOP_S_STAT`.

Support of only two scope types does not mean that a script is partitioned into two scopes, as scopes of different types can alternate consecutively. It is even possible to have scopes of the same type be consecutive.

In general, dynamic scopes (scopes of type `CHOP_S_DYN`) are used to implement physically-based forward simulations, whereas static scopes (scopes of type `CHOP_S_STAT`) are used for kinematic animations. The mechanisms implemented for a dynamic scope are general enough to be used to simulate a static scope, yet for efficiency reasons this is not desirable. Therefore static scopes are normally used when relative position specifications for the objects are sufficient. The mechanisms of a dynamic scope allow the user to specify forces on objects, which can be integrated into velocities and further into positions. For further discussion of the differences between dynamic and static scopes see Section 3.7 (The STATE Package, page 52).

3.6.3 The `CHOPinvalids` Data Structure

`CHOPinvalids` are used to store invalidation intervals. This invalidation information needs to be returned to all dependent data structures in the callback list. The dependent data structures use the `CHOPinvalids` to properly invalidate any data they may have cached that has become invalid as a result of the changes in the chop. These data structures therefore need to know over what time interval data was changed.

The `CHOPinvalids` structure stores at most one invalidation interval. One `INTinterval` is sufficient as it can be complex; see Section 3.3 (The INT Package, page 32). It is also possible that no invalidation interval is stored, if the data is not invalidated but *mapped.* For this purpose the `CHOPinvalids` structure can also store a list of mappings (see Section 3.3, The INT Package, page 32). It is possible that a `CHOPinvalids` structure will store both an invalidation interval and a list of mappings. An example of how mappings can be used to retain cache values was given in Section 3.3 (The INT Package, page 32).

Mapping is preferred to invalidating as it is often more efficient. However, as can be seen from the above mentioned example in Section 3.3 (The INT Package, page 32), mapping can only be performed when a number of conditions are met. The conditions are:

1. Only the time component of a ctpt is changed.
2. The change in the time component is small enough so that the order of the ctpts in the chop is not changed.
3. The `IMPobj` containing the ctpt is of type discrete.
4. The preceding chops do not interfere with the mapping interval.

The last condition is hard to verify. It is for this reason that mappings are currently not used in UGA.

3.6.4 Caches and Their Invalidation

Caches and the concept of caching have been mentioned throughout the document. Therefore we will not repeat the reasons why caches are essential to UGA.

A cache is an entity that stores data associated with times or intervals. Notice that this concept of a cache fits nicely into the chop paradigm. For efficiency, our caches can only use discrete interpolation methods. Other than this restriction on interpolation methods, and the flag that denotes that a chop is a cache, cache chops are the same as regular chops.

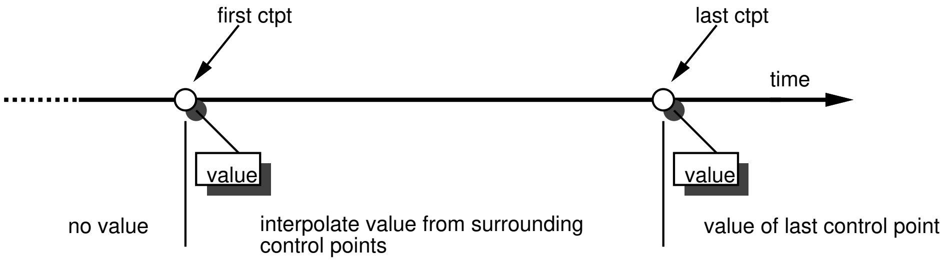

Figure 3.9: Persistent values for regular chops.

Cache values normally are valid only up to a certain time, whereas normal chops usually have *persistent* values. That means that in general when a chop has its last ctpt at time t, the value of that ctpt persists from t to infinity. The value of the chop is defined at every time between time t and infinity to be the value of the last ctpt (see Figures 3.9 on pages 50). This approach was chosen to ensure that all changes are persistent and thereby avoids discontinuous behavior. For example, if the last ctpt on a translate chop specifies a translate value of $[3,7,8]$, then this value should continue to influence its object until infinity, rather than letting the object snap back to the origin.

By convention we use ctpts as delimiters of intervals. Thus, whenever it is necessary to specify data that is only valid over certain time intervals, i.e. it is non-persistent, we have to insert two new ctpts. The first ctpt is inserted at the beginning of the interval the data is valid over. The second ctpt, inserted at the end of that interval, is needed to limit the effect of the first ctpt. The second ctpt will have no value associated with it. Nonetheless it is of type `CTPT_TYPE_VALID`. See Figure 3.10 (page 51) for an illustration. Caches very often store non-persistent data, as most cached data is only valid for a certain interval. Non-persistent data is installed in a chop by calling `CHOPset_range()`. The function `CHOPset_range()` actually also removes any previously existing ctpts between those two newly inserted ones to insure the desired effect, as it can not assume that it is called with a newly created chop.

Cache chops can be invalidated over an interval of time with the routine `CHOPinval_interval()`. Invalidating an interval is analogous to setting a value in a cache over an interval (see above). The only difference is that the

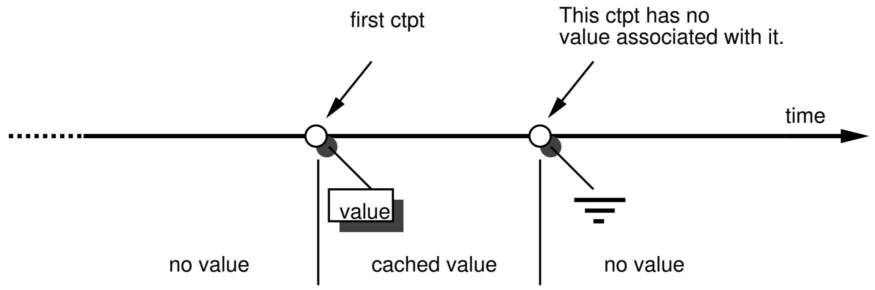

Figure 3.10: Non-persistent values for cache chops.

value stored for invalidating is a no-value token, implemented via a `NULL` pointer.

3.6.5 Data Inquiry

The functionality for inquiring a value from a chop is implemented by the `CHOPinq()` procedure. The function `CHOPinq()` returns a `CHOPeval` data structure as result. The `CHOPeval` data structure consists of a `SXFform` and an `INTinterval`. The `SXFform` is needed to be able to communicate the result of the inquiry to the caller of the routine. The `INTinterval` describes over what time interval the returned result is valid.

To compute its return value `CHOPinq()` loops through all tracks of the chop. For each track it finds the spanning `IMPobj`. The spanning `IMPobj` is defined to be the `IMPobj` of the inquired chop whose start and end ctpts bracket the inquiry time. The function `CHOPinq()` calls `IMPinterpolate()` to return the interpolated value at the inquired time, using the spanning `IMPobj`. The returned value is inserted into the `SXFform` that is to be returned as part of the result of `CHOPinq()`. IMP not only computes the actual value, but also what interval this value is valid over. These intervals are intersected and returned to the caller. The returned interval is, for example, used in STATE to decide what the valid interval for a cache is.

It is possible for `CHOPinq()` to return a `SXFform` of value `NULL` to show that no valid data is stored in the chop (useful to denote an invalid cache).

3.7 The STATE Package

The STATE package provides the mechanisms for maintaining and interpreting the "state" of an object. An object's state is determined by the chops that have been applied to it. The STATE package stores the list of chops that apply to an object in a `STATEstate` structure.

The most important operation performed by STATE is determining the value of an attribute of an object at a specific time. As mentioned in Section 2.2.3 (Inquiring Data, page 16), this operation is called a *state traversal* because it involves stepping through the chops in a state to figure out how they affect the attribute. The value of an attribute is accumulated in a `SXFform` in a `CHOPeval` structure. The `CHOPeval` also contains the validity interval which specifies the time period during which the accumulated data is valid.

Some of the chops in the `STATEstate` are caches. These caches make state traversals more efficient by storing precomputed data. Caches add essentially no semantic meaning to the state. In other words, a state traversal would produce the same results without the caches, but it would be slower in most cases. The only semantics associated with a cache have to do with invalidation, which will be described later in this section.

3.7.1 Data Structures

The chops in a `STATEstate` can be grouped into *scopes*. A scope is a set of semantically-related chops, as mentioned in Section 3.6 (The CHOP Package, page 43). The STATE package stores the chops of a scope in a `STATEscope` structure. The `STATEstate` structure stores a list of `STATEscope` structures. The scopes maintained by STATE are related to the `CHOPscopes` maintained by CHOP. The difference is that a `CHOPscope` can contain chops that belong to several objects. A `STATEscope` corresponding to such a `CHOPscope` stores only the chops from the scope that belong to a particular object.

Recall from Section 2.2.2 (Adding a Chop, page 15) that the priority of a chop is essentially the line number on which the chop occurs in the script describing the current model. While every chop has a priority associated with it, the priority does not always make a difference. The `constants` field of the `STATEstate` structure points to the list of chops for which priority is not important. This kind of chop is discussed in Section 3.7.3 (The Importance of Chop Priorities, page 56).

Each of the chops in an object's state is abstract in that it specifies operations that can be interpreted differently by objects from different classes. To interpret chops, a set of routines known as *chop methods* are used. The set of chop methods is determined by the current class of the object. The `STATEfunc_table` structure stores the set of chop methods currently being used to interpret the chops in a particular state. Other methods are also stored in the `STATEfunc_table`, as will be described in Section 3.8 (The GOBJ Package and Object Classes in General, page 66). The `STATEfunc_table` structure is stored in a `STATEbehavior` structure. The `STATEstate` structure for an object points to the `STATEbehavior` structure currently being used to interpret the chops associated with that object.

It is possible for one chop in a state to change the interpretation of any chops that follow it. A chop makes such a change by specifying changes to the `STATEbehavior` structure associated with the state. The most common example is the `rep` chop that appears at the beginning of all the example FLESH scripts in this document. It typically has one ctpt whose value is the name of an object class. When a `rep` chop is evaluated, it sets the `STATEbehavior` referenced by the current `STATEstate` to the `STATEbehavior` structure associated with the specified object class. More details about object classes and their methods will be given in Section 3.8 (The GOBJ Package and Object Classes in General, page 66).

The `STATEref` structure represents a reference to an object[4] A chop that references an attribute of an object contains a ctpt whose value is a `STATEref` (i.e., the `VALUitem` associated with that ctpt has a `VALU_pointer` that points to the `STATEref`). This `STATEref` has fields that point to the chop making the reference and the referenced object's state. The `STATEeval_ref()` routine uses these fields to evaluate the `VALUitem` that makes the reference. One subtle detail of reference evaluation will be described later in Section 3.7.3 (The Importance of Chop Priorities, page 56).

3.7.2 Standard State Traversal

This section describes the basic details of the state traversal process. For simplicity, it is assumed that all the chops in the state are in a single scope.

[4] The representation of references is likely to change in the near future to provide increased control over referenced information.

It is also assumed that no dynamic simulation is being performed. The more elaborate case of multiple scopes and dynamic simulation will be considered in Section 3.7.5 (Traversal for Dynamic Simulation, page 62).

State traversal is handled by the routine `STATEtraverse()`. A state traversal is made in response to some client's inquiry for some data (attribute) of an object. The various specific ways that a client can make an inquiry are not important here; their description is saved for Section 4.6 (More Details of the STATE Package, page 84). A data inquiry asks STATE to compute a specific data value as of a specific inquiry time, scope and chop priority within that scope. The scope and chop priority specify where the traversal will stop, i.e., the last chop in the state whose effects will be considered. All chops that are considered are evaluated as of the inquiry time. The inquiry time, scope and priority are stored in a `STATEinquiry` structure passed to the inquiry routine.

The first step of a traversal is to look for any caches that can reduce the amount of work to be done during the traversal. The search for appropriate caches starts at the chop nearest the priority specified by the inquiry. This chop is the chop with the highest priority that is less than the inquiry priority. Since the chops in a state are sorted by priority, a binary search for the inquiry priority is used. After establishing this starting position, the traversal routine looks forward and backward in the state to see if there are any valid caches that will satisfy the data inquiry. Looking backward may find a cache that was built at an earlier priority. If no chops between that cache and the starting position invalidate the cache (by changing the data it stores), the cache can be used. In addition, it may be possible to find a valid cache by looking forward in the state. As an example, say the last chop that modifies an object's polyhedral boundary representation (b-rep) is at priority 100. Then the b-rep will be valid from priority 100 onward until the end of the state. An inquiry for the b-rep at priority 200 will cause a cache to be built at priority 200. A subsequent inquiry at priority 150 will be able to use the value cached at priority 200. If a valid cache is found, then its data will be copied and returned, and the traversal terminates.

If a cache that immediately satisfies the inquiry cannot be found, a valid cache at priority less than the inquiry priority will still be useful. The traversal can start at that cache and consider only the chops between that cache and the inquiry priority. If no such cache is available, then the traversal must start with the lowest priority chop in the state.

When traversal begins, the object's `STATEbehavior` must be set. The set of methods in this table is determined by the class of the object, and this class can be affected only by the chops of lower priority than the initial traversal position. If the initial traversal position is at the beginning of the state (i.e., the beginning of the scope for the purposes of this section), then the initial `STATEbehavior` is assumed to be that of the general object class, "gobj"; this class is described in Section 3.8 (The GOBJ Package and Object Classes in General, page 66).

The chops in the scope must now be visited in order of increasing priority. As each chop is encountered, the appropriate chop method in the `STATEbehavior` for the type of chop is called. The chop method is passed the `STATEstate`, the chop (a `CHOPchop`), the time of the inquiry and the `CHOPeval` structure that holds the data being gathered and its validity interval.

Any other data needed by the chop method may be obtained by a recursive call to `STATEtraverse()`. As an example, consider what a deformation chop does when a `TRIPobject` is being inquired. Deformations warp an object's shape in world space. The deformation chop thus needs to know how the object-space information in the `TRIPobject` relates to world space. This relationship can be determined only if the current transformation matrix (CTM) for the object is known. The deformation chop obtains this CTM by making a recursive inquiry.

The chop method modifies the `SXFform` in the `CHOPeval` to add the effects of this chop to the data being collected. The method bases its effects on the value interpolated from the chop's ctpts; this value is obtained by calling `CHOPinq()`. The method also modifies the `INTinterval` in the `CHOPeval` that stores the validity interval. The new validity interval is the intersection of the existing validity interval and the interval returned by `CHOPinq()`. This returned interval describes the time period over which the chop's effects are valid, and it is determined by the IMP package as described in Section 3.5 (The IMP Package, page 38). The new validity interval is thus the time period over which the effects of all the chops visited so far are valid. The chop method can also change the `STATEbehavior` of the object. A deformation chop, for example, makes this sort of change; more details of this sort of change will be given in Section 3.8 (The GOBJ Package, page 66).

Traversal continues until the initial inquiry priority has been reached, at which point the gathered data is returned. If, during the traversal process, any caches of the same data type as the inquired data are encountered, then they are updated. A graphical depiction of the basic traversal process is given in Figure 3.11 (page 57). This Figure shows an inquiry of the color of the object `zelda` described in the FLESH script in Figure 2.2 (page 14).

3.7.3 The Importance of Chop Priorities

During the traversal of a state, chops with a low priority value are visited before chops with a high priority value. This ordering of chops is important because the functions represented by chops do not necessarily commute. The standard example involves translation and scaling (about the origin): scaling and then translating does not produce the same results as translating and then scaling, as shown in Figure 3.12 (page 58).

It is possible to bypass the priority ordering of chops in special situations. When a chop is added to a state, the chop can be designated a *constant* chop. The `constants` field in the `STATEstate` structure points to a list of constant chops associated with that state. The list of constant chops is checked at the beginning of the state traversal process. If any of these constant chops are of the same type as the inquired data, then the value of that constant chop is evaluated as of the inquiry time and returned. This "short-cut" mechanism makes a number of inquiries more efficient. The "resolution" of an object (a parameter that affects the number of faces in a polyhedral approximation to the object), for example, is usually specified completely by one chop. The chop that specifies the resolution value can be accessed immediately if it is placed in the list of constant chops. Since the short-cut mechanism causes a constant chop to override any normal chops in the state, constant chops must be used with care. It is up to the client that adds a constant chop (by calling `STATEadd_constant()` instead of `STATEadd_chop()`) to ensure that the constant chop is really appropriate.

Evaluating a reference involves inquiring data from the referenced object. The priority used in this inquiry comes from the priority of the chop containing the reference. If the chop has priority π, the inquiry will be made with priority $\pi - 1$. To see why this priority is used, consider a self-referential chop, e.g., a chop that tells an object to rotate around its own current position. Evaluating this chop will cause a recursive inquiry to get the object's position. A rotation affects an object's position, so this recursive inquiry cannot be as of the chop's priority π; the result would be another attempt

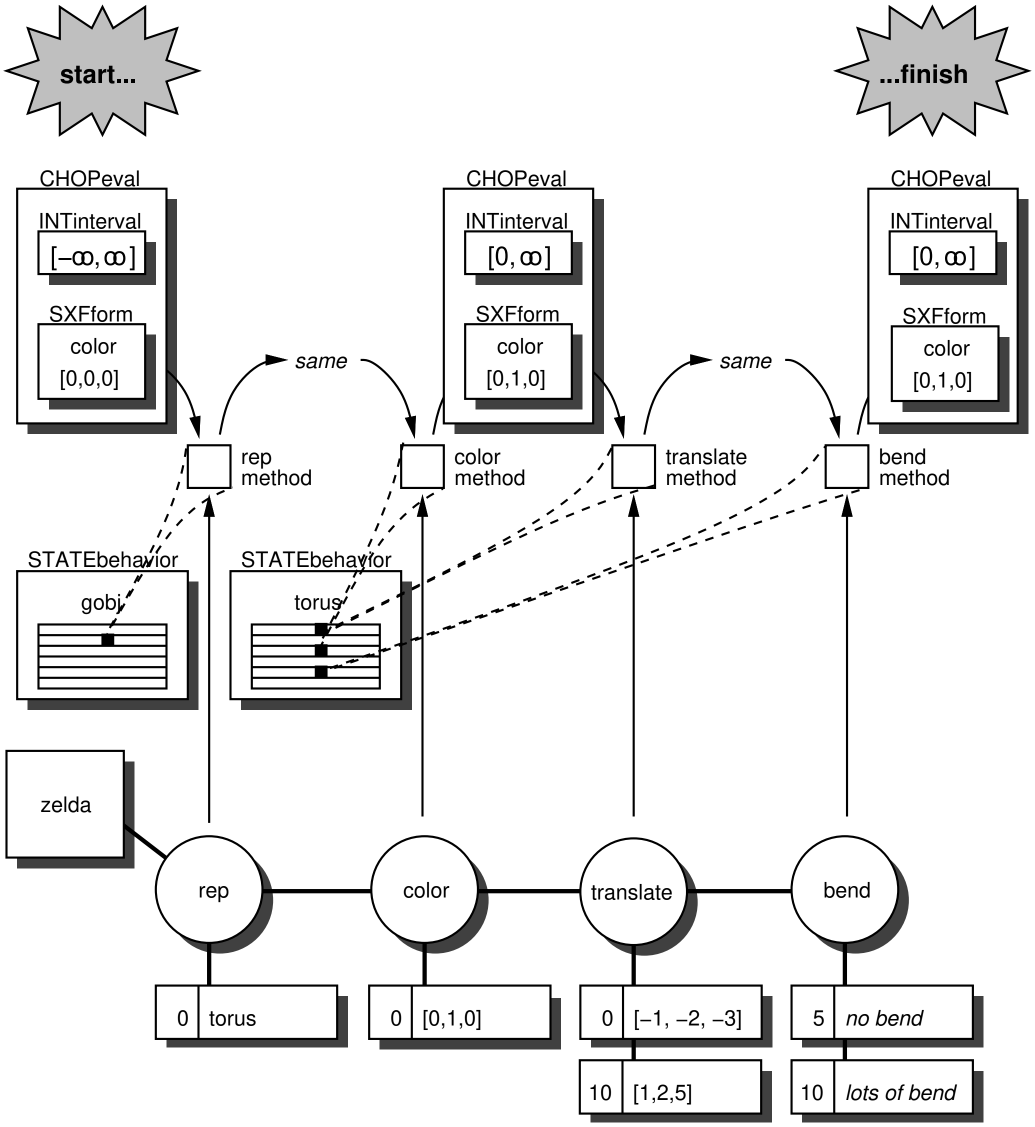

Figure 3.11: The state of object `zelda` being traversed. The state was described in the FLESH script in Figure 2.2 (page 14). The data being inquired is color as of time 0 and the priority of the end of the state. (It is assumed that there are no caches so traversal starts at the beginning of the state.)

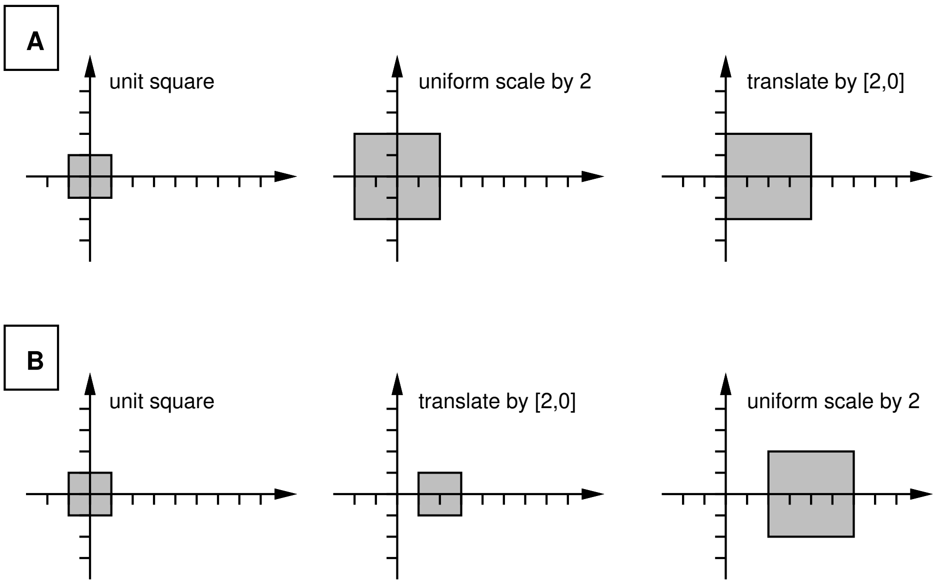

Figure 3.12: The operations of translation and scaling (about the origin) are not commutative. Figure A shows the results of scaling first, and Figure B shows the results of translating first.

to evaluate the chop, which would cause another recursive inquiry, leading to infinite recursion. Inquiring at priority $\pi - 1$ will allow all the chops up to the rotation chop to affect the object's position, causing the correct position to be computed.

It is sometimes useful to be able to treat the actions of several chops as an atomic operation. Such a set of *parallel* chops are made to act as a unit by giving them all the same priority. Parallel chops make it easier to implement a controller that detects collisions and makes appropriate responses. This controller uses a special `newtonian` chop to communicate responses to the objects it controls. The `newtonian` chop sets all the properties of an object related to newtonian dynamics; for this discussion, the relevant properties are the object's position and its momentum. A sample script is shown in Figure 3.13 (page 59).

The controller `preventer` will be called whenever the `newtonian` chop of either `holmes` or `moriarty` is evaluated. At this time, the controller will

```
holmes    : rep       0 = sphere ;
moriarty  : rep       0 = cube   ;
preventer : rep       0 = collision_controller
            /* detect collisions amongst "member" objects */
            member    0 = [SET, [holmes, moriarty]]
;
start_parallel
   holmes   : newtonian 0 = preventer.newtonian(holmes)   ;
   moriarty : newtonian 0 = preventer.newtonian(moriarty) ;
end_parallel
```

Figure 3.13: A FLESH script fragment showing a collision-detection-and-response controller that controls two objects.

inquire the positions of its "members" and check for interpenetration; if it finds any, it will translate the colliding objects (to counteract interpenetration) and change their momentum so they will bounce off each other. If the `newtonian` chops for `holmes` and `moriarty` do not have the same priority—say `moriarty`'s chop has priority π and `holmes`'s chop has priority $\pi - 1$—then `moriarty` will not receive the correct momentum transfer. Specifically, consider what could happen when `moriarty`'s `newtonian` chop is evaluated. This chop references `preventer`, so an inquiry will be made to `preventer` with priority $\pi - 1$. This priority will be used by `preventer` when it inquires the positions of `holmes` and `moriarty`. Inquiring `holmes`'s position at priority $\pi - 1$ will cause its `newtonian` chop to be evaluated. This chop's effects will update `holmes`'s position to counteract the collision, and the collision will disappear before `moriarty` gets a chance to respond to it! A solution is to give both `newtonian` chops the same priority π. Then `preventer` will inquire their positions at priority $\pi - 1$ and the collision will be detected when each `newtonian` chop is evaluated.

3.7.4 Invalidation and Revalidation

As part of maintaining cached data, STATE is also responsible for invalidating data caches. A cache in a state stores a value that is dependent on the chops of lower priority. If a lower-priority chop is added, removed or modified, the cached value may become invalid. For example, if a translation chop is added in at a lower priority than a CTM cache, the CTM cache must be invalidated.

The `CHOPchop` structure maintains a list of states that share a chop. Whenever the chop is changed, the callback routine `STATEchop_changed()` is invoked. This routine is called once for each `STATEstate` that shares the chop. As mentioned in Section 3.6 (The CHOP Package, page 43), the routine is passed a `CHOPinvalids` structure. This structure contains an `INTinterval` that describes the period of time over which the change has effect; this period is known as the *invalidation interval*.

The `STATEchop_changed()` routine starts by obtaining the initial *invalidation set*. This set consists of all types of data that are affected by the chop that was changed. For example, changing a deformation chop will invalidate any b-reps (`TRIPobjects`) cached after the chop; it will also invalidate any cached CTMs, since a deformation can change the position of an object in space. Notice that the contents of the invalidation set depends on the interpretation of the changed chop. Remember also that the interpretation of the chop depends on the object's class and is encoded in the methods of the `STATEbehavior`. Consequently, each object class must define a table to give the invalidation set for each type of chop. This table is an array of type `STATEbuf_tbl` in the `STATEfunc_table` structure mentioned in Section 3.7.1 (Data Structures, page 52). This array contains an entry for each type of chop (i.e., each type of `SXFform`). The `STATEbehavior` that contains the `STATEfunc_table` is obtained by calling `STATEsetup_method()`.

The `STATEchop_changed()` routine next traverses the chops in the state, starting at the chop that changed. At each cache chop encountered, the `CHOPinval_interval()` routine is called. This routine invalidates the cache if it is affected by the invalidation interval; details of this process are described in Section 3.6 (The CHOP Package, page 43).

References between objects make invalidation somewhat tricky. If object A references an attribute of object B, and that attribute of B is changed, then the caches in A after the point of the reference could become invalid. This problem is solved by introducing a cache in the state of B. This cache stores the value of the attribute of B referenced by A. The cache also stores in its callback list a pointer to the chop from A that references B. This arrangement of references (for the particular case in which A is referencing B's color) is depicted in Figure 3.14 (page 61). When the referenced at-

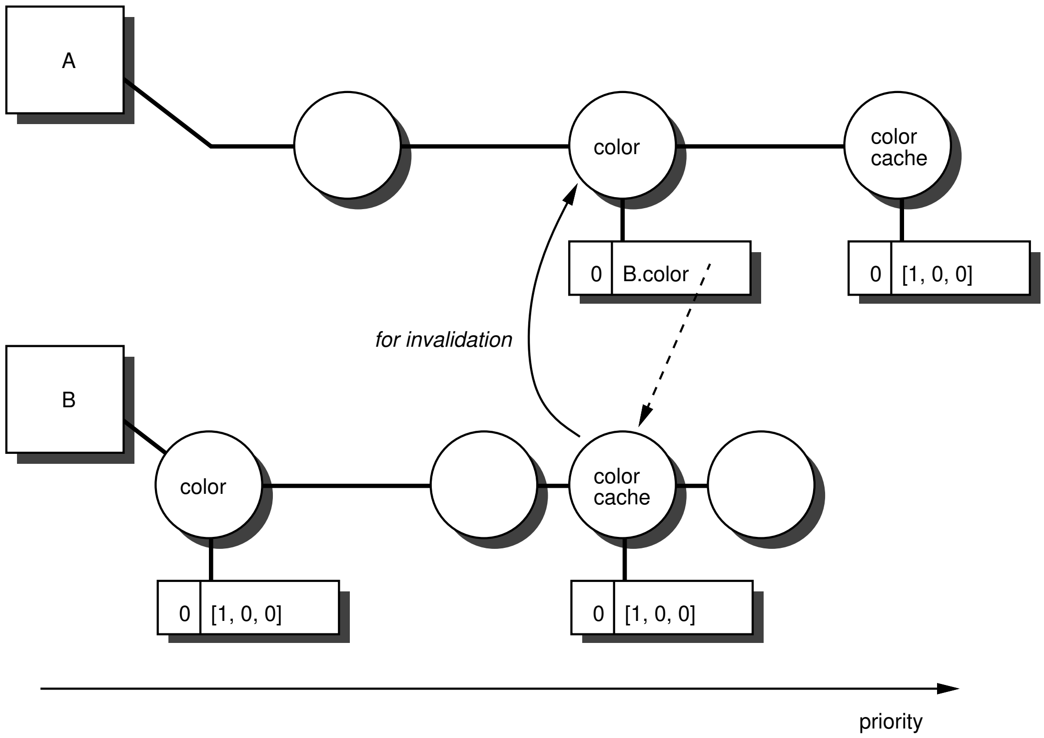

Figure 3.14: The color of A is a reference to the color of B. To support invalidation, B contains a cache that points to the referencing chop in A.

tribute of B changes, the cache in B will be invalidated. The referencing chop of A will be found via the pointer, and it will also be considered to have changed. The caches of A affected by that referencing chop are then correctly invalidated.

It is possible for `STATEchop_changed()` to encounter a chop that makes any further invalidation unnecessary. Such a chop is said to *revalidate* the invalidated data. Any chop that makes an absolute change to an attribute will revalidate that attribute. A color-specification chop, for example, makes an absolute change (as mentioned earlier in this section). Consider a change to the first of two color-specification chops. Any color caches between the two chops will be invalidated. But a cache after the second chop is affected only by the second chop. This cache should thus not be invalidated. The second color chop has revalidated any subsequent color caches.

Like invalidation, revalidation depends on the interpretation of the chops, and hence on the object class. The `STATEfunc_table` structure contains a second `STATEbuf_tbl` array that stores class-specific revalidation information.

3.7.5 Traversal for Dynamic Simulation

Dynamic simulations involve a sense of "history" that is not present in the standard UGA state traversal process. Consider a set of balls that have forces applied to them at time 0. If the balls are allowed to bounce off one another, the position of any ball at time $t > 0$ can be computed only by simulating the motion of all the balls from time 0 to time t. The standard traversal process has been extended to handle this sense of history. A *dynamic scope* is used to designate chops to which the extended traversal processes should be applied. Dynamic scopes will cause simulation over time to take place automatically for any inquiry time t.

The difficult part of the simulation is the integration of a ball's acceleration (computed from its net force and torque) and velocity to get its position and orientation. This integration must be performed numerically because the closed-form function for the ball's acceleration is not known (we only know specific accelerations at a few specific sample times). The simplest form of numerical integration uses Euler's method [PRES88]. Euler's method approximates the integral of a function over an interval of parameter values as the sum of the functions value at many parameter values along that interval. This form of integration is not guaranteed to be particularly accurate, but it is simple to implement and hopefully accurate enough for many simulations. It is also the basis of more accurate methods such as the Runge-Kutta methods.

Integration by Euler's method can be used as follows. If the acceleration and velocity of an object at time t are known to be a_t and v_t, respectively, then the velocity of the object at time $t + \Delta t$ can be approximated as

$$v_{t+\Delta t} \approx v_t + a_t[(t + \Delta t) - t] = v_t + a_t \Delta t. \tag{3.1}$$

(Rotational quantities are being ignored for the sake of clarity.) Similarly, if the position, velocity and acceleration of an object at time t are known to be p_t, v_t and a_t, then the position of the object at time $t + \Delta t$ can then be approximated as

$$p_{t+\Delta t} \approx p_t + v_t \Delta t + \frac{1}{2} a_t (\Delta t)^2 \tag{3.2}$$

Note that the positions, velocities and accelerations mentioned are really vector quantities.

Given initial conditions p_0, v_0 and a_0, the position p_t can be computed iteratively for any $t > 0$. Assume that $\Delta t = 1$. The first iteration computes p_1 directly using Equation 3.2. To compute p_2 during the next iteration, v_1 is needed; it can be approximated from v_0 and a_0 using Equation 3.1. At the ith iteration, v_{i-1} is computed from v_{i-2} and a_{i-2} using Equation 3.1, and then p_i will be computed from p_{i-1}, v_{i-1} and a_{i-1} using Equation 3.2. It is assumed that the acceleration is known *a priori* at each time step; since force is the product of mass and acceleration, this assumption is equivalent to assuming that the forces are known at each time step.

This kind of integration is performed by a "dynamic" chop within the dynamic scope. When the position of an object (as expressed by the object's CTM) is inquired, the state traversal will enter the dynamic scope and encounter the dynamic chop. The dynamic chop's method will look through the dynamic scope for all chops that apply accelerations and velocities to the object. It will then use Euler's method to iteratively compute the object's position. The dynamic chop uses the same algorithm to compute the object's velocity if it is inquired. Note that the values of the accelerations and velocities used in Euler's method can be specified by any other chops in the dynamic scope, but these chops do not actually participate in the execution of Euler's method (so they do not make any direct changes to the CTM or velocity when they are inquired).

It would be inefficient to require the dynamic chop method to restart its integration from time 0 every time a position is inquired. If a dynamic simulation had been run as of a time not long before the current inquiry time, it would be more efficient to start the integration at that earlier time. The STATE package uses a caching convention to make this efficient integration strategy possible.

Each iteration of the Euler's method algorithm will cause a cache to be produced. The cache will store the position p_t and velocity $v_{t-\Delta t}$ computed during this iteration. Notice that the velocity is one time step "behind" the position as explained above. This cache will be placed at the end of the dynamic scope. When the next iteration (at time $t + \Delta t$) encounters the beginning of the scope, it will retrieve the values from the cache. The cached $v_{t-\Delta t}$ will be used to approximate v_t; this new v_t, the cached p_t and the a_t known *a priori* will then be used to approximate $p_{t+\Delta t}$. The cache at the beginning of a dynamic scope provides a semblance of physical continuity between iterations. In other words, physical reality is approximated only if a position and velocity have the same value at the beginning of a time step that they had at the end of the previous time step. Figure 3.15 (page 65) illustrates this approximation of continuity. The traditional UGA evaluation model does not support this continuity: the value of an object's attribute at an inquiry time is defined by the effects of the object's chops as of the inquiry time only. Extending UGA to handle iterative physical simulations thus required the new caching convention.

The routine `STATEtraverse_dynamic()` coordinates a traversal through a dynamic scope. One of its important duties is to help the dynamic chop's method start the iteration for Euler's method. When an inquiry at time t_i for the CTM of an object invokes `STATEtraverse_dynamic()`, the routine looks for the caches at the end of the dynamic scope. It finds the cache at the latest time t that is earlier than t_i, because the iteration can begin at that time; if no cache is found, the iteration must start at time 0. The `STATEtraverse_dynamic()` routine places the `SXFform` obtained from the cache (or an identity `SXFform` if no cache could be found) in the `CHOPeval` structure that was passed in to hold the inquiry's results. The routine then starts traversing the dynamic scope. When the dynamic chop's method is invoked, it will get from the `SXFform` in the `CHOPeval` the p_t value it needs to compute $p_{t+\Delta t}$ using Equation 3.2.

The values for v_t and a_t are also needed to use this equation. These values must be recursively inquired using a standard STATE inquiry routine. The inquiry for v_t will recursively invoke `STATEtraverse_dynamic()`. At the start of this traversal, `STATEtraverse_dynamic()` will again look at the caches at the end of the dynamic scope for the value $v_{t-\Delta t}$. If it is found, it will be used to initialize the `SXFform` in the `CHOPeval` as described earlier. When the dynamic chop is encountered, its method will be called to compute v_t using Equation 3.1. The value $v_{t-\Delta t}$ will be available as the initialized `SXFform` value, so the computation can be completed with no more recursive inquiries. Finally, the inquiry for a_t will be resolved by recursively traversing the scope and calling the methods of any chops that affect acceleration. Note that the dynamic chop does not affect this inquiry

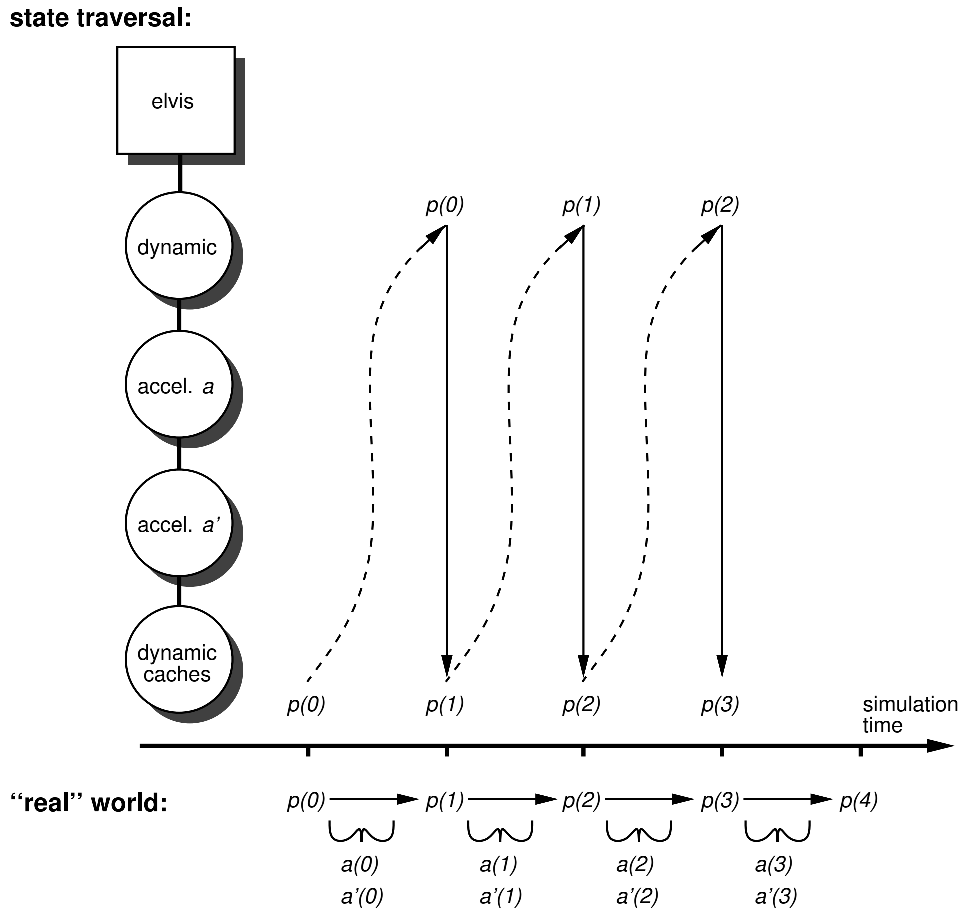

Figure 3.15: The top part of this diagram shows how the position of `elvis` at time 4 can be integrated by three passes over a dynamic scope. (The position at time t is denoted $p(t)$.) Two accelerations, $a(t)$ and $a'(t)$, create the change in position. The bottom part of this diagram shows how each of the passes corresponds to a discrete interval of simulated time.

at all because no integration needs to be performed.

When the original traversal of the dynamic scope is complete, the caches at the end of the scope will be updated to contain p_t and $v_{t-\Delta t}$. The dynamic chop's method is then ready to advance to the next step of the iteration. Before doing so, the time step Δt should be adaptively updated to ensure that it is not so big that inaccuracy is introduced or so small that efficiency is compromised. Updating Δt is not a simple process, however, and requires information about all the operations being performed by chops in the state. Mechanisms to correctly update Δt have not yet been added to the current implementation of UGA. Clients of UGA can overcome this problem, however, by keeping track of their own time step. A client can explicitly inquire position at each of the sample points dictated by its own time step. At the start of each such inquiry, `STATEtraverse_dynamic()` will find the cache built at the end of the previous inquiry, so continuity between time steps will be maintained automatically. Whatever specific mechanism is used, repeated passes are made through the dynamic scope to simulate the cumulative effect of accelerations (forces) over time.

When `STATEtraverse_dynamic()` services the recursive inquiry for v_t, it actually does more than initialize the `SXFform` in the `CHOPeval` with $v_{t-\Delta t}$ and let the dynamic chop's method compute v_t from Equation 3.1. It also inquires the value of velocity v'_t as of time t and the priority of the *beginning* of the dynamic scope. This other velocity v'_t represents the effects of any chops in scopes *preceding* the dynamic scope on the velocity at time t. The sum $v_t + v'_t$ is then returned by `STATEtraverse_dynamic()` as the complete velocity at time t. Modifying the values that were computed during the previous iteration may seem like a strange idea, because it is physically unrealistic (it is equivalent, in a sense, to changing the initial conditions of the simulation at each step). This capability is supported because the "unrealistic" effects obtained may be important for artistic reasons. The values of p_t and a_t are similarly modified by inquiries of the position and acceleration as of the beginning of the dynamic scope.

3.8 The GOBJ Package and Object Classes in General

The GOBJ ("*g*eneral *obj*ect") package implements the "gobj" object class, the default object class in UGA. The gobj class currently represents a "point" object. Such an object has no volume or surface. Point objects are primarily useful as the control points of a path object. Representing control points as full-fledged UGA objects promotes orthogonality; any animation technique that can control the position of a more standard object like a cube can be used to modify the control points (and the shape) of a path. Path objects are described in more detail in Section 5.2 (Paths, page 91).

A more important aspect of the gobj class is its role as the prototypical object class in UGA. Every other object class is based on the gobj class. Correspondingly, the package that implements any other object class is based on the GOBJ package that implements the gobj class. The GOBJ package thus holds special importance in UGA, and is considered part of the UGA kernel.

3.8.1 Features of All Object Classes

An object class is basically just a collection of methods. An object class's methods allow it to interpret the abstract chops, inquire data, and perform other class-specific operations (e.g., intersect a ray with an object belonging to the class).

Each class defines several arrays of pointers to its methods. These arrays are stored in a `STATEfunc_table` structure. The `STATEfunc_table` structure is stored in a `STATEbehavior` structure. These structures are part of the STATE package because methods are used during the state traversal process, as described in Section 3.7 (The STATE Package, page 52)[5].

Four of the fields of the `STATEfunc_table` have already been described in this document. An inquiry method is called to obtain data from an object, as mentioned in Section 2.2.3 (Inquiring Data, page 16). The `STATEfunc_table` contains an array of these methods, indexed by the `SXFform_type` corresponding to the data. A chop method is called to evaluate a chop, as described in Section 3.7 (The STATE Package, page 52). An array of these methods is also contained in the `STATEfunc_table`, as already described. Invalidation information is also stored in an array in the `STATEfunc_table`, as is revalidation information; these arrays were also mentioned in Section 3.7.

The `STATEfunc_table` also contains an array of *edit methods*. A class can associate one of these methods with each `SXFform_type`. The edit methods are meant to support the high-level editing operations mentioned in Section 2.2 (Database Overview, page 12). A high-level operation to edit a particular attribute (i.e., a particular `SXFform_type`) of an object would

[5] The GOBJ package is "higher-level" than STATE, i.e., GOBJ includes the STATE header file but not vice versa. Thus it is not possible for the method table to be a GOBJ data type.

be invoked by calling the appropriate edit method from the `STATEbehavior` referenced by that object. The current version of UGA does not fully exploit edit methods, but their importance is expected to increase as the implementation of UGA is refined.

The final array of methods contained in the `STATEfunc_table` contains the *arbitrary methods.* The methods in this array are meant to handle any class-specific operations that are not handled by any of the other types of methods. The STATE package defines the `STATEarb_meth_type` enumerated type that contains an element for each of the kinds of arbitrary methods that can be specified by an object class. The `STATE_ADD_CHOP` element, for example, gives the index of a chop-adding method in the arbitrary method array. The "agg" object class uses this method to handle the adding of a chop to an aggregate object. An explanation of what the agg class does in this situation can be found in Section 5.1 (Aggregate Objects, page 89).

Each object class has several different *subclasses*[6]. Currently there are three subclasses:

1. implicit, e.g., a sphere represented with an implicit equation $x^2 + y^2 + z^2 = r^2$;
2. deformed, e.g., a bent sphere represented with a polyhedral boundary representation (b-rep);
3. volumetric, e.g., a sphere represented with a three-dimensional voxel array.

The methods associated with one subclass of a class can be different from the methods of another subclass of the same class. A class maintains a `STATEfunc_table` structure for each of its subclasses; these structures are stored in an array in the `STATEbehavior` structure for the class. Thus the different subclasses of a class act much like different classes. Subclasses are supported because they correspond to a standard set of "variations" that are supported by most object classes.

By default, an object is represented by the implicit subclass of its class.

[6]Note that the term "subclass" in this context does *not* mean a derived class in an inheritance hierarchy. The term is not really appropriate, but it is used for historical reasons.

The object's methods[7] decide when to change to another subclass. For example, the deformation chop method for the sphere class changes the object to the deformed subclass. One effect of this change is a switch to a different ray-intersection method for the sphere. The method for the implicit sphere can find the intersection by solving a quadratic equation, whereas the method for the deformed sphere must intersect the ray with a polyhedral approximation to the bent sphere.

3.8.2 The Object Class Hierarchy

The object classes in UGA are organized into an inheritance hierarchy. Inheritance works as follows. An object class that wants to inherit from a parent class P calls the P`new_behavior()` routine. This routine returns a copy of the `STATEbehavior` structure for the parent class. The child class can then replace any elements of the `STATEfunc_table` or `STATEbuf_tbl` that it wants to override.

Note that it is not possible for a child class to add a new kind of method or invalidation/revalidation scheme that does not exist in its parent class. There must already be an entry in the `STATEfunc_table` or `STATEbuf_tbl` for anything added by the child class. The parent class need not have anything meaningful in its entry (e.g., it can have a `NULL` pointer for a method that it does not use), but the entry must still be present. This limitation was deemed acceptable because it makes the implementation simpler.

There are three basic reasons for having a child class overwrite the methods of a parent class:

1. The method is not meaningful to the parent class. For example, a torus has an inner radius (for the hole). The torus class thus needs a method that interprets the chop that specifies this radius. The parent class of the torus class is currently the gobj class, for which the concept of inner radius is meaningless. The gobj class thus does not define an inner-radius method (although it does have space for one in its `STATEfunc_table` so that the torus class will be able to fill it in).

2. The child class can make assumptions that allow the method to be implemented more efficiently. An example is the method that tessel-

[7] It is often convenient to say "the object's methods" instead of "the methods of the object's class." The same convention holds for invalidation/revalidation table information.

lates an object (i.e., breaks its b-rep into many little faces so it can be deformed). The gobj tessellation method must inquire the b-rep and apply a general tessellation algorithm that can handle arbitrary geometries. A cube, however, has a simple geometry that can be tessellated with a simpler algorithm. The cube class thus overrides the general tessellation method.

3. The method has a different meaning for the child class. When a member of the text class is scaled, for example, it might be desirable to have the letter spacing change in a non-linear manner.

The gobj class is the most general object class, so it is at the root of the inheritance hierarchy. Since a child class can only override methods and invalidation/revalidation information from its parent class, the `STATEfunc_table` and `STATEbuf_tbl` for the gobj class contain an entry for every method and every invalidation/revalidation scheme used anywhere in UGA.

3.8.3 Some Important Methods

A few of the methods in the `STATEfunc_table` merit discussion at this point. The most important method for any class is the "chop-behavior" method. For an object of class X, this method is implemented by the X`chop_behavior()` routine. This method is called to evaluate a `rep` chop whenever it is encountered in the state of an object belonging to class X. A `rep` chop always specifies the name of a class, X', as its ctpt; see the rep chops in Figure 3.13 (page 59), for example. The X`chop_behavior()` routine changes the object's class from X to X'. Making this change involves two operations: making a `STATEstate` point to the `STATEbehavior` for a torus, and initializing data specific to class X'.

These two operations are most easily understood through an example. Consider again the object `zelda` from Figure 2.2 (page 14), and assume that a client is inquiring the inner radius of `zelda` as of time 0 and the priority of the end of the script. As mentioned in Section 2.2.3 (Inquiring Data, page 16), the client obtains an inquiry method with a call to the `STATEsetup_method()` routine. To determine the appropriate method, this routine must have `zelda`'s `STATEbehavior` as of the inquiry priority. So `STATEsetup_method()` inquires it. For this inquiry, `zelda` starts as a member of the default class, gobj. When the `rep` chop is encountered, `GOBJchop_behavior()` will be called. This routine will notice that the `rep` chop's value at time 0 is "torus"[8], and will make `zelda`'s `STATEstate` point to the `STATEbehavior` for the torus class. No other `rep` chops will be encountered, so `zelda` will still have the `STATEbehavior` of a torus when the inquiry is complete. When `STATEsetup_method()` returns, the client can retrieve the inner-radius-inquiry method from the `STATEfunc_table` in `zelda`'s `STATEbehavior`.

When this inquiry method is invoked, another traversal of `zelda`'s state will begin. At the start of this traversal, `zelda` will revert to being a member of the gobj class. Thus the `GOBJchop_behavior()` routine will be called when the `rep` chop is encountered. This time, `GOBJchop_behavior()` will make `zelda`'s state point to the `STATEbehavior` for the torus class and then it will call `TORUSchop_behavior()`. This routine will notice that the inner radius is being inquired, so it will put the default inner radius value into the `CHOPeval` (through which the inquiry's results will be returned). If no subsequent chops modify the inner radius, the inquiry will return the default value. In general, the X`chop_behavior()` routine makes certain that an inquiry for any data specific to the class X will get at least the default value.

Another important chop method handles a `parent` chop. When an object has a `parent` chop, it becomes the *extension* of some other *prototype* object. All the chops associated with the prototype now also affect the extension. More details of this idea are described by Zeleznik *et al.* [ZELE91]. The `parent` chop method for the class X is the routine X`parent_chop()`. When it is called during an inquiry of some data from the extension object, the routine inquires that same data from the prototype object. The data from the prototype is then considered to be data from the extension, and the traversal continues on to the next chop.

3.9 The OM Package

The OM ("*o*bject class *m*anager") package keeps track of the object classes being used in UGA. In a sense, OM is not essential to the operation of UGA.

[8] The `rep` chop's ctpt will actually have the torus class's `STATEbehavior` as its value. The FLESH parser converts a class name to the appropriate `STATEbehavior` when it parses a `rep` chop.

It does, however, provide a few convenient services that do not fit directly into the other packages.

The main data structure supported by the OM package is the `OMhandle`. The `OMhandle` structure contains an array of `STATEbehavior` structures, one for each class known to UGA. The array is indexed by the `OMclass_type` enumerated type, which contains one element for each of the object classes. The `STATEbehavior` structure for class X is added to the array by the X`init()` routine (every class X defines such a routine which must be called once when UGA is being initialized).

The OM package allows some object classes to be ignored when an UGA executable is being compiled. This capability is useful for debugging. A "streamlined" version of UGA that includes only one object class is sufficient for debugging operations that treat all object classes in a similar manner. This streamlined version will take less time to link, an important consideration when repeated compilations are needed to find and correct bugs. To support this capability, a cpp preprocessor constant is defined in OM for each object class available in UGA. All code that *explicitly* references a class X (e.g., a call to the X`init()` function) is bracketed by cpp conditionals so that code is compiled only if the OM constant for X is defined. To prevent an object class from being linked into the UGA executable, a programmer need only un-define the appropriate OM constant.

3.10 The REG Package

The REG package allows clients to register interests in specific database events. When one of these events occurs, REG notifies the client through a callback routine.

Currently database events are generated whenever a chop or a state has been modified, added or deleted. However, since most clients only want to know about very specific subsets of the total set of database events, the REG package allows each client to specify an interest in a specific event subset.

To specify an interest in an event subset, a client calls the routine `REGcreate()`. The parameters to this routine define the event subset in which the client is interested. Since the routine takes a variable number of parameters, the client only sends the fields that are necessary to define its interest. The REG package defines a number of event subsets:

- a specific chop, e.g. all changes to the third chop in an object's state;
- a specific state, e.g. all changes to a specific object;
- an object class type, e.g. all changes to all objects of the cube class;
- a chop type, e.g. all changes to all `color` chops;
- a time interval, e.g. all changes occurring during that time interval;
- one or more of a set of actions, e.g. whenever a ctpt or chop is added or deleted;
- a specific tag event (see below).

The `REGcreate()` routine provides clients with a very simple interface for specifying interests of the form of "all changes to objects of type 'path', " or "all changes to object x between time 0 and 10."

The tag field is special, as it allows a client to send an arbitrary "tagged" message to the database. Tagged message events can be generated by any client and go only to REG, which makes no attempt to interpret the message, but instead forwards it to all other clients interested in that particular type of tag event.

In an effort to make database transactions appear atomic, REG does not call back any clients whose interests have been triggered by a database event until `REGflush()` has been called. `REGflush()` is called by the client.

By modifying a number of chops before flushing the set of triggered interests, systemwide efficiency is gained since an interest that is triggered more than once will only be called back once. However, if multiple events happen between calls to `REGflush()`, undesirable effects may occur. For example, consider a client that is called back based on its interest in the modification of a particular chop, even though a later event in the same transaction had destroyed that chop. The implementer of clients has to be aware of this situation. A simple safeguard would be to ensure that `REGflush()` is called immediately before any deletion request is sent.

Chapter 4

More Details of the Kernel

This chapter completes the description of the database packages by filling in some details not mentioned in the previous chapter. This chapter also contains some of the "lore" that influenced the implementation decisions for these packages.

4.1 The MEM Package

Memory allocation is a basic operation used by all parts of UGA, yet it is an operation that can be costly if done inefficiently. Profiling of the previous version of UGA indicated that calls to `malloc()` occupied a significant portion of the run time, so a new memory management strategy was used in the current implementation of UGA.

The MEM package creates and manages free-lists of constant-sized records. Whenever a constant-sized record is needed (as opposed to an array that may need to be resized with `realloc()`), a MEM allocation routine is called. Routines to reclaim allocated records are also provided by MEM.

The pattern of memory allocations and deallocations produced by the UGA system makes the free-list approach desirable. Most UGA packages use a limited variety of different (constant-sized) data structures, but they repeatedly allocate and deallocate a large number of these structures. The limited variety of structures makes it reasonable to maintain a separate free-list for each type of structure. The management of these lists is more efficient than the general memory management implemented by `malloc()` and `free()`. Calls to `malloc()` must still be made to add new elements onto an exhausted free-list, but the number of calls can be kept small by using a *preallocation* strategy: whenever memory for one new element must be allocated, enough memory for several new elements is allocated with a single `malloc()` call.

The MEM package implements the ideas just outlined, with one exception. It actually maintains a separate free-list for each size (number of bytes) of structure, not for each structure type. Different structure types that happen to be the same size can thus be allocated from the same free-list. This sharing of free-lists is hidden from the packages that use MEM.

The preallocation strategy also hides some implementation details of `malloc()`. The number of new elements preallocated by MEM is always a multiple of the block size that `malloc()` allocates most efficiently. This optimal block size is operating-system-dependent, but MEM prevents its users from having to know this detail.

There is one situation in which the sharing of free-lists is not efficient. A routine may want to allocate many data structures that must all be freed when the routine terminates. (Contrast these temporary structures with the persistent structures that many routines allocate, e.g., a `CHOPchop` structure allocated by `CHOPcreate()`.) Explicitly freeing each structure at the end of the routine can be time consuming. What the routine really needs is one contiguous chunk of memory from which to allocate all its own free-list elements. When the routine is finished, the entire chunk can be freed quickly without affecting any other free-lists. In MEM, such a chunk is called a *private pool*. Any package that uses MEM can request the creation of a private pool. Cells can then be requested from this private pool, and the entire pool freed with one call to MEM.

The MEM package not only makes memory allocation more efficient, it also helps during debugging. Whenever a structure is allocated, MEM can log the UGA package that requested the structure. This information can be used to find packages that allocate unreasonable amounts of memory, or that never free the memory they allocate ("memory leaks").

4.2 More Details of the VALU Package

4.2.1 Paths Through Trees

As already mentioned in Section 3.1 (The VALU Package, page 23,) the VALUget_subtree() routine is called when access is needed to a particular node in an evaluated VALU tree. This access is needed by SXFval_to_form() when it is building a SXFform from a VALU tree. The VALUget_subtree() routine takes as parameters a VALUitem (assumed to be a tree) and a path to a particular node in the tree. The path is specified as an array of integers[1]. The ith element of the array indicates where the path goes at the ith level of the tree; if the ith element is the integer j, then the path goes to the jth child of the node at the ith level. The end of the path is marked by the constant VALU_FIELD_END. If, for example, the VALUitem is a list of three lists, the first element of the second sublist could be reached by specifying [1, 0, VALU_FIELD_END] as the path (note that the indices start at zero).

4.2.2 Other Data Structures

The eval_data field in the VALUid structure is a stack used during VALUitem evaluation. Each time VALUeval() is called, the calling routine can push some data onto this stack. This data will be available for use at any time during the evaluation. Currently, the eval_data stack is used by CTPTeval() to evaluate a VALUitem as of a particular time. This use of the stack is described in Section 4.4 (More Details of the CTPT Package, page 80). The eval_data field is a stack in case the evaluation of a VALUitem spawns new evaluations that need their own special values. This situation arises when the VALUitem contains a VALU_pointer that points to a STATEref. Evaluating this STATEref will cause the referenced object to be inquired; this inquiry will ultimately cause CTPTeval() to be called again.

The VALUspec structure allows some "type-checking" of VALUitems. A VALUspec can be used to specify the template that a node in a VALU tree should follow, e.g., whether it should be a VALU_atom or a VALU_list, and how many elements it should contain if it is a VALU_list. A hierarchy of VALUspec structs can be constructed to specify the template for

[1] The elements of the array are actually of type VALUfield to allow type checking, but the type VALUfield is equivalent to the type int.

an entire tree. The `VALUcheck()` routine can be called with a `VALUitem` and a `VALUspec` to determine if the `VALUitem` matches the specified template. The `VALUspec` structure was originally designed to be used by the `SXFval_to_form()` routine. The routine knows the particular type of form into which it must convert a `VALUitem`, so it could call `VALUcheck()` to make sure that the `VALUitem` is formatted in such a way that the conversion will be meaningful. Using a `VALUspec` is more trouble than it is worth for complicated `SXFforms`, however, so the power of the `VALUspec` is not fully exploited in the current implementation of UGA.

4.3 More Details of the CORE and SXF Packages

4.3.1 The core Array

The `SXFform` structure contains a field `core` that is an array of pointers to `COREcore` structures. Different form types can incorporate different numbers of core types, so the size of this array depends on the particular form type being represented. The `SXFform` structure is actually declared to have only one element in its `core` array. When `SXFcreate()` allocates the memory for a `SXFform`, it consults the `SXFform_def` structure for the form type to determine the number of elements—call it n—that should really be in the `core` array. It then requests enough memory for a `SXFform` structure plus $n-1$ pointers to `COREcores`. This block of memory is cast to a `SXFform`. The cast effectively increases the size of the `core` array (which is the last field in the `SXFform`) to the appropriate size, n elements. The `SXF_FORM_CORE()` macro that accesses a particular pointer from the `core` array exploits the fact that C will not complain if an array is indexed past its declared upper bound.

4.3.2 Interaction with the VALU Package

The `SXFform_def` structure contains an array of `SXFcore_entry` structures, one for each core in the definition of the particular `SXFform`. The path needed by `VALUget_subtree()` to extract this core's value from an evaluated VALU tree is specified in the `SXFcore_entry` structure. This path is stored as an array, as mentioned in Section 4.2 (More Details of the VALU Package, page 77).

There may be cores in the SXFform's core array that do not correspond to nodes in an evaluated VALU tree. These cores are used for temporary storage. An example of temporary storage is present in the SXFform for shading parameters. Shading parameters consist of a color specification (an ordered triple giving red, green and blue intensities) and a opacity specification (also a triple with red, green and blue components). Whenever shading parameters are used, it is also necessary to have a DRAWobj that represents the color and transparency in a form suitable for hardware rendering; see Section 6.1 (The DRAW Package, page 97). The DRAWobj does not correspond to any node in the VALU tree from which the color and transparency vectors are taken, but it should still be in the SXFform. The num_fields field of the SXFform_def indicates that total number of cores in the core array of the SXFform. The num_valu_fields field of the SXFform_def indicates how many of these cores correspond to nodes in the VALU tree, i.e., how many are not used for temporary storage.

The SXFval_to_form() routine has already been discussed. There is also a corresponding SXFform_to_val() routine. This routine is needed during the evaluation of a chop that references an object (e.g., Figure 2.6 on page 20). Such a chop will have a ctpt whose VALUitem involves a VALU_pointer that points to the object. When the ctpt is evaluated, the VALU_pointer must be evaluated into an elementary VALUitem. The STATE package manages this VALU_pointer via the routines in a VALUuser_type. Evaluating this VALU_pointer results in a data inquiry of the referenced object. This inquiry, like any inquiry, will produce a SXFform which must be converted into a VALUitem. The SXFform_to_val() routine performs this conversion. A COREcore_to_val() routine is also implemented, and it is called by SXFform_to_val() to convert the cores associated with the form.

4.3.3 Invalidation

The STATE package handles invalidation of caches, as described in Section 3.7.4 (Invalidation and Revalidation, page 59). Recall that each object class contains a STATEbuf_tbl. This data structure is an array with an element for each type of chop (each SXFform_type) that indicates what caches can be invalidated when a chop of that type is changed.

To keep the size of each STATEbuf_tbl to a minimum, UGA uses the concept of *invalidation groups*. An invalidation group is set of cache types

(SXFform_types, again) that are all invalidated by changes to chops of specific types. The "geometry group," for example, consists of the cache types that could be invalidated by changes to chops that affect the vertices of a polyhedron. The set of possible invalidation groups is defined by the SXF package, and each invalidation group is assigned a unique index. The STATEbuf_tbl element for a particular chop type stores a bit vector whose ith bit is set if changes to that type of chop can invalidate members of the ith invalidation group. The total number of different invalidation groups is limited so that the bit vector can be of reasonable length[2].

When STATEchop_changed() is invoked, it consults the STATEbuf_tbl for the class of the current object. The routine finds the invalidation groups associated with the type of the changed chop so any caches belonging to those groups can be invalidated. The groups to which a cache belongs are found in the SXFform_def structure for the form type stored in the cache. The SXFform_def struct contains a groups field. This field is another bit vector that indicates the invalidation groups to which caches of this form type belong.

4.4 More Details of the CTPT Package

4.4.1 Time-Parameterized Evaluation

Section 3.4.2 (Evaluating a Ctpt, page 37) described why the evaluation of a ctpt that references another object must be parameterized by an inquiry time t. The reference to be evaluated will be part of a VALUitem, and VALUeval() will be called to evaluate it. Since VALUeval() does not take an evaluation time as a parameter, the eval_data field of the VALUid structure is used. Recall from Section 4.2 (More Details of the VALU Package, page 77) that this field is a stack of data that can be accessed during the evaluation of a VALUitem. The CTPTeval() routine pushes the time t onto this stack. It also pushes on an infinite INTinterval. This interval is the initial validity interval for the ctpt's value; it will be adjusted during the evaluation of the VALUitem to contain the time over which that VALUitem's value is actually valid.

[2] Currently, the bit vector is stored as an integer of type long.

4.4.2 Caches Within Ctpts

The repeated re-evaluation of a ctpt can become computationally expensive. The expense can be particularly high if the value of this ctpt involves a reference to another object; evaluation of this ctpt then requires a recursive inquiry of one or more properties of that other object, and any objects on which it is dependent. To improve the efficiency of the `CTPTeval()` routine, a `CTPTpoint` has a `cache` field. When the value of the ctpt is a `VALUitem`, this field caches the `SXFform` that was built the most recent time this `VALUitem` was evaluated. The `CTPTpoint`'s `cache_int` field contains a interval that records the time over which the cached `SXFform` is valid. Once the cache has been created, an evaluation of the ctpt at a time within the `cache_int` interval can be resolved by merely returning the cached `SXFform`.

The caching scheme mentioned above will be correct only if some mechanism supports invalidation of caches. In particular, recall the discussion from Section 3.4.2 (Evaluating a Ctpt, page 37) of how the script from Figure 2.6 (page 20) would be evaluated. When `fellini`'s orientation is inquired at time t, `depardieu`'s position will be computed. This position will be cached at the ctpt in `fellini`'s `orient` chop, and the `cache_int` field will be set to an interval containing t. When a chop is added to `depardieu` that changes its position during that interval, it is essential that the ctpt cache no longer be valid.

The `CTPTupdate()` routine will invalidate a `SXFform` cached by a ctpt. This routine is called by `CHOPinval_interval()`, which is called by the `STATEchop_changed()` callback routine as described in Section 3.7.4 (Invalidation and Revalidation, page 59). The `CTPTupdate()` routine is passed the invalidation interval, and it compares this interval to the ctpt's `cache_int` interval. The current implementation of `CTPTupdate()` completely invalidates the cached `SXFform` if the two intervals overlap at all. An alternative implementation could subtract the invalidation interval from the `cache_int` interval, thus potentially leaving the `SXFform` valid for at least some period of time. This approach was not chosen because the extra implementation complexity was not considered worthwhile.

4.5 More Details of the CHOP Package

4.5.1 More Data Structures

The CHOP package defines a data structure called a CHOPhandle. There is always only one CHOPhandle. A CHOPhandle stores information about all chops created, e.g. a pointer to the callback function STATEchop_changed(). When CHOPinit(), which creates a CHOPhandle, is called, the caller must pass in a function pointer to be called for the callbacks. That function pointer points to STATEchop_changed(). These callbacks are needed to avoid header file include circularities.

The CHOPhandle contains the head and tail of a linked list of CHOPscopes. The ordering of the list determines the relative priority of the chops in different scopes.

Finally, the CHOPhandle contains a hash table that associates character string names with chops. This association makes it possible to reference a chop by name, if the chop has been added to this hash table. In general, not all chops have names. The association of a name with a chop is a convenience to make it possible for clients (for example a user interface) to access particular chops.

Each CHOP_guts contains an interests field. This field is for the exclusive use of the REG package. The REG package uses the interest field to associate client interests with individual chops instead of using a hash table.

4.5.2 Sharing of Chops

The decision to store the data of the CHOP_guts in a separate data structure (instead of directly in the CHOPchop) was made to support the sharing of chops. The sharing of chops is needed to support aggregate objects; see Section 5.1 (Aggregate Objects, page 89). An aggregate object actually pushes all its chops down to all members of the aggregate object. To avoid duplication and subsequent maintenance of these chops, all these CHOPchops share the same CHOP_guts. The CHOP_guts maintains a reference count to know when to destroy itself.

It is also possible for a single chop to be part of several object's states. The state of each object simply stores a reference to the same chop. This feature saves space and allows a client to change several objects at the same time.

4.5.3 Clipping Intervals

Another mechanism implemented for aggregate objects is the idea of *clipping intervals*. A clipping interval "clips" a chop from a state. With a clipping interval it is possible to render a chop ineffective for certain time intervals. Normally, the clipping interval is set to be `NULL` which is understood to have the same effect as an interval $(-\infty, +\infty)$, i.e. the chop is always taken into account. However, if the clipping interval is set to a smaller interval, then the chop will not be taken into account unless the inquiry time lies within the clipping interval. [3]

Clipping intervals are needed for the aggregate objects, since membership in an aggregate object can be for a limited time. For example, it is possible for an object to be a member of an aggregate object from time 0 to time 10 only. Because the chops for the aggregate object are pushed down to all members, regardless for what time interval that object is a member of the aggregate object, we needed a way to clip the effects of these chops to the time interval the object actually is a member of the aggregate object. The clipping interval of a chop originating from an aggregate object and that has been pushed down to a member object is always set to the interval of time the object is a member of the aggregate object. For more details on how clipping intervals are used with aggregate objects see Section 5.1 (Aggregate Objects, page 89).

Clipping intervals are implemented as `INTintervals` and can therefore be complex. The clipping interval is stored directly in the `CHOPchop` data structure. Any inquiry made to a chop is first checked against the clipping interval. All inquiries are handled by the `CHOPinq()` routine. If the inquiry time lies within the clipping interval, then the inquiry proceeds as described in Section 3.7 (The STATE Package, page 52). If the inquiry time lies before the first simple interval of the possibly-complex clipping interval, then `CHOPinq()` returns `NULL`, signaling that the inquired chop is inactive. If the inquiry time does not fall inside the clipping interval and is after a simple interval, then `CHOPinq()` changes the inquiry time to be the end time of

[3] This statement is not quite correct, as will be obvious from the exact definition of the effects of a clipping interval later in this section.

that simple interval.

4.6 More Details of the STATE Package

4.6.1 Inquiry Methods

The standard routine that inquires data from an object is `STATEinquire()`. This routine takes a `STATEinquiry` structure as an argument. This structure defines the parameters of any inquiry:

1. the type of data being inquired;
2. the time at which data should be computed;
3. the location in the state at which the data should be computed (a priority);
4. whether the data should be cached after it is computed;
5. a `CHOPchop` that should be called back if the computed data changes;
6. a flag denoting whether the actual data structure that was cached should be returned (if a cache is found), or just a copy of it.

To actually compute the desired data, `STATEinquire()` calls a state traversal routine (either `STATEtraverse()` or `STATEtraverse_dynamic()`, depending on the context). The computed data is stored in a `SXFform` in a `CHOPeval` structure.

The standard form of inquiry just mentioned is not always the most efficient way to obtain data from an object. Consider, for example, inquiring the polygonal boundary-representation (a `TRIPobject`) of a torus object. The `STATEstate` of the torus will contain a `rep` chop that makes the object a member of the torus class. It will then contain chops that set the resolution of the torus (the coarseness of the polygonal approximation) and the inner radius of the torus. When `STATEinquire()` calls `STATEtraverse()` to compute the `TRIPobject`, the traversal will start before the `rep` chop. The `rep` chop will be encountered, and its chop method will be called to update the `TRIPobject`. This method will not know anything about the resolution and inner radius of the torus (which are specified later), so it will create a TRIPobject for a default torus. The traversal will then continue on to the resolution chop. If this chop specifies a value other than the default, the chop method will throw out the TRIPobject that has already been computed and replace it with a TRIPobject of the correct resolution. The default inner radius will still be used, however. The traversal will then encounter the inner-radius chop, and the TRIPobject will have to be destroyed and rebuilt again. This repeated destruction and rebuilding of TRIPobjects is inefficient.

An inquiry method tailored to the torus class can be more efficient. This method can recursively inquire the resolution and inner radius from the object's STATEstate before any TRIPobject is built. Then a single TRIPobject can be built according to the correct specifications.

Inquiry methods specific to object classes are supported by UGA. These methods are referenced through an array in the STATEfunc_table structure, as mentioned in Section 3.8 (The GOBJ Package and Object Classes in General, page 66). If an object class wants to use its own inquiry method for a particular SXFform_type, it puts a pointer to the method in the array; the SXFform_type value serves as the index of the pointer in the array. Otherwise, the array entry at that index contains a NULL pointer. A client obtains an inquiry method by calling STATEsetup_method(). As described in Section 3.8.3 (Some Important Methods, page 70), this routine sets the STATEbehavior for the object whose data is being inquired. The client then obtains the inquiry method pointer from the array in the STATEfunc_table (contained in the STATEbehavior). If a specific method is mentioned in the array, the client can call it directly. If a NULL pointer is found instead, the client must call STATEinquire() to perform a standard inquiry. The STATE package provides some macros that simplify this process; these macros are described in the next section.

The STATE package also provides a routine that simplifies the implementation of class-specific inquiry methods. Every such inquiry method must perform some of the same tasks, e.g., decide whether or not the requested data has already been cached, and decide whether any newly-computed data should be cached. The STATEdata_inq_shell() routine can be called to perform these common tasks. It searches the STATEstate for a valid cache of the requested data. If such a cache exists, it is immediately returned. If no cache exists and the caller wants the computed data to be cached, then STATEdata_inq_shell() creates a cache for the data. The

STATEdata_inq_shell() routine then invokes another routine that is passed into it as a parameter. This second routine does the real work of the class-specific inquiry method, the work that is not common to all inquiry methods. When this routine returns, STATEdata_inq_shell() caches the computed data (if told to do so) and adds the callback routines that will invalidate the cache.

4.6.2 Inquiry Macros

The STATE package provides several macros to simplify the task of getting an inquiry method and using it to inquire data.

- STATE_INQUIRE() is the most common and safest way to inquire from a STATEstate. This macro assumes the caller knows nothing about the class of the object whose STATEstate is being inquired. The STATEsetup_method() routine is called to set the STATEbehavior for the priority in the state where the request is made. The inquiry method array in the STATEfunc_table is checked, and a class-specific method or STATEinquire() is called, as described in the previous section. The results of whatever inquiry call is made will be a CHOPeval structure containing a SXFform and an INTinterval, both of which must be freed by the caller.

- STATE_INQUIRE_CACHE() does the same thing as STATE_INQUIRE(), except that the SXFform in the CHOPeval is the same SXFform that has been cached. Thus the requested data must have been cached, and the caller must *not* free the SXFform. The INTinterval should still be freed, however.

- STATE_QUICK_INQ() is similar to STATE_INQUIRE(), except that it assumes that the STATEbehavior of the STATEstate has already been set correctly. Remember that the correct STATEbehavior depends on the priority of the request, because any chop could change the STATEbehavior. This condition can be guaranteed if a call has just been made to STATE_INQUIRE() as of the same priority and on the same STATEstate, and no other STATE calls have been made in between.

- STATE_QUICK_INQ_CACHE() is the same as STATE_QUICK_INQ() except that, as with STATE_INQUIRE_CACHE(), the SXFform in the CHOPeval is the same SXFform that has been cached. This SXFform must therefore not be freed by the caller.

Chapter 5

Examples of Interesting Object Classes

The most general object class, gobj, has already been described in Section 3.8 (The GOBJ Package and Object Classes in General, page 66). This chapter describes a few details of some of the other object classes that have been implemented in UGA. The description is at a high level, but it gives a glimpse at how non-trivial object classes interact with the other packages of UGA.

5.1 Aggregate Objects

An aggregate object is a collection of objects that are treated in a uniform fashion. The objects that are grouped together in an aggregate object are called the *members* of that aggregate object. Consider an animated model of the solar system, for example. Objects would be created to model the sun and all the planets that orbit around it. The sun itself actually orbits around the center of the galaxy, dragging the planets along with it. A simple way to handle this galactic orbit is to group the sun and the planets into one aggregate object, and rotate that aggregate object around the galaxy.

An aggregate object in UGA is a member of the "agg" class, implemented by the AGG package. The members of an agg object are specified by `member` chops added to the agg object. This approach allows the membership of an agg object to change over time. If, for example, the solar system model were to depict a galactic disaster that caused the Earth to hurtle off to the far reaches of the universe, this behavior could be accomplished by removing the Earth from the solar system agg object and then moving the Earth independently. The `member` chop allows three ways to specify the membership of an agg object at a particular instant in time:

1. add an object to the group;
2. remove an object from the group;
3. replace the membership of the group with a set of objects.

When a chop is added to an agg object, the effect must be as if every member of the agg object were given the chop individually. This effect is achieved by adding a reference to this chop to the `STATEstate` of every member of the agg object. This propagation of chops is performed by a special method defined by the AGG package. The method is one of the "arbitrary methods" described in Section 3.8 (The GOBJ Package and Object Classes in General, page 66).

A propagated chop should affect a member object only during the time interval in which the object is a member of the agg object. This interval is stored as a *clipping interval* associated with the object's reference to the chop. Clipping intervals are part of the `CHOPchop` struct; see Section 3.6 (The CHOP Package, page 43) for details. When a state traversal of the object encounters the shared chop, it will determine if the inquiry time is contained in the clipping interval. If so, the effects of the chop at the inquiry time will be computed. If the inquiry time is after the end of the clipping interval, the chop is evaluated as of the end of the clipping interval; this policy is necessary so that the changes made to an object while it was a member of an agg object do not disappear as soon as the object leaves the agg. If the inquiry time precedes the start of the clipping interval, the chop is ignored.

The AGG package maintains its own internal data structures that describe the membership of an agg object. These data structures allow the clipping intervals of the shared chops to be changed if a `member` chop is changed. For example, say the chop adding the moon to the solar system from time 0 to time 100 is changed so the moon is a member only until time 90. The clipping intervals on every chop the moon shares with the other members must now be changed so these chops stop having effect at time 90.

The REG package, described in Section 3.10 (The REG Package, page 72), notifies the AGG package whenever a `member` chop has changed. Each agg object acts like a client and registers with REG an interest in any changes to its chops. When a change is made, REG invokes a callback routine that allows the agg object to update the appropriate clipping intervals.

The AGG internal data structures are stored in "aggstuff" chops in the `STATEstate` of the agg object. Each `member` chop has an aggstuff chop associated with it. The aggstuff chop allows the old value of the `member` chop to be determined even after the `member` chop has been changed. Both the old and new values of the `member` chop are needed to determine how the membership has changed. The AGG package is the only part of UGA that ever manipulates aggstuff chops. The aggstuff chops would be visible to anyone who looked for them, but it is assumed that only the AGG package will actually look for them.

5.2 Paths

The path object class defines a path (in n-space) which might be used to define object trajectories or the shape of such other object classes as prisms, revolves or ducts.

A path object depends on a set of control points. The control points are used to define the shape of the paths. Control points for the path are defined by the `move_point` FLESH statement. This statement takes real numbers as coordinates for the control point, not a point object. This difference is important as path objects are therefore not aggregate objects. However it is common to set the control points of a path to be the positions of other objects (see Figure 5.1 on page 92 for an example). Because of the dependency network implemented in UGA it is then possible to interactively change the shape of the path by moving the objects that define control point positions.

The decision to implement the FLESH statement `move_point` such that it takes real numbers instead of point objects as parameters was made for the following reasons. By making a path a non-aggregate object it can be handled more efficiently by UGA. Time-efficiency is considered essential,

```
a_point: translate  0 = [-1, -2, -3] : linear
                   10 = [ 1,  2,  3]
;

a_path : rep           0 = [pathClass,implicit]
         curve         0 = [(LINEAR, 3)]
         /* The path consists of one linear */
         /* curve with 3 control points.    */

         move_cntrl_pt 0 = [[1, 1], [0, 0, 0]]
         /* move the first cntrl pt in the   */
         /* first curve to {0,0,0}.          */

         move_cntrl_pt 0 = [[1, 2], [2, 3, 7]]
         move_cntrl_pt 0 = [[1, 3], a_point.translate]
;
```

Figure 5.1: A path object.

because path objects are mostly used in interactive settings. The data dependency network in UGA is faster than the aggregate object mechanisms. Space is saved, because if a path control point is fixed or does not need to be changed interactively, then the user can set the control point to be a vector of constant values. The overhead of creating and maintaining a point object for such a control point is saved.

An example of a simple path object is given in the FLESH script in Figure 5.1 (page 92). The script creates two objects: a point and a path. The point moves over time. The path is a simple polyline with three control points (and therefore two lines). The first two control points are at fixed positions, whereas the third one is the position of the point object. Since that object moves over time, the path will change over time. Note that the path references the position of the point object, not the point object itself.

Path objects are frequently used to define object trajectories in space over time. A path object therefore acts very much like an interpolation method. The path interpolates its control points. The advantage of using a path object to do the interpolation, instead of the interpolation methods described in Section 3.5 (The IMP Package, page 38), is that the animator is able to interactively change the path to fit her needs. Since path objects are based on the CAP package (see the CAP package description [WLOK90]), a wide variety of parameters is provided so that controlling the shape of the path becomes easy.

To be able to use path objects for other object's trajectories over time, all path objects can export an `orient` attribute. For details about the syntax of this functionality refer to the FLESH manual [GRIM91].

5.3 Revolves

Revolve objects are solids traced out by a path revolved around an axis. Therefore, part of the state of a revolve object is a `set_path` chop that references a path object. Once again, note that only a special characteristic of the path object is imported into the revolve object (namely a data structure called a `CAPpath` [WLOK90]), so a revolve object is not an aggregate object.

The paths that are revolved can be either two or three dimensional. If the path is three dimensional, then a *rotational projection* about the axis of revolution is performed to convert the path into two dimensions. Figure 5.2 (page 94) illustrates an example of a rotational projection.

If the axis of revolution is not explicitly given, then it will be computed as the vector from the first point in the path to the last point in the path. If an axis is explicitly given and it is not the same as the computed axis, then two points are implicitly added to the path, at the base and end of the explicit axis of revolution. Adding these points insures that the revolved object will be solid.

There are two kinds of resolution associated with a revolve object. They are the latitudinal and longitudinal resolutions. The latitudinal resolution of the revolve object determines how many degrees apart each revolve profile will be. It is specified by a FLESH change operator of the revolve object (refer to the FLESH manual [GRIM91]). The longitudinal resolution of the revolve object is taken to be the same as the resolution of the passed in path. As with other objects, the resolution is only used for polygonally based computations. For more information on revolve objects see the FLESH manual [GRIM91].

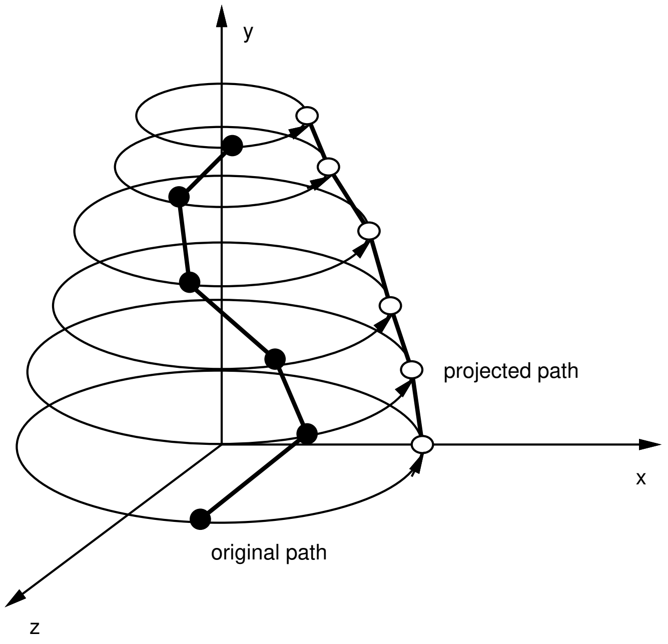

Figure 5.2: The rotational projection of a path.

5.4 CSG Objects

A CSG (constructive solid geometry) model is represented in UGA as a single object. The CSG object class is implemented by the CCSG ("class of *csg* objects") package.

The Boolean expression tree that defines a CSG model is specified by `structure` chops in the state of a CSG object. Each ctpt of a `structure` chop represents the Boolean expression in infix notation at a particular moment in time. Together, these ctpts define how the structure of the CSG model changes over time. The variables in the Boolean expressions are names of other objects, which get converted to references to these objects.

Since any object name can be used as a variable in the Boolean expression, a CSG object can reference itself. Any such reference evaluates to the CSG object up to but not including the `structure` chop making the recursive reference. Thus a CSG model can be built by a sequence of `structure` chops that successively add more elements to the same CSG object.

The algorithms that inquire various kinds of data from a CSG object all use the `CCSG_traverse()` routine. This routine recursively performs an operation at each node of the CSG tree. The routine is parameterized so it can perform a variety of operations. It can perform a Boolean set operation on polyhedra at each node, thus producing as a net result the polyhedral boundary-representation (b-rep) of the entire CSG object. It can build a `RAYobj` that references the `RAYobjs` for the children of each node, thus producing a ray-traceable representation for the entire CSG object. It can inquire the color of each object at a leaf of the tree; these color specifications will be added to a one-dimensional array that represents the color of each node in depth-first order.

If each subtree of the CSG model is represented by a named object, changes to the CSG model can sometimes be processed more efficiently. For example, let the root node of an expression tree subtract a sphere from another elaborate CSG object. If the position of that sphere is changed and the b-rep of the whole model is inquired, the b-rep of the other CSG object will be cached so it need not be recomputed. Such caching allows reasonable space-time tradeoffs in some interactive modeling situations.

It is interesting to notice what happens when a CSG object is deformed and its b-rep is then inquired. The b-rep inquiry will start a state traversal to compute a `TRIPobject`. When the `structure` chop defining the expression tree is encountered, an expensive CSG b-rep evaluation algorithm will be invoked to produce a `TRIPobject`. The state traversal will then continue on to the deformation chop, which will deform the single `TRIPobject` computed by the previous chop. If the `TRIPobject` is cached as of the `structure` chop, the parameters of the deformation chop can be changed without causing the CSG b-rep to be rebuilt from scratch each time. If the deformation parameters are changing frequently, this approach can be much more efficient than the alternative of deforming the leaves of the tree and then recomputing the Boolean expression each time.

Chapter 6

Beyond the Kernel of the Database

This chapter will describe elements of UGA that are not fundamental parts of the database. It is a bit of a catch-all chapter for topics that are important, but that do not fit cleanly into other chapters.

The DRAW and RAY packages are hardware-dependent optimized code packages. The user of these packages is totally isolated from the underlying hardware details. In addition these packages can be isolated out of the UGA framework easily to be used in other projects. The section on networking explains our strategies to distribute UGA over several machines.

6.1 The DRAW Package

A user of the database frequently wishes to see the geometric objects in the database. When making changes, or even when just viewing a pre-made script, display of these objects must be fast and efficient. In terms of the database, objects have inquiry methods that return a data format that can be efficiently drawn by graphics hardware.

The DRAW package supports these data types. In addition, DRAW provides routines for drawing these data types in a window. These windows are handled by the VAMP package, providing a hardware-independent software interface to graphics hardware. Likewise, DRAW itself provides a hardware-independent interface to 3-D drawing[1].

6.1.1 The Purpose of DRAW

For a dynamic and flexible system like UGA, standard retained-mode systems seem to be unnecessarily limiting. Many operations can perform much better if they take advantage of data coherency. Consider a deforming object. The connectivity of the polygons in its polyhedral boundary representation (b-rep, implemented as a `TRIPobject`) will not change through the course of the deformation. Only the vertices and normals of the polygons will move. Systems like PHIGS+ will not allow this kind of fine-grained editing.

In addition, standard retained mode systems set up their own database structure, with their own database traversal. The UGA system performs its own database traversal, and duplicating this work in the graphics display code is a waste of resources.

Finally, "standard" retained mode systems are far from standard, requiring far too much special-casing to handle differences in hardware and in how a particular vendor chose to implement the "standard". For these reasons, the DRAW package supports hardware drawing. This library is built on top of several different hardware libraries[2]. The DRAW package itself handles optimizing UGA data types for efficient display on various architectures.

The DRAW package does not enforce a particular data hierarchy. It is thus possible for an application to use its own data structures that can simply reference `DRAWobj`s. The application is not forced to use the higher-level DRAW data structures, for example `DRAWlists`. This freedom allows the UGA database to take maximum advantage of its own structure and information-management schemes. But DRAW's optimizations of data structures *can* be retained for later reuse and, most importantly, modification.

6.1.2 The Design of DRAW

Central to the concept of the DRAW package is the `DRAWobj`, a data structure that provides a uniform interface to a variety of drawing primitives and

[1] While software such as PHIGS+ is considered a standard, we have found that different versions of such "standard" can vary widely across platforms and vendors.

[2] Currently, DRAW supports HP's Starbase, and Sun's XGL.

attributes. A DRAWobj is not simply used to store graphics primitives, such as b-reps or polylines, but is also used to store changes in the drawing state, such as colors, lights, or the default shading mode.

Calls that create DRAWobj's use UGA data types to construct hardware-dependent versions of the data used to draw a primitive. This data is editable with UGA packages for greater functionality. A DRAWobj for a b-rep, for example, is constructed from a TRIPobject. The DRAWobj retains a reference to the TRIPobject even after a hardware-dependent polyhedron has been constructed. It is therefore possible to modify the geometry of the TRIPobject (e.g. change the coordinates of its vertices) and tell DRAW to update the hardware-dependent polyhedron.

Non-hardware specific data is maintained inside a DRAWobj with the stuff field. This field hides object-type-specific data, like a TRIPobject, or a MAT3mat for a transformation matrix, from users of the package that do not need to know about it. For example, a simple hardware renderer could ask each object in the database for its DRAWobj, and then draw it into a VAMPwindow, without ever needing to know the actual steps necessary to render that particular object correctly.

Each DRAWobj contains a list of hardware-specific data. This data is specialized to the format any one architecture wants. Hardware-specific functions are called using the appropriate data. The data is not created until the object is drawn on that particular architecture.

6.1.3 The Execution Model

A DRAWobj is created by calling the appropriate routine with the appropriate data. The creation routine is called with an optional window to draw in. The routine returns a pointer to the DRAWobj. The DRAWobj can be modified by calling the creation routine again, with new data, but this time passing in the old object. All objects can be drawn with the same DRAWexecute() call, and can be freed with the same DRAWdestroy() call.

When executed, the object looks through its list of hardware-specific data for data appropriate to the window (hardware) it has been told to draw itself into. If no appropriate hardware-specialized data is found, it creates it, and adds it to its list. It then uses the data in hardware-specific calls.

Some simple DRAWobjs do not store hardware versions of themselves, but simply translate themselves to an appropriate form on the fly. Whether hardware data is made for a particular DRAWobj type depends on whether the difficulty and time of converting the generic data outweighs the memory cost of storing it. An example is the DRAWobj for color. Most routines that set colors for graphics hardware take three floats. These floats can be generated quickly from the RGB components in a SXFform for color, so there is no need to store them again.

As a more complicated example, consider lights. Lights do nothing on a wire-frame display, but might need data on a shaded display. Accordingly the internal list of hardware-specialized data of the DRAWobj that represents the light will contain a version for the shaded display, but never a version for the wire-frame display, since the representation is trivial. The data for the wire-frame version of the light is not stored even when the object is drawn in both displays at the same time. This example actually occurs quite frequently: In a shaded display the "selected" object can be highlighted by drawing it a second time using the wire-frame representation in a different color.

6.2 The RAY Package

The RAY package provides basic ray-tracing functions for a variety of primitive objects. Each object class in UGA provides (or inherits) methods that intersect a ray with an object belonging to the class and perform shading calculations based on such an intersection. These object-class-specific methods call routines from RAY to do the actual work. The RAY routines themselves make no reference to SXF, CTPT, STATE or any of the other kernel packages. This implementation was chosen so that the RAY package could be incorporated into applications other than UGA with a minimum of effort. This implementation also makes it simpler to compile a version of UGA that includes no ray-tracing code, an option which is useful when debugging parts of UGA that do not involve ray tracing.

A ray-traceable object is represented by a RAYobj structure. This structure has fields for the various matrices used during ray-object intersection calculations, e.g., matrices to transform world-space points to object-space points, world-space directions to object-space directions, etc. The structure also contains a union for the auxiliary data of specific primitive objects. The

`RAYobj` for a polyhedral primitive object, for example, stores a BSP tree to make the intersection calculations more efficient.

Before an object from the UGA database can be ray-traced, a `RAYobj` for that object must be built. That object's class will provide a method that builds a ray-traceable representation of that object. This method knows what RAY primitive object type most closely matches the object class, and calls the RAY routine that builds a `RAYobj` for that primitive type. Any data (e.g. the matrices or the BSP tree) needed to build that `RAYobj` are inquired from the UGA database and passed in to the RAY routine. The RAY routine can thus be independent of the UGA database.

A ray that can intersect a `RAYobj` is represented by a `RAYray` structure. This structure contains the origin and direction of the ray. The results of a ray-object intersection computation are stored in a `RAYinter` structure. The `RAYinter` structure contains the distance along the ray to the intersection point and an indication of whether the ray was entering or exiting the object at the intersection point.

For each primitive object type, RAY defines a routine that takes a `RAYobj` and a `RAYray` and computes a `RAYinter`. For each primitive type, there is also a routine that computes various surface properties from a `RAYinter` structure. These surface properties include the surface normal at the intersection point and the u and v coordinates at the intersection point (for pattern mapping and other surface-oriented mappings).

The RAY package provides no space-subdivision methods that make it more efficient to intersect one ray with many objects. It is expected that such multi-object efficiency measures would be built on top of RAY.

6.3 Networking

The discussion so far has implicitly assumed that the UGA database and any clients that use it run as a single UNIX process. In some situations, however, it is desirable to have separate processes for the database and its clients. If a client performs a significant amount of work that does not directly involve the UGA database, then the overall performance could be improved by running the client and the database concurrently on different machines.

There are some problems with this idea, however. We currently believe that for many interactive graphics applications, the amount of work performed by a client external to the database should be minimized. Much of this work can be shifted to controller objects within the database. This shift is advantageous because it promotes flexibility; more of the application can be controlled with the standard mechanisms provided by the UGA system. As an example of this idea, we are currently exploring how user-interface widgets can be expressed as objects in the UGA database. The goal is to allow the user interface that controls a model to be as easily configured and animated as the model itself. When this project is complete, much of the work traditionally performed by interactive modeling applications will become work performed by the UGA database. The performance gained by running the modeling application concurrently with the database will thus be minimal.

The obvious solution is to exploit concurrency within the database itself. If data must be inquired from two objects that are not dependent on one another, it should be possible to perform the two inquiries in parallel. We would like to be able to exploit this kind of fine-grained parallelism, but difficulties involved in the implementation make it an open research problem.

As an interim solution, we have implemented only a basic networking system that supports coarse-grained parallelism (with the entire database running as a single process). The system allows clients to be dynamically started and terminated on their own machines. Messages between a client and the UGA database are managed by an asynchronous transaction manager. A series of requests for data can be grouped into a single transaction block (packet) to reduce communication overhead. The transaction manager provides network transparency, hiding the details of communication from the client. The networking system works on several architectures[3] and can be ported to other systems.

The following sections briefly describe the different software packages that implement the networking system.

[3] Currently, it runs on Sun and Hewlett Packard workstations.

6.3.1 The EVENT Package

The EVENT package manages communication *events*. It allows a software package to register sockets and timers as event sources and notifies the package when events on those sources are pending. Events can be either requests from some other source to read information, or error signals. A package that registers for an event provides a callback routine and some arbitrary data; when the event occurs, EVENT calls the routine with the data as its parameters.

6.3.2 The STREAM Package

The STREAM package provides an abstraction called a *service*. A service is an interface between the database an a set of clients. Multiple client processes can establish a connection to a service. The STREAM package coordinates communication between the database and the clients, handling all peer connections and error cases.

6.3.3 The TRX Package

The TRX package implements the concept of *transactions*. Transactions are the main abstraction used for network communication with the UGA database. A transaction is defined as a block of data that has a unique identifier, a source and a destination. A transaction can be initiated either a client or the database; the TRX package does not differentiate between the two sources. It provides a standard method for opening, closing and replying to transaction blocks.

When a client opens a transaction, TRX obtains a free transaction block from the current pool of blocks. If a free block is not available, the pool of blocks is expanded. The client then fill the transaction block with the data it wishes to send. Before closing a transaction the client must indicate whether return data is expected from the transaction. If no return data is expected, closing the transaction will simply flush it to the specified destination. If return data is expected, the client may register a callback routine to get called when the return data arrives. If no such callback routine is registered, the return data will be cached until the client, by means of the polling routines, obtains the return data.

The receiver of a transaction (typically the OIL package, described in the next section) must have previously registered a callback routine to get called whenever a transaction is received. This routine should process the transaction block and may optionally create a reply transaction block. The reply transaction block will automatically be sent back by TRX to the originator of the transaction.

6.3.4 The OIL Package

The OIL package provides the high-level interface for clients. This package provides routines that simulate the primary routines from the STATE package, the CHOP package and the other main database packages. A client can call the OIL routines just as if it were calling database routines in its own process. The OIL package then uses the TRX package to start the appropriate transactions to access the remote database.

The OIL package packs the arguments to the database routines and any data they return into packets for efficient network communication. As mentioned in Section 3.2 (The CORE and SXF Packages, page 27), each `SXFform` defines a routine that will pack core types associated with that form type. Each `SXFform` also has a corresponding routine to unpack the a packet.

Multiple calls to the database can be communicated in a single network packet. If this approach is taken, all the data returned by those calls will be returned in one packet. Blocking remote procedure calls can also be simulated by sending only one call per transaction block, but this approach reduces performance.

Bibliography

[BARR81] Al Barr, "Superquadrics and Angle-Preserving Transformations," *IEEE Computer Graphics and Applications*, January 1981, pp. 11-23.

[GRIM91] Cindy Grimm and Mitch Henrion, "Description of the FLESH Language," Computer Graphics Group internal documentation, Brown University, Providence, RI, July 1991. Online: `/pro/uga/doc/new_scefo`.

[HUAN91a] Nate Huang, "A Guide to TRIP," Computer Graphics Group internal documentation, Brown University, Providence, RI, July 1991. Online: `/pro/uga/doc/trip`.

[HUAN91b] Nate Huang, "Writing a New Object Class in UGA," Computer Graphics Group internal documentation, Brown University, Providence, RI, July 1991. Online: `/pro/uga/doc/oc`.

[PRES88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, Cambridge, 1988.

[STRA91] Paul S. Strauss, Michael J. Natkin, Nate Huang, "The Programming Environment for the Brown Computer Graphics Group," Computer Graphics Group internal documentation, Brown University, Providence, RI, July 1991. Online: `/pro/uga/doc/standards`.

[WLOK90] Matthias M. Wloka, "A Guide to CAP," Computer Graphics Group internal documentation, Brown University, Providence, RI, July 1990. Online: `/pro/uga/doc/cap`.

[ZELE91] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes and Andries van Dam, "An Object-Oriented Framework for the Integration of Interactive Animation Techniques," Proceedings of SIGGRAPH '91 (Las Vegas, Nevada, July 29-August 2, 1991). In *Computer Graphics*, July 1991, ACM SIGGRAPH, New York. Online: `/pro/graphics/papers/siggraph91`.