BUMP, a Motion Description and Animation Package

Steven Anders Oakland

Master's Project Report

Under Direction of

Professor Carlo H. Séquin

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

September 1987

ABSTRACT

This report describes **Bump**, the Berkeley Unigrafix Movie Package. It is an animation-generating preprocessor for the Berkeley UNIGRAFIX geometric modeling and rendering system. **Bump** allows motions to be defined in the abstract, using a number of different motion styles such as linear interpolation and B-splines. These abstract motions can then to be applied to objects within UNIGRAFIX scenes. When used in conjunction with the existing UNIGRAFIX rendering packages and other tools, **Bump** provides a simple and effective mechanism for the production of animation sequences.

September 1987

1. INTRODUCTION

The UNIGRAFIX graphics package is a tool for the creation of images in a variety of styles [8]. Over the course of about five years, it has grown in complexity as many different contributors have added new modules. UNIGRAFIX currently consists of a small library of shapes, a large number of filters and generators, an interactive shell, and several renderers, including a recently completed ray tracer [5]. It can be used to create intricate scenes of almost anything that can be represented by polygons.

UNIGRAFIX has been described in more detail in a number of Master's reports and miscellaneous documents [7, 8]. For the reader of this paper, it is sufficient to consider UNIGRAFIX as a collection of tools for the definition and rendering of three-dimensional scenes built of polygons.

However, UNIGRAFIX has no way of expressing motion, is unable to produce anything but single, static frames. This is unfortunate, as the addition of motion makes computer graphics much more useful and interesting. For example, a rotating model of a molecule is far easier to comprehend than static views of the same molecule, and many geometrical transformations make more sense if one can see them in progress. The transformation of a cube through a cube octahedron into an octahedron is much clearer when shown as a continuous action than as a series of static snapshots.

The Berkeley Unigrafix Movie Package, or **bump**, introduced in this paper, is an animation preprocessor for UNIGRAFIX which supplements the existing tools with time dependencies. With only five additions to the current UNIGRAFIX scene description language, it allows the user to define deformations and motions of arbitrary complexity and apply them to conventionally-defined UNIGRAFIX objects. These specific statements will be defined in more detail below, and are also explained in the **bump** tutorial, Appendix C of this paper.

This paper will first discuss the decisions made during the planning stages of **bump**, in section 2. It will then describe **bump**'s external appearance, as seen by a typical user, in section 3. Section 4 presents a view of the internals of the system. Finally, section 5 contains a review of the project, with comments on its deficiencies and suggestions for future work.

There are a number of appendices as well. Appendix A provides several examples of animation produced by **bump**, listing the original scene

descriptions and motion specifications as well as presenting the resulting pictures. Appendix B is a BNF description of the five **bump** extensions of UNIGRAFIX. Appendix C is a tutorial for the package, designed for the typical user. Appendix D contains the **bump** manual pages, and appendix E briefly describes the hardware and other facilities that were used for the production of the demonstration videotape.

Bump is designed to be an easy-to-use addition to the collection of UNIGRAFIX tools. It strives to present a simple user interface and requires only minimal additions to the UNIGRAFIX language, while providing a motion description language rich enough to represent any action that the user can conceive of.

2. DESIGN OF BUMP

We first had to define the abilities and limitations of the system, which required decisions as to what sort of motions it should be capable of, what types of objects would be capable of movement, and how the motions would be applied to the objects. The treatment of time also posed problems. How would the starting time and duration of motions be defined? Should it be possible for motions to be applied more than once, at different times and with different durations?

Beyond these basic decisions on the semantics of the system, there were also many questions of syntax. At the most basic level, how would motions be described? We had to make the motion description language as extensible as possible, so that any later additions would be easy.

Throughout the design phase, the major emphasis was on the user interface, with much consideration also given to ensuring that the resulting animation package would continue the existing UNIGRAFIX philosophy. This philosophy, as well as it can be pinned down, is "provide sufficient primitives, and no more." **Bump**, in accordance with this, provides minimal yet sufficient means of describing motions.

The solutions to the above questions were arrived at with a combination of ideology and practicality. For example, it was decided that motions should be considered to be time dependent transformations. Motions could not be applied to fundamental entities such as object definitions and vertices, but only to instances and arrays of definitions. The reason for this was primarily a practical one; in the current implementation of the UNIGRAFIX data structure, only instances and arrays contain transformation lists to which new transformations

could easily be appended. Allowing other objects to move would have required that **bump** compute transformation matrices on its own, duplicating the routines already available elsewhere.

We decided that the definition of a motion and its application would be separate statements, paralleling the existing UNIGRAFIX treatment of objects. The definition of an object and a motion would be semantically similar, while the motion's application to an object would be the equivalent of the instantiation of an object via an instance or array statement. With this choice, we tried to continue the existing UNIGRAFIX philosophy as much as possible.

The design and construction of **bump** was top-down, in that we first established what the package should look like and what it would do, with the assumption that we would later find some way of implementing it. There were several early, major decisions which shaped **bump**:

2.1. Bypassing of the ASCII level.

In previous UNIGRAFIX work, the link between all modules was the ASCII scene description language, commonly known as **UFO**.* All object modifier programs expected their input to be expressed in **UFO**, and generated **UFO** output. This provided easily-understood interfaces, as the language was terse, well-defined, easy to parse and to generate, and readable by humans. It was also particularly well-suited to the construction of pipelines of filters at the UNIX† shell level. However, a great deal of time was being spent by each program in translating the ASCII input into an internal data structure, and in retranslating the new data back into ASCII once the module completed its work. While the time required for this is not overly distressing for single frames, such bottlenecks become intolerable when one must generate thirty frames for a second of videotape. It was decided that bump would bypass the **UFO** level as much as possible, and work directly with the UNIGRAFIX data structure obtained from the new **UG** parser [6]. Once a renderer that can directly read this data structure is completed, this decision will cut the time of rendering each frame roughly in half.

---

* UFO stands for Unigrafix FOrmat.

† UNIX is a trademark of Bell Laboratories.

2.2. Decoupling of motion definition and application.

An essential decision was that motions could be first defined in the abstract, and later applied to specific objects. Their starting times and duration would be specified at the individual applications. This made it possible to apply motions to more than one object, to apply motions at different times in the scene, and most interestingly, to scale the duration of a motion to any amount of time in global terms. This decoupling of motion definition from application nicely parallels the existing treatment of objects in **UFO**, where objects are first defined and later instantiated, with the different instantiations of a single object typically having different initial transformations, such as translations, rotations, and/or scaling.

2.3. Absolute frame of reference.

Another question was in what coordinate system the transformations should take place. If, for instance, a rotation about the y-axis were applied to an object, which y-axis should this be taken to mean? The y-axis of the space in which the object resides, or the y-axis of the object itself? The conclusion was that motions would be relative to the space in which the moving object was defined, a choice which was consistent with the semantics of transformation concatenation in UNIGRAFIX.

2.4. LISP-like motion description syntax

It was decided that the **UFO** extensions comprising the **bump** motion descriptions would resemble LISP. This was a format particularly well suited to the lists of times and value pairs used in describing motions. It was also easily extensible. As all statements would be delimited by a set of parentheses, it would be easy for a program to disregard all non-understood statements and pass them intact to the next step of processing. Furthermore, this decision was consistent with plans to eventually convert the entire **UFO** language into LISP syntax [9].

2.5. Concatenation of motions

The question of what should happen to a moving object once its motion was completed was also considered. One possibility was that objects would only be visible so long as there was a motion being applied to them; when the motion expired, the object would vanish. This would have required a special sort of pseudo-motion, 'static,' to allow non-moving objects to remain in the scene.

This scheme did not match the existing semantics of UNIGRAFIX, in which objects are visible at all times. It was decided that an object would move so long as a motion was acting on it, and would then remain in the position and orientation where it was deposited at the last frame of the motion. This had several advantages, one of which was that it made it much easier from the user's viewpoint to concatenate motions, as each successive motion would automatically begin exactly where the its predecessor expired. It also made the results of multiple repetitions of a movement more reasonable. For example, three repetitions of a 90° rotation would result in the object rotating from 0° to 270°, instead of from 0° to 90° three times in succession, with a snap back to 0° after each.

A related question concerned what effect a motion should have before it became active. Though this may sound meaningless at first, it was a matter of serious discussion. Consider a rotation which carries an object from 10° at time 5 to 30° at time 10. What rotation does this imply for the object before time 5? One suggestion was that, since the effects of motions remain constant after they expire, their effects should be similarly constant before they begin. This scheme would result in the object being rotated 10° before time 5, rotating from 10° to 30° in the interval between times 5 and 10, and remaining at 30° thereafter. It is graphically represented by line 'a' in figure 2.0.

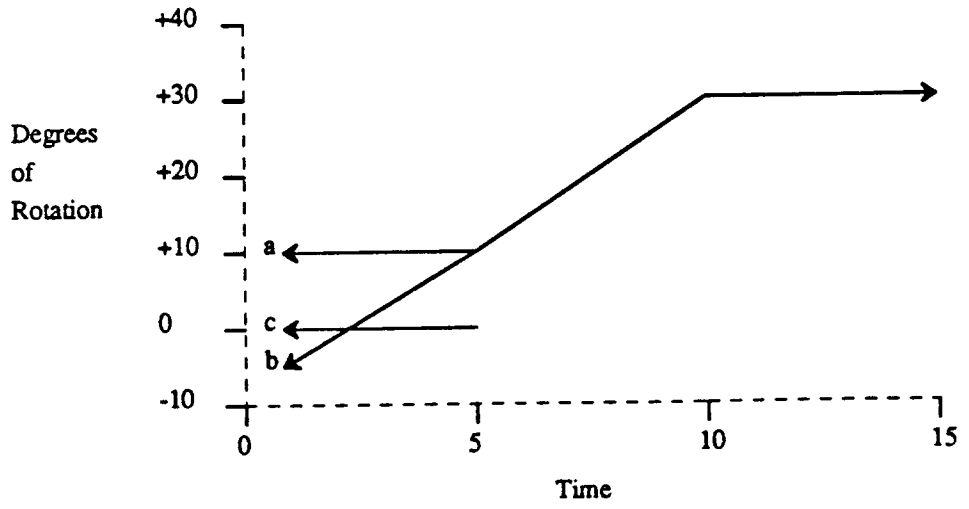


Figure 2.0

Although this had the appeal of treating both extensions of the motion symmetrically, it was dismissed for the practical reason that it would make complex motion sequences very difficult to design, as the effects of a motion

appended to the sequence could propagate backwards and change the position and orientation of the object from the first frame on.

Another proposal was that the motion should be interpolated backwards, as is shown by line 'b' in figure 2.0. This would have resulted in the object being rotated -10° at time 0, and was rejected due to the great difficulty this would have caused in designing complex motions.

The system eventually decided upon was that motions would have no effect until the first time at which they were applied. At this time, the motion would snap to its first value and proceed from there, remaining fixed at the motion's final value, as is is shown by line 'c' of figure 2.0. Although this solution treated the front and back extensions of the motion differently, its practicality for defining motions was so great that this quirk was forgiven. It allowed jumps to the motion's initial position, if desired, and ensured that the effects of one motion would not propagate backwards. If the motion director preferred to avoid jumps, this could be done by the simple expedient of designing each motion to start with the identity transformation.

2.6. Omitted features.

Several features were initially considered for **bump**, but eventually rejected as they were deemed to be inconsistent with the existing UNIGRAFIX philosophy.

One of these was the concept of 'linking' one object to another, so that the linked object would follow the motions of the other. This would be useful, for example, when a robot arm picked up an object and carried it from one place to another. This was rejected, as it was decided that such a tool was beyond the set of primitives essential for animation. Also, the required mathematics of constraint solving would have been too involved to implement at this time.

Another rejected notion was the concept of the camera as a 'flying eye.' In this scheme, the camera would be considered to be an object in the scene like any other, and motions would be applied to it just as they were applied to all other objects. The camera itself would be an imaginary four-sided pyramid, with the viewer's eye at the vertex, and the corners of the visible scene defined by the rays from the pyramid's vertex through the corners of its base. Translating and rotating this pyramid would be the equivalent of moving and turning the camera, while scaling its height would correspond to zooming it in and out. Scaling in other axes would change the screen's aspect ratio, an

interesting effect but one of limited utility.

This scheme was rejected in favor of a scheme of describing camera movements which was more closely related to the current method of specifying the camera position. This has not yet been implemented, but is explained in some detail as a proposed extension to **bump** in the final section of this paper.

2.7. The Bump language extensions

It was finally decided that **bump** would add five new statements to **UFO**. **Mdef** and **mapply** supply the core functionality, respectively providing the ability to define motions in the abstract and apply these motions to specific objects, while **move** simultaneously defines a motion and applies it, as a shorthand for motions that are used only once. For generating movements that could not be easily described via **mdefs**, two escape mechanisms are provided. The first of these, **meval**, allows the evaluation of expressions within **UFO** files by making numeric fields time-dependent. This makes it possible, for example, to change the coordinates of vertices without regard for the geometric consequences. This can create non-planar faces, self-intersecting solids, and other objects which are impossible for the current renderers to display. However, it also supplies the user with the power to directly manipulate the scene at a very low level. Secondly, **mexecute** was introduced to allow calls to arbitrary generator and filter programs, with the result of the call inserted into the output file. This last command was meant to serve as a 'back door,' providing for almost any functionality desired. Together, these five extensions create a small but effective motion description toolbox, which has been proven capable of producing animated sequences of considerable complexity.

Bump was influenced in its early stages by the **BAGS** project at Brown University, which had similar goals [3, 11]. The combination of **Bump** and the underlying UNIGRAFIX tools is very similar in functionality to **BAGS**. They share a number of fundamental concepts, such as the separation of an entity's definition from its instantiation, and the use of lists of times and values to describe motions. There are also some basic differences, such as **BAGS'** heavy dependency on constructive solid geometry, and UNIGRAFIX's close links to the UNIX operating system.

3. USER'S VIEW OF BUMP

3.1. General

A typical file using **bump** contains several object definitions and instances, and possibly some lighting and color statements, specified in the standard UFO fashion [7]. In addition, it will contain at least one **bump** statement. Bump statements have a LISP syntax, and a complete BNF description of the various types can be found in Appendix B. Several examples of typical **bump** files can be found in Appendix A, which contains some of the many files used to generate the UNIGRAFIX demonstration videotape.

The required command-line arguments to **bump** are the name of the input file, the initial time value, the time increment per frame, and the number of frames to make. There are several optional arguments. The **-p** option gives the prefix of the output file name, telling **bump** where to put its output. If no output prefix is given, **bump** will put each frame into a temporary file in /usr/tmp, render it, and then delete the temporary file. The **-r** option is used to give a file name containing the rendering command line. If this option is used to provide a file name, **bump** will read this file for each frame and cat the current output file through the command line it contains. This rendering file can be made time-dependent via use of mexecute's side-effects. If no rendering file name is given, **bump** renders the frames through UGplot with default arguments, puts them into the Ikonas frame buffer, and sends a VAS-IV record signal to /dev/tthy4.

3.2. The move statement

The following is a complete, simple **bump** file, defining a cube which in the interval from time 0 to time 1 turns at a constant rate from an angle of -135° to an angle of +45° about the y-axis. Simultaneously, it moves along the x-axis from x=0 to x=100.

```
def cubedef;
    include cube;
end;
i cubeinstance (cubedef -ry -45);
(move cubeinstance 0 1
    (linear -ry (0 -90) (1 90))
    (linear -tx (0 0) (1 100)))
```

The word 'cubeinstance' following the keyword **move** indicates the object to which the **move** is being applied. The number '0' immediately following is the starting time of the motion. Before global time 0, the motion will have no

effect. At time 0, the motion becomes active, immediately adding a rotation of -90° about y and a translation of 0 in x to the cubeinstance's transformation list.

The '1' following the 0 indicates the duration of the motion. This particular motion has a duration of 1, meaning that the motion will be completed at time 1 and will not change thereafter.

The two lines below the 'move,' each beginning with the word 'linear', define the motion. This motion has two components. The rotation about the y-axis is defined by the first component, which states that the motion style is to be linear, that the operation is to be a rotation about y, and that the value of the rotation is to be -90° at time 0 and +90° at time 1. These rotations are concatenated onto the -45° rotation specified in the cubeinstance's instantiation, resulting in a total rotation of -135° at time 0 and a rotation of +45° at time 1. The second component describes the translation in x in a similar way.

In this example, the components were conveniently declared to start at the first time the motion became active, and to halt at the same time the motion did. It is sometimes useful to use times outside of the motion's range in the component specifications. The start time and duration given in the motion definition can be thought of as defining a window into the values listed in the motion's components. In the example below, this windowing feature is used to eliminate the two endpoints of a spline.

```
def cubedef;
    include cube;
end;
i cubeinstance (cubedef);
(move cubeinstance 5 5
    (cspline -tx (0 3) (5 3) (10 -3) (15 -3))
    (cspline -ty (0 0) (5 3) (10 6) (15 9)))
```

Since the above **move** statement begins at time 5 and has a duration of 5, only the component values at times 5 and 10 will be interpolated. The values at times 0 and 15 will fall outside of the window, but will still be active in shaping the curve.

The **move** statement shown here is useful when a motion is to be used only once. **Move** is the simplest of the **bump** statements, and perhaps the most commonly used.

3.3. The mdef and mapply statements

If a motion were to be used several times, either applied to a number of objects, or multiple times to a single object, an **mdef** to define the motion followed by an **mapply** applying it to specific objects would be more appropriate. The following example creates two revolving planets, one at the origin and the other five units away on the x-axis. Both rotate about the origin from -90° at global time 0 to +90° at global time 1. As the first planet was instantiated at the origin, this is merely a rotation about its center. But since the second planet was translated before the rotation was applied, it will move in a circle of radius five about the origin.

```
def protoplanet;
    include planetfile;
end;
i planet1 (protoplanet);
i planet2 (protoplanet -tx 5);
(mdef spin 0 1
    (linear -ry (0 -90) (1 90) ))
(mapply planet1
    (spin 0 1));
(mapply planet2
    (spin 0 1));
```

All time values given within an **mdef** are local to the **mdef**. It is possible for a motion to start at any global time, and to be scaled to any duration in global time units. This is illustrated in the following example, identical to the previous one, except that planet1 spins from time 5 to time 15, while planet2 spins from time 10 to time 30. The same spin motion is used, but the duration is different in the two **mapplys**.

```
def protoplanet;
    include planetfile;
end;
i planet1 (protoplanet);
i planet2 (protoplanet -tx 5);
(mdef spin 0 1
    (linear -ry (0 -90) (1 90) ))
(mapply planet1
    (spin 5 10));
(mapply planet2
    (spin 10 20));
```

When the motion is applied, the starting time (5) of the first **mapply** is mapped to the starting time (0) of the **mdef**, and the **mapply**'s ending time of 15 (initial time 5 + duration 10) is mapped to the **mdef**'s ending time of 1 (initial time 0 + duration 1). This results in the motion being stretched to a duration of 10 for the first **mapply**, and in a similar way, to a duration of 20 for the second. The length of motions specified in **move** statements cannot be scaled in this way, as their times are given in global values when the motion is defined.

It is also possible to change the duration of the motion within its **mdef** in order to provide a finer time grain. These values, however, are local to the **mdef**, and will have no effect on its duration in global time. The example below is identical to the previous one, except that the duration of the motion, within the **mdef**, has been changed from 1 to 100 for purposes of illustration. It will have exactly the same effect as its predecessor.

```
def protoplanet;
    include planetfile;
end;
i planet1 (protoplanet);
i planet2 (protoplanet -tx 5);
(mdef spin 0 100
    (linear -ry (0 -90) (100 90)))
(mapply planet1
    (spin 5 10));
(mapply planet2
    (spin 10 20));
```

There are also two predefined pseudo-motions which can be applied to objects. These are **appear** and **disappear**, which make objects visible and invisible. Objects are visible by default and only become invisible when a **disappear** is acting on them, so **appear** is actually unnecessary. It is included for completeness. These pseudo-motions are applied in exactly the same way as ordinary motions, as shown in the following example. Here, a cube becomes invisible from time 5 to time 10.

```
def protocube;
    include cube;
end;
i cube1 (protocube);
(mapply cube1
    (disappear 5 5))
```

3.4. The meval and mexecute statements

There are two other **bump** extensions to UNIGRAFIX, **meval** and **mexecute**. Both allow the user to use time-dependent algebraic expressions in place of constants in standard UFO statements. These expressions are written in C-language syntax, and may make use of the operators { ( ) - + > < <= >= ?: * / }. The expressions to be evaluated are delineated by the reserved character '$', and may contain the variable 'T', equal to the current global time. **Meval** is designed to let the user avoid motion specifications entirely, and hand-code the desired transformations. **Mexecute** is a general escape mechanism, allowing calls to the shell from within a **bump** file.

3.4.1. The meval statement

Meval allows arbitrary transformations, with no consideration of the logic of the geometry involved. For example, the following code segment shows **meval** being used to define a square whose top two vertices move together from time 0 to time 1. Note that the **meval** is enclosed by a def statement, made necessary by the implementation of **meval** and **mexecute**.

```
def quadrilateral;
    (meval
        v A $1-T$ 1 0;
        v B 1 -1 0;
        v C -1 -1 0;
        v D $T-1$ 1 0;
        f (A B C D);
    )
end;
```

Meval gives the user the power to describe changing objects which could not be defined otherwise. This power is not without its dangers, for it is easily possible to create tremendous geometric blunders with **meval**, such as non-planar faces or self-intersecting solids, which are impossible for the existing renders to display properly.

3.4.2. The mexecute statement

Mexecute issues shell commands from within a **bump** file, and injects the result of the command into the file. This provides the ability to call arbitrary programs with time-dependent arguments, and place their output into an animated sequence. In this next example, we show **mexecute** being used with the UNIGRAFIX program ugtrunc, to define a shape which is truncated from a cube into a cube octahedron as time progresses from 0 to 50, and restored to a cube from 50 to 100.

```
def shape;
    (mexecute
    cat cube | \
      ugtrunc -t $(T < 50) ? (T/50) : (2-(T/50))$
    )
end;
```

Caution must be exercised when using **mexecute**. Although the above example works in theory, the shapes returned from ugtrunc with truncation values of exactly 0 or 1 will cause the renderers great difficulty. In addition to such boundary condition problems, one should be aware that any error messages or diagnostics sent to standard output by the program called by the **mexecute** will be injected into the file, almost certainly causing syntax errors when the file is reparsed, and making **bump** halt abruptly. But if due care is taken,

mexecute can be the most powerful statement in **bump**'s vocabulary, allowing the introduction of arbitrary time-dependent elements.

Furthermore, one can use **mexecute** solely for its abilities to change the environment outside of the **UFO** file being animated. For example, the UNIGRAFIX generator program mkworm first reads an 'axfile,' which defines a wire in space, and then sweeps a polygon along this wire, outputting the resulting shape. A program which creates a time-dependent axfile for mkworm could be called from a **mexecute** within the **UFO** file and given the current time as a command-line parameter. If this program does not write to standard output, it will have no effect within the **UFO** file. This could be followed by another **mexecute** call to mkworm, which would generate a worm along the newly-created axfile and insert it into the **UFO** file, with the final result being a worm which squirms.

4. INTERNALS OF BUMP

This section gives a more detailed explanation of the implementation of **bump**. A good understanding of this section is essential for anyone planning to modify **bump**. There are two phases of **bump**'s work: A preprocessing step, done once at the beginning of a sequence, and a loop executed for each frame produced.

4.1. Preprocessing

4.1.1. Initial parse

After having opened its input file, **Bump** then gives the file pointer to the UNIGRAFIX parsing routine UGread(), which parses it and returns a pointer to a **UG** data structure.

Briefly, the UNIGRAFIX data structure is a set of circular linked lists, where each list contains one type of statement. All the face statements are in one list, all the edges in another, and so on. Each statement has two lists of pointers, the 'to' and 'from' lists, where the 'to' list contains pointers to all statements which refer to this statement, and the 'from' list has pointers to all statements referred to by this statement. These pointers form a tree which branches at each def statement. There is an implicit def pointing to the entire file, which is instantiated by the camera. This data structure, recently developed by Mike Natkin, is more fully described in the UG(3G) man page [6].

All of **bump**'s operations from now on are modifications to this data structure. UGread does not understand **bump**'s statements, but is intelligent enough to place all unrecognized statements within parentheses into a special extension of the **UG** data structure, a UG_ESCAPE_STATEMENT. **Bump** then searches the data structure for these escape statements.

4.1.2. Processing of mdefs and moves

Bump first processes all **mdef** and **move** escape statements in the **UG** data structure. For each **mdef** which it finds, it builds an internal representation of the motion, adds it to its list of defined motions, and deletes the **mdef** statement from the data structure. The internal representation of a motion will be described below. For any **moves** it encounters, it builds a motion representation corresponding to the one described by the **move**, stores it in the list of defined motions, and replaces the **move** statement with an **mapply** referring to the newly-constructed motion. For example, the **move** statement below,

```
(move cube 10 10
    (linear -ry (10 0) (20 90))
```

becomes the **mdef/move** pair,

```
(mdef Internalmdefxxx 10 10
    (linear -ry (10 0) (20 90))

(mapply cube (Internalmdefxxx 10 10))
```

4.1.3. Internal representation of a motion

A motion's representation consists of the (local) initial time and duration of the motion, and a list of the individual actions comprising the motion. These entries can be either transformations or references to other motions. A nested motion reference consists of the name of the motion, the (local) time at which it will start, its duration, and the (optional) number of repetitions. A transform is somewhat more complex. It consists of the type of transform (-rx, -sa, etc), the motion style (currently one of 'step', 'linear', 'bspline' or 'cspline'), a count of the number of parameters to the transform, and a list of time values with their associated parameters. A transformation typically has only one parameter, but up to sixteen are allowed. This makes it possible to interpolate between four by four matrices. A motion definition such as the following,

```
(mdef slide-and-spin 0 10
    (linear -tx (0 0) (10 100))
    (spin 0 5 2))
```

which describes a motion of sliding in x from 0 at time 0 to 100 at time 10, while spinning twice with a duration of 5 for each spin, is internally represented by a structure similar to figure 3.0.

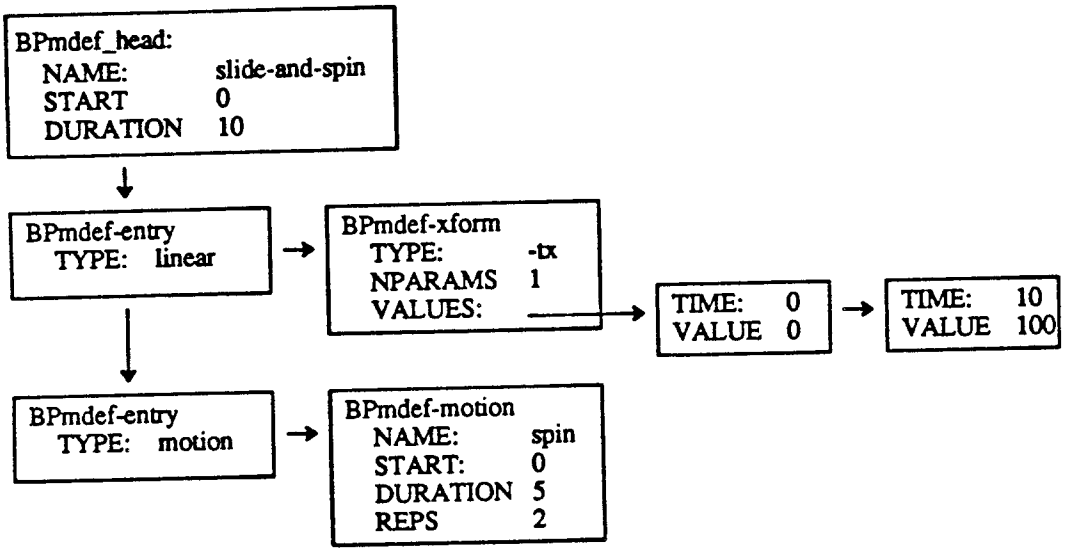


Figure 3.0: Internal motion representation

4.2. Runtime

After the preprocessing is completed, **bump** goes into a loop to produce the consecutive frames of its animation. Once it has generated a frame, it outputs the frame in one of two ways. If an output file prefix was specified in the command line, the UFO description of the current frame is written into a file whose name begins with the given prefix and is suffixed with the frame number. If no output prefix was specified, **bump**'s default action is to immediately render the frame via UGplot, put it into the Ikonas frame buffer, and command the VCR to record the frame.

This default can be changed by specifying a rendering file on the command line. A rendering file will contain the text of a shell command line, presumably one which calls a renderer with several arguments. If the name of one is specified, **bump** will open the rendering file for each frame, read its current contents, and cat the ASCII description of the current scene through the

command line given in the rendering file. For example, if the rendering file contained,

```
ugplot -sa -ep 10 20 -30 -va 40
```

Bump would then execute a line similar to the following after every frame.

```
cat bumpout.xxx | ugplot -sa -ep 10 20 -30 -va 40
```

4.2.1. Stored state

Before making any changes to the data base, **bump** stores the current state of the object being changed, so that the data structure can be restored after the frame has been rendered. For instances or arrays, the stored information is a pointer to the current end of the transformation list for this object, so that the transformations appended specifically for this frame can be easily deleted. For statements which are not to appear in the current frame, such as any invisible instances or arrays and any **bump** statements, a pointer to the statement is saved so that it can be returned to the scene later. In the case of **mevals** and **mexecutes**, the original contents of the enclosing def are retained, so that the def can be restored to its original state after the frame has been rendered.

4.2.2. Applying motions

The most complex work is done when motions are applied to objects, via an **mapply**. Much of this complexity is due to nested timeframes. As motions are allowed to nest indefinitely, it is possible to have an arbitrary number of nested timeframes. At a minimum, there are two: the global time, and this time normalized to the range of the motion that is being applied. To illustrate, assume that you are applying a motion beginning at global time 5 and with a duration of 10. This makes the motion's range, in global time, from 5 to 15. However, the motion's definition may specify a different range, perhaps from 0 to 100. Thus, global time 5 would be mapped to local time 0, global time 10 to local 50, and global 15 to local 100. These nested timeframes make it possible to stretch a motion over any duration in global time.

The first step in processing an **mapply** is to check whether or not it is active at this time. If the current (global) time is equal to or later than the start time specified in the **mapply**, it is active. If the **mapply** is active, BUMP generates a list of transformations representing the results of the named motion at this time, and appends it to the object's transformation list. For arrays, the

transformations apply to the incremental transform between array elements. If **mapply** is not active, then no transformations are appended for this **mapply** at this time.

Generating the list of transformations is the heart of **bump**. **Bump** first searches its list of defined motions for the motion referred to. This is a simple linear search, as it seemed unlikely that a single scene would ever contain enough different motions to make hashing necessary. In the most complex scenes of the **bump** demonstration video, no more than a half-dozen motions were ever defined at one time, so this appears to have been a reasonable assumption.

4.2.3. Prior repetitions

Bump first determines how many, if any, repetitions of the motion have been completed before this time. This is done by subtracting the motion's start time from the current global time, and then repeatedly attempting to subtract the motion's duration, as specified in the **mapply**, from the remainder. If this is possible, and if the number of times this has been done is still less than the number of repetitions called for in the **mapply**, a transformation list representing the transformation at the motion's final time is obtained from BPmake_tform() and appended to the moving object's transformation list. This will be done once for each prior repetition which has run to completion. The mechanisms for generating these transformation lists will be covered in detail below.

Once the transformations for any prior repetitions have been appended, **bump** tackles the current repetition. The global time remaining, once the starting time and the time required for all prior repetitions have been subtracted, is normalized into the **mdef**'s range. This is done by dividing the remaining time by the duration specified in the **mapply**, multiplying the result by the **mdef**'s duration, and adding to this product the **mdef**'s starting time. If the result is beyond the motion's ending time, it is set to this ending time, with the result that moving objects stop in place once a motion expires.

4.2.4. Building transform lists

The routine BPmake_tform() takes a pointer to a motion, and a time value within the motion's range, and builds and returns a list of transformations representing the motion at this time.

A motion contains a list of components, each of which can be either a call to another motion, or a transformation. Nested motion calls may result in any number of entries being added to the parent motion's transformation list, while each transformation will add at most one.

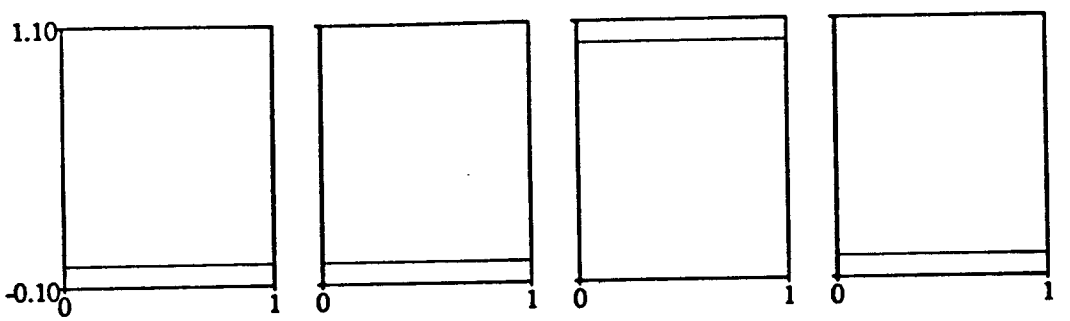
In the case of nested motions, BPmake_tform() calls another routine to recurse on the nested motion, using the time normalized into the parent motion's time as the new 'global' time. The recursion will return a list of transformations, which BPmake_tform() will append to its list of transforms for the parent motion.

In the case of a transformation, **bump** calculates its value at this time and appends a matching entry to the motion's transformation list. Values are calculated according to the motion's style. **Bump** currently supports four different motion styles, which are stepped motion (STEP), linear interpolation (LINEAR), an interpolating cardinal spline (CSPLINE), and an approximating b-spline (BSPLINE). In addition, if one of the two pseudo-motions **appear** and **disappear** is encountered, no transformation is added, but the visibility flag for the current object is set or unset accordingly.

4.2.5. Calculating transformation values

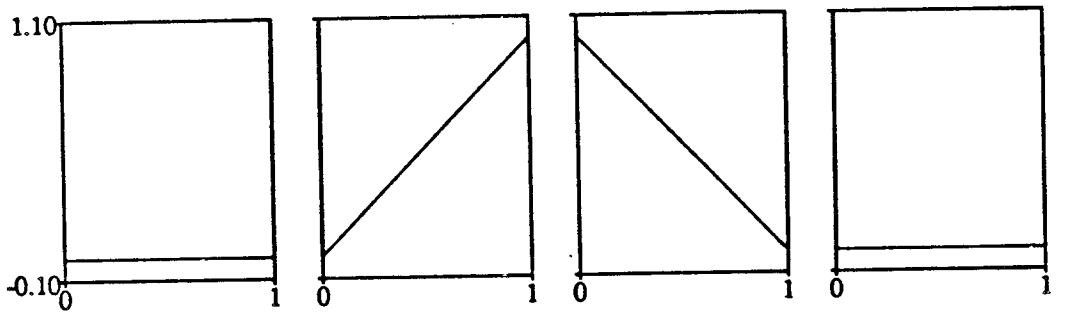
The steps in determining a transformation value at a specific time are as follows: First, **bump** finds which interval of the component this time is within. For example, if the component were specified with the time-value pairs ((0 1) (1 2) (2 4) (3 8)), the time 2.5 would fall in the interval between 2 and 3, with the corresponding values 4 and 8. If the specified time is earlier than all of the time values in the list, this component of the motion is not active, and no transformation is added for it. If the specified time is later than all of the time values, the computation continues as if it were equal to the latest value, ensuring that objects lock in position once their movements run to completion.

Bump then selects the set of basis functions appropriate to this motion style. The values for all styles of motions are calculated in the same way, the only difference being in the basis functions which are used. Stepped motion has the simplest basis functions, while the two types of splines have the most complex. In all cases, the basis-function function for a specific style of motion returns the value of the i-th basis function at u, where i is from 1 to 4, and u ranges between 0 and 1. Graphs of the basis functions are presented in figures 4.0 through 4.3.



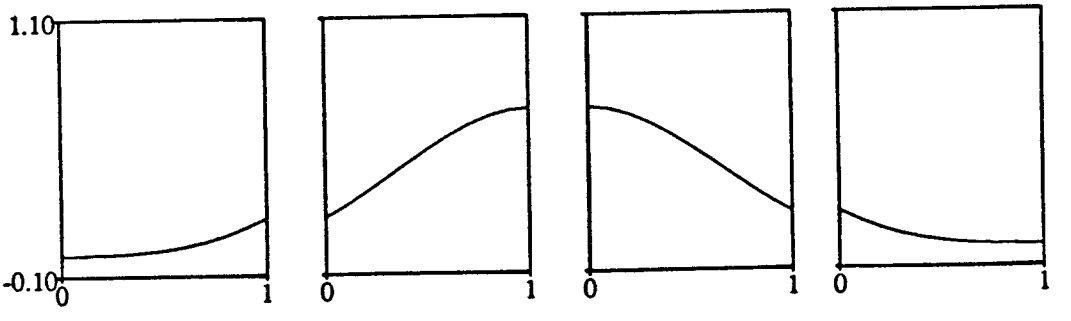
$f_3(u) = 0$ $f_2(u) = 0$ $f_1(u) = 1$ $f_0(u) = 0$

Figure 4.0: Stepped motion basis functions



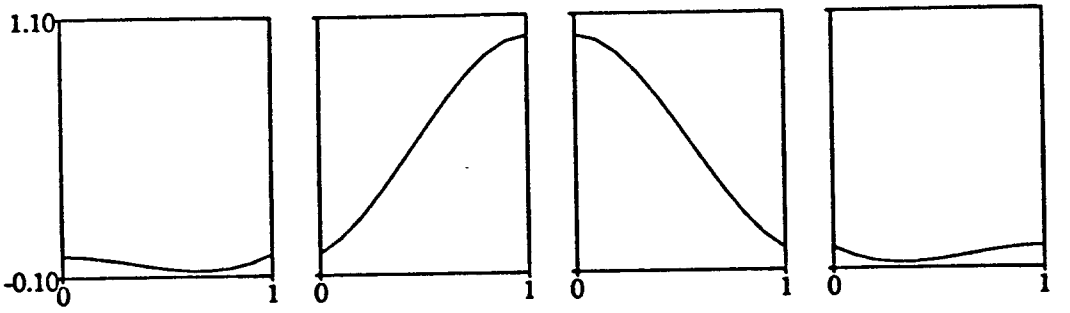
$f_3(u) = 0$ $f_2(u) = u$ $f_1(u) = 1-u$ $f_0(u) = 0$

Figure 4.1: Linear interpolation basis functions



$f_3(u) = \frac{1-3u+3u^2-u^3}{6}$ $f_2(u) = \frac{4-6u^2+3u^3}{6}$ $f_1(u) = \frac{1+3u+3u^2-3u^3}{6}$ $f_0(u) = \frac{u^3}{6}$

Figure 4.2: B-spline basis functions



$$f_3(u) = \frac{-u^2+u^3}{2} \qquad f_2(u) = \frac{u+4u^2-3u^3}{2} \qquad f_1(u) = \frac{2-5u^2+3u^3}{2} \qquad f_0(u) = \frac{-u+2u^2-u^3}{2}$$

Figure 4.3: Cardinal spline basis functions

The final value for any computation is computed in the classic spline manner, [2] where the final value is the sum of the values at the various nearby intervals, weighted by their corresponding basis functions. Specifically,

$$value(i,u) = f_0(val[i-2]) + f_1(val[i-1]) + f_2(val[i]) + f_3(val[i+1])$$

Where *val*[*i*] is the value at the right end of the current interval, *val*[*i*-1] the value at the left end of the current interval (also the right end of the previous interval), and so forth. If we are at one end or another of the list of values, so that there is no value for either the next or prior interval, this imaginary value is considered to be zero.

Once the transformations have been appended, **bump** checks whether a **disappear** is acting on the object at this time. If so, an invisibility flag is set so that the object will be removed from the data structure before the frame is rendered. Conversely, if an **appear** is acting on the object, the invisibility flag is unset, cancelling any previous **disappears**. Objects are visible by default.

4.2.6. Meval and Mexecute

The evaluation of **meval** and **mexecute** statements is fairly straightforward, compared to the complexities required for motions. In the first phase of both, **bump** finds the def enclosing the **meval** or **mexecute**. This is done in a horrendously inefficient manner, iterating through the list of all def statements until **bump** finds one whose contents pointer matches the UGfile containing the **meval** or **mexecute**. This was made necessary by the construction of the UG data structure, which includes both pointers from statements to their

enclosing UGfiles and pointers from defs to the UGfiles they enclose, but lacks pointers from UGfiles to the enclosing defs. If a large number of defs are used in a UFO file, this implementation will cause bump to slow linearly with the number of defs. A more efficient method of locating the defs is suggested as a future improvement in section 5.1 of this paper.

Once bump locates the enclosing def, it writes the contents of this def into a file in /usr/tmp. As it writes, it searches for any expressions delineated by dollar signs ("$"). It evaluates these expressions, and writes the result of this evaluation in place of the original expression into the temporary file. The actual evaluation is done by a simple yacc grammar [1] which can handle the operations and symbols { ( ) - + > < <= >= ?: * / } but does not understand trig functions.

In the case of an meval, the resulting output is then reparsed via UGread(), and the pointer of the enclosing def is set to the structure returned from this UGread() [6]. Mexecute is similar to meval, except that after the expressions have been evaluated, the resulting text is given to the shell for execution. The shell's output is redirected to a second temporary file, and this second file is reparsed and inserted into the enclosing def. This can cause problems if the shell returns text which is not legal UFO text, as UGread() will return a syntax error and exit when it reads it.

It is necessary to enclose mexecutes and mevals within def statements, due to the above-described implementation of the two statements. They need not be the only statements within the def, but one should note that the entire enclosing def is reparsed for each time value, so care should be taken to put as little as possible within a def containing one of these two statements.

4.2.7. Rendering the frame

Once the data structure has been modified to represent the scene at the current time, the pointer to this structure is given to a rendering routine. There are currently two different ways of presenting the output frames. If a output file prefix was given on the command line, the rendering routine merely calls UGwrite to write out the frame in standard UFO format to a numbered file with this prefix, and these frames are rendered and recorded at a later time.

Alternatively, if no output file prefix was specified, the rendering routine puts the frame into a temporary file in /usr/tmp, renders it, and then deletes it. By default, bump renders the frames through UGplot with default arguments, puts them into the Ikonas frame buffer, and sends a VAS-IV record signal to

/dev/ttyh4. If a rendering file was specified by the user, **bump** uses the shell line contained within this file as the command line through which the frame should be rendered.

All these schemes involve converting the data structure into **UFO** ASCII format before giving it to the renderer. Eventually, we plan to connect **bump** directly to the renderers, saving a great deal of time by simply passing the data structure pointer.

4.2.8. Restoring the data structure

When the rendering routine completes, the data structure is restored to its original state. This involves linking back in any statements which were unlinked earlier, such as invisible objects and all **bump** escape statements. Also, the transformations which were appended to any moving objects' transformation lists are now truncated back off. Once the data structure has been restored, the next frame is calculated.

5. DISCUSSION AND CONCLUSIONS

In the creation of the demonstration video, **bump** has clearly shown that it and the supporting UNIGRAFIX environment are adequate for the creation of interesting animation. However, the experience of designing and animating the video has also pointed out that there are several improvements that would make animation easier and make possible even more interesting creations.

5.1. Deficiencies of BUMP

A notable deficiency of **bump** is that there is currently no way to move the camera. While a UG statement describing the camera does exist, **bump** does not know how to animate it. This was a decision forced both by the pressures of time and the limited short-term utility of such an feature; at the time **bump** was being written, no renderers existed that could have used the camera statement had **bump** been capable of handling it.

It is possible, though awkward, to create camera movement by running **bump**'s **UFO** output files through a rendering routine which keeps track of the current time and changes the camera parameters to the renderer accordingly. For the demonstration scenes in which the camera moves, this was the method used. Another method of creating camera movement is to use the side effects of **mexecute** to create a time-dependent rendering file. However, both these methods make it very difficult to design camera movements involving

anything other than simple linear interpolation.

Bump lacks several other abilities, which are not as essential, but would be desirable for some types of animation. These include the ability to change light sources with time, and to change the colors of objects. Also, the addition of a few more motion styles would be useful. Especially nice would be "accelerate" and "decelerate" styles, suitable for introducing an object to the scene or removing it. For more cartoon-like animation, splines that interpolate but overshoot their control points in a predictable manner would be appropriate [4].

While the current implementation of the motion styles themselves has been elegant and effective for the simple cases it has been so far used on, the system has several shortcomings. The functions which compute the basis functions do not have access to the list of specified values at each time, so they cannot change their basis functions according to these values. This makes it impossible to implement more complex splines which take these values into account, such as P-splines [10]. Also, **bump** implicitly assumes equal knot spacing, while knot spacing dependent on the Euclidean distance between parameter values would allow smoother motions [2].

Motion blur would be a worthwhile addition to **bump**. This cannot be implemented solely at the renderer level, since the addition of blurs requires a knowledge of the previous positions of the moving objects. The current renderers accept only a single frame at a time, and thus lack the information necessary for motion blurring. Blurring, if it is to be added at all, must be put into the scene by the process that is calculating the motions in the first place. As **UFO** currently has no provision for describing blurs, this would require heavy modifications to the scene description language.

Three other deficiencies of **bump** lie in the implementation of **meval** and **mexecute**. Currently, the entire contents of the enclosing def are reparsed for each time value. This was a decision based on the convenience of switching the def's contents pointer, compared to the difficulties of replacing only those elements affected by the **meval** or **mexecute**. However, the contents are reparsed in isolation, with no access to the context of the surrounding file. This makes it impossible, for example, for faces to refer to vertices or colors defined earlier, outside of this def. While this can be worked around by the knowledgeable programmer, it is a serious defect in **bump**, as apparently-correct definitions can fail, due to the existence of a **meval** or **mexecute** in the def.

Secondly, one should note that there is an implicit def surrounding the entire **UFO** file, which is instantiated by the camera. Following this train of thought, it should be possible for a "free-floating" **meval** or **mexecute**, outside of any explicit def statement, to work properly, albeit at the cost of reparsing the entire input file for each time value. This is not implemented, due to the fact that there is no representation of this implicit def in the UG data structure, so there is no corresponding pointer for **bump** to switch.

A final difficulty in the implementation of these two statements is found in the mechanism for locating their enclosing defs. As mentioned in section 4.5, **bump** finds the enclosing def statement by iterating through the list of all def statements until it finds one with a contents pointer that matches the file containing the **meval** or **mexecute**. If a large number of defs are used in a file, this implementation will cause **bump** to slow linearly with the number of defs. A more efficient method would be to keep a separate list of all defs containing **meval** or **mexecute** statements. This list could be generated at the time the first frame was generated, using the current inefficient method. In the creation of subsequent frames **bump** would be able to restrict its search to this newly-constructed list, thus shortening the time required. If further speedups become necessary, the list could be stored in a data structure more appropriate for quick access, such as a tree or hash table.

Another addition to **bump** could be a **.bumprc** file, similar to **.ugirc** and **.mailrc**. This file would contain values for various environment settings, such as camera specifications and rendering options, the location of the recorder and the signal that should be sent to it, and any other appropriate information.

5.2. Supporting environment problems

There are a number of problems in the UNIGRAFIX environment which hamper the creation of effective animation. These should be addressed at some point.

A minor point is that it was difficult to preview the animation, as UGplot is incapable of rendering wireframe scenes for the Ikonas. This made it necessary to preview all frames with shaded faces, requiring roughly four times as long as would have been necessary with wireframe rendering.

Bump is currently forced to translate its frames back into the **UFO** format, as there are no renderers that can take a data structure pointer directly as input. Though not fatal, the time required by this process and its inverse as the renderer reads the **UFO** file combine to slow the process of displaying a single

frame by roughly a factor of two.

A more serious problem is that no renderers currently understand the data structure's camera statement, making it impossible to define camera movements except by means of the cumbersome process described above. Also, all current renderers except the costly **UGRAY** explode when something passes behind the camera. Until a new renderer is completed without these flaws, or an old one modified, it will be impossible to produce animation with interesting camera movements.

5.3. Conclusion

The decision that **bump** should avoid ASCII scene descriptions and work directly on the new **UG** data structure should be especially beneficial in the long run, although it was more difficult to implement in this way than on the more straightforward ASCII level. Also, the decision to use a LISP-like syntax for **bump**'s motion description language will mesh well with future plans for converting the **UFO** language into a similar syntax.

In the production of the demonstration video, **bump** has been proven capable of producing interesting and informative animation. Evaluated in the light of its original goals of providing a variety of animation extensions to UNIGRAFIX, while preserving the existing UNIGRAFIX philosophy and requiring minimal changes to the existing scene description language, it seems that **bump** can be judged a success.

Acknowledgements

This work was supported in part by the AT&T Information Systems One Year On Campus program, and by Tektronix, Inc.

I would like to thank my research advisor, Dr. Carlo Séquin, for his advice, suggestions, and encouragement throughout the development of **bump**. Professor Dave Anderson provided a valuable outside viewpoint. Mike Natkin wrote the **UG** parser, without which **bump** would not have been able to function, and Greg Couch maintained a sane working environment.

Mike Francisco of the Berkeley Educational Television Office provided invaluable help in assembling the demonstration videotape.

REFERENCES

1. *UNIX System V Support Tools Guide*, AT&T Information Systems, April 1984.

2. R. H. Bartels, J. C. Beatty and B. A. Barsky, *An Introduction to Splines for use in Computer Graphics and Geometric Modelling*, Morgan Kaufmann Publishers, inc., Los Altos, California, 1987.

3. E. Chang, Brown Animation Generation System (BAGS), Brown University Graphics Group Report, 1986.

4. J. Lasseter, Principles of Conventional Animation Applied to 3D Computer Animation, *Proceedings of SIGRAPH '87* , 1987, 35-43.

5. D. M. Marsh, UgRay: An Efficient Ray-Tracing Renderer for UniGrafix, Computer Science Division, EECS, UCB, Berkeley, CA 87/360, Fall 1987.

6. M. Natkin, UG(3G) man page, on line documentation, degas.berkeley.edu.

7. C. H. Séquin, M. Segal and P. Wensley, UNIGRAFIX 2.0 User's Manual and Tutorial, Computer Science Division, EECS, UCB, Berkeley, CA 83/161, Fall 1983.

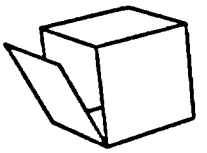
8. C. H. Séquin, The Berkeley UNIGRAFIX Tools Version 2.5, Computer Science Division, EECS, UCB, Berkeley, CA 86/281, Spring 1986.

9. C. H. Séquin and D. M. Marsh, Unpublished work, 1987.

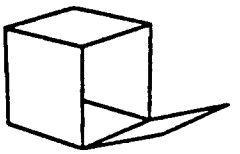
10. C. H. Séquin, Procedural Spline Interpolation in UniCubix, Computer Science Division, EECS, UCB, Berkeley, CA 87/321, Spring 1987.

11. P. Strauss, A Tutorial Guide to the SCEFO Language, Brown University Graphics Group Report, 1986.

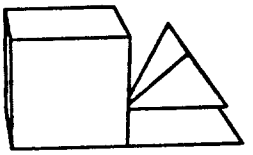
Appendix A: Examples of Animation



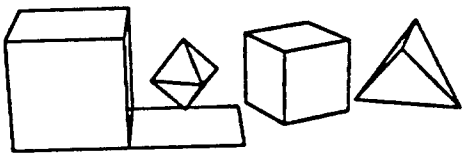
time = 5



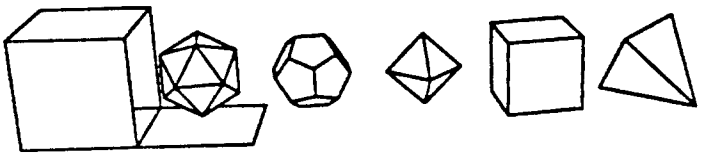
time = 15



time = 25



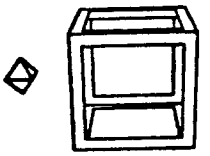
time = 35



time = 45

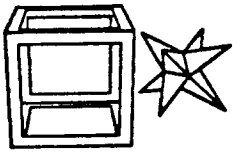
```
{first part of the meval sequence.  Box produces platonics.  Takes 100 frames.}

l sun 0.7 -1 1 -1 ;
l hlightsun 0.2 -1 0 -1 ;
l glow 0.1 ;

c orange 1 5 1 ;
c yellow 1 55 1 ;

def box;
v XYZ   1 1 1;
v XY    1 1 -1;
v XZ    1 -1 1;
v YZ    -1 1 1;
v X     1 -1 -1;
v Y     -1 1 -1;
v Z     -1 -1 1;
v N     -1 -1 -1;
f ( Y YZ XYZ XY );
f ( XY XYZ YZ Y );
f ( Z XZ XYZ YZ );
f ( YZ XYZ XZ Z );
f ( N Z YZ Y );
f ( Y YZ Z N );
f ( N X XZ Z );
f ( Z XZ X N );
f ( N Y XY X );
f ( X XY Y N );
end;

def lid;
v XYZ   1 1 1;
v XY    1 1 -1;
v XZ    1 -1 1;
v X     1 -1 -1;
f ( X XY XYZ XZ );
f ( XZ XYZ XY X );
end;

(mdef open 0 100
        (linear -tx (0 -1))
        (linear -ty (0 1))
        (linear -rz (0 0) (100 -90))
        (linear -ty (0 -1))
        (linear -tx (0 1)))

(mdef containerturn 0 100
        (linear -ry (0 180) (100 0)))

(mdef turn 0 100
        (linear -ry (0 0) (100 180)))

def container;
i box (box);
i lid (lid);
(mapply lid (open 0 20))
end;

i container (container orange -sa 1.5);

(move container 0 100
        (containerturn 0 20)
        (linear -tx (0 -12) (20 6) (70 -24))
        (disappear 60 50))

def tetra;
```

```
        include tetra;
end;

def cube;
        include cube;
end;

def octa;
        include octa;
end;

def icosa;
        include icosa;
end;

def dodeca;
        include dodeca;
end;

i tetra (tetra yellow);
i cube (cube yellow);
i octa (octa yellow);
i dodeca (dodeca yellow);
i icosa (icosa yellow);

(move tetra 0 100
        (disappear 0 20)
        (turn 0 100)
        (linear -tx (0 6) (100 6))
        (cspline -tz (60 0) (100 -20))
        (cspline -tx (60 0) (100 5))
        (cspline -ty (60 0) (100 15)))

(move cube 0 100
        (disappear 0 25)
        (turn 0 100)
        (linear -tx (0 3) (100 3))
        (cspline -tz (60 0) (100 -20))
        (cspline -tx (60 0) (100 5))
        (cspline -ty (60 0) (100 -5)))

(move octa 0 100
        (disappear 0 30)
        (turn 0 100))

(move dodeca 0 100
        (disappear 0 35)
        (turn 0 100)
        (linear -tx (0 -3) (100 -3))
        (cspline -tz (60 0) (100 -20))
        (cspline -tx (60 0) (100 -5))
        (cspline -ty (60 0) (100 -5)))

(move icosa 0 100
        (disappear 0 40)
        (turn 0 100)
        (linear -tx (0 -6) (100 -6))
        (cspline -tz (60 0) (100 -20))
        (cspline -tx (60 0) (100 -5))
        (cspline -ty (60 0) (100 15)))
```

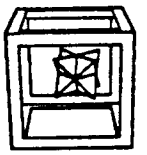


time = 0

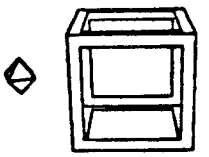
time = 25



time = 50

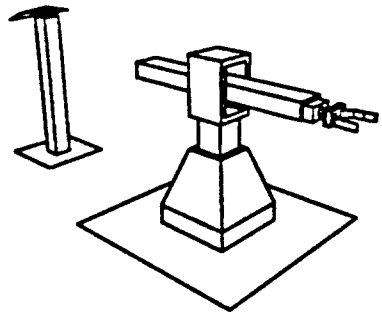


time = 75

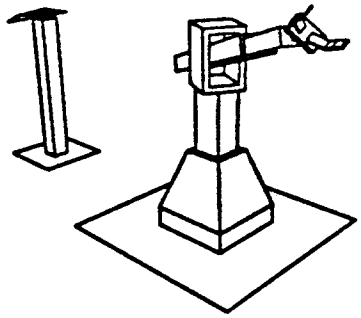


time = 100

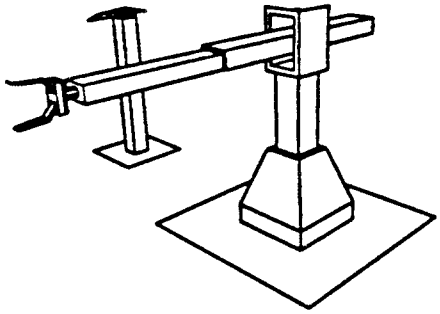
```
{ fourth of the meval demo sequence.  Octahedron | ugstar.  Takes 125 frames }

l sun 0.7 -1 1 -1 ;
l hlightsun 0.2 -1 0 -1 ;
l glow 0.1 ;

c yellow        1 55 1 ;
c orange        1 3 0.975;

def filter;
        include filter;
end;

def octa;
        include octa;
end;

def larger;
        (mexecute
        (cat octa | ugstar -h \
        $(T <= 12.5) ? 0 : ((T >= 37.5) ? 2.5 : (2.5 * (T-12.5) / 25))$))
end;

def smaller;
        (mexecute
        (cat octa | ugstar -h \
        $(T <= 62.5) ? 2.5 : ((T >= 87.5) ? 0 : (2.5 - (T - 62.5) / (25/2.5)))$))
end;

def filters;
        i (filter orange -sa 3 -ty);
end;

i filters (filters -ty 1 -tz -3);
i octa (octa yellow);
i smaller (smaller yellow);
i larger (larger yellow);

(mdef backandforth 0 125
        (linear -ry (0 45) (125 135))
        (linear -ty (0 1) (125 1))
        (linear -tz (0 -3) (125 -3))
        (cspline -tx (0 -6) (50 6) (100 -6)))

(move octa 0 125
        (backandforth 0 125)
        (disappear 12.5 75))

(move larger 0 125
        (backandforth 0 125)
        (disappear 0 12.5)
        (disappear 50 1000))

(move smaller 0 125
        (backandforth 0 125)
        (disappear 0 50)
        (disappear 87.5 1000))
```



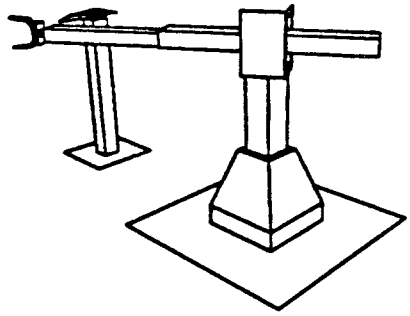
time = 0.5



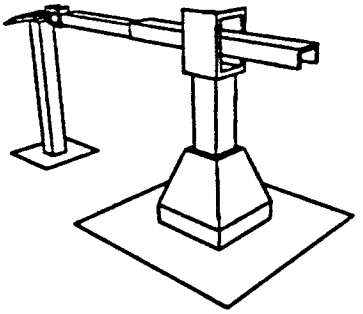
time = 1.5



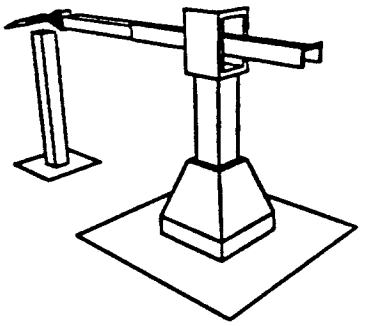
time = 2.5



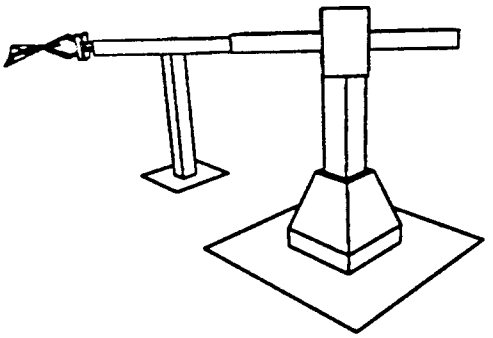
time = 3.5



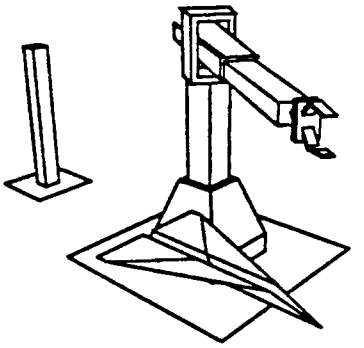
time = 4.5



time = 5.5



time = 6.5



time = 7.5

```
c yellow 1 60 1;
c colfloor .8 40 .75;
c gray 0.5 40 0.5;

l sun 0.7 -1 3 -2;
l glo 0.2;

def robot;
        (mexecute mkrobot \
                -h $(T < 2) ? (65 + T * 15) : ((T < 5) ? 95 : \
                                        ((T < 6) ? (95 + (T-5) * 10) : 105))$ \
                -r $((T<1) ? 0 : ( (T<4) ? (60*(T-1)) : ( (T<6) ? 180 : \
                                        (180 + 90 * (6 - T)))))$ \
                -d $(T < 1) ? (35 - 70 * T) : ( (T < 2) ? (35 * (T-2)) : 0)$ \
                -e $(T < 1) ? 5 : ( (T < 2) ? (5 + 55 * (T-1)) : 60)$ \
                -t $(T < 1) ? 0 : ( (T < 2) ? (90 * (T-1)) : 90)$ \
                -a $(T < 4) ? 30 : ((T < 5) ? (5 + 25 * (5-T)) : \
                                        (T>7) ? (5 + 25 * (T-7)) : 5)$

        )
end;

def plane;
        include plane.shrunk;
end;

def platform;
        include cube;
end;

def floor;
        v       floor1  -1 0 1;
        v       floor2  1 0 1;
        v       floor3  1 0 -1;
        v       floor4  -1 0 -1;
        f       (floor1 floor2 floor3 floor4) colfloor;
end;

i (floor colfloor -sa 20 -tx -140 -ry -0.01);
i (platform gray -sx 5 -sz 5 -sy 45 -tx -140 -ty 45);
i plane (plane yellow -sa 3 -rx 90 -tx 140 -ty 91 -tz 8);
i (robot);

(move plane 0 9
        (linear -ry (0 180) (6 180) (7 90))
        (linear -tx (0 0) (7 0) (9 392))
        (linear -ty (0 0) (5 0) (6 10))
        (disappear 9 1000))
```

Appendix B: BNF Grammar of Language Extensions

BNF grammar for the BUMP extensions to Unigraflx

Definitions:

```
[]      indicate optional parts
''      indicate literal parts

start, time, duration, value    : real numbers
repetitions     : integer
obj_name        : arbitrary string
motion_name     : arbitrary string
text    : arbitrary text

white space is always ignored except to separate fields

bump_stmt       : move_stmt
        | mdef_stmt
        | mapply_stmt
        | meval_stmt
        | mexecute_stmt

move_stmt       : '(' 'move' obj_name start duration
                                '(' m_style t_form value_list ')' ')'

mdef_stmt       : '(' 'mdef' motion_name start duration
                                '(' m_style t_form value_list ')' ')'

mapply_stmt     : '(' 'mapply' obj_name component_list ')'

meval_stmt      : '(' 'meval' text ')'

mexecute_stmt   : '(' 'mexecute' text ')'

component_list  : '(' component_entry ')' [component_list]

component_entry : motion_name start duration [repetitions]

value_list      : '(' value_entry ')' [value_list]

value_entry     : time param

param : number [number [number [number ... up to 16 of them ... ]]]

t_form : '-sx' | '-sy' | '-sz' | '-sa' | -sv
        | '-tx' | '-ty' | '-tz' | '-ta' | -tv
        | '-rx' | '-ry' | '-rz' | '-ra' | -rv
        | '-mx' | '-my' | '-mz' | '-ma' | -mv
        | '-M3' | '-M4'

m_style : 'step'
        | 'linear'
        | 'bspline'
        | 'cspline'
```

Appendix C: Tutorial

Bump tutorial

The Berkeley UNIGRAFIX Movie Package, or **bump**, is an animation-generating front end to Berkeley UNIGRAFIX. It adds five new 'motion' commands to UFO, the UNIGRAFIX ASCII scene description format, namely **move, mdef, mapply, mexecute** and **meval**. These will be more fully described below.

To generate an animated scene in the simplest way, the user first creates an extended-UFO file containing objects and their movements. He then feeds this file to **bump** along with the starting time value, the number of frames, the time increment between frames, and the prefix of the output file names. **Bump** then gives the extended UFO file to the new UNIGRAFIX parser, recently completed by Mike Natkin, which constructs a matching data structure and places any statements it does not recognize, presumably concerned with motion, in a special escape statement.

Bump then processes the motion statements, and for each time increment recalculates the time-variant parameters of any moving objects, inserts these new values into the data structure, and calls an output routine. This loop continues until the specified number of frames have been computed.

Typically, the output routine merely prints the data structure as a standard UFO file, one for each frame of animation. Once **bump** has completed, these can be individually rendered and displayed by the user. There are two command-line options that can change this behaviour, namely **-r** and **-p**, which will be described in the "Sample Runs" section of this tutorial.

1. MOVE - Specify a Motion

Motions are time-dependant transformations. In particular, they are composed of the fundamental operations of translation, rotation, and scaling. A motion is described by a list of transformations and their values at specified times. Intermediate values are calculated with various styles of interpolation. The general form is:

```
( move object_name start_time duration
      ( motion_type transformation { (time param_values) } ))
```

Motions computed by **bump** are appended to the moving object's transformation list. As an example here is a turning page:

```
i page (protopage -ry -45);
 . . .
(move page 0 1
      (linear -ry (0 -90) (1 90) ))
```

In the interval from time 0 to time 1 the page swings at a constant rate from an angle of -135 degrees to an angle of +45 degrees.

2. MDEF - Define a Motion.

Motions can also be defined in the abstract and given a name, so that they can be applied multiple times or to more than one object. Once a motion has been defined, it can be applied to any object in the scene with the **mapply** statement to be discussed later. These abstract motions have their own local time frames within which all their individual motion elements are defined. When such a motion definition is applied to an object, its time frame can be scaled to stretch over any desired time duration in the context of the time frame in which the application is specified.

As a simple example, here is a motion describing a spin about the y-axis. At time zero, the rotation is zero degrees, and by time one, the object will have turned 360 degrees. The motion type is specified as linear.

```
(mdef spin 0 1
      (linear -ry (0 0) (1 360)))
```

Any object can now be made to spin once about its y-axis, by applying this motion to it via **mapply**. The pair of numbers immediately following the motion name represents the 'visible' time interval of the

motion. When the motion is applied, its local time will run through this range, and any transformations with time values out of this range will not appear. This is especially useful in specifing invisible control points on splines, as on this more complex motion definition, which describes an object rolling along an S-curve in the x-y plane.

```
(mdef roll 1 1
      (linear -rz (1 0) (2 360))
      (cspline -tx (0 0) (1 50) (2 100) (3 150))
      (cspline -ty (0 0) (1 -50) (2 50) (3 0)))
```

The first transformation applied to the object will be a rotation about z. It is then translated in the x-y plane by the next two lines, causing it to travel in an S-curve. Since the motion starts at internal time 1 and has a duration of 1, the spline control points associated with times 0 and 3 will not be interpolated. However, they will still be active in shaping the curve.

There are currently four motion types, namely **step, linear, cspline, and bspline. Step** allows objects to jump from one position to the next, with no continuity between positions. **Linear** specifies a simple linear interpolation between control points. **Cspline** is an interpolating cardinal spline. **Bspline** is an approximating b-spline, which provides the smoothest moves but is the most difficult to design.

3. MAPPLY - Apply a Motion to an Object.

Mapply is how motions are are applied to objects. Motions can be applied only to instances and arrays. The general format of an **mapply** is:

```
(mapply object_name
      ( motion_name  starttime  duration  [optional number of applications] ))
```

```
i copycube (protocube -tx 10);
(mapply copycube
      (spin 0 45 3))
```

This will swing the copycube three times around the y-axis at a distance of 10 because the motion is with respect to the coordinate system in which copycube is described. The first spin starts at global time 0, the next at 45, and the third at time 90 and terminates at 135. Each spin will begin where the prior one left off.

Applying a motion to an array will affect the incremental transform between the individual the array elements. For example,

```
a cubearray (protocube) 10 -tx 1 ;
(move cubearray 0 1
      (linear -tx (0 0) (1 9))
```

will result in an array of cubes spaced one unit apart at time zero, and spaced ten units apart at time one. If you would like to move the entire array, you would have to do the following:

```
def cubearray_def;
      a cubearray (protocube) 10 -tx 1 ;
end;
i cubearray_instance (cubearray_def);
(mapply cubearray_instance
      (spin 0 45 3))
```

An array of spinning cubes can be produced as follows:

```
def spinning_cube;
      i temp_cube (protocube);
      (mapply temp_cube (spin 0 45 3))
end;
a cubearray (spinning_cube) 10 -tx 3;
```

Multiple motions can be applied; assuming that the object 'ball' and motions 'bounce' and 'pop' have already been defined, here is a ball bouncing several times and then popping. Each of the four bounces will start from the final position of the previous one, and take 50 time-units. At time 200, the ball will pop once, taking ten units to do so.

```
(mapply ball
      (bounce 0 50 4)
      (pop 200 10))
```

4. Nested Motions

A **mdef** can contain nested motions. As an example, here we have an object stepping along a diagonal line every 5 time-units, and spinning for 10 after having taken the third of the steps.

```
(mdef leap_and_spin 0 25
      (step -tx (0 0) (5 10) (10 20) (15 30))
      (step -ty (0 0) (5 10) (10 20) (15 30))
      (spin 15 10 1))
```

There also are two predefined pseudo-motions, **appear** and **disappear**. When these are applied to an object, it is inserted into or removed from the scene. This can be useful for deleting objects which have moved behind the camera, or objects which are out of sight and need not be rendered to save time. Objects are visible by default whenever there is no **disappear** acting on them, which makes the **appear** pseudo-motion redundant. It is included for completeness. The use of these pseudo-motions looks like this:

```
(mdef twinkle 0 10
      (disappear 0 5)
      (appear 5 5))

(mapply object_name
      (twinkle 0 10 50))
```

This will result in the object disappearing for five time units every ten time units, with a total of 50 such twinkles. The **appear** in the **mdef** above is not really necessary, as the object will appear anyway when the disappear expires. However, **appear** can be used to override a **disappear**, if desired. These pseudo-motions can also be used in **move** statements, as shown in the following example which makes the object disappear from time 0 to time 5.

```
(move object_name
      (disappear 0 5))
```

5. MEVAL - Evaluate Time-dependent Fields.

meval allows the insertion of arbitrary time-dependant text. Within an **meval**, the UFO text can contain algebraic expressions referring to the global variable T, the time, and these expressions are evaluated for each frame. The fields to be evaluated are designated with the special character '**$**'. In the following example, we define a squid with three time-dependant tentacles. In the time from 0 to 1, the tentacles rotate about different axes with different phase offsets.

```
def body;
      include body_file;
end;
def tentacle;
      include tentacle_file;
end;
def squid;
      i (body);
(meval
      i (tentacle -rx $360*T + 120$);
      i (tentacle -ry $360*T + 240$);
      i (tentacle -rz $360*T + 360$);
      )
end;
```

6. MEXECUTE - Call Shell With a Generator.

Mexecute allows the insertion of output from arbitrary generators and filters. This is a powerful tool, allowing the definition of objects that change shape with time. For example, here is a cube undergoing greater and greater truncation as time progresses (presumably from 0 to 100).

```
def shape;
      (mexecute
            ugtrunc -t $T/100$ < cube)
end;
```

Both **meval** and **mexecute** require an enclosing def. This is due to the implementation of the two statements, which work by evaluating and reparsing the entire contents of the def containing the **meval** or **mexecute**, and replacing the def's prior contents with the result of the reparse.

7. SAMPLE RUNS

This section gives several examples of the more unusual ways of using **bump**. These are primarily concerned with different ways of specifing or dynamically changing the output routine. The **-r** *filename* command line option allows the user to name a file which is read for each frame to privide the rendering command line. For example, the command line,

```
bump inputfile 0 1 5 -r renderfile
```

will cause **bump** to read inputfile and then produce five frames of animation. When it is done calculating each frame, it will read renderfile and then cat the current frame through the command line that renderfile contains. If the contents of renderfile were,

```
ugplot -ep 10 20 -30
```

this would result in each frame appearing on the user's screen, viewed from the point (10 20 -30). When combined with **mexecute**, the **-r** option can provide a way to move the camera. If the extended-UFO file contained the statements,

```
def foo;
      (mexecute echo "ugplot -ep 10 20 -30 -va $T*5+5$" > renderfile)
end;
```

and the original command line was

```
bump inputfile 0 1 5 -r renderfile
```

the final result would be that frame number 0 would be rendered with view angle 5, frame number 1 with view angle 10, and so forth.

The previous two examples are incomplete, in that each frame is merely dumped to the screen and then forgotten. If one plans to record the sequence, it is necessary to use more complex renderfiles, such as the one created by the statements,

```
def foo;
(mexecute echo "ugplot -di -sf -ep 10 20 -30 -va $T*5+5$; mv rast.iv /usr/tmp/raster.$T$" > renderfile)
end;
```

This will render each frame with a different view angle and move the raster files to distinctly-named files in /usr/tmp. One can go even further and automate the entire rendering and recording process with a rendering file such as,

```
ugplot -di -sf; \
/c/grafix/bin/iv rast.iv; \
echo \\' > /dev/ttyh4; \
sleep 6;
```

This is in fact, exactly the default rendering sequence used by **bump** if no rendering file name is given on the command line.

An alternative method of simulating camera movements is to use **bump** to create consecutive scenes in individually-named UFO files, and render them via a shell that gradually changes the parameters to the renderer. This is what was done for the **bump** demonstration video. For example, if the command line were,

```
bump inputfile 0 1 5 -p foo
```

bump would produce five (presumably different) UFO-format output files named "foo.0" through "foo.4." The user could then write a simple shell to render and record these files sequentially with different parameters to UGplot for each one. Note that the **-p** option will override the **-r** option; if an output prefix is specified, **bump** will not render the frames but will merely write them to a file.

8. Camera movement

Currently, **bump** supports camera movement only in the indirect ways described above. If a more direct treatment of camera movement were to be added in the future, the following would be a simple way to describe it in a manner consistent with the rest of **bump**.

The camera would be animated via interpolations of the parameters which define it. These parameters would be referred to in **move** and **mapply** statements as if they were instances. Camera parameters are either points or scalars, and can be manipulated via normally-defined motions. Points such as '-ep', would be moved about with translations, while scalars such as '-va' could be scaled. This is demonstrated in the next few examples. The first one would 'slide' the camera right from -5 on the x-axis to +5 over one hundred time units.

```
(mdef track_right 0 100
      (linear -tx (0 -5) (100 5)))

(mapply -ep
      (track_right 0 100))

(mapply -vc
      (track_right 0 100))
```

Note that we must explicitly tell the camera to move its view center. Making the camera circle the origin at a distance of 50 while zooming in from va=90 to va=30 would be like this:

```
(move -ep 0 100
      (step -tx (0 50))
      (linear -ry (0 0) (100 360)))

(move -va 0 100
      (linear -sa (0 90) (100 30)))
```

Since the camera position is specified like a point (with -ep) in the global world system, the above turn motion would rotate the camera around the global x-axis keeping it aimed on the origin. Scaling '-va', assuming it was set to 1 initially, would generate the effect of zooming in.

9. For further reading

The man page for **bump** offers a condensed version of this tutorial. For a look at the inner workings of **bump**, see the more lengthy report, "BUMP, a Motion Description and Animation Package," by S. A. Oakland.

Appendix D: Manual Pages

NAME

bump – animate a UNIGRAFIX file

SYNOPSIS

bump *infile initial increment number* [*-p prefix*] [*-r rendfile*] [*-d*]

DESCRIPTION

Bump reads an extended-UFO UNIGRAFIX file and produces an animated sequence. It either generates a series of standard-UFO files, or sends the frames one-by-one to a renderer for immediate recording.

COMMAND LINE ARGUMENTS

infile name of extended-UFO input file

initial value of time at start of sequence

increment
time increment per frame

number
number of frames to be produced

-p prefix
optional. Prefix which will be used in generating names of output files. If no prefix is given, the frames will be immediately rendered and then deleted.

-r renderfile
optional. Name of file containing shell command line through which frames should be sent. By default, **bump** will call ugplot to render the frame for the Ikonas, display it, and send a VAS-IV record signal to /dev/ttyh4. If **-p** is turned on, this argument is ignored.

-d optional. Turns debugging on, produces unflattened output files, and provides a running commentary on what **bump** is doing. The more **-d**'s, the more commentary.

EXTENSIONS TO UNIGRAFIX

Bump recognizes five extensions to UNIGRAFIX, namely **move, mdef, mapply, meval,** and **mexecute.** **Mdef** is used to define motions in the abstract. **Mapply** applies these motions to specific objects, while **move** both defines a motion and applys it, as a shorthand for motions which are used only once. **Meval** provides evaluation of expressions within **UFO** files. **Mexecute** allows calls to arbitrary generator and filter programs outside of the **UFO** file. The general forms of these extensions are as follows:

move

(**move** *named_object start_time duration*
(*motion_style transformation* (*time value*) (*time value*) ...))

mdef

(**mdef** *motion_name start_time duration*
(*motion_style transformation* (*time value*) (*time value*) ...))

mapply

(**mapply** *named_object*
(*motion_name start_time duration* [*repetitions*]))

meval

def *definition_name*;
(**meval** ...any UFO text, containing '**$**...**$**' where evaluation is desired...)
end;

mexecute

def *definition_name*;
(**mexecute** ...a shell command line, containing '**$**...**$**' where evaluation is desired...)
end;

EXAMPLES

The following is an extended-UFO description defining a cube which spins from 0 to 360 degrees in the time from 50 to 150:

```
i cube (protocube);
(mdef spin 0 1
        (linear -ry (0 0) (1 360)))
(mapply cube (spin 50 100))
```

Move provides a shorthand for doing the same thing:

```
i cube (protocube);
(move cube 50 100
        (linear -ry (50 0) (150 360)))
```

Creating five frames of animation, at T={0,25,50,75,100}, and putting the output in files named /usr/tmp/foo.{0,1,2,3,4} would require the following command line:

```
bump samplefile 0 25 5 -p /usr/tmp/foo
```

If the output file prefix is omitted, **bump** will put the output into a temp file and immediately render it. By default, it will render the scene for the Ikonas via UGplot, put it into the Ikonas buffer, and send a record command to /dev/ttyh4, where the VAS-IV is presumably listening.

This behavior can be modified with the **-r** *renderfile* option. If the name of a rendering file is given, **bump** will read this file for a shell command line, and then cat the UFO description of the current frame through this command line.

MOTION STYLES

Bump currently recognizes four different motion styles. These are:

step, for stepping from value to value with no intermediate positions.

linear, for linear interpolation between subsequent values.

cspline, for cardinal spline interpolation between values.

bspline, for b-spline approximation of values.

The pair of numbers immediately following the motion name in an **mdef** represents the 'observable' time interval of the motion. This is useful for specifing invisible control points on splines, as on this motion definition which describes an object rolling along an S-curve in the x-y plane.

```
(mdef roll 1 1
        (linear -rz (1 0) (2 360))
        (cspline -tx (0 0) (1 50) (2 100) (3 150))
        (cspline -ty (0 0) (1 -50) (2 50) (3 0)))
```

The first transformation applied to the object will be a rotation about z. It is then translated in the x-y plane by the next two lines, causing it to travel in an S-curve. Since the motion's internal time starts at 1 and the duration is 1, the control points associated with times 0 and 3 will not be interpolated. However, they will still be active in shaping the curve.

BUGS

Mexecute does not expand shell expressions like "~ug/lib" and "$HOME". These must be spelled out manually.

The **bump** process tends to grow slightly with each frame. This seems to be due to memory fragmentation, as it is careful to free all space that it allocates. It is therefore sometimes necessary to break up long sequences into several smaller runs to avoid exceeding the maximum process size.

Comments are not allowed within **bump** statements.
Error diagnostics can be cryptic. Syntax errors will sometimes cause core dumps.
Evaluator for **meval** and **mexecute** is simpleminded and does not know about trig functions.

FILES

~ug/bin/bump
~usr/tmp/bump.xxxxx
~usr/tmp/bumpout.xxxxx.x

SEE ALSO

babybump (UG), ugplot (UG)

AUTHOR

S. A. Oakland

Appendix E: Production Details

Bump has been used to create a three-minute animated videotape demonstrating the current status of the UNIGRAFIX project. The roughly six thousand frames comprising this movie were generated over a period of two months, using a Vax 750 (degas) and a Vax 8650 (vangogh) displaying frames on an Ikonas frame buffer in 480 by 512 resolution. The Ikonas RGB signal was converted to NTSC format by a Tektronix 1474 color sync generator and a Bosch TCE-2000 digital color encoder, and recorded by a Sony VO-5850 ¾ inch videocassette recorder operated by a Lyon/Lamb VAS-IV controller connected to the Vax 750.

This raw footage was assembled into a completed videotape by Mike Francisco of the Berkeley Educational Television Office. During post-production, we discovered a number of problems with the Graphics Lab animation hardware. Specifically, the videocassette recorder's head switching is visible, resulting in a band of static at the bottom of the picture. Also, the color sync generator is very poorly adjusted, resulting in a signal that cannot be played through a time base corrector. This makes it impossible to use sophisticated editting equipment, and very difficult to edit the tape at all or even copy it! Finally, the color encoder is also mistuned, and is generating an "illegal" color signal.

The final demonstration videotape is only barely acceptable, and it is essential that the three devices mentioned above be fixed before any other serious animation is begun.

The background music for the video is a piece called "Xango", from the album "Forthcoming" by Sky. Since I used it without permission, we will run into serious copyright problems if the videotape is distributed outside the university. The master videotape has the voiceover and music on separate tracks, so it will be easy to substitute other music or eliminate it completely if necessary.