Interactive Kinematic Objects in Virtual Environments

Randall G. Keller

Master's Project
under the direction of Carlo Séquin

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

December 18, 1997

Abstract

A new extension package entitled WALK-KIN has been integrated into the existing WALKTHRU/WALKEDIT system. It provides three separate elements that combine to create a more interesting and engaging virtual environment.

The first element is the introduction of the GLIDE compiled-code paradigm for behavior scripting. In UniGrafix, the parsed scripting mechanism has a limited expression semantics allowing only primitive behavior to be defined. The new system is both faster and more expressive since it is compiled and uses C as a full scripting language. The new extension allows authors to create content objects capable of performing limited simulations in real-time within the virtual environment.

The second element is the integration of this scripting mechanism into the WALKTHRU/ WALKEDIT environment. Using the new WALK-KIN extensions, kinematic objects, which are more complex than previously possible and operate faster, can be placed within the virtual WALKTHRU environment and manipulated using the existing WALKEDIT object association primitives.

The third element is a pair of interaction event handlers. The first is an interactive script execution mechanism that enables time-varying objects to respond to user interaction. An execution context has also been provided to the updating function enabling modification of object properties. A second event handler makes it possible to link the various content objects of the virtual environment to related information in other files or databases. The system functions by associating a URL with each prototype or instance of a virtual element. When a user selects such an object, the page at the designated URL is displayed by a Web browser.

Contents

1 Introduction

The Berkeley Architectural WALKTHRU system [11,12,27,28] was developed to enable a user to traverse a complex virtual building environment in real-time. The WALKTHRU system is both interactive and time-varying, but both of these aspects are quite limited. Interaction with the system is restricted to the user's navigation of the 3D environment; the user may interactively determine the path through the virtual world, but objects are not reactive. The time-varying aspect of the virtual environment is similarly limited to a small number of moving objects with simple behaviors.

The interactive aspects and the handling of time-varying objects were developed as separate extensions to the WALKTHRU. The WALKEDIT extension [4,5] addresses the interaction issue by providing a paradigm for interacting with and manipulating interior objects within the 3D virtual environment in real-time, and the WALKTHRU-CFAST extension [3], was created to improve the simulation capabilities of the visualization environment. The latter is a merger between the CFAST [21] fire simulation system and the WALKTHRU visualization application. Both of the original programs operate on highly occluded architectural environments, as does the merged system. The architecture of the merged system was constructed in such a way that other simulation systems can be connected to the WALKTHRU visualization system in a generic fashion.

The primary focus of the work presented in this report is to develop a system for defining and interacting with kinematic objects in virtual environments, in particular within the WALKTHRU system. This work builds upon the basic WALKTHRU/WALKEDIT framework. In this context, kinematic refers to an object that moves with respect to time but does so in a non-physical manner. A clock whose arms move through a rigid-body transformation (in this case a time-based rotation) is an example of a kinematic object. In contrast, a clock, which consists of a complicated physical simulation involving a pendulum and a gear mechanism, is a dynamics simulation and beyond the scope of this work. Finally as an example of a possible interaction mechanism, the clock object's arms could respond to the user, allowing them to be set with mouse clicks and drags. Clearly the presence of such interactive kinematic objects can add a great deal to an otherwise static virtual environment.

As part of the work accomplished in this report, the WALKTHRU and WALKEDIT systems have been extended to support interactive kinematic objects by providing an interaction mechanism and an authoring environment. This new extension is called WALK-KIN for "kinematic WALKTHRU" and is built into the WALKEDIT framework as a package for the WALKTHRU renderer and interaction system. Examples of interactive kinematic objects built using the extension's mechanisms include a kinematic sculpture that can be activated by selecting its pedestal base with the mouse, and an office chair that can be interacted with in a physically plausible manner. These simple example objects represent a proof of concept; the same mechanisms could be used to construct something as sophisticated as an autonomous agent implemented as an interactive kinematic object.

1.1 Motivation

As motivation for adding kinematics and interactivity, we consider virtual reality systems in general. We begin by defining a framework for distinguishing between a variety of virtual reality systems. Then we examine the elements of the WALKTHRU and related systems and discuss their features within this domain. Finally, we consider the location of the improved WALK-KIN system in this classification framework.

1.1.1 What Makes a Good Virtual Reality System?

The answer to this question is still the topic of debate and research, but some clarifying work has been done. Zeltzer [30] proposes a taxonomy for virtual reality systems that provides a basis for understanding existing systems as well as the new WALK-KIN extension. Zeltzer decomposes the essential elements of a virtual reality (VR) system along three distinct axes: *Autonomy*, *Interaction*, and *Presence*. These three dimensions range from 0 to 1 and thus form a unit cube, which can be used to categorize the space of all possible VR systems. The concept of *Autonomy* represents the complexity of response that an object exhibits when exposed to external events. *Interaction* categorizes both the number and the speed at which model parameters can be accessed and altered by the user and other virtual agents. *Presence* is a rough measure of both the quality and number of senses effectively stimulated by the virtual environment interface. Alternate definitions of *Presence* have been proposed. Slater [25,26] argues that the notion of *Presence* can be further subdivided into two distinct parameters, a quantitative measurement of the immersion of the virtual environment (something akin to Zeltzer's relationship between stimuli and senses) and a subjective measure of the psychological feeling imparted on the user by the environment itself. This psychological aspect, the second parameter, becomes very important to our discussion of *Presence*. In considering our virtual environments; we propose that interaction with a kinematic object can increase the user's sense of *Presence* in the virtual environment if the interaction occurs in a natural and plausible fashion. For example, opening a door by selecting and dragging its handle with the mouse is an intuitive and natural interaction mechanism, while opening it using a menu driven mechanism is awkward and lessens the illusion of virtual reality and the user's notion of *Presence*.

The original WALKTHRU system contains few moving objects and no means of interacting with these kinematic objects. Thus the WALKTHRU system ranks quite low on all three of the virtual reality measures. Subsequent work involving interaction was developed for the placement of interior objects within the virtual environment and is contained in the WALKEDIT extension to the WALKTHRU. The WALKEDIT extension increases the *Interaction* of the system by allowing the positioning parameters of interior objects to be manipulated in real time, but it has no effect on the other two aspects of the system. In contrast the WALKTHRU-CFAST simulation and visualization system augments the *Autonomy* value of the system through the simulations, but not the interactive elements of the virtual environment. Therefore, the primary aim of this work is to increase the WALKTHRU system's value along both of these axes. Towards this end, the new WALK-KIN extension enables the creation and use of interactive kinematic objects within the WALKTHRU's virtual environment. Finally, we also claim that the use of interactive kinematic objects in a plausible and realistic fashion can augment the user's sense of *Presence* in the virtual environment.

1.2 Project Goals

The primary aim of this project is to incorporate interactive kinematic objects into the WALKTHRU/WALKEDIT virtual environment. Interactive kinematic objects capable of expressing interesting behaviors and reacting to the user in real-time will augment the virtual experience making it both more engaging and realistic. In order to achieve this aim we have separated the work into three separate goals:

1.) **Improve the expressibility and performance of the scripting mechanism for time-varying objects**: We achieve this by changing our object behavior scripting system from a parsed, limited expression syntax to a complete and compiled programming language.

2.) **Develop and implement a framework for defining interactive kinematic objects:** Here our goal is to provide an author with the ability to create virtual world analogs of functioning objects. Simple objects, such as windows and doors that open and close, or an elevator with call buttons, add to the model's realism. In addition, such objects can be linked with an existing simulation in order to enhance its interactivity and functionality.

3.) **Given an interaction mechanism for objects in the 3D world, create useful event handlers to activate kinematic interactive objects in a variety of modes:** We incorporate the mechanisms for moving objects from the WALKEDIT [4, 5] package and provide an enhanced system for accessing particular object properties.

1.3 Overview of Work

The work presented in this report concerns an extension to the existing WALKTHRU framework called WALK-KIN. It provides both an authoring environment and an interaction mechanism enabling interactive kinematic objects in the WALKTHRU's virtual environment. The behavior scripting and interaction techniques provided by WALK-KIN were chosen to enhance the notions of *Autonomy* and *Interaction* discussed above. The WALK-KIN extension is the result of three related pieces of work; each piece of work solves one of the aforementioned goals:

1.) **Adding Kinematic Objects:** In order to understand the basic requirements of kinematic objects, a number of different languages and standards are examined. In particular, the UniGrafix language and its time-varying script extension BUMP [19] are scrutinized, since they are the underlying structure of the ASCII representation of a WALKTHRU model. In addition, the interactive BUMP viewer, UGMovie [6], is analyzed as a mechanism for scripting and also used as a testbed for performance analysis. These performance measurements demonstrate a need to move toward a more powerful scripting mechanism. Therefore, the GLIDE [13] scripting paradigm of generating and executing C code is incorporated into the UniGrafix [7] language. Furthermore, a number of comparison measurements are made between the new GLIDE-based system and the original BUMP system.

2.) **Modifying the WALKTHRU:** The WALKTHRU parser and the database environment are modified to mirror the changes performed in the above work done with BUMP and UGMovie. The GLIDE paradigm of compiling C code and evaluating behavior functions is incorporated into the WALKTHRU environment. The final system allows an object to be concisely described in a UniGrafix file and a related C behavior function file and then added to a WALKTHRU database. The generated C code is then linked into the WALKTHRU viewing application, and the time-varying objects are bound to their updating functions.

3.) **Creating Interaction Event Handlers:** The notion of interaction in the WALKTHRU and in WALKEDIT is explored through the development of two interaction event handlers. One event handler links interior objects in the virtual environment to URL's for World Wide Web documents. This handler enables interactive selection of objects to display relevant information in a standard Web browser. The other handler provides a generic mechanism for associating an object with its interactive behavior functions, allowing a user's mouse clicks to interact with a kinematic object.

1.4 Report Organization

Section 2 reviews related work and discusses the previous WALKTHRU systems as well as GLIDE, UniGrafix, and other 3D modeling languages. Sections 3-5 of this report are each devoted to one of the three goals discussed above. Section 3 discusses the shift from a parsed primitive scripting language used by the previous systems (BUMP, UGMovie and the WALKTHRU) to using C as a scripting language. It details the modification of the UniGrafix parser to incorporate the features developed in the GLIDE language. Section 4 explains the limitations of the WALKTHRU system as well as the modifications made to the database and parsing system necessary for the WALK-KIN extension. Section 5 examines the selection mechanism and event handling elements of the WALKTHRU/WALKEDIT environment. This section also examines the implementation of the two new event handling mechanisms developed for the WALK-KIN extension, as well as providing a number of example interactive kinematic objects created using this new extension. Section 6 discusses extensions to this work and other related issues and Section 7 concludes this work.

2 Related Work

This section is divided into two main parts: an examination of 3D scene description languages and previous work related to the WALKTHRU system. The first sub-section discusses kinematic scene description languages used at Berkeley in the past (UniGrafix, BUMP, and GLIDE) and general purpose 3D modeling standards (VRML 1.0/2.0 and Open Inventor). The second sub-section covers related virtual reality research systems developed at Berkeley: UGMovie, the WALKTHRU, WALKEDIT, and WALKTHRU-CFAST.

2.1 Kinematic Scene Description Languages

In this section a number of different scene description languages are examined. The focus here is on a language's facilities for specifying behaviors and interaction. In particular, we consider the expressiveness of a language's scripting system, and the generality and usefulness of its interaction mechanism. In addition, we consider specific implementation issues for the various languages and standards discussed.

2.1.1 UniGrafix and the BUMP Extensions

The UniGrafix [7] language is a scene description language capable of representing polyhedra as well as non-manifold objects and piecewise linear wire segments. The original language is static and does not support any form of time-varying values. In order to augment the language, an extension package named BUMP (Berkeley UniGrafix Movie Package) [19] was developed. BUMP extends the static UniGrafix language by providing a facility for substituting expressions in the place of floating point values within particular statements. The scripting language for these expressions is a subset of C's math expression syntax. The scripting language allows for simple math operations and provides a few math functions as well as a conditional construct, e.g. the in-line `if` statement. The expressions themselves must appear within special BUMP extension statements and must be delineated by dollar sign symbols.

The BUMP system is implemented as a modification to the static UniGrafix parser. The modified parser contains new syntactic constructs, which are separated from the original elements of UniGrafix by encapsulating them within parenthesis. Inside the parenthesis of these extension statements time-varying elements of the scene can be defined and used. Some examples of the extension statements provided by the BUMP package are: `(mdefine…)`, `(mexecute…)`, and `(meval…)`. The `(mdefine…)` construct enables motions to be defined while the `(mexecute…)` expression allows external scripts to be invoked using the C function `system`. For our purposes the `(meval…)` extension is of primary interest. Using `(meval…)` constructs an author can use time-varying dollar sign expressions inside scene graph geometry and create kinematically animated objects. See Figure 2 for an example of a moving triangle defined using BUMP-extended UniGrafix. In the example the dollar sign expressions appearing in the `v` statements represent time-varying values defining the vertices: `v1`, `v2`, and `v3` that constitute the face `f1`. By using the time-varying values for the coordinate locations, the vertices move with time and likewise the corresponding face moves and changes shape.

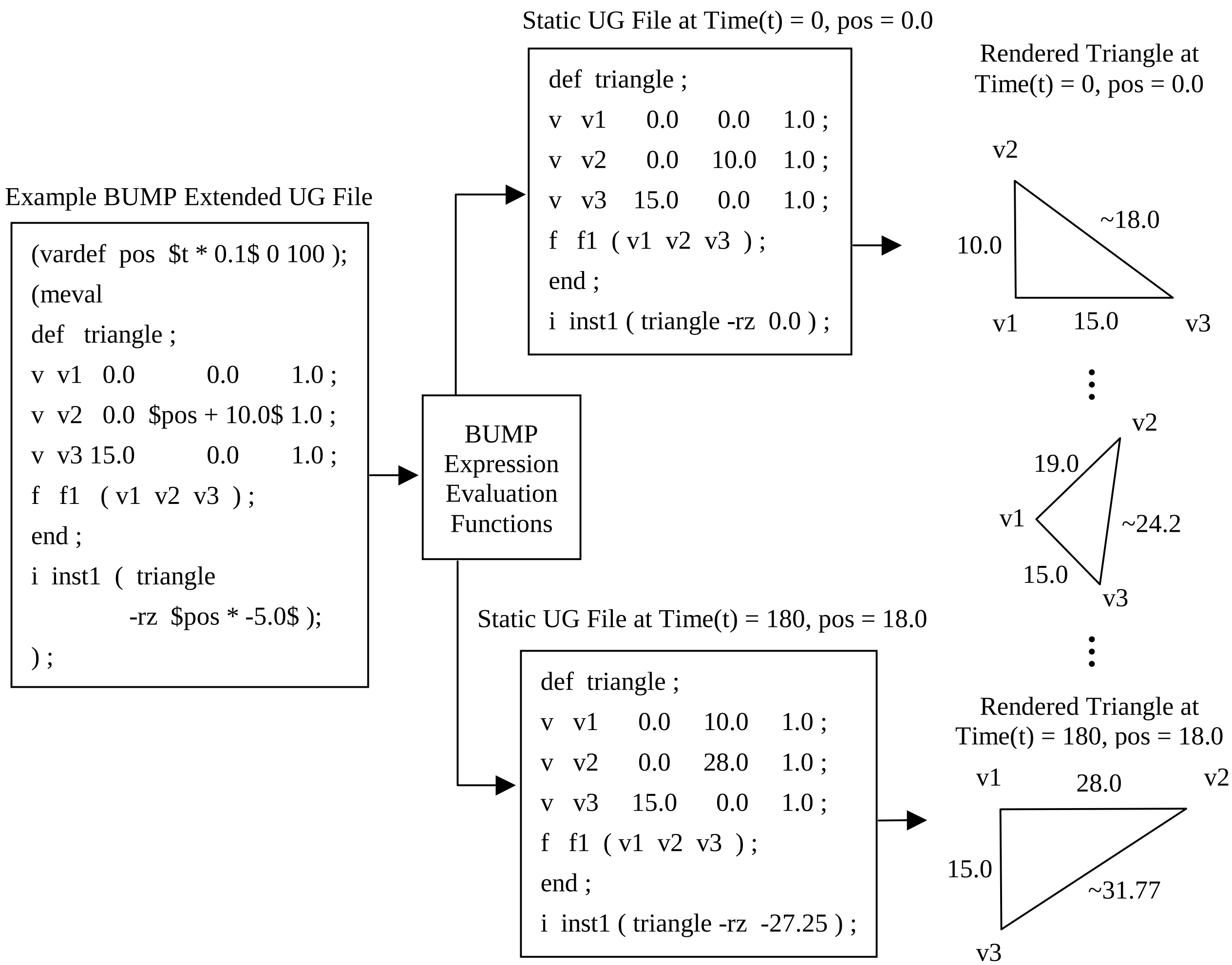

Figure 2.1: Example of the BUMP systems animation of a time-varying UniGrafix file

The BUMP system functions as a preprocessor and takes as input a BUMP extended UniGrafix file (i.e., a file that contains dollar sign expressions encapsulated within BUMP extension statements). BUMP then removes the time-varying statements and evaluates them using its library of parsing and evaluation functions. The BUMP package also has an understanding of variables that are created through the `(vardef…)` statement. Certain variables such as `Time` are always defined and can be used to make time-varying objects. For each frame, the dollar sign expressions within BUMP extension statements are evaluated and replaced by this computed value producing a static set of frame specific values. Finally, the BUMP statement keywords and parenthesis are removed from the file leaving only static UniGrafix statements. Thus, the system effectively creates a series of snapshots, or static UniGrafix files, describing the scene for evolving values of frame number and these snapshots can then be gathered together to form an animation. Figure 2.1 demonstrates this process on an example scene containing a moving triangle. The end result is a series of static UniGrafix files describing the scene for particular values of the variable `Time`, which are ready to be parsed and rendered to produce an animation. Since the output of BUMP is a series of static UniGrafix files, the rendering and animation of a scene are completely separated and thus a simple static UniGrafix Scene viewer can be used to display each frame of the animated scene.

The BUMP system of defining objects has three distinct disadvantages. First and foremost, the system can be very slow to execute since it parses all of the time-varying expressions and outputs a new static scene description every frame. Second, the scripting language for the time-varying elements is only a subset of C's mathematical expressions and thus it is not sufficiently expressive to permit the creation of complex motion or behavior. Finally, this system of defining expressions provides no facility for interaction: objects can not be constrained to move according to a user's mouse motions. Due to these limitations, the work done in building the WALK-KIN extensions has replaced this parsed BUMP paradigm with a more efficient and powerful system involving code generation and compilation.

There is, however, one important advantage to the BUMP system's reliance on parsing the time-varying expressions every frame. Due to the re-parsing, these time-varying expressions can be altered interactively by the user and new expressions with new types of motion can be tested directly without degradation in performance. The ability to redefine behavior while the system is visualizing is significant since it augments the BUMP systems *Interaction* value.

2.1.2 The GLIDE Scene Description Language

The GLIDE (Graphics Language for Interactive Dynamic Environments)[1] [13] scene description language was an early attempt to overcome the limitations of the BUMP system and to develop a better language for the creation of interactive kinematic objects and scenes. GLIDE is an extension of a static scene description language, SDL [24], and was originally developed for an undergraduate computer graphics class, primarily as a means for creating time-varying and interactive scenes. The language utilizes C as the expression script for its kinematic elements. Instead of parsing the expressions to evaluate them, GLIDE transforms an expression's C code into compiled functions. These compiled functions are then executed to update the time-varying elements of the scene graph. The primary reason for using compiled code was performance: a system that parses its expressions will take considerably longer to execute than one utilizing compiled code. Like the BUMP system, GLIDE provides a series of variables for controlling the time-varying nature of expressions. In particular, the frame number and the time since the beginning of animation are represented as variables (`FRAME` and `TIME`). In addition, some interactivity is provided through the variables `MOUSE_X` and `MOUSE_Y`; these represent the x and y location of the mouse relative to the graphics window. Using these variables, scenes can be created in which an object's motion is coupled to the user's mouse motion.

---

[1] The GLIDE language semantics and syntax were originally created and implemented by Maryann Simmons and Ajay Sreekanth.

The current implementation of GLIDE relies heavily on a two-pass parser. The time-varying expressions for objects in the scene, which are C code fragments, are extracted from the scene description file during a first pass of the parser and placed in a C code file. The first pass of the parser outputs this C code file but does not actually build a scene graph. The output file contains a single `switch` statement where each `case` element corresponds to a time-varying expression in the original scene description file. In each of the `case` elements of the `switch` statement a piece of C code is stored. A particular C code fragment is the updating function for a specific time-varying statement. The `switch` statement is controlled by an `ID` variable that enables a time-varying statement in the scene graph to find its updating function in the `switch` statement by using its `ID`. Once the C code is generated, it must be compiled and linked into the actual application, which will be animating the scene and its objects. The second pass of the parser is the mechanism that actually builds the scene graph and is normally incorporated into a renderer or other scene viewing application. In addition to a standard scene graph, the second pass parser is responsible for associating a time-varying element with its correct updating function. The current implementation achieves this by storing a generic function pointer corresponding to the `switch` statement function with every time-varying element of the scene graph. During the updating of the time-varying elements, a statement's `ID` is passed as an argument to the statement's updating function. Since the case parameter (the `ID` for a time-varying statement) depends on the parse order, statements must not be re-ordered between the first and second phases of parsing.

The primary advantages gained by using the GLIDE system are the increased evaluation speed and the more expressive scripting language. Since the expressions are preprocessed by the first pass of the parser and converted into compiled code, we can save a considerable amount of CPU time relative to a parsed system. As will be demonstrated later, the costs of parsing (relative to simply executing expressions) can be quite significant, and these savings can dramatically increase performance. Furthermore, the increase in expressiveness gained by using C as the scripting language is quite significant. Using a system such as GLIDE, it is possible to embed arbitrarily complex expressions within the kinematic scene, enabling very complex behaviors or simulations.

There are also disadvantages to the GLIDE paradigm. First, we must parse the file twice and create a C code file that must then be compiled into the application. This adds a considerable amount of time to the initial viewing of a scene containing time-varying objects. In contrast to BUMP, we are no longer able to interactively change the time-varying expressions for a given value while a GLIDE renderer is executing. Due to the manner in which GLIDE implements the parsing and conversion into C code, a viewer using GLIDE is unable to animate more than one scene without re-linking new updating code. Finally, the very presence of C as the scripting language presents two important pitfalls. First, the parser itself no longer determines whether the syntax is correct but defers this checking to the compiler. Second, since the scripting language now allows looping constructs and memory operations, a given scene's kinematics may cause execution errors, for example a segmentation fault or non-termination can occur.

2.1.3 Open Inventor and VRML

Unlike the previous systems described in this section, the 3D environment specifications VRML [2,17] and Open Inventor [29] are not Berkeley specific. As of the writing of this report, VRML 2.0 has in fact become a standard for virtual reality modeling on the World Wide Web, and a number of browsers have recently been developed for viewing VRML worlds. Open Inventor is a 3D scene description language developed by Silicon Graphics. In addition to geometry statements, Open Inventor contains elements for defining behaviors and interaction. The fundamentals of Open Inventor were carried over more or less directly into VRML 1.0, which is almost completely a subset of Open Inventor. VRML 2.0 is the successor to VRML 1.0 and differs in many respects, most notably it contains time-varying elements. Furthermore, VRML 2.0 deviates considerably from Open Inventor's mechanisms for time-varying objects, behaviors and interaction.

Open Inventor

In contrast to the relatively simple languages discussed above, Open Inventor provides for both interaction and behavior explicitly within the language syntax and semantics. Interaction in Open Inventor is achieved through the presence and use of `sensor`, `dragger`, and `manipulator` nodes in the scene graph. `Sensor` nodes sense the location of the user and act as interaction mechanisms by allowing a user's proximity to a given location to affect objects in the scene. Similarly, `dragger` and `manipulator` nodes take direct input from the user in the form of mouse events and transform them into values for use in the virtual environment. In all three cases `Connections` (links between the fields of objects) are used to propagate the data through the scene graph.

Open Inventor also provides a mechanism for behavior or function execution. `Engines` are responsible for behaviors and consist of simple functions embedded within a scene graph object. `Engines` may also be connected together via the `connection` mechanism and by combining them, we can simulate some interesting behaviors. In addition to the simple predefined `engines`, Open Inventor allows arbitrary math expressions to be defined using a simple math script much like the BUMP syntax. Given this notion of interaction and the ability to generate time-varying information using the `engine` framework, Open Inventor scenes can describe complex interactive settings which can in general be much more interesting to the user then any BUMP extended UniGrafix scene. It should be noted that the system of `engines` is not sufficiently complicated to allow for iterative constructs or recursion and therefore these `engine` nodes do not constitute a full programming language.

VRML

As a first attempt at defining a standard for virtual reality on the World Wide Web, the virtual reality modeling community adopted a subset of Open Inventor and termed it VRML 1.0 (Virtual Reality Modeling Language). For this first version, all of Open Inventor's behavior and interaction related elements were removed. In addition, many of the basic node groups: polygon primitives, transformation types, and material properties were reduced to a bare minimum. The resulting paired down version is an essentially static scene description language that contains two extra Web related constructs. These extra constructs are the `WWWanchor`, a mechanism for linking a piece of geometry with a URL, and the `WWWinline`, a mechanism for allowing scene graph elements to reside at other locations on the World Wide Web.

Recently, the standard set forth by VRML 1.0 has been extended and augmented with interaction mechanisms and behavior capabilities and has become VRML 2.0. VRML 2.0 is essentially a time-varying interactive scene description language containing statement types for behaviors and interaction. The revised language provides for time-varying interactive objects through the addition of `script` nodes (used to create behaviors), `interpolator` nodes (used to create simple kinematic animations) and `sensor` nodes (used to create interactive objects and to perform tasks based upon the locations of elements or the value of time).

After studying the two time-varying standards, Open Inventor and VRML 2.0, it becomes clear that they differ in their abilities to provide behaviors and interaction. On the one hand, Open Inventor has a more general interaction system than VRML 2.0's limited number of predefined `sensor/manipulator` nodes. On the other hand, the `script` node mechanism of VRML 2.0 is more powerful then the `engine` system used by Open Inventor. In VRML 2.0 `script` nodes contain arbitrary scripts written in a subset of the scripting language Java™[15]. Since the subset of Java™ used is a full programming language, the complexity of possible behaviors is much richer then the simple math syntax and function execution provided by Open Inventor `engine` nodes. An obvious question is why not use one of these particular systems as the underlying mechanism for this new work? The short answer is that it was too much effort to shift the whole WALKTHRU project to use either of these standards, since the WALKTHRU system is so heavily dependant on the underlying UniGrafix scene description language, for a further discussion see Section 6.

2.2 Berkeley Visualization Systems

The following related systems were all developed at UC Berkeley. The first, UGMovie, is an interactive scene viewer for the extended UniGrafix language. The other three systems are the WALKTHRU and two extensions to the base system. The two extensions are WALKEDIT, a 3D environment editing extension, and WALKTHRU-CFAST, a combination of the WALKTHRU and a fire simulator. These various systems are explored because of their direct relevance to the new WALK-KIN extension developed as part of this report work.

2.2.1 A Kinematic Scene Viewer: UGMovie

The BUMP extension to UniGrafix augments the language and provides a means of producing animations offline. UGMovie is an application for interactively viewing and potentially altering these kinematic UniGrafix files. UGMovie[6] employs the BUMP library code to evaluate the time-varying scenes and uses a simple GL renderer to display the results each frame. In addition, UGMovie provides the capabilities for altering the expression strings interactively, allowing the user to modify a kinematic scene while the application is running. This ability to modify time-varying expressions interactively is useful during the initial creation and prototyping process of building an animated scene. In addition, the ability to alter these behaviors while visualizing the scene represents access to a whole new series of model parameters. Finally, in contrast to a GLIDE-based system, UGMovie's reliance on BUMP allows any properly defined kinematic UniGrafix file to be loaded and viewed without additional work (i.e., there is no compilation step necessary).

Since UGMovie is built using the BUMP library it inherits the same drawbacks of that language, namely slow parsed time-varying expressions with a limited behavior capability. In addition, UGMovie understands only a subset of the extended BUMP scene syntax. In particular, the new escape statements provided by BUMP must be encapsulated within a definition statement in UGMovie. This is more restrictive than the original BUMP parser which allows the new escape statements to be placed at any location within the time-varying UniGrafix file.

2.2.2 The WALKTHRU System

The WALKTHRU is a system [11, 12, 27, 28] that allows a user to navigate inside a three dimensional model of an architectural environment. The system provides the user with real-time navigation of very large 3D models. One such 3D environment is the UniGrafix model of Soda Hall, which is a relatively complete, furnished model of the Computer Science building at UC Berkeley consisting of approximately 1.5 million polygons. The WALKTHRU represented a significant research advancement since it enabled a user to navigate through a very large geometric model in real-time. Since the system considers only potentially visible objects for rendering (a superset of those actually visible), as long as the number of such objects is reasonable for the machine, real-time performance can be guaranteed regardless of the actual total model size. This system utilizes UniGrafix as its initial scene description format and builds a database of scene elements from this file description. Given UniGrafix as the underlying modeling language, the BUMP extended syntax is used for the few time-varying objects appearing in the Soda Hall model. Like UGMovie, the WALKTHRU's time-varying objects inherit the same drawbacks as BUMP.

Rather than support the whole BUMP syntax, the WALKTHRU followed UGMovie and only implemented a very small subset of the extended syntax. In fact only `(meval…)` constructs are supported, and within these only time-varying instance statements are permitted. Since the WALKTHRU's parser only allows kinematic instance statements, some behaviors that are supported by UGMovie, such as time-varying colors or vertices, can no longer be constructed. This limitation represents a serious reduction in the behaviors available to the WALKTHRU's time-varying objects. In addition, the WALKTHRU environment does not allow behaviors to be modified while the model is being visualized. Even more limiting, the alteration of any of the underlying UniGrafix descriptions for WALKTHRU objects requires re-parsing the original and rebuilding the database, which is a computationally expensive process.

2.2.3 The WALKEDIT System

The WALKEDIT system [4,5] represents an important addition to the original WALKTHRU architecture. This program allows the user to interactively manipulate 3D objects in the WALKTHRU's virtual environment. It was developed primarily as an interactive modeling tool that allows a user to position furniture within the virtual environment. Its main contribution is to allow simple 2D mouse motions for placing furniture objects in the 3D virtual world. This disambiguation of the 2D mouse motions into 3D virtual world motions is achieved through *object associations* [4], whereby objects in the 3D world are constrained by associations to move on various planar surfaces within the model. For instance, by default, a chair in an office stays constrained to remain on the plane of the office floor. Similarly, a picture frame is constrained to move in the plane of an office wall.

In addition to these static constraints, a given object's constraint plane can change according to a user's mouse motions. In general, objects associate themselves with the appropriate constraint plane that is currently visible to the user at the selected location (i.e., the one which is closest to the user along a ray cast from the selection point into the model). Since the user, the object, or both may move, the 2D constraint plane can change dynamically according to changes in visibility. For example, a book will move from the floor to the top of a desk (i.e., change its constraint plane from the floor to the desk) if the ray cast from the user into the model intersects the desk plane before it intersects the floor plane. Similarly, if a book is dragged off of the edge of a desk, it will drop to a horizontal surface directly below it, and this new surface will become its constraint plane. The ability of the system to change constraint planes automatically is a big benefit to the ease and speed of furnishing the virtual environment with content objects.

2.2.4 The WALKTHRU-CFAST Simulation Architecture

The major contribution of the WALKTHRU-CFAST [3] framework is the development of a general architecture for connecting a visualization mechanism together with a simulation system. This work combines the Consolidated Model of Fire and Smoke Transport (CFAST) [21] developed by the National Institute of Standards and Technology's (NIST) as a simulator and the Berkeley WALKTHRU system as its visualization mechanism. The current system allows a user to interactively observe the output of the fire simulation system CFAST from within the WALKTHRU's virtual environment. A user may navigate the halls of a building as the simulator updates the environment, enabling the user to see the smoke and fire grow as a function of time. The combination of these two systems allows CFAST's fire simulation results to be visualized within an actual 3D architectural context generated by the WALKTHRU: making it possible for even a novice user to interpret the complicated results of the simulation. In addition, the system provides the user with other methods of visualizing the data. Currently the system includes two additional modes: a thermal vision mode, where heat levels are displayed instead of the default smoke and fire scenes, and a toxicity gauge displaying the levels of toxins in the air at the user's present location.

The WALKTHRU-CFAST system suffers from one primary drawback: only the time parameter of the simulation can be altered while the system is executing. Using interactive kinematic objects we can provide access to another important model parameter: connectivity. The CFAST simulation is governed by the connectivity of the various spaces of the virtual environment, this controls the flow of heat, smoke and fire between the different cells of the simulation's model. By employing the mechanisms of the WALK-KIN extension we have created a new interactive objects which directly controls the connectivity of the environment. This new object, the interactive door, allows the user to interactively open and close portals connecting cells in the environment. The future hope is that the developers of CFAST will provide an API to control their simulation interactively and then it will simply be a mater of calling these simulation control functions from within the behavior elements of the door. Thus, with this enhancement the CFAST simulation could be altered interactively by manipulating the door object from within the virtual environment. Such a link between interactive kinematic objects and the simulation is a good example of why these objects are useful; they allow the user to affect the outcome of complex simulations from within the virtual environment.

3 Kinematic Objects

In order to achieve the goal of fast and complex kinematics, we begin by modifying the existing framework of the WALKTHRU to incorporate a new, more expressive language for its scripting mechanism. Since the WALKTHRU system employs UniGrafix(UG) as its scene description language and uses the BUMP extensions to implement its time-varying objects, we first analyzed how these could be modified to achieve our goals. Given our desire for expressability, speed, and interactivity, it became apparent that a system such as GLIDE, which uses compiled as opposed to parsed code, would be more useful than the BUMP system. Therefore, we decided to also study the addition of GLIDE-like capabilities to the UniGrafix scene description language. In addition, we examine the related costs and benefits of utilizing a compiled code updating system versus a parsed system. The results of these performance measurements prompted the incorporation of the new GLIDE-like scripting mechanism into the WALKTHRU environment; this incorporation will be discussed in Section 4.

3.1 Adding Complex Kinematics to UniGrafix

As a first attempt at merging the GLIDE paradigm with the UG environment, we modified the UGMovie program, which is a display engine for time-varying UniGrafix scenes. This application was chosen to demonstrate the feasibility and the performance increases we could obtain by moving from the old interpreted scheme to a new compiled code paradigm. Two separate alterations to the UGMovie application were necessary for it to make use of compiled C code rather than parsed expressions. First, the original UniGrafix parser had to be modified to operate in a two pass fashion, the first pass producing the C code for time-varying statements and the second pass actually building an augmented scene graph in which kinematic objects have a link to their updating functions. Second, the UGMovie application's previous time-varying expression updating code had to be modified to use these new C code function calls instead of the BUMP library parsing and evaluation routines.

3.1.1 Modifying the UniGrafix Parser

In the original BUMP scheme, the time-varying expressions were parsed and evaluated every frame. In contrast, we convert these expressions into compiled code and simply reference them in order to avoid re-parsing. In order to adopt this compiled code paradigm we modified the UniGrafix parser in four separate ways:

1.) A new C code outputting mode of the parser was created

2.) An augmented scene graph was incorporated into the separate second pass parser

3.) A time-varying statement expression list was constructed

4.) All of the BUMP extension package language elements were removed

The GLIDE scene description language relies on a two-pass parser and this heavily influenced the design decisions involved in the new UniGrafix two-pass parser. In accordance with GLIDE's system, the new UG parser exists in two different forms. One of the parser's modes only emits C code, while the other builds the scene graph and the time-varying expression list.

Creating a C Code Output Parser

The mechanism employed for generating the output C code file based on an input UG file is similar to the system used by GLIDE. The time-varying expressions in the UG file are written out to another file called `ug.dynam.c`. Since these expressions are found in the original UG file and then written to the C file without modification, there is an implicit requirement that the syntax of the time-varying expressions in the UG file be valid C expression syntax.

Input: example scene of a moving triangle in the new extended UniGrafix file format

```
vardef  pos  $ float $ $ pos = Time * 0.1 $ ;
def   triangle ;
v   v1  $ pos  +  3.0 $   10.0      1.0 ;
v   v2  $ pos  * -2.0 $     0.0      1.0 ;
v   v3     15.0     $ -2.0 * pos $  1.0 ;
f   f1   ( v1  v2  v3  ) ;
end ;
i inst1  (  triangle -rz  $ pos * -5.0 $ ) ;
```

Output: generated by first pass parser based on input, this is the output C code file, ug.dynam.c

```
/* User Defined Global Variables */
float pos;

/* Update function for Global Variables */
void update_global_variables (void) {
Time = Time + 1;
pos = ( float )( Time * 0.1 );
}

/* Main Time-Varying Expression Updating Function */
void UG_DYNAM_FUNC0(int which, float *floatval){
        switch(which) {
        case 1:
                floatval[0] = (float)(pos + 3.0 );
                floatval[1] = (float)(10.0);
                floatval[2] = (float)(1.0);
                break;
        case 2:
                floatval[0] = (float)(pos * -2.0 );
                …
        case 3:
                floatval[0] = (float)(15.0);
                floatval[1] = (float)(-2.0 * pos );
                …
        case4:
                floatval[0] = (float)(pos * -5.0);
                floatval[1] = 0.0;
                floatval[2] = 0.0;
        }
}
```

Figure 3.1: Example of C code produced from an extended UniGrafix file by the first pass of the new parser, arrows show UniGrafix statement--C code correspondence. See Appendix A for the formal definition of the new extended UniGrafix language format.

As a concrete example, see Figure 3.1, which demonstrates the operation of the first pass of the parser on a kinematic scene containing a moving triangle. In practice, the parser goes through the same steps when building the C code file as it would in building the scene graph, however, static statements are simply ignored with respect to outputting code (the face statement in the example of Figure 3.1 is static and does not effect the generated C code).

The structure of the output C code file follows the same form used by GLIDE. A single function called UG_DYNAM_FUNC0 comprises most of the file and contains a single `switch` statement. Each time-varying statement appearing in the original UG file corresponds to one of the cases present in this `switch` statement (the arrows in Figure 3.1 demonstrate the correspondence between UniGrafix statements and generated C code). A time-varying statement's updating code, the C code encapsulated by its dollar sign expression, is copied into one of the cases of this `switch` function. All of the cases of this `switch` statement are referenced by a number that corresponds to the parse order of the time-varying statements appearing in the original file (this number corresponds to the `which` variable in the example of Figure 3.1). The careful reader will also note that all values are returned in the `floatval` variable. For the case of vector values this is reasonable, but for instances which return only one value the same mechanism is used in order to decrease the number of calling arguments to this generic function. Thus, for single valued elements such as instances the first floating point element of `floatval` is used to return the calculated time-varying value. During the second pass of the parser this number will be associated with the object in the scene graph, giving access to the object's particular updating code. Parsing the original UG file to create C code and then re-ordering time-varying elements of the file will result in undefined behavior since the time-varying statements are linked according to the parse order, and the original order is not stored with the output UniGrafix file.

Under the new system, all of the floating point values in `vertex`, `color`, and `instance` statements can now be replaced directly with dollar sign expressions. In order to achieve this simplification in syntax, a new type of time-varying floating point object was added to the UniGrafix parser. These kinematic floating point objects store three quantities: a default value, a floating point number, and a generic updating function. In addition, each time-varying expression stores a decision variable that references particular code within the generic updating function. An object's generic updating function is simply a pointer to a large case statement, the one shown in ug.dynam.c Figure 3.1. In addition, the decision variable stored with each time-varying expression allows the actual piece of updating code to be found within the generic updating function. See the direct correspondence between dollar sign C code fragments and the particular case statements generated in the conversion shown in Figure 3.1. The new object is treated like the other parser objects, and it is passed around as a floating point value. Finally, these time-varying floating point objects are linked into the UniGrafix scene graph and are updated using the stored generic function pointer and the case statement value. Both the first and second pass of the new UniGrafix parser uses these kinematic floating point objects.

The Augmented Scene Graph Structure

The second pass parser builds the scene graph and creates a linked list of all time-varying statements. See Figure 3.2 for an example of the statement list created for the example scene of the moving triangle. Now instead of outputting C code for a time-varying value, the second pass of the parser links the object in the scene graph to the compiled code via an auxiliary pointer. This pointer in the data structure of the scene graph statement serves two purposes: it tells us whether a statement is time-varying or static, and it also provides us with a link to the updating function necessary to obtain a new value for each successive frame.

The Time-Varying Expression Statement List

For the purposes of performance analysis a data structure similar to one used by the GLIDE system is incorporated into this second pass parser. The data structure is a list of all of the time-varying statements in the UG file, separated by type (see Figure 3.2 for an example). The creation of such a structure allows us to run through and update only these statements without having to search for them in the scene graph. Clearly, the rendering engine could have been modified to check and see if the statement was in fact time-varying and then simply update it before rendering. Doing so, however, would have complicated the performance measurements because the updating phase and the rendering phase would now be intermingled. Instead, by using this scheme, we achieve complete separation between updating and rendering and also save time by not having to traverse the whole scene graph during updating.

For each element of the time-varying statement list, we store a pointer to the updating function (the `switch` statement), the case number to which this particular statement refers, and a pointer to the actual UG statement structure (scene graph element). In order to update the time-varying elements of a file, we simply traverse this statement list, and for each statement we evaluate the updating function and place the returned values directly into the stored scene graph element. Since the order of traversal is important, we initially partition the statements by type. In the case of time-varying instance statements, it is necessary to update all of their time-varying transforms before they themselves are considered. Therefore, transform statements are processed first, and then the updating of instances simply involves recalculating their composite matrix, since they do not contain an explicit updating function. The order of traversal of the other elements is currently arbitrary since the updating of such statements can be performed independently.

Removal of the BUMP Syntax

In the BUMP system the extension statements are delineated by parenthesis for easy removal and updating purposes. Rather than keeping these syntactic elements, the extension system is simply removed and replaced with a GLIDE-like system in which any floating point value in a supported statement type can be replaced with a time varying expression encapsulated in dollar sign characters. In addition, all of the other elements of BUMP are also omitted; gone from the new system are the `(mdefine…)` and `(mexecute…)` statements as are the various `move` commands. Although these constructs are no longer present, the abilities of these various commands can now be directly implemented in the C code of the time-varying expressions. Functions for performing the interpolation can be written in C and included in the UniGrafix file, and the `(mexecute…)` command can be simulated using the C function `system`.

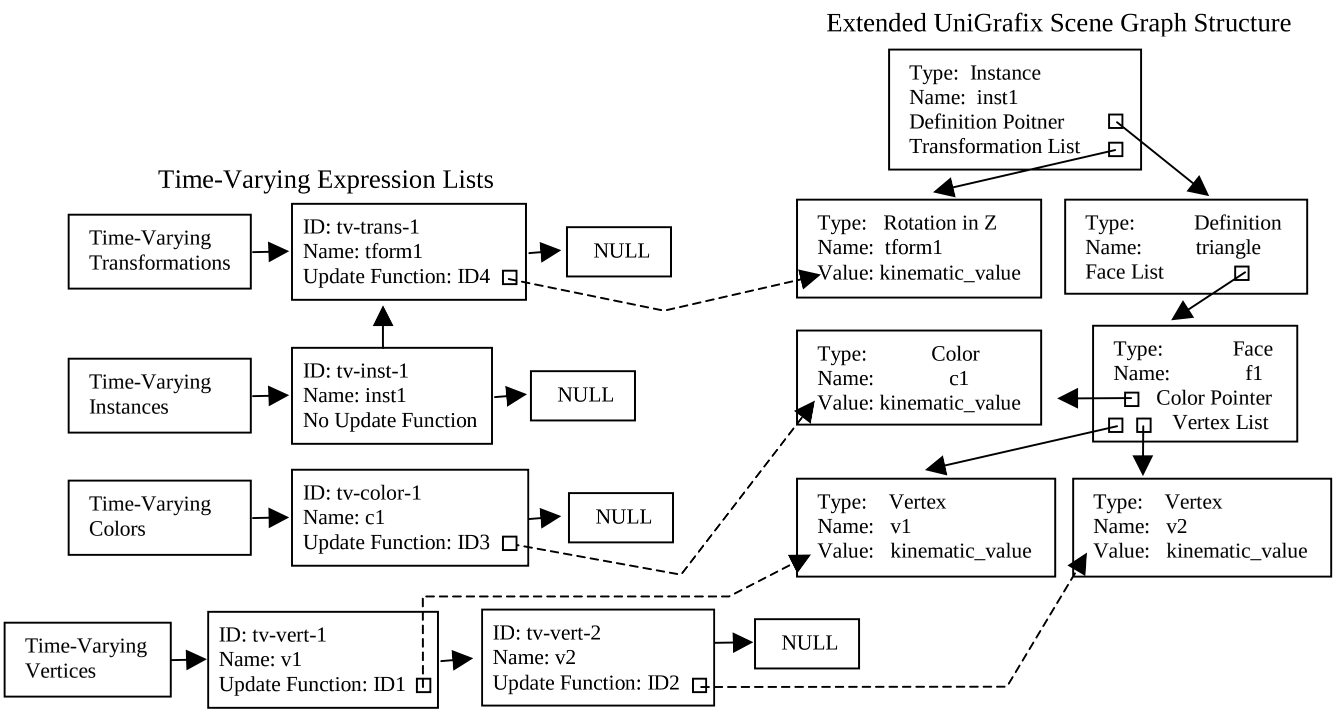

Figure 3.2: Scene Graph and Time-Varying Expression list created by the second pass parser. Structures based on the moving triangle scene of Figure 3.1. The dashed arrows indicate correspondence between the time-varying updating functions and the relevant kinematic values appearing in the scene graph.

Advantages and Disadvantages of the New Parser

One of the added benefits of using this system is that the contents of a dollar sign expression are no longer limited to the small subset of C math expressions BUMP supported. Now the contents of a dollar sign expression can be any valid piece of C code, which results in much greater expressiveness. Furthermore, dollar sign expressions can now contain functions that return floating point values. Using these functions, complex decisions can now be performed, thereby allowing arbitrarily complex behavior to be simulated in an external C file and referenced by geometry in a time-varying UG file. The syntax of this new GLIDE-like UniGrafix is presented in Appendix A: New Syntax for Time-Varying UniGrafix.

Given that we gain a considerable amount of flexibility and expressiveness by using C as the scripting language for these kinematic elements, what new things can we achieve employing such a system? Clearly, simple time-varying expressions of the form used in the original UGMovie files (such as `old_cubogear.ug` in Appendix B) can easily be mapped into this new system (see the file `new_cubogear.ug` in Appendix B). In addition we can make use of the ability to execute arbitrary C functions to return the floating point values for dollar sign expressions. This allows a conciseness in the writing of kinematic UG files that was previously only realizable using a preprocessor (see these examples `original_jug.ug`, `post_cpp_jug.ug`, and new_jug.ug in Appendix B). Aside from ease of use, the new paradigm allows for previously unrealizable complexity in the simulation model governing the behavior of objects. The file grav.ug and the related C file grav.c are an example of a simulation governing scene geometry. In the UniGrafix file the geometry makes references to a C function which returns the particular coordinates of a simulation object. The simulation (in this case four bodies obeying a gravity-like set of attractive forces) is actually performed in the C file and governs the movement of the scene's elements. The files are included in Appendix B. The primitive scripting language of the original BUMP extension does not permit a scene of such complexity to be created.

Since the new parser is based so closely on the GLIDE parser and compiled code system, it inherits all of its shortcomings and disadvantages. Namely, those mentioned in Section 2.1.2: the input file is parsed twice, the inability to interactively alter the scene's objects, the new parser no longer verifies input, and the scripting language provides two potential pitfalls (non-termination and potential pointer errors, i.e. segmentation faults and illegal de-referencing). The most important of these disadvantages are the potential pitfalls of the scripting language and the inability to alter scene objects interactively. The scripting language dangers can be removed by using a different scripting language such as Java™ [15] or VRMLscript [2,17], while interactivity can be enabled by incorporating dynamic linking of code libraries.

3.1.2 Modifications to the UGMovie Application

Once the parser was modified, three other modifications were necessary to update the UGMovie application to use the new syntax and compiled code paradigm for time-varying objects:

1.) All of the BUMP extension package function calls were removed from UGMovie

2.) The new UniGrafix parser was incorporated into UGMovie

3.) The first pass parser was encapsulated into an executable

These modifications were necessary to move from the parsed domain of BUMP into a completely compiled code system for updating time-varying objects. The current UGMovie application runs under SGI's IRIX operating system version 4.0.5.

Removal of the BUMP Extension Elements from UGMovie

First, all of the BUMP function calls were removed from the UGMovie application. In particular, the time-varying statement updating routines, the initialization routines, and the user interaction elements have all been modified. This has two primary effects on the UGMovie application. On the one hand, this greatly reduces the number of file input/output demands relative to the BUMP system and improves the overall performance. On the other hand, the BUMP features providing variable editing and behavior redefinition while visualizing the scene are no longer present. Originally, UGMovie achieved the updating of kinematic objects by removing all of the (meval…) type constructs (all information between the parenthesis) and writing these to temporary files and then evaluating these expressions using the BUMP function calls. The BUMP function calls then returned a static UniGrafix file that was spliced into the actual scene graph. These file input/output operations were performed every frame and caused a considerable portion of the performance problems. In addition, since the cost of re-parsing and evaluation is a function of expression length, UGMovie restricts the syntax to allow `(meval…)` statements within `def`/`end` pairs only. This allows the `(meval…)` expressions to be removed and re-parsed easily without requiring a re-parsing of the whole UniGrafix file.

Incorporating the New UniGrafix Parser

Next, the new UniGrafix parser was incorporated into the UGMovie program, thereby allowing this application to parse the new time-varying UG file syntax. In addition, since the new parser builds a new data structure, the time-varying statement list discussed previously, a traversal mechanism for this list was added to the UGMovie application in the place of the original BUMP evaluation functions. Trace information of the UGMovie executable demonstrates performance problems precisely because of this constant expression re-parsing and evaluation. In contrast the new updating scheme using the traversal mechanism and the compiled code paradigm removes this re-parsing and improves the performance of the application by a factor of 2.8 to 10.0 depending on the graphics intensity of the scene file.

Creation of a Stand-alone First Pass Parser

Finally, a relatively simple wrapper program was written to encapsulate one version of the new UniGrafix parser set to produce time-varying expression C code as output. As mentioned previously, the new UniGrafix parser exists in two distinct modes. The first produces C code from a UniGrafix file, which contains time-varying statements, and does not build an explicit scene graph data structure. The second does not produce any code but rather builds a scene graph and a time-varying statement list based on the input file. The former version of the parser now appears in a new wrapper executable called `dynUG`, while the later is incorporated into the new UGMovie application as the file parser. This new program, `dynUG`, takes a time-varying UG file using the new syntax as input and produces the `ug.dynam.c` file as output (for an example `ug.dynam.c` file see Appendix B). The output C code file (`ug.dynam.c`) must be compiled and linked into the new UGMovie application in order to view the time-varying UG file using this new viewing application. Under the current scheme, the new UGMovie executable must be recompiled in order to alter the time-varying expressions of a particular file, or to load in a different time-varying scene. Finally, a script has been written which allows the new UniGrafix scenes to be viewed easily using the new UGMovie. The script performs the necessary parsing, linking, and compilation, as well as loading the file into the new UGMovie executable (see Appendix C:1 for the location of this script and its tutorial). The current script and executable operates under version IRIX 4.0.5 of the operating system.

3.2 Dynamic Linking

The recompilation step of the new UGMovie executable, or for that matter GLIDE, is not strictly necessary. Using the dynamic linking functions provided by Silicon Graphic’s IRIX operating system versions 5 and later, it is possible to augment the current system to allow arbitrary scene files to be loaded by the UGMovie application without the need for re-compilation. The incorporation of dynamic linking has not been done yet due to operating system incompatibilities, but outlined below are the necessary elements required. Currently, the new UGMovie application compiles under IRIX 4.0.5 and the necessary dynamic linking features require versions 5.x or later of the operating system. In order for UGMovie to recompile under the newer versions of IRIX the Motif [10] user interface must be upgraded or the entire interface replaced by a different widget package such as TCL/TK [20] which will compile under the new versions of IRIX. Once the UGMovie application's user interface is upgraded, the addition of the dynamic linking capabilities is a relatively small task, however the upgrading of the interface could take a considerable amount of time. The use of dynamic linking is important. It enhances the compiled code paradigm making it much more interactive. As discussed in Sections 1 and 2, this is an improvement within the virtual reality taxonomy system [30] and to the concept of *Interaction* discussed there.

In order to achieve dynamic linking in UGMovie, we would only need to add a mechanism for associating a shared library with a particular input file. The `dynUG` parser could be modified to produce a shared dynamic link library instead of a piece of C code. We could then use the name of the dynamic file as a mechanism for creating the shared library. For example, the `octabug.ug` file when parsed and analyzed by a new dynamic linking `dynUG` would produce a shared library called `liboctabug.so` instead of the C code file. The UGMovie application would then make use of this `liboctabug.so` and obtain access to all of the time-varying expressions referenced in the UniGrafix file `octabug.ug`.

Having altered the first pass parser `dynUG`, now the actual UGMovie application would also need to be modified to make use of certain dynamic linking functions specific to the IRIX operating system. The two relevant functions are `dlopen` [8] and `dlsym` [9]. Under this scheme, when a file is loaded into the application, a call to the function `dlopen`, which loads a dynamic link library, would be performed. This function takes the name of a shared library as its argument (from above the name would be based on the input UniGrafix file). As the UniGrafix parser built the loaded file's scene graph, we would perform a lookup on any referenced functions to find them in the dynamic link library. The lookup function `dlsym` would be used with a function's name as its argument; it returns a function pointer corresponding to the named function in the dynamic link library. Once we have the function pointer, we can simply substitute this pointer into the time-varying statement list element for that particular statement. Having done this, updating proceeds normally since the correct function pointer has been added to all of the time-varying statements in the scene graph. Given this capability, the UGMovie application could load any valid UG file without being re-compiled. In addition, by using such a framework, we are now able to interactively alter the code bound to a given statement while the application is running. In order to achieve real interactivity, the code would need to be in the original shared library, but if the user were willing to wait for a new shared library to be created, then any arbitrary code could be added to a scene's objects without exiting the application. The overall performance effect of this scheme would be a slightly longer loading time and perhaps a longer access time to run a particular function in the shared library.

3.3 Performance Results

In order to analyze the performance differences between the old method employed by the BUMP extension package and the new GLIDE paradigm applied to UGMovie, we have used a number of profiling and timing tools to compare the two methods. Instead of simply reporting the increases in frame rate obtained by using the new paradigm, we also estimate the improvements in performance and determine the absolute possible speed for the given test animations, if the expression updating took no computation time. We do this in order to gauge the success of our shift from the old parsed system to the new compiled system and also to determine how well the new system works in absolute terms.

3.3.1 Analysis of Pixie and Prof Trace Data

Since the graphics and rendering capabilities of UGMovie remain unaltered by our modifications, an obvious choice of analysis/measurement tools is the UNIX profiling utilities *pixie* [22] and *prof* [23]. These utilities generate instruction trace information and return statistics on the number of cycles executed for all of the functions in the traced executable. For the purposes of comparison, all of the various measurements were performed on both executables animating the same series of five test files. All of the test files exist in both formats, the original BUMP enhanced UG syntax and the new syntax developed for the compiled paradigm. The test files used for comparison purposes all describe relatively simple time-varying objects such as moving sculptures. These animated objects were chosen since they are all elements in the WALKTHRU and they have different graphics complexities as well as different behavioral complexities. All of the performance tests described in this section were executed using one processor of a 33MHz SGI VGX machine (R2000A/R3000 CPU and R2010A/R3010 FPU) with 4 Mbytes texture memory and 128 Mbytes of main memory.

The primary hypothesis made before performing these measurements was that under the new paradigm the updating of time-varying expressions would take almost no computation relative to the old parsed method. Therefore, since the rendering of the geometry is unaffected by this new system of updating, we should be able to estimate the improvement in speed of the animation by simply removing the cost of updating time-varying expressions from the original parsed system. This idea was used to analyze the trace information obtained using the UNIX profilers *pixie* and *prof* . In Table 3.3 we present information relevant to the test files. In particular, we present estimates of the percent of cycles used to update the time-varying expressions and compare them to actual measurements of the updating costs for the old parsed system of evaluation. From Table 3.3 we see that the percentage spent updating is related to the amount of information to be parsed as well as the geometric complexity of the scene. For scenes where the amount of parsing is small relative to the geometric complexity (`Cubogear`), we notice the percent of total computation spent on parsing is correspondingly smaller than scenes where the opposite is true (`Spheres`).

MODEL NAME	1 MEVAL CHARACTERS	2 SCENE POLYGONS	3 ESTIMATED PERCENT SPENT UPDATING[2]	4 MEASURED PERCENT SPENT UPDATING[3]
Spheres	9362	61	95.5 %	95.8 %
Canaries	9362	982	79.3 %	79.5 %
Octabug	4512	128	91.8 %	91.3 %
Hingebug	4896	544	83.7 %	85.5 %
Cubogear	1569	3002	31.1 %	35.5 %

Table 3.3: Characteristics of time-varying UniGrafix test animation files for the parsed system.

In Table 3.3 the estimated percentage column (col 3) represents the percent of the total computation performed updating the time-varying expressions using the original application. These estimates were obtained by analyzing the trace information and removing cycles caused by functions responsible for this updating. The functions removed from the total cycle count fall into three related categories. First, those functions responsible for parsing and evaluating the time-varying expressions were removed (`BP_yyparse, BP_yylex, Bpmeval, UGfind_ref, UG_yylook,` and `UG_yyparse`). Second, various string related functions and file input/output functions were removed (`strcpy, strlen, strrchr, _atod, _atof, input,` and `_doprnt`). These string functions were used by functions in the first category and by other auxiliary functions too numerous and computationally insignificant to merit removal themselves. Finally, the memory behavior differs significantly between the parsed and the compiled code systems and thus the cycles contributed by the `malloc` function needed to be dealt with separately. Since a good portion of the cycles executed by `malloc` are caused by the Motif interaction code, which is shared by both versions of the application, it was necessary to keep track of these counts, since they have nothing to do with the expression updating. Therefore, `malloc` was not simply removed from the old system count data but rather only the difference between the new system's `malloc` data and the old system's was removed. This difference corresponded to the amount of improvement between the two different system's memory behavior.

Once the total number of cycles was determined, we could estimate the percent of cycles actually spent doing the updating relative to the total number of cycles executed. This estimate appears in column 3. Finally, we determined the real improvement by simply tracing the new system and

---

[2] The values in this column are estimated from the original parsed system trace information.

[3] The values in this column are measured from the new parsed system using timing tools.

comparing its total number of cycles executed versus the total executed in the original application using the old system. Our fundamental assumption is that the new system will remove precisely these labeled function calls and its memory performance will be improved relative to the original system. Due to the close agreement between columns 3 and 4 we are confident that the labeling of elements was correct and that new system removes these elements and improves the memory behavior by the amount estimated.

Based on the percentages measured and listed in Table 3.3, we are now able to estimate the total number of cycles that the new system ought to execute, if our assumption that the primary cost of updating in the original is completely removed from the cycle counts. Table 3.4 provides this cycle count information. The first column lists the original application's cycles executed for 600 frame runs of the various test file animations. The estimated column (col 2) is simply the product of the original and one minus the measured percentage (col 4) found in Table 3.3. In addition, the new application was also traced using the profiling tools; and the actual cycle measurements obtained are reported in column 3. We note that there is close agreement between our estimates of the total number of cycles executed (col 2) and the total cycles actually measured (col 3). The last column (col 4) reports the actual improvement in frame rate that we expect to see if the cycle times measured are directly related to an animation's frame rate.

MODEL NAME	TOTAL NUMBER OF CYCLES (millions of cycles)			4 EXPECTED SPEEDUP FACTOR BASED ON TRACE/PROFILING MEASUREMENTS
	1 MEASURED OLD SYSTEM	2 ESTIMATED NEW SYSTEM	3 MEASURED NEW SYSTEM	
Spheres	1601	72	68	20.0
Canaries	2037	422	419	5.0
Octabug	1749	144	152	12.0
Hingebug	2168	352	315	6.0
Cubogear	2066	1423	1333	1.5

Table 3.4: Trace measurements for 600 frames of test animations.

In order to obtain frame rate measurements for the animations, it is necessary to convert from the cycle domain into the time domain. As a first attempt, the cycles themselves were translated into time based on the 33 MHz processor speed of our test machine. These results do not match the observed time since the timing measurements were actually preformed in multi-user mode. Due to this complication, the actual execution times were used, and the estimates for the new systems execution time were derived using the percentages of Table 3.3, in exactly the same way they were used to determine the estimates of Table 3.4. The old system's measured execution time (col 1), and an estimate (col 3) and measurement (col 4) of the new system's execution time are reported in Table 3.5. All time measurements were performed using 600 frame animations of the test files. The reader will notice that the agreement between the results when mapped to the time domain is not nearly as good as in Table 3.4. After seeing these results it became clear that the profiling tools were not modeling the execution accurately enough for these purposes. Based on this complication we attempted to utilize another profiling tool *glprof* [14], which is an SGI specific application. The *glprof* profiling tool is capable of simulating the graphics pipeline and after analyzing traces for these test files it became clear that this is not the case for the other tools *pixie* and *prof*. Thus, the original trace measurements are not accurately modeling the performance behavior of the new system since its limiting factor is the graphics pipeline. However using the *glprof* tool alone was not possible, since it does not accurately model an application's use of the CPU, and therefore it would give erroneous results for the original parsed system. Caught between these two limitations we decided to move to a new simpler system of measurement, which does not use cycle count information at all.

MODEL NAME	1 MEASURED EXECUTION TIME FOR OLD SYSTEM (sec)	2 PERCENT OF CYCLES SPENT UPDATING	3 EXPECTED EXECUTION TIME FOR NEW SYSTEM (sec)	4 MEASURED EXECUTION TIME FOR NEW SYSTEM (sec)
Spheres	100	95.8 %	4	10
Canaries	128	79.5 %	26	30
Octabug	146	91.3 %	13	20
Hingebug	150	85.5 %	22	22
Cubogear	167	35.5 %	108	60

Table 3.5: Comparison of expected time versus measured time.

3.3.2 Analysis of Time Measurements

The new measurements are based on modifying the actual applications and on decoupling the rendering of the animations from the actual updating of the time-varying expressions. In order to achieve this, both the old UGMovie application and the new one were modified to run in three separate modes: normal mode, rendering only mode, and updating expressions only mode. By running the application in updating only mode, we could obtain a reasonable estimate on the time of execution for just the updating and not the rendering portion of the animation. In addition, running the application in render only mode gives a measurement of the static complexity of the scene, which was then used as a baseline to compare against the rendering cost of an animation. These measurements are shown in Table 3.6.

MODEL NAME	ORIGINAL SYSTEM			NEW SYSTEM		
	1 MEASURED TOTAL TIME	2 MEASURED TIME UPDATING	3 MEASURED TIME RENDERING	4 MEASURED TIME UPDATING	5 EXPECTED TOTAL TIME	6 MEASURED TOTAL TIME
Spheres	100	91	10.0	3.4	12.4	10.0
Canaries	128	101	30.0	3.5	30.5	30.2
Octabug	146	132	10.1	8.9	22.9	20.0
Hingebug	150	133	20.1	9.1	26.1	22.4
Cubogear	167	107	60.0	6.3	66.3	60.0

Table 3.6: Measurements (in seconds) for estimating the total time for the new system.

The first column of Table 3.6 contains the total time measured for a 600 frame animation of the specified test file using the original UGMovie system. The second column contains the total execution time for the application run in update only mode, so no animation was actually rendered for these test cases. From the first two columns we can estimate the time spent rendering the scene by simply taking the difference between the total cost and the updating cost. When we compare this calculated estimate for time with the measured execution time of the application (run in display only mode, column 3), we find in all five cases that there is very close agreement. The fourth column contains the execution time values for the new system running in update only mode. The estimate of the total execution time for the new system running in the full mode is presented in column 5. The values presented in column 5 are simply the sum of column four with the difference between columns one and two. Finally, the actual measurements for the new system running in the full mode are presented in column 6 of Table 3.6.

One important observation from Table 3.6 is that while the relative improvements in updating execution time between just columns 2 and 4 are rather large, the actual difference between the total execution times, columns 1 and 5, is not as substantial. Consider the `Cubogear` case in which the original updating cost was 107 seconds and the new cost is 6.3 seconds, these measurements alone would suggest a speedup in frame rate of almost 17 times. However, the actual final estimate is much less dramatic with the difference being only about a factor of 2.5. The disparity between these results occurs because in a rich scene such as the `Cubogear` the rendering aspect of the scene, as opposed to its simulation, dominates the display cost. Recall that the modifications made to the application do not effect the rendering of the scene.

MODEL NAME	1 MEASURED FRAME RATE FOR ORIGINAL SYSTEM	2 EXPECTED FRAME RATE FOR NEW SYSTEM	3 COMPUTED ABSOLUTE RENDER-BOUND FRAME RATE	4 MEASURED FRAME RATE FOR NEW SYSTEM	5 IMPROVEMENT FACTOR BETWEEN OLD AND NEW SYSTEM
Spheres	6.0	48.4	66.7	60.0	10.0
Canaries	4.7	19.7	22.2	19.9	4.2
Octabug	4.1	26.2	42.9	30.0	7.3
Hingebug	4.0	23.0	35.3	26.8	6.7
Cubogear	3.6	9.2	10.2	10.0	2.8

Table 3.7: Comparison of expected, absolute and measured frame rates (frames per second).

Table 3.7 summarizes all of the relevant frame rates for an easy comparison of estimated and measured performance for the two systems. Columns 1 through 4 represent the frame rates both measured and estimated. The values in column 1 were obtained by dividing the total number of frames (600) by the values in column 1 of Table 3.6. Similarly, the number of frames (600) divided by the values of column 5 of Table 3.6 generate those in column 2, and finally 600 divided by the difference between columns 1 and 2 of Table 3.6 yields column 3. Conceptually, the third column represents the maximum possible frame rate given the current renderer if no other processing is performed, i.e., if the updating of time-varying expressions and geometry took no computation or matrix operations. Finally, the improvement factor represents the ratio of new measured frame rates versus the old system's frame rates for the given test files.

3.3.3 Comparison of the Performance Measurements

Comparing the estimated with the measured frame rates in Table 3.7 (columns 2 and 4) we find strong agreement between the values. Furthermore, we note that the measured system (col 4) always performs better than our estimate of performance (col 2). Since the estimates did not account for the SGI graphics pipeline, the use of such a system actually yields a performance improvement because the updating of time-varying expressions, which is primarily CPU driven, can be executed concurrently with various graphics operations.

Next we turn our attention to the real question; how much did the new system improve relative to the original? From Table 3.7 we can clearly see a marked improvement between columns 1 and 4 as represented by the improvement factor listed in the last column. Clearly, the new system is considerably faster than the original; we see, for example, a factor of 10 speedup in the `Spheres` case. Since our main goal was to speed up the updating of time varying elements in the scene we should see how the new system compares to the fastest possible rendering of the scene assuming no update costs. By comparing columns 3 and 4 of Table 3.7 we note that for three out of the five cases we are very close to being completely dominated by the rendering time. In the other two cases (`Octabug` and `Hingebug`) we achieve a slower frame rate because the complexity of the updating is relatively high compared to the rendering cost. The primary problem with these two files is the number of time-varying instances and transforms relative to the static scene complexity. The computation of the updating functions and the multiplication of transform matrices become a significant element of the total cost in these two cases. In contrast to these two models, in the `Cubogear` scene the limiting performance factor is rendering time. This suggests that in order to obtain the best relative performance out of our system we should try to balance rendering complexity and updating complexity, thereby keeping both the rendering pipeline and the updating processor busy most of the time.

4 The WALKTHRU Authoring Environment for Kinematic Objects

In virtual reality systems, time-varying objects provide interesting elements for a user to interact with. The virtual environment as a whole seems more real when clocks keep time and trees sway in the wind. For these reasons, numerous present day systems and standards incorporate time-varying elements into their environments. In particular, the WALKTHRU system has been enhanced by using the BUMP extended syntax to incorporate primitive kinematic objects into its virtual reality environment. We abandon the BUMP system because of its limited expressibility and because of the performance penalties discussed in Section 3, and we adopt a GLIDE-based system instead.

A new GLIDE-like parser has been incorporated into the WALKTHRU's model building utilities. The WALKTHRU's updating and evaluation functions have been modified to handle the new compiled code object types. In addition, the authoring environment and the display routines have been altered to allow the WALKEDIT package to be used as an interaction mechanism with these new kinematic objects. At the end of the section we present some examples of new compiled code objects in the WALKTHRU's virtual environment.

4.1 The Original Kinematic Objects in the WALKTHRU

The original Berkeley WALKTHRU system supports a limited variety of time-varying objects. The types of objects permitted are a subset of those allowed by UniGrafix with the BUMP extension statements. In particular, the subset includes objects that move in a rigid-body fashion using time-varying transformations, but not object that have deforming vertex transformations or whose color changes with time. In order to understand the WALKTHRU system's orignal BUMP based mechanism for updating time-varying objects, we first need to review the creation of the object database. None of the various WALKTHRU systems actually take UniGrafix as direct input; they are all based on a visibility cell database, which is a decomposition of the model into cells and visibility information. The cells contain objects as well as lists of other visible cells. The visibility information is used by the WALKTHRU to prune the potential space of visible objects to allow real-time navigation of the virtual environment regardless of actual model size. In the WALKTHRU system, the applications `wkcreate` and `wksplit` are used to build this visibility cell database; then `wkadd` is used to add the various content objects into these visibility cells.

The WALKTHRU's original `wkadd` utility creates animation structures for each time-varying component in the 3D model. For a time-varying object, the animation structure contains a linked list of all of the transformation statements (both static and time-dependent); for a static object the animation structure is empty. These transform elements are the same as those appearing in the original UniGrafix instance statement defining the time-varying object. The transform statements appearing in the linked list are stored as strings, as are the dollar sign expressions originally appearing in the UniGrafix file. Both static and kinematic transforms are inserted into this list by `wkadd`. Figure 4.1 demonstrates an example UniGrafix object and the corresponding animation structure created by the original `wkadd` for a WALKTHRU database file.

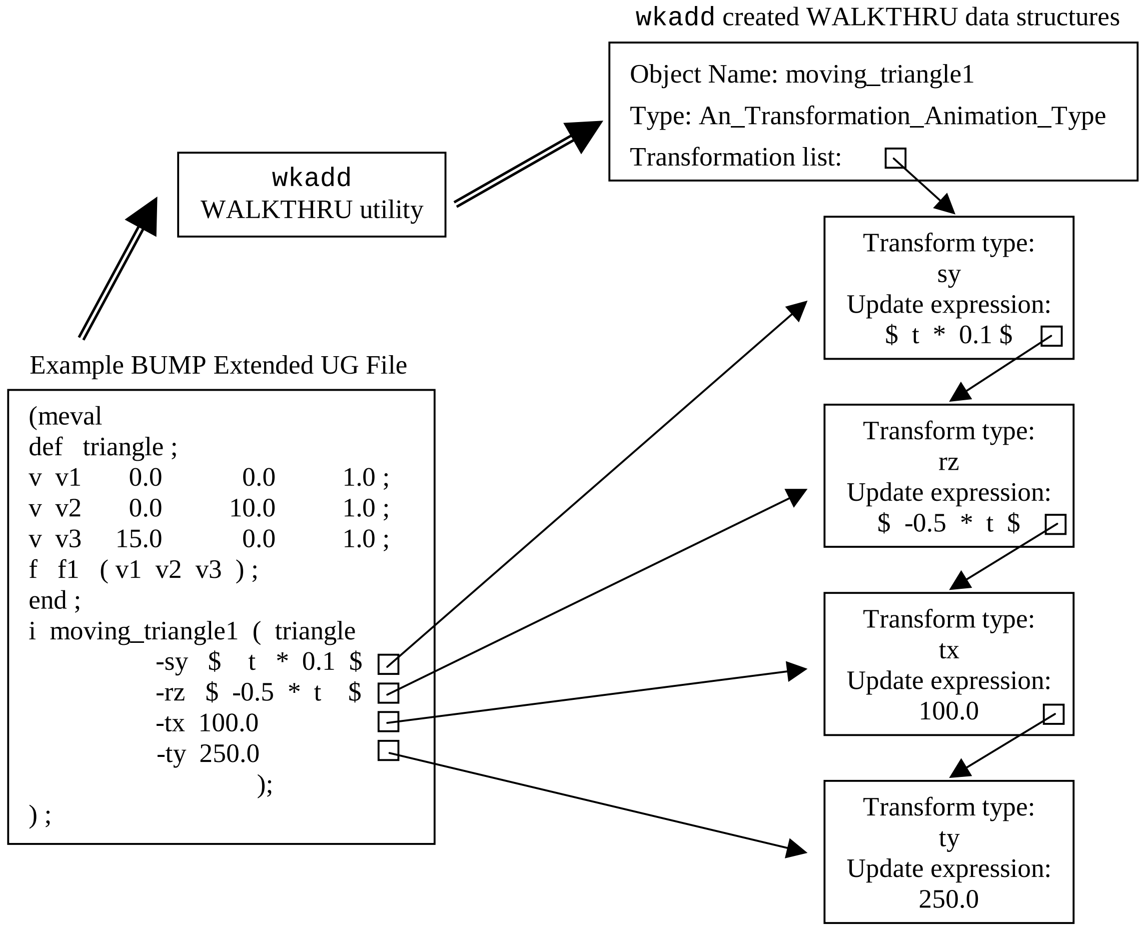

Figure 4.1: Example of original WALKTHRU animation structure created from a BUMP extended UniGrafix file.

The updating function for kinematic objects is a simple parser and matrix multiplier that operates on a kinematic object's animation structure and in particular on its linked list of transformation elements. In essence, the updating function employs two separate parsers and a full matrix multiplication for each of the time-varying objects present in the scene. First, the updating function evaluates the dollar sign expressions using the BUMP parser and evaluation functions. Next, the transformation strings are parsed to determine the type of transform, i.e., whether they are a rotation, translation, scaling, or arbitrary matrix transform. Finally, all of these transform matrices (both static and time-varying) are multiplied together to compute a composite transformation. The composite transformation is then copied into the object's transformation structure. This object transformation is the matrix used to position an object's geometry within the WALKTHRU prior to rendering. See Figure 4.2 for a graphical depiction of this updating process performed on the animation structure of a time-varying object.

WALKTHRU Animation Data Structures

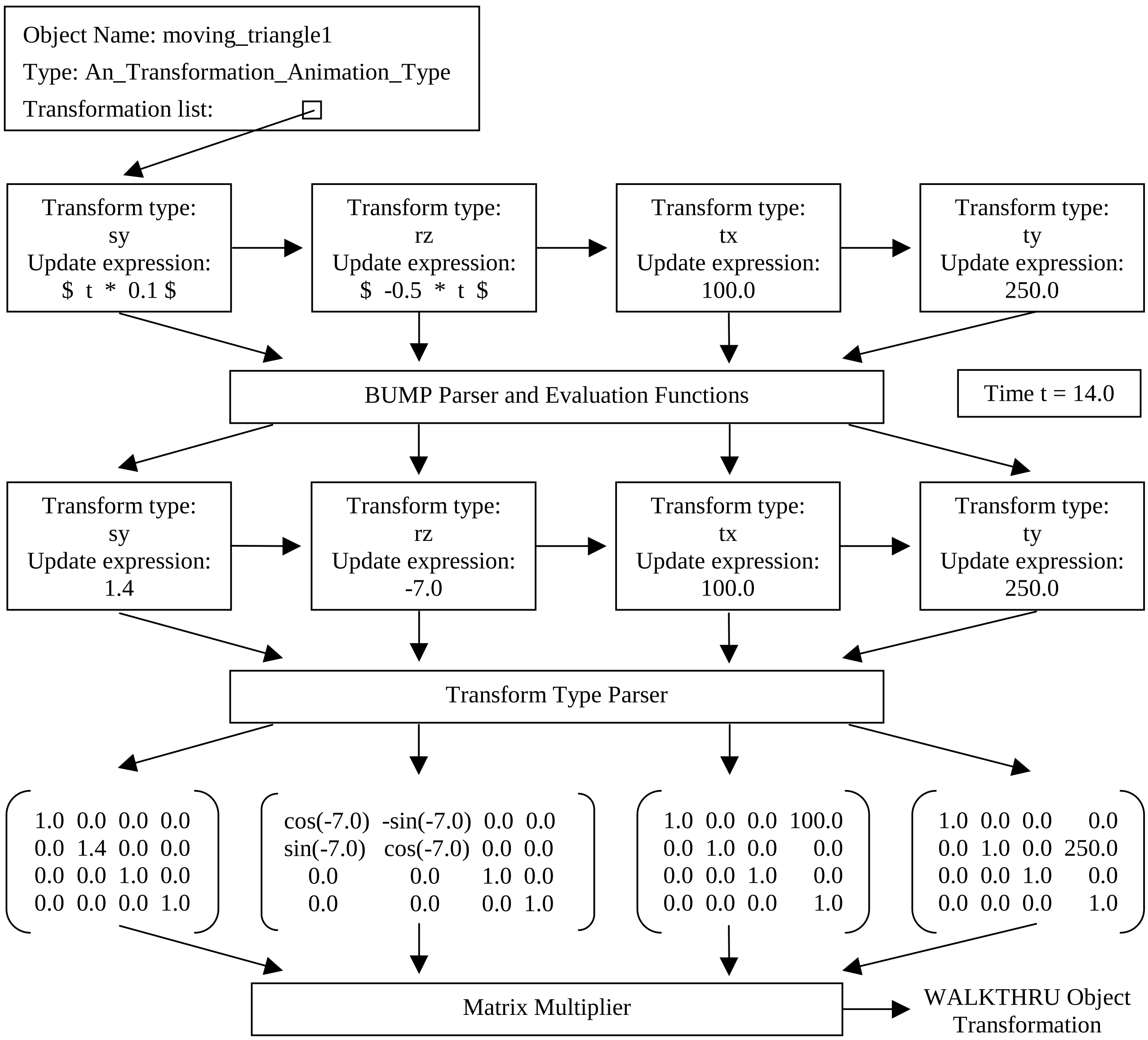

Figure 4.2: Updating process using BUMP for a WALKTHRU animation data structure

The WALKTHRU system's implementation of time-varying objects has a number of limitations. First, the WALKTHRU's visibility decomposition relies heavily on a two-tier structure of cells and objects, where only the objects can be time-varying. The reliance on objects as the only type of primitive geometry structure, and the few elements of BUMP supported, limit the kinds of kinematic components an author can incorporate into the model. For example, time-varying objects permitted in BUMP and UGMovie that change their color or modify their individual vertex positions could not be added to WALKTHRU databases using the original `wkadd`. The absence of time-varying color is simply a fault of the incompleteness of the parser and updating routines, while the inability to alter lower level geometry is a problem with the WALKTHRU system. Originally designed for maximum performance, the WALKTHRU system stores all of its elements in display list structures. Thus the explicit connectivity information contained in the UniGrafix winged-edge data structures, as well as the original scene hierarchy have been lost. Even if the WALKTHRU were altered to support all or most of the BUMP system extensions, the performance findings of Section 3 demonstrate that a compiled system such as GLIDE is both more expressive and considerably faster than the BUMP system. Therefore, simply augmenting the existing system is probably not worth the effort.

Finally, there is the question of interaction. In the WALKTHRU there are two fundamental problems that prevent the construction of interactive objects. Previous to the WALKEDIT work, there was no mechanism for transmitting messages to particular objects based on the user's mouse events. In addition, the subset of the BUMP expression script language that the WALKTHRU implements has no means of allowing these time-varying expressions to be modified by user interaction. Therefore we will use WALKEDIT's interaction paradigm and this new more general compiled-code mechanism to augment the current capabilities of the virtual environment and to provide for the creation of interactive time-varying objects.

Because of the limitations of the WALKTHRU, the only time-varying objects which appear in the virtual environment are those that have no interaction and whose defining expressions are simple equations of time. In the original model of Soda Hall there are only two classes of kinematic objects, a number of non-interactive moving sculptures and an elevator which simply ascends and descends through the building stopping at every floor. Clearly these objects are not particularly interesting, nor are they useful other than as simple animations. The following work presented in this section aims to improve the current state of these elements and to enable much more interesting and complicated objects to be created and incorporated into the system.

4.2 Additions and Alterations to the WALKTHRU's Animation Library

The WALKTHRU system has a library devoted to dealing with the time-varying aspect of kinematic objects, called `libanim`. This library defines a single type of kinematic object as well as some other related elements, such as a `time` variable and a mechanism for updating these time-varying objects. In order to augment the current system with the new compiled-code paradigm, a new type of time-varying object was added to the library. The compiled kinematic object type differs from the BUMP version in three ways. First, the new system decomposes the initial UniGrafix instance of an object into two separate matrices. One is the static placement matrix, which consists of all of the non-dollar sign transforms of the instance statement multiplied together. The `wkadd` application is responsible for creating this matrix and placing it in an object's description within the WALKTHRU database. The other matrix is the time-varying behavior matrix, which is generated from a linked list of UniGrafix kinematic transform elements. The linked list itself is formed from the dollar sign transforms of the object's instance statement and is created by the `wkadd` application. The second difference from BUMP is that the linked list elements are compiled updating functions instead of the previous expression strings. Finally, an interaction context is provided to the updating functions allowing objects to exhibit interactive behavior.

4.2.1 Altering the Animation Structure

The first change to the animation structure of `libanim` was the addition of this new type of compiled-code object. The new object type is added to the enumerated types for the animation structure and is called `AN_COMPILED_TFORM_ANIM_TYPE`. Aside from this change the actual structure of an animation is very similar to the BUMP implementation. We recall from the previous discussion (Section 4.1 and Figure 4.1) that an original time-varying UniGrafix instance statement is decomposed into a linked list of transformation elements with values stored for each element. There are two differences between the new paradigm and the BUMP system. First, the only elements stored in the list for these new objects are time-varying transforms; static transformation components are no longer stored in this list. In addition the meaning of the values stored has changed; for the new paradigm these values are links to compiled updating code instead of being expressions that BUMP would parse and evaluate.

The values stored in the new compiled kinematic animation structures are now simply numbers acting as links to the compiled updating code. A given number associates a particular piece of updating code with its transform. These numbers are generated based on the parse order of the original UniGrafix file. Therefore, the order of time-varying expressions should not be changed between the first and second passes of the parser. Again as in Section 3.1.2, the wrapper application `dynUG` is executed on the time-varying scene to generate the C code file `ug.dynam.c`. This generated C code file is then linked into the animation library `libanim` when the WALKTHRU itself is compiled and linked into a rendering system. Figure 4.3 presents the new data structure with an example UniGrafix object converted into this WALKTHRU format.

4.2.2 Changes to the Kinematic Object Updating Routines

Instead of evaluating expressions during an update, as in the BUMP system, the new updating function executes the pre-compiled and linked code. The animation library `libanim` decides based on the type of animation object found and determines whether it is one of the old BUMP defined objects or one of the new GLIDE-based object types. For these new compiled-code objects, the update process executes a generic function with an object specific argument, which is based on the parse order of the original statements. We recall from Section 3, that the code produced and output in the `ug.dynam.c` file by the wrapper application `dynUG` contains a single `switch` function used to update all time-varying elements. One of the arguments to this function, the `which` argument, is used by the `switch` statement to choose between the various code elements in a time-varying scene. For the WALKTHRU, we store a different value of `which` for each transform of an object; thus, each transform can reference its own unique piece of code appearing in the `ug.dynam.c` file. Figure 4.3 demonstrates this link between updating code and the WALKTHRU data structures.

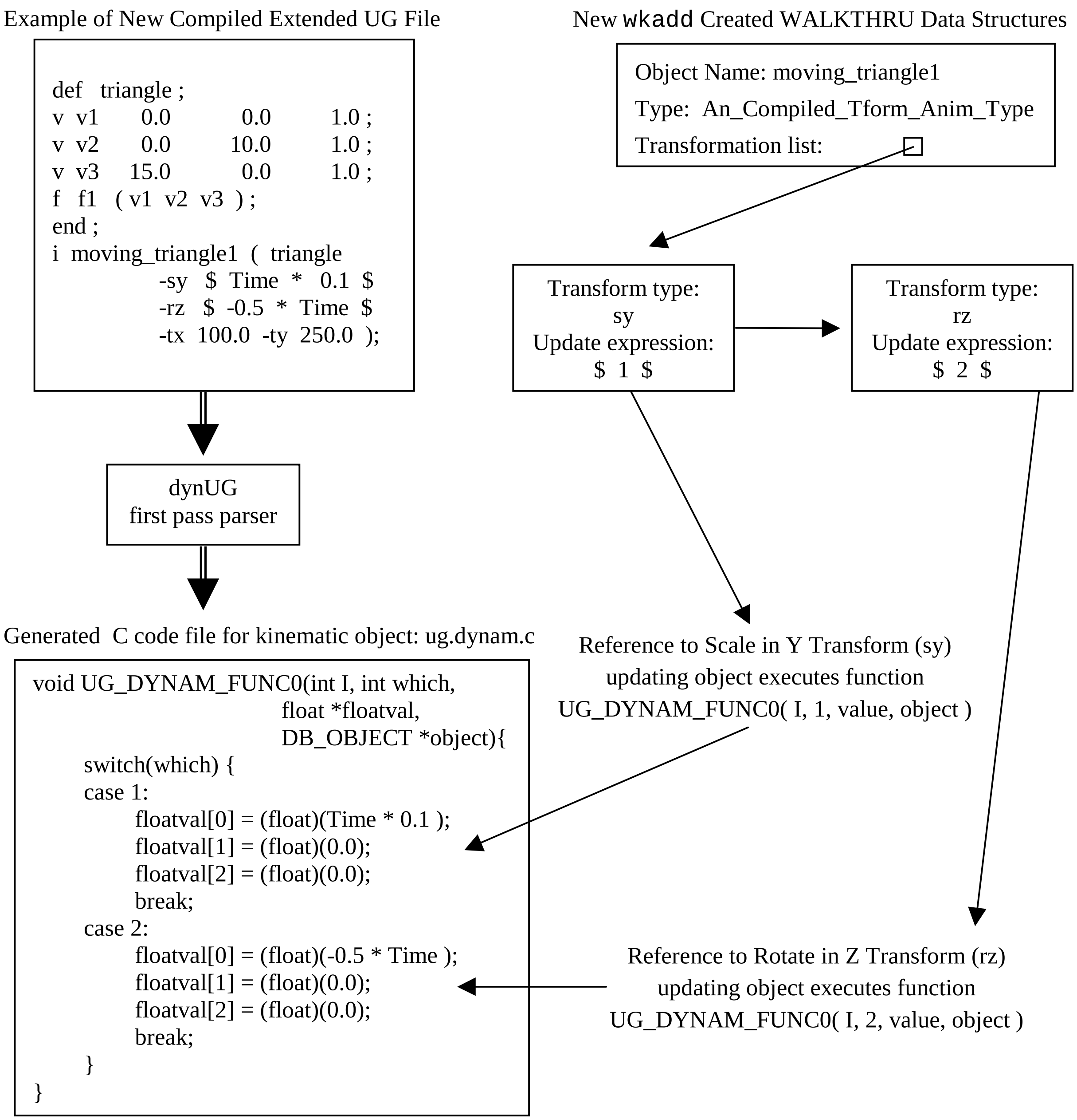

Figure 4.3: Example UniGrafix file converted into new WALKTHRU animation structure with the associated code generated by `dynUG`

The final enhancement of the new updating system is the ability of compiled updating code to modify object properties other than geometric transformations. Using the new system, an object's color and texture, in addition to its transformations, can vary with time. We enable this final enhancement by augmenting the animation evaluation function to pass an object's pointer into its transformation updating functions. This allows a transformation element to alter its parent object's data structure and in particular the color or texture properties of the parent object. Access to a variable called `object`, which is a pointer to the parent instance statement, from within the context of a dollar sign expression allows these properties to modified. See section 4.4.3 for an example of a multi-colored and multi-textured object that cycles through a variety of colors and textures.

4.2.3 Modifications to Enable WALKEDIT to Manipulate Kinematic Objects

WALKEDIT's primary purpose is to allow intuitive and realistic object placement and manipulation within the virtual environment. Due to incompatibilities the user could not move BUMP objects using the original WALKEDIT manipulation system. If the user tried to move an object, the object would revert back to its original location whenever it was re-drawn. In the original system, there was a conflict over the use of an object's positioning matrix, which is used to determine the location of geometry within the virtual environment. Both BUMP and WALKEDIT would modify this matrix and overwrite each other's results causing this strange interaction behavior.

In order to reconcile this conflict, a number of options were considered. The best possible solution would be to simply add a new matrix to the object data structure. This was not implemented, but in such a scheme WALKTHRU/WALKEDIT objects would have two separate matrices, one for WALKEDIT to use as a positioning matrix and another one for the time-varying behavior elements. The kinematic system would remain unchanged; it would produce an absolute positioning matrix for an object, which would then be multiplied by this new incremental fine-positioning matrix. With such structures, objects could be moved while being updated, and the original system of defining objects would not need to be altered. This approach has the serious disadvantage that it requires modifying the actual object data structures of the WALKTHRU, invalidating all pre-existing WALKTHRU database files and necessitating large systematic changes to most WALKTHRU libraries.

For simplicity and backwards-compatibility, we implemented a different solution that alters the meaning and effect of the new time-varying kinematic instance statement; it leaves all other statement types unchanged. This solution separates the main positioning matrix into two parts: a static placement matrix, which is stored within the object's data structure, and a new time-varying behavior matrix that is *not* stored since we did not want to modify the WALKTHRU object data structures. Instead, the new time-varying behavior matrix is regenerated every frame. The WALKEDIT manipulations in the new WALK-KIN system operate as before upon the static matrix. In this system, the time-varying matrix is pre-multiplied so that its transformations are performed in the object's local coordinate space. This modification changes the behavior of objects in two ways. First, the static elements of the transformation statements are *not* regenerated after the object is added via the `wkadd` program. In effect these transformation statements become a placement matrix for the object and can only be subsequently altered by the WALKEDIT manipulation procedures. Second, the time-varying transforms are based on the object's local coordinate system regardless of their ordering with respect to the static transformations. The rationale behind this scheme is that the time-varying elements help to define the object itself, whereas the other static transforms simply place an already existing whole object within the virtual environment. Figure 4.4 illustrates the kinematic updating / manipulation pipeline for the new WALK-KIN package with data and matrices for the various stages.

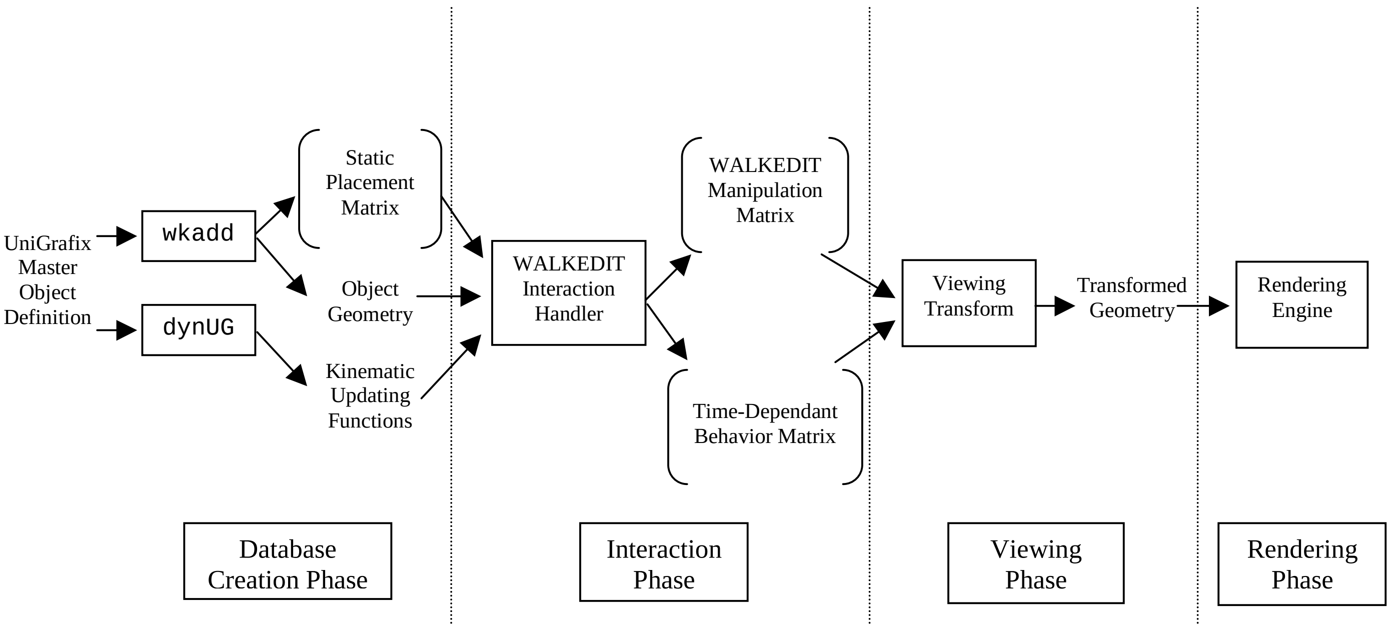

Figure 4.4: The new interaction / manipulation pipeline for kinematic objects, processes are shown as boxes and data are shown as text or as matrices.

4.3 Modifications to the WALKTHRU Parser

In order to incorporate the new compiled-code kinematic objects into the WALKTHRU's virtual environment, the application `wkadd` was augmented. The `wkadd` application adds interior objects to a WALKTHRU visibility cell database. In particular, `wkadd's` parser is enhanced so that it can parse and create new elements linking scene geometry to compiled code. The new `wkadd` parser functions much like the second pass parser discussed in Section 3. Appendix C:8 contains a reference to the location of a tutorial and notes on the `wkadd` script and the building of the new WALKTHRU/WALKEDIT system using the WALK-KIN extensions.

For each compiled kinematic instance the new `wkadd` application creates a WALKTHRU object with an animation structure. The animation structure is of the new compiled type (`AN_COMPILED_TFORM_ANIM_TYPE`) and contains the linked list of transform elements. Note that a compiled kinematic instance is a UniGrafix instance statement that contains dollar sign expressions that are valid C code statements. During parsing, `wkadd` separates all of the transform elements into static and time-varying components. All static transformation elements are composited together to form the static placement matrix. Since transformations are not in

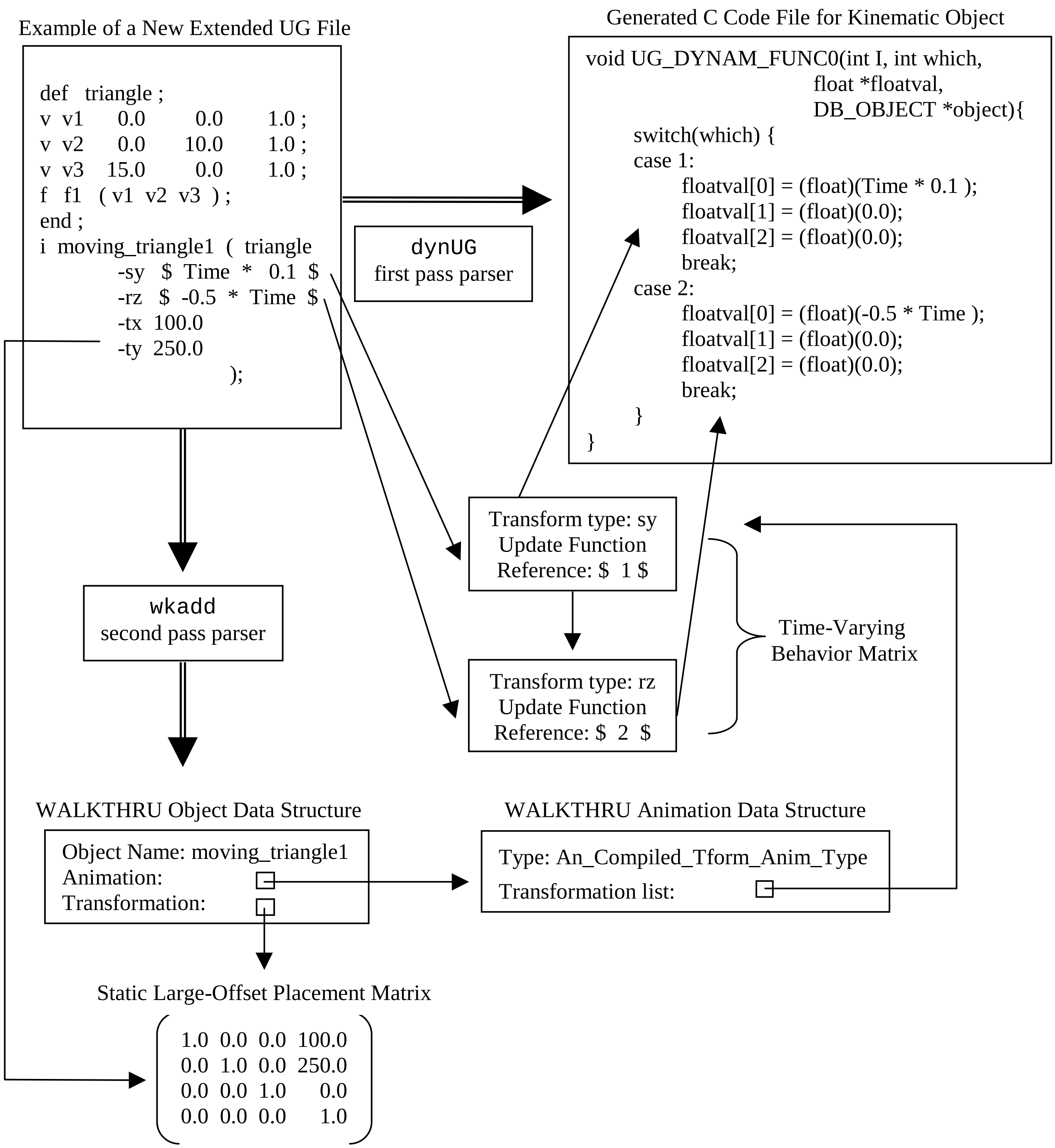

Figure 4.5: The process of creating a new compiled-code kinematic WALKTHRU object using the new `wkadd` and `dynUG` utilities

general permutable, the user needs to include all static transforms that must be done before the dynamic updating inside dollar sign expressions to preserve the original ordering of multiplication. After multiplying the static transforms, the time-varying transform elements are placed inside the linked list of transform elements, which together form the time-dependent behavior matrix.

Finally, the code generated by the first pass parser, `dynUG`, is compiled and linked into the new WALKTHRU / WALKEDIT environment via the animation library `libanim`. The linked list of transform elements in a compiled object differs from the parsed case: in the compiled case expressions are not stored in these linked list elements. Instead of expressions, the `wkadd` parser stores the transform type, the code reference number, and a generic time-varying or slider-dependent updating function. The code reference stored is simply the value of the `which` variable discussed in Section 3.1, and the generic function is a pointer to the generated function created by the `dynUG` application. Figure 4.5 demonstrates the `wkadd` and `dynUG` process of parsing a kinematic object into a compiled WALKTHRU animation object.

4.4 Examples of Kinematic Objects

Here we present a number of examples that use the new compiled kinematic objects. The first group is a re-implementation of the previous BUMP defined WALKTHRU objects, demonstrating that this system is capable of exactly the same behavior as the BUMP system. The later examples demonstrate capabilities which are well beyond those achievable using the simple BUMP expression language script. The third example demonstrates the power of providing an execution context and the use of C as the compiled scripting language. This particular kinematic instance statement modifies object traits not supported by WALKTHRU BUMP objects, such as altering the object's texture and color using the `object` variable provided by the extended execution context.

4.4.1 The Simple Hingebug and Cubogear

In the original model of Soda Hall a number of simple moving sculptures appear throughout the building. As a first test of this new system, we decided to re-implement these objects using the new compiled-code scheme and the new extended UniGrafix parser. The two objects used are the `hingebug` and the `cubogear`; they behave in exactly the same fashion as in the original BUMP implementation. The expressions that drive these kinematic objects are simply the original BUMP expressions re-formulated to be valid C expression syntax. These objects were re-implemented to demonstrate that the new system could express the original implementation's elements and also to show the ease with which original BUMP defined objects could be re-implemented in the new system. The locations to both of these objects are given in Appendix C:2.

4.4.2 A Complicated Pseudo-Physics Simulation (the Gravity Example)

In the original system there are serious limitations caused by the primitive scripting language and its lack of certain constructs, most notably iteration and recursion. The new system can handle more complicated behaviors and simulations. A gravity simulation was built to demonstrate some of the capabilities of the new compiled-code paradigm. The simulation consists of a simple force based computation of gravity between four spherical objects treated as point-masses for the force computations. The spheres interact via the force of gravity and rotate around one another, since they begin the simulation with initial velocities. Finally, a simple repelling force simulating inelastic collision between the objects is used to keep them from getting too close to one another. The location for this example component is given in Appendix C:3.

4.4.3 A Multi-Colored, Multi-Textured Object

The final object discussed here utilizes the new `object` variable and the access it grants to object properties inaccessible through the BUMP system. The execution context provided by the new system allows an object's time-varying transforms to alter its parent object's properties. As a test case for exploring these new capabilities, a new simple object with complicated texture behavior is created. It is a simple cube that displays different textures and colors through time. Since the cube's time-varying behavior does not involve moving geometry, the transform used to control it must return an identity matrix. Thus, we control the cube through a transform statement, whose code alters the surface properties of color and texture as a side-effect, and which does not move the object but instead returns an identity matrix. Later in the report this object is generalized and enhanced with some interaction mechanisms, and also the texture images are enhanced into movies (see Section 5.3.3 for the video viewscreen object). At this stage the object simply cycles through a series of textures while altering the colors used to display them, demonstrating the new capabilities of texture and color modification. See Appendix C:4 for the location of this example object.

5 Interactive Kinematic Objects in the WALKTHRU

Although simple time-varying objects may be useful from a designer's point of view and can provide a more interesting virtual reality experience, they are unable to actively engage the user. A moving sculpture that evolves through time may be interesting to observe, but it is essentially passive and static in content—simply something to be viewed. The user has no control over what the object does, and the object cannot respond to the user. To extend this system and provide for a more active experience we need interactivity.

In order to create interactive objects we need to address two issues:

1) How does the user select an object?

2) Given a selected object, execute the appropriate behavior code.

There exists a wide spectrum of possible solutions, varying in both complexity and functionality. In a complex system the user might open a door by manipulating a virtual model of a hand and creating forces using a data glove to manipulate the door's handle and push the door open. A less complex system might use a pseudo-physical system in which the user's mouse controls a virtual 'stick' through which the user can create forces in the virtual environment. In a simpler system a keyboard stroke or an external menu-based control system might allow a door to be opened. We have chosen an intermediate solution. Our interaction model provides a point and click selection mechanism and event handlers that support a general behavior execution system capable of performing arbitrary computations.

5.1 Overview of the System

The WALK-KIN interaction system operates in the two steps mentioned above. The first step uses a mechanism identical to the WALKEDIT selection system to determine the selected object. In the second step, the new WALK-KIN interaction system invokes the appropriate event handler determined by the mode of operation (i.e., the type of query being performed) and the type of object selected.

The first part of the object selection process operates by generating a `selected object` and a `query type`. From Figure 5.1 we see that event information flows from the WALKEDIT selection mechanism through the WALKTHRU and into the Interaction Event Manager. The `selected object` is determined by projecting the user's mouse coordinates into the virtual environment. The `query type` requires no explicit computation and is directly based on the mode of operation. Both of these pieces of information are combined together to form an object interaction event. The Interaction Event Manager invokes a particular WALK-KIN event handler for each of the different event types. Currently three different event handlers are linked into the system: the *kinematic interactive object* event handler, the *WWW query object* event handler and the original WALKEDIT object manipulation mechanism. The behavior of the event handler itself is dependent on the `selected object` information and other properties of the object structure. The possible behaviors are respectively: execute an object's interaction script, determine and display the URL for the object, determine the owner or occupant associated with the object, or move this object using the WALKEDIT manipulation system. Based on these two pieces of information (`selected object` and `query type`) the WALK-KIN Interaction Event Manager invokes the appropriate event handler.

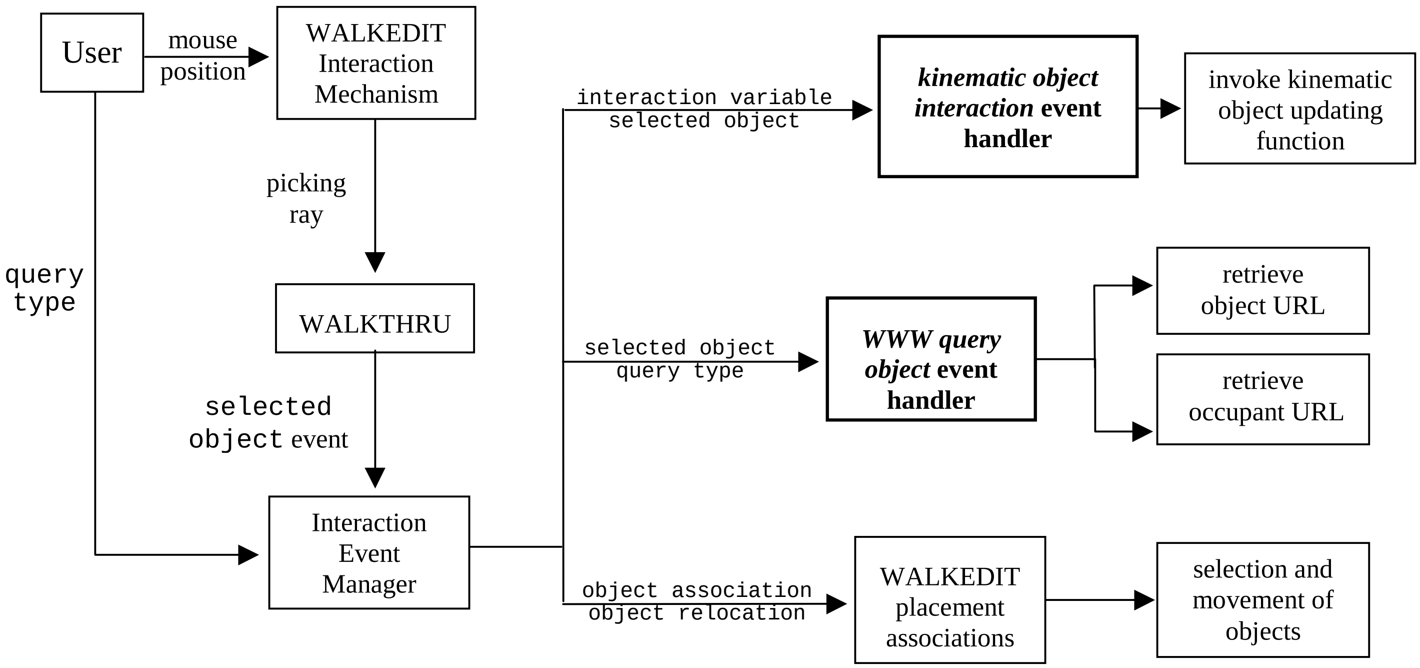

Figure 5.1: Flow of information between the WALK-KIN interaction mechanism and the new WALK-KIN event handlers.

The *kinematic interactive object* event handler provides a link between the interaction mechanism itself and all of the new compiled-code kinematic objects present in the virtual environment. It can serve as a simple ON/OFF switch or it can be used to change various attributes of the object itself. Furthermore, since the behavior script executed is compiled C code, it can access other WALKTHRU objects through shared state or it can provide access to an integrated simulation such as the CFAST fire simulator.

The *WWW query object* event handler allows the user to access external information about various objects by associating these objects with specific URLs. This handler can thus provide links between the WALKTHRU virtual environments and arbitrary Web based databases such as an architectural database or an occupancy database. Finally, the third event handler is simply the original WALKEDIT manipulation package and the related *object association* and *object relocation* functions [4]. The *object associations* link different interior objects together enabling them to be manipulated by the user as a conceptual whole. Similarly, *object relocation* functions define how a user's mouse motions manipulate and move interior objects within the virtual environment.

5.2 Object Selection

A widely accepted way of selecting an object is for the user to point at an object in view and to click the mouse button to select it. The scheme for picking objects in 3D used in the original WALKEDIT [4,5] package operates in exactly this fashion. Picking is performed by casting a ray from the user's eye through the mouse position on the view plane and into the model. This picking ray is then intersected with all of the currently visible objects. The object that is closest to the viewer along this ray is returned as the selected object. This picking scheme relies on two specific functions: `map2d`, an SGI specific function that determines the picking ray, and a ray intersection function that performs the necessary intersections and determines the object closest to the user in the virtual environment. By incorporating the event handlers into the WALKEDIT system as a package, we inherit the WALKEDIT selection mechanism.

5.3 The WALK-KIN Interaction Event Manager

The new WALK-KIN interactive kinematic object package consists of the *interaction event manager* and the related event handlers. We have chosen the WALKEDIT system as the basis for our new WALK-KIN package for three reasons. First, by adding WALK-KIN as an extension to the WALKEDIT system we obtain access to all of its interaction mechanisms and in particular the new kinematic objects obey the *object association* placement system. Next, we are able to use the WALKEDIT interface mechanism and incorporate our elements into its TCL/TK [20] user interface. Finally, by placing all of the compiled-code functions and variables within a package, we can activate, deactivate and alter object controlling variables or the interaction mechanism all while the application is visualizing a virtual environment.

The new package, WALK-KIN, processes selected objects before they are sent to the WALKEDIT event handlers. When an event is received by WALK-KIN, the *interaction event manager* decides whether to call an event handler or to simply pass the elements on to the WALKEDIT interaction system. The decision is based on the type of query (`query type`) the user initiated as well as the kind of object selected (`selected object`). In the case of a static object, the user can either manipulate it via the WALKEDIT mechanism or they can query it using the *WWW query object* mechanism. For time-varying objects they have a third option: to activate it or otherwise interact with the behavior function bound to the object.

The WALK-KIN package is also responsible for passing environment variables to the various handlers. In order to interact with an object, the behavior code needs an interaction variable (`interaction`) and access to the object's data structure (`object`). These two pieces of information allow the updating scheme (discussed in Section 4.2.2) to permit changes in behavior. For example, the interaction variable can act as an on/off switch or otherwise keep track of user interactions. The object pointer enables the behavior code to modify object information other than just the transform values of its instance statement (see Section 4.2.2 for a discussion of the `object` variable).

5.3.1 *WWW Object Query* Event Handler

The ability to link static objects within a virtual environment to other elements of data (e.g., intrinsic properties of the object or related issues) is clearly a desirable feature. For an architectural visualization like the WALKTHRU, linking the model to an architectural database and allowing access from within the virtual environment could prove to be very valuable, since it allows the user to access design issues and concerns directly from within their 3D context. This type of linking naturally augments both the issue database and the virtual environment. We decided to merge the database and the virtual environment by creating links to information on the World Wide Web (WWW) from within the original WALKTHRU application.

Providing such a link raises two problems. First, how can we integrate a Web browser into the WALKTHRU framework? And second, how can we translate object and occupancy information into relevant URLs to be loaded by the browser? The two browsers that we chose to support were Netscape Navigator™ and NCSA Mosaic. The mechanisms used to control the browsers differ, so separate routines were created to interact with each of them.

In the case of Netscape's Navigator™, the application provides an explicit remote control mechanism. We use this mechanism to link the WALKTHRU with the Web browser.[4] A facility to interface external applications with a currently running Navigator™ browser is provided through the use of the command:

```
netscape -raise -remote 'openURL( URL-STRING )'
```

where the string *`URL-STRING`* specifies the URL to be loaded. This command will fail with a non-zero status value if a browser is not running on the system. In this case, a new browser is invoked with the command:

```
netscape 'openURL( URL-STRING )'
```

In either case, the command is issued by the *WWW query object* event handling routines using the C `system()` function, which issues a shell command from within a C program. The remote control process for the Navigator™ browser is illustrated in Figure 5.2.

For Mosaic, a slightly more complicated technique is required, since a simple mechanism for externally controlling the application is not provided.[5] Mosaic contains a special interrupt signal handler that provides access to its remote control features. If the process ID (`pid`) of the browser is known, an application can send the signal `SIGUSR1`. When Mosaic receives this signal it

---

[4] The relevant information is located at `http://home.ntescape.com/newsref/std/x-remote.html`

[5] Information for manipulating Mosaic can be found at `http://www.gene.cinvestav.mx/remote-control.html`

executes the commands contained in the file `/tmp/Mosaic.pid` on the machine with the browser running locally. In our case the text file consists of two lines: the first contains the keyword `goto` and the second contains `file://URL-STRING`, where *`URL-STRING`* represents the URL to be loaded and displayed. The *WWW query object* event handler starts a new Mosaic process when the first query object event occurs in order to obtain the browser's `pid`. Subsequent object query events simply store a new URL in the text file `/tmp/Mosaic.pid`, and then send the `SIGUSR1` signal to the browser. The signaling and file input/output process for the Mosaic browser is demonstrated in Figure 5.3.

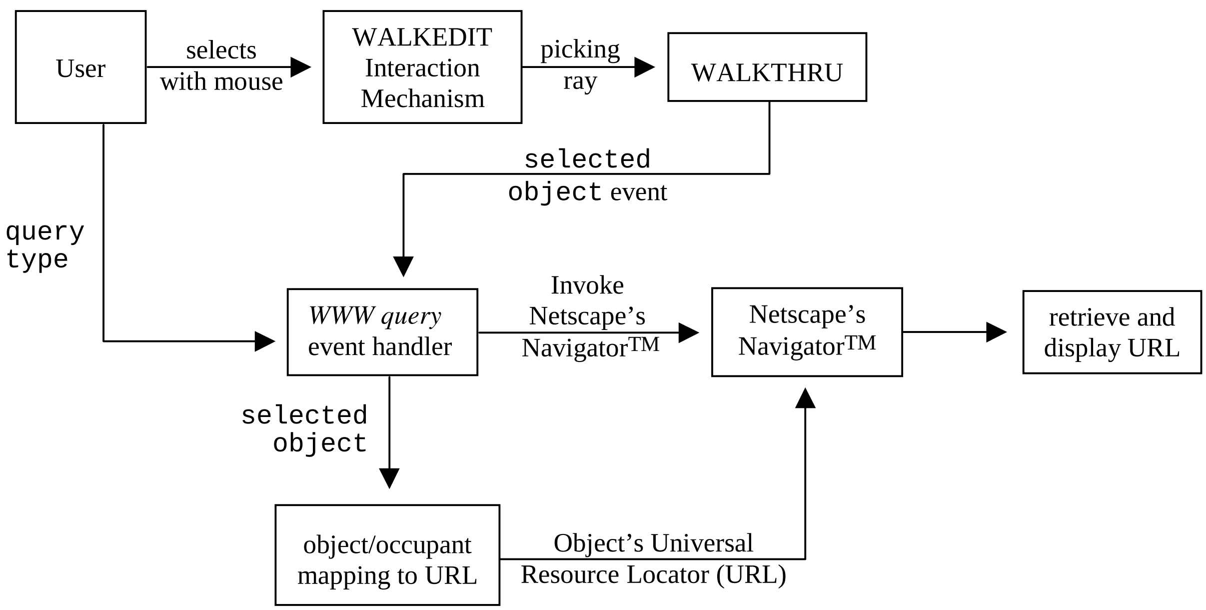

Figure 5.2: The *WWW query object* event handler controlling Netscape's Navigator™

The *WWW query object* event handler's final task is to determine the URL to present. To date we have linked two types of information for users to access by performing a query: object data about content items and occupancy information about Soda Hall. When the user selects an object to request object data, the event handler identifies the class of the object. A table is used to map object class names to URLs. In our simple implementation, all of the different classes of objects present in the Soda Hall model have been entered in the table. Corresponding HTML pages for each object have also been created. The HTML pages act as placeholders for useful information about the different object classes and are a temporary proof of concept for the implementation. It is certainly possible to have the event handling mechanism generate arbitrary query strings for either a Web-based database system or some other kind of database.

In a similar fashion, when the user queries an object for occupancy information, the name of the object is parsed and a room number corresponding to its location is generated. In the current WALKTHRU database of Soda Hall, each object stores its office room or corridor number as part of its name. This is an artifact of the method used to define the building, and of the instancing hierarchy of the UniGrafix language. We exploit this property by using an object's name to derive its office number. A table is used to map office numbers to occupancy information by processing a building roster of Soda Hall. This roster associates a person with a given office number. The HTML page displayed for the occupancy query includes an occupant's name and personal homepage.

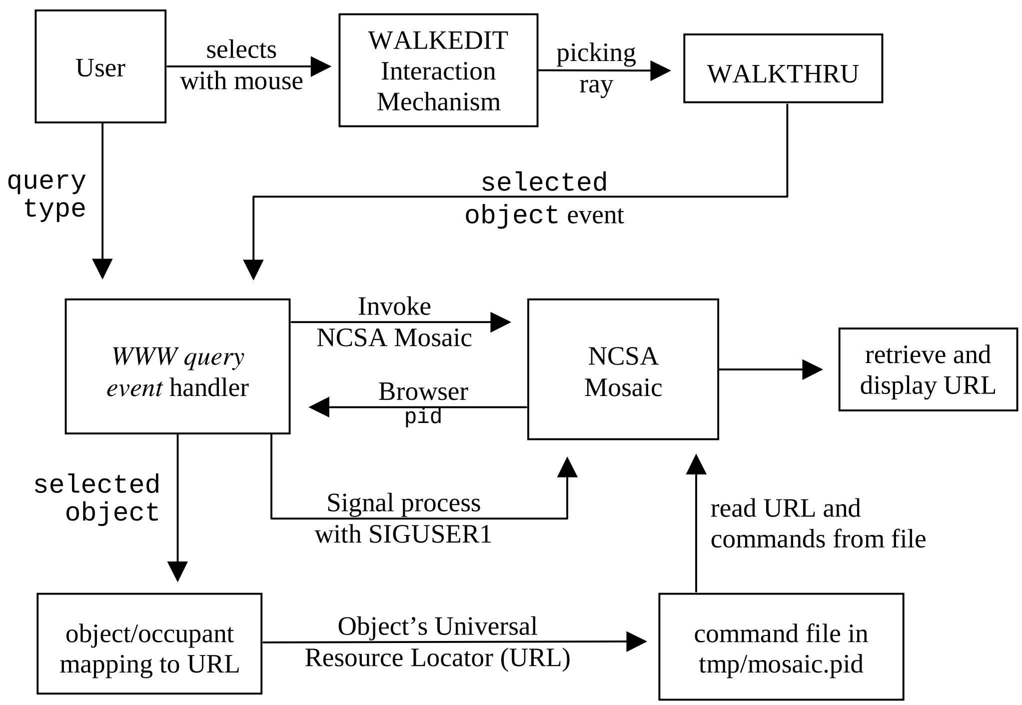

Figure 5.3: The WWW query event handler communicating with NCSA Mosaic

5.3.2 Kinematic Object Interaction Event Handler

The *kinematic object interaction* event handler is the main mechanism enabling kinematic interactive objects in the WALKTHRU. The event handler takes a selected object and finds its associated updating code. This function is executed within an interaction context that enables state-based objects (those objects defined with local variables) to be defined. Finally, the updating function alters the state of the object and its properties as well as the transformation matrix positioning the object. The context permits an interactive behavior script to alter an object's color and texture properties and the matrix manipulation allows objects to be moved.

The interaction event handler for kinematic objects behaves in a similar manner to the *WWW query object* event handler. For this handler, the data is a pre-compiled update function rather than the *WWW query object*'s simple URL. Kinematic objects in the WALKTHRU function by means of the generic updating functions that evaluate their animation structures and return a matrix value (see Section 4 for details). Aside from the issue of what is done with this matrix and what it represents (which is the focus of Sections 4.2.2 and 4.3), we must also consider when it gets invoked.

Ordinarily, a kinematic object's updating function is called just before it is drawn in order to update its geometric position. This is the case for BUMP time-varying objects as well as the new compiled variant. In both cases, a function (`ANEvaluateAnimationTransformation`) is called on the kinematic object. This function is able to evaluate both parsed and compiled kinematic objects and it returns the time-dependent behavior matrix (discussed in detail in Sections 4.2.2 and 4.3). To implement interactivity, we need to call this function whenever the user selects the object.

However, because of the very presence of the new interaction scheme and its out of sequence updating calls, we cannot simply call the updating function. Consider a kinematic object that is not interactive: under this scheme when the user selects it, the object will be updated one additional time during that frame. Thus, the object's internal state would be off, and some of the updates would not have been displayed; for a state-based object that cycles through a series of different visual states, some of these may not be displayed because of this updating behavior. In addition, there would be no way to use the selection mechanism to pass different information to an object. Since interactive objects need to respond to the user and may need to change their internal state, e.g., turn on when selected, they must have two different types of updates: a generic draw update and an interactive selection update.

The WALK-KIN event handling mechanism solves this problem by passing information to the object's updating code. Two different pieces of data are passed into the function providing an execution context: one is an interaction variable and the other is a pointer to the object (this `object` pointer is discussed at length in Section 4.2.2). The animation evaluation functions are modified to add these two additional elements of data and the `dynUG` application is altered accordingly.

The `dynUG` application has been augmented so that the generic C code function `UG_DYNAM_FUNC0` has two additional arguments: `interaction` for the interaction variable and `object` for the parent object's pointer. The value for `interaction` is true only if the object has just been selected and it is false during all other updates. For a particular element of transform code, the value of `object` is a pointer to the defining instance's object data structure. Providing these elements in the generated C code enables any dollar sign expression to reference these two variables. Thus, a simple piece of C code appearing in a transform's updating expression[6] can implement an ON/OFF toggle switch by using the `interaction` variable. Figure 5.4 depicts the interaction process and the various cases of execution for the updating code.

Given this interaction variable, an object can have two different code elements: one executes if

---

[6] For example, a simple interaction mechanism controlling an on/off state variable might be: `if (interaction) {on = !(on) }`

the object is selected and the other executes during a normal update. Since the behavior function executed is stored as its own C code, it can have local state as in the `on` variable example[6], or it can have other more complicated state-based behavior. Finally, objects can be coupled through their C code state and thus their behaviors can affect one another. An example of this coupling is the juggling sculpture (see Appendix C:5 for source location) which consists of a planar rectangular object and three spheres. The picture plane object is interactive and selecting this component activates the three spheres by modifying a shared on/off state variable.

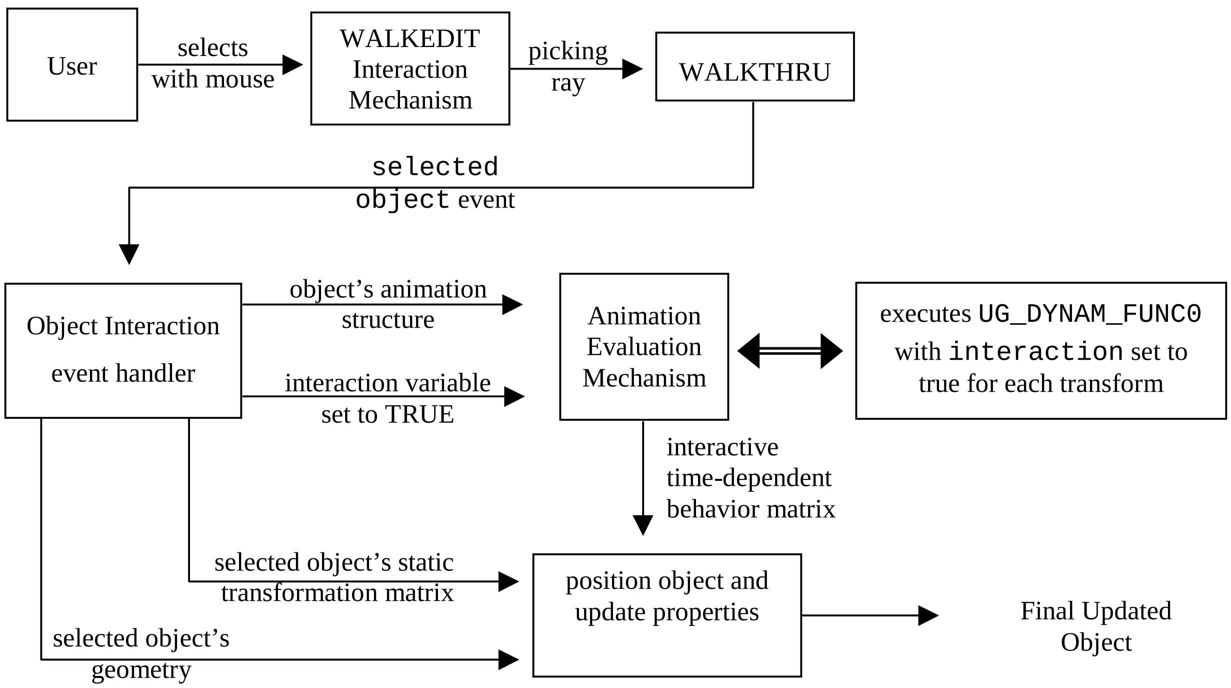

Figure 5.4: Flow of information between the Kinematic Object Interaction event handler and the WALKTHRU system

5.4 WALKEDIT and *Object Associations*

In order to demonstrate the power of using both the new interaction event handlers and the original WALKEDIT object association package, we decided to define new elements that could be used to control the interactive manipulation of objects. One behavior that was deemed necessary and interesting was the creation of a hinge. In this implementation, the hinge is not an object in the virtual environment but rather it is both an *object association* and an *object relocation* procedure built into the WALKEDIT framework. The hinge is a basic component of many articulated objects and is particularly useful in building elements to represent operational doors.

The hinge mechanism functions by grouping two objects as well as defining a constraint on the types of motion between these two objects. The hinge's *object association* creates a conceptual pair containing the 'anchored' parent object, that provides the reference frame, and the 'hinged' child element that can move with respect to the parent. The hinge's *object relocation* procedure constrains the user's interaction along a line perpendicular to the hinge axis, creating a natural hinge type of motion for the 'hinged' element. The other 'anchored' element is not constrained in this fashion and may have other *object associations* and relocations applied to it.

In the current implementation, the *object relocation* procedure is generic and three slight variants exist. The *relocation procedure* can be used for many different objects and is generic in the sense that it performs a local constrained rotation around an object's local axis. Currently, the three variants simply apply the rotation around a different principle axis within the objects local coordinate system and are named: `On_Hinge_X`, `On_Hinge_Y`, and `On_Hinge_Z`, with the last letter denoting the principle axis of rotation. The *object relocation* procedure transforms the user's mouse movement events into a rotation about the hinge axis. Currently, no alignment of the 'hinge edge' with edges on the associated object is performed and thus care must be taken when defining and placing the object. For example, the `On_Hinge_X` relocation procedure uses the `X`-axis of the object's master coordinate system as the 'hinge edge' and so user mouse motions rotate the object around this edge. Thus for a door to appear correctly, it must be defined so that its 'hinge edge' is this relocation axis; and it must also be placed in the virtual environment so that its 'hinge edge' is coincident with the corresponding 'hinge edge' of the associated 'anchor' object.

The *object associations* are fixed on a per object basis and are thus not generic and currently take an arbitrary object and associate it with the closest named object of the instance type defined in the association itself. For example, an `On_DoorFrame` association takes any object and associates it with the closest 'doorframe' object in the virtual environment, while the `On_Seat` association operates in the same fashion with the closest 'seat' object.

5.4.1 Example of an Interactive Office Chair Created Using *Object Associations*

In order to test the hinge association and determine the usefulness of using *object associations* to define interaction, a simple office chair was constructed. The chair exhibits a multitude of different behaviors and demonstrates the value of using associations to define interaction paradigms for both static and kinematic objects. The associations and interaction mechanisms

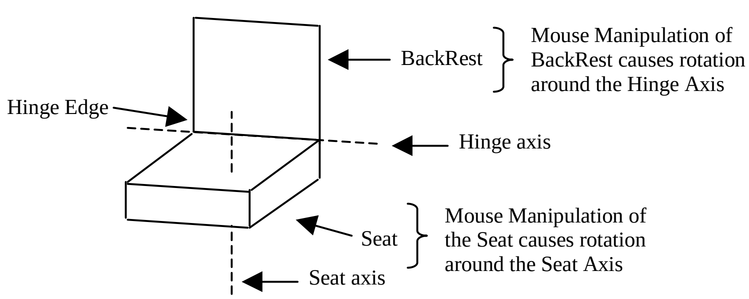

Figure 5.5: Depiction of Chair object and its components. Rotation axes are shown as dashed lines.

for the virtual chair are modeled on an office chair that has a movable seat and pivoting backrest. In particular, the back of the chair can be manipulated about a hinge connecting it to the seat and the seat can be rotated around the base column. Finally, the base of the chair can be manipulated around the virtual space: turned, pushed and even stacked with the other two components following suit as a coherent whole. Figure 5.5 demonstrates the new components of the chair object and their corresponding rotation axes.

The chair object is implemented through three separate *object associations* and *object relocations*. The back is associated with the seat via the `On_Seat` association, which causes the back to follow the movements of the 'seat' object rigidly. The `On_Hinge_X` procedure relocates the back relative to the hinge allowing it to be manipulated as if a real hinge existed between it and the associated 'seat' object. In addition, the 'seat' object itself is constrained by the `On_Horizontal_No_Trans` association and the `Pseudo_Gravity_On_Center` relocation. The association moves the seat object to rest on a horizontal surface below it, while the `Pseudo_Gravity_On_Center` relocation causes the seat to drop directly down as if by gravity. Finally, the base of the chair is constrained to remain on the floor and obey the basic pseudo-gravity behavior through the `Pseudo_Gravity` relocation and the `On_Horizontal` association. The difference between these two sets of associations and relocations is that in the seat case they were modified to disable translation and only allow rotation around the central axis, otherwise these two sets (`Psuedo_Gravity/On_Center` and `On_Horizontal/No_Trans`) are identical.[7] In the current implementation the hinge motion is achieved by taking the x motion of the mouse and mapping it as the rotation about the hinge axis, we hope to extend this to be a projected mapping of the x,y mouse motion perpendicular to the hinge axis in the near future.

5.4.2 Example of an Interactive Door Created Using *Object Associations*

The hinged door was built in order to integrate the notions of interactive objects with the simulation paradigm developed in the WALKTHRU-CFAST project. It uses the generality of the hinge relocation procedure and a slightly modified object association to associate a 'door' object with the 'doorframe' object. The 'door' has the `On_DoorFrame` *object association* and thus it is paired with the closest 'doorframe' object in the virtual environment. Furthermore, the 'door' object relocates according to the `On_Hinge_Z` *object relocation* so that the user's mouse interaction rotates the 'door' around its `Z`-axis, creating the required motion for opening and closing. Finally, the 'doorframe' object acts as a regular piece of furniture using the `On_Horizontal` association to remain on horizontal surfaces and the `Pseudo_Gravity` relocation to act as a 'real' object by falling to horizontal surfaces.

5.5 Examples of Interactive Kinematic Objects

Given all of these mechanisms, our final task was to create a variety of interesting example objects and place these within the WALKTHRU environment. In the following sub-sections we

---

[7] The modifications of the standard associations and relocations were performed by Richard Bukowski.

focus on the combination of interactive elements and kinematics. In particular, we will discuss kinematic objects that the user can activate or deactivate using the interaction mechanism discussed in Section 5.3.

5.5.1 Interactive Moving Sculptures

As a first exploration into these new interactive kinematic objects, we augmented the existing models, `hingebug` and `cubogear`, with the *kinematic object interaction* mechanism. The originals are simply self-contained, moving objects that are always on and evolving through their cyclical behavior. We have changed these sculptures, so that the pedestals act as ON/OFF switches. When the object is activated it will begin its cyclical motion, and when it is deactivated the object will remain in its new position.

The mechanisms used to build these two objects follow directly from Section 5.3.2. The time-varying expressions behave in the same manner as with the BUMP system, but have now been converted into the new compiled-code system as C code expressions. The pedestal object acts as an interactive kinematic mechanism for storing the sculptures internal state. The sculptures are no longer driven by the time variable `t`, but instead have a private notion of time. In these two example sculptures, the variables are `hingetime` and `cubetime` respectively. The private time variable controls all of the kinematic expressions responsible for the movement of a particular sculpture. The locations for the demo files of the `hingebug` and `cubogear` complete with pedestals can be found in Appendix C:6.

5.5.2 A Video Playback Viewscreen

As another example of an interactive kinematic object, we present a video playback called `viewscreen`. We developed this object to demonstrate the combination of all three new elements of our system: 1.) a compiled-code system allowing powerful operations such as texture definition and binding, 2.) an interaction mechanism capable of activating objects, and 3.) an environment granting access to parent object's properties through the `object` variable. In contrast to the previously mentioned simple sculpture objects, the video `viewscreen` does not use time-varying transformations to modify its geometry; rather the texture bound to the object itself is modified by the updating code. Using this mechanism, we are able to achieve real-time video playback as a time-varying texture map displayed on a polygon in the virtual environment. Video playback represents a more interesting example of interactive kinematic objects, since this object's content varies over both time and user input. The presence of the evaluation context makes this system general so that any object could be modified to play video on its surface. The current implementation is limited to altering the texture at the object level and does not allow polygons of an object to be textured differently.

The video playback system works by taking a series of image files in the SGI `rgb` file format representing consecutive frames of a video and texture mapping them onto the polygons of an object. The texture maps are cycled in frame order to achieve the effect of video playback. Our system provides real-time performance on a RealityEngine™ II [1] dual 150 MHz processor machine by pre-loading the textures into memory and updating the currently bound texture using GL-specific library commands [16]. The pre-loading of textures is required to support real-time playback, although subsequent work demonstrated that a playback rate of ten frames per second is possible without pre-loading (this is described in more detail below). There are a number of problems with pre-loading textures, the most significant of which is the limited amount of texture memory, and therefore the inability to provide real-time playback of arbitrary length video. Given our current configuration of 4Mb of texture memory and 128 Mb of main memory, we can only pre-load approximately 700 frames of video (where each frame is a 256 x 128 x 3 byte color `rgb` image). This is insufficient for even a small video segment. Playback normally runs at between 24 and 30 frames per second, so a limit of 700 frames supports at most 30 seconds of video stored in memory. What we ultimately want to provide is the ability to display compressed video that is stored at a distant location on the World Wide Web. This would allow video objects to mirror the static URL work discussed in Section 5.3.1. Given such a system, the user could bind video clips on the Web to a `viewscreen` in the virtual environment, and have acceptable frame rates after a partial download of the source video from the remote site.

The video playback system has currently been incorporated into a `viewscreen` interactive kinematic object. The object's C code encapsulates the salient features of the video playback mechanisms and performs the loading and binding as a pre-processing task in the WALKTHRU using the WALK-KIN extension. The interactive binding and real-time playback is achieved through an interactive identity transform that accesses its parent object's data structure (see Section 4.4.3 for details). Instead of manipulating a transform matrix, this identity transform alters the `viewscreen's` texture pointer to point to the next frame of video. If the local on/off state variable is 'on', playback proceeds by fetching the next frame of video and binding it to the `viewscreen's` object definition. The location for example code and demo UniGrafix files for this object are in Appendix C:7.

In order to further investigate the texture pre-loading requirement, we built another test system on a newer version of the SGI operating system IRIX version 5.0.3. Doing this allowed us to use new GL texture-mapping capabilities specifically designed for the Reality Engine II architecture. Our new system does not pre-load the textures into memory. Instead, it waits until a request is made and then loads and binds textures during the display of the video. This removes both the initial overhead of pre-loading the images and reduces the texture memory requirements to only those needed for the currently bound textures. We implemented all of this using new versions of the standard GL texture mapping functions (`tex-def-2d` and `tex-bind-2d` with the `FAST-DEFINITION` flag set). This technique provides a playback rate of 10 frames per second. If the current WALKTHRU system utilized these new features, video playback could occur from a remote site on the Web after only a partial download. Video on the Web is generally stored in a compressed format, usually MPEG [18]. We believe that uncompressing the video will not have a significant negative impact on the frame rate, since efficient software solutions exist to perform this operation. For example, the decompression of MPEG video generally runs on a midrange machine at approximately 20 frames per second for frames larger than the ones we currently display (320 x 240). The marginal increase in cost per frame to support compressed video will probably not effect our original performance estimate of ten frames per second.

The use of SGI's new fast-defined textures imposes one additional restriction on our system: the texture sizes must be powers of two in both dimensions. The image frames of our video must be modified to match this requirement. If the images are not powers of two in both dimensions, image scaling can be applied. This technique can cause significant degradation in the video frame quality, and in addition the actual operation of texture mapping video onto a polygon causes a loss of image quality.

6 Discussion and Future Work

During the development and implementation of the WALK-KIN extension two major design decisions were made. First, we analyzed a series of different options for enhancing the existing scripting mechanism and incorporated a new scripting system into the WALKTHRU. Second, during the integration and alteration of the WALKTHRU we considered using a different scene description language and in the process altering the underlying graphics database structure, but we decided that this change was not prudent at this time.

In the initial phases of this work, we considered a number of different languages for the scripting mechanism used to define kinematic objects. The languages Java™, C++, C, LISP, VRML script, and a compiled variant of the original BUMP system were all examined. At the beginning of the project performance was considered most important, which translated directly to the choice of a compiled rather than a parsed scripting system. Once a compiled system had been adopted and the speed of script execution was no longer a primary concern, the need to easily express complicated behavior became important. An ideal scripting system addressing both the performance and expressiblity issues would be a variant of the Java™ language. Java™ supports just-in-time-compilation, is platform independent, offers the dynamic alteration convenience of a parsed/interpreted language, and has the speed of a compiled system. For the WALK-KIN implementation, however, the actual language chosen was C, since Java™ did not emerge as a useful finished product until after C had been incorporated into the scripting system. The next logical step for future work would be to incorporate a just-in-time-compiled variant of the Java™ language into the system. If such a major modification is not practical, then the current scripting system should be augmented through dynamic linking (as discussed in Section 3.2), since this will improve the interactivity of the system as well as the authoring environment.

A number of possible choices were considered for this project's graphics platform and the underlying database. For the past 10 years, we have utilized SGI's graphics system and the Berkeley UniGrafix language as the underlying data structure and scene graph hierarchy. Periodically we have been considering switching to an externally supported system. The current WALKTHRU system is not portable outside of the SGI workstation domain at Berkeley, since it is written in GL and relies on UniGrafix as its scene description language. Other potential systems we considered were OpenGL, OpenInventor, VRML 1.0/2.0, or the new Java3d system. The last three of these systems support interaction elements and have a scripting system already in place. The presence of a scripting and interaction system would naturally effect the choices made above regarding the scripting mechanism employed by our new system. For example if we chose VRML 2.0, then we could use Java™ as the scripting mechanism and also use VRML's behavior defining elements and interaction devices to create interactive kinematic objects.

Moving from UniGrafix to one of these open standards would necessitate substantial modifications to all of the WALKTHRU's libraries as well as requiring a fundamental change in the WALKTHRU's database structure. The new database structure would need a whole new set of elements to support the interaction and kinematic constructs present in any of these graphics description languages. Finally, none of the three standards outlined above is a clear winner in this arena, and therefore we should wait at least until Java3d is well established before considering which, if any, of these systems we want to ultimately adopt. For these reasons the current system has remained faithful to the Berkeley UniGrafix language and the corresponding original WALKTHRU/WALKEDIT database.

In the current implementation of WALK-KIN a series of useful elements were omitted due to time constraints. These elements would help to create a more author friendly environment for the creation of interactive kinematic objects. The omitted elements include the ability to reference arbitrary UniGrafix statement values and the BUMP package's motion definition constructs. The ability to reference geometry values from outside of their normal scope was addressed by other UniGrafix tools and would provide a valuable addition to the authoring environment. Such referencing should allow dollar sign expressions to reference any statement values deep down in the scene-hierarchy making any sort of link between object behaviors transparent in the UniGrafix file. Currently, such linking can be achieved only through included C files. This obscures these relationships because it requires careful reading of the geometry description and all of the related C script files. The original BUMP system had the ability to define a variety of pre-defined motion prototypes. Among the BUMP elements that have not yet been re-implemented are the defined motion statements (`mdef` and `mapply`) as well as the presence of spline-based motion paths. Similarly, a more general offset path behavior that made use of the new kinematic elements could also be defined and implemented in a future version of the language. The addition of the extended referencing capabilities and/or the original BUMP constructs would both require modifying the UniGrafix parser.

One of the new extensions of the WALK-KIN package is the ability to create an interactive kinematic object capable of controlling a simulation. An interactive kinematic door in the context of the CFAST fire simulator would allow the user to alter the flow of heat and smoke between two rooms, and thereby affect the evolution of the fire simulation. Currently, our interactive door object has no link to the fire simulation, since the simulation itself does not provide the appropriate programming interface to allow dynamic updating of the simulation space in a dynamic manner. At the completion of this project, the best approach to providing this kind of control over the evolution of a fire would be to save the state of the entire simulation when the door was manipulated, and then restart the simulation using this saved state on a new cell/portal layout with the door's portal modified. Clearly this is neither a fast nor a clean implementation of the link between interactive kinematic objects and the WALKTHRU-CFAST simulation system. What we really need is a programming interface for the CFAST simulation that allows the simulation to be dynamically altered during execution. We understand that some work at NIST is aimed in this direction.

7 Conclusions

The WALK-KIN extension provides a framework for defining and scripting interactive kinematic objects for a WALKTHRU virtual environment. Toward this end, we implemented and evaluated three complementary mechanisms. First, the existing scripting mechanism was augmented to improve its expressiblity and performance. Second this new scripting mechanism was integrated into the WALKTHRU/WALKEDIT environment, providing the framework necessary for creating interactive kinematic objects. Third, we implemented a new interaction event manager using the WALKEDIT selection mechanism and two event handlers. One of the event handlers controls user interaction with the kinematic objects, and the other is used for linking external information with the virtual model.

For the first task, we analyzed the original UniGrafix language and the BUMP scripting system. We then incorporated the GLIDE compiled-code paradigm into the existing UniGrafix language. We evaluated our new system by comparing it to the original BUMP system and measured the various performance gains made. These comparison measurements showed that our new compiled-code system yields impressive speedups relative to the old parsed system. In all of the example scenes tested, the new system executes between 3 and 10 times faster than the original. Furthermore, in all but one of the examples, the complexity of the graphics, and not the execution of updating code, limits the frame rate and the overall performance of the animated object.

The second task was accomplished by combining the new compiled-code paradigm with the WALKTHRU/WALKEDIT virtual environment and by creating a framework for defining objects using this new scripting mechanism. The new WALK-KIN paradigm enables new types of kinematic objects that are more complex and whose update functions execute faster. In addition, these new objects have been fully integrated into the WALKEDIT object manipulation system and can be *associated* and *relocated* within the virtual environment just like other static content objects. Finally, the `wkadd` application was modified so that these new compiled-code kinematic objects can be incorporated into the WALKTHRU database of a virtual environment.

Our third extension was the addition of an interaction event mechanism and two new interaction event handlers for the WALKTHRU/WALKEDIT system. Using the first event handler, objects can be activated/deactivated and in general may respond to the user in a variety of complex ways. The system’s object modification capabilities have been enhanced to provide interactive time-varying manipulation of color and texture properties. All of these elements have been combined together in a number of interactive kinematic example objects. A second interaction event handler provides a link between content objects and the World Wide Web through a query retrieval and Web browser display event handler.

A combination of all of these mechanisms can be used to create complex virtual objects. The new scripting mechanism provides increased expressiblity and improved performance. In addition, the interaction mechanisms and link to WALKEDIT creates a more usable and interesting environment. Finally, we believe that all of these elements used in conjunction create a more compelling and realistic virtual experience for the user.

Acknowledgements

This work was completed under the direction of my advisor Carlo Séquin, whose contributions and guidance were instrumental in finishing this project. In addition, I want to thank Prof. Landay for being my second reader. This project was built upon the foundation of the WALKTHRU created by Seth Teller and Tom Funkhouser, and the WALKEDIT system created by Richard Bukowski. Additional elements of the UGMovie application, implemented by Gregory Couch, and the BUMP package, created by Steven Oakland, were used for performance analysis and prototyping.

Aside from the direct contributors, I would like to thank all of the members of the Berkeley Graphics group. In particular I would like to thank Maryann Simmons, who from the beginning of my graduate career has always found time to help me, and in particular for reading and editing parts of this final report, and for specific technical help. Also Ketan Mayer-Patel, Christopher Healey, and Christina Tempelaar-Lietz deserve special attention for other related technical support. In addition, Rick Bukowksi and Laura Downs deserve thanks for all of their WALKTHRU/WALKEDIT related guidance and assistance. In general I would also like to thank other members of the graphics group current and past: Sara McMains, Rick Lewis, Mark Brunkhart, Ajay Sreekanth, Amy Hsu, and Jordan Smith.

During work on this project I was supported by the National Science Foundation grant number: CDA 94-01156 and the project itself was partially supported by NEC and by the ONR MURI grant N00014-96-1-12200.

Finally, I would like to dedicate this project to my immediate family for providing me with intellectual support and especially to Jack Keller, whose vision and imagination prompted me to go into science and engineering.

References

[1] Akeley, Kurt. RealityEngine Graphics. *Proceedings of SIGGRAPH 93 (Anaheim, California, August 1-6, 1993).* In *Computer Graphics* Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 109-116.

[2] Ames, A., Nadeau, D.R., and John L. Moreland. *The VRML Sourcebook*. New York. Wiley, 1996.

[3] Bukowski, R. and Séquin, C. Interactive Simulation of Fire in Virtual Building Environments. *Proceedings of SIGGRAPH 97* (Los Angeles, California, August 3-8, 1997). In *Computer Graphics* Proceedings, 1997, ACM SIGGRAPH, pp. 35-44

[4] Bukowksi, R.W. and Séquin, C.H. Object Associations: A Simple and Practical Approach to Virtual 3D Manipulation. *Proceedings of the 1995 Symposium on Interactive 3D Graphics* (Monterey, CA, April, 1995), pp. 131-138.

[5] Bukowksi, R.W. *The WALKTHRU Editor: Towards Realistic and Effective Interaction with Virtual Building Environments*. Master's Thesis, Computer Science Division (EECS), Technical Report UCB/CSD-95-886, University of California, Berkeley November 1995.

[6] Couch, Gregory S. Berkeley UNIGRAFIX 3.1 and the Ugmovie animation previewer. Technical Report, UCB/CSD-94-831, University of California, Berkeley, September 1994.

[7] Couch, Gregory S. Berkeley UNIGRAFIX 3.1-data structure and language. Technical Report UCB/CSD-94-830, University of California, Berkeley, September 1994.

[8] *dlopen* manual pages, In *IRIX Programmer's Reference Manual*, Release 5.3. On-Line Documentation. Mountain View, CA. Silicon Graphics, Inc.

[9] *dlsym* manual pages, In *IRIX Programmer's Reference Manual*, Release 5.3. On-Line Documentation. Mountain View, CA. Silicon Graphics, Inc.

[10] Ferguson, Paula M. *Motif Reference Manual*. Sebastopol, CA. O'Reilly and Associates, 1993.

[11] Funkhouser, T.A. and Séquin, C.H. Adaptive Display Algorithm for Interactive Frame Rates during Visualization of Complex Virtual Environments. *Proc. of SIGGRAPH '93* (Anaheim, CA, Aug. 1993), pp. 247-254.

[12] Funkhouser, Thomas A. *Database and Display Algorithms for Interactive Visualizations of Architectural Models*. Ph.D. Thesis, Computer Science Division (EECS), University of California, Berkeley 1993.

[13] Geis K., and L. Pereira, "CS199 Final Report: GLIDE, Version 2.0", Spring 1995.

[14] *glprof* manual pages, In *IRIX Programmer's Reference Manual*, Release 4.0.5. On-Line Documentation. Mountain View, CA. Silicon Graphics, Inc. May 1992.

[15] Grand, Mark. *Java Language Reference*. Cambridge, O'Reilly, 1997.

[16] *Graphics Library Programming Guide*, Document Version 2.0. Document Number 007-1210-030. Mountain View, CA. Silicon Graphics, Inc. 1990.

[17] Hartman, J. and Wernecke. *The VRML 2.0 Handbook: building moving worlds on the web. Reading, Mass*. Addison-Wesley, 1996.

[18] Mitchell, Joan L. et al. *MPEG video: compression standard*. New York, Chapman and Hall, 1996.

[19] Oakland, Steven Anders. *BUMP, a motion description and animation package*. Master's Thesis, University of California at Berkeley, 1987.

[20] Ousterhout, John K. *Tcl and the Tk toolkit*. Reading, Mass. Addison-Wesley, 1994.

[21] Peacock, R.D., Forney, G.P., Reneke, P. et al. CFAST, the Consolidated Model of Fire Growth and Smoke Transport. NIST technical note 1299, U.S. Department of Commerce, Feb. 1993.

[22] *pixie* manual pages, In *IRIX Programmer's Reference Manual*, Release 4.0.5. On-Line Documentation. Mountain View, CA. Silicon Graphics, Inc. May 1992.

[23] *prof* manual pages, In *IRIX Programmer's Reference Manual*, Release 4.0.5. On-Line Documentation. Mountain View, CA. Silicon Graphics, Inc. May 1992.

[24] Séquin, C. H., "SDL: Scene Description Language", Lecture Notes, cs184, Fall 1991.

[25] Slater, M., Usoh, M., Chrysanthou, Y. The Influence of Dynamic Shadows on Presence in Immersive Virtual Environments. in *Virtual Environments '95*. Selected papers of the *Euographics Workshops* 93 and 95. 1995. p. 8-21

[26] Slater, M., A. Steed and M. Usoh. Steps and Ladders in Virtual Reality. in *ACM Virtual Reality Science and Technology (VRST)*, eds. G. Singh and D. Thalmann, World Scientific. 1994 p. 45-54.

[27] Teller, S.J. and Séquin, C.H. Visibility Preprocessing for Interactive Walkthroughs. *Proc. of SIGGRAPH '91* (Las Vegas, Nevada, Jul. 28-Aug. 2, 1991). In *Computer Graphics*, 25, 4 (Jul. 1991), pp. 61-69.

[28] Teller, Seth J. *Visibility Computations in Densely Occluded Polyhedral Environments*. Ph.D. Thesis, Computer Science Division (EECS), University of California, Berkeley, 1992.

[29] Wernecke, John. *The Inventor Mentor*. Reading, Mass. Addison-Wesley, 1994.

[30] Zeltzer, D. (1992a) Autonomy, Interaction and Presence. *Presence: Teleoperators and Virtual Environments*, 1 (1), pp. 127-132.

Appendix A: New Syntax for Time-Varying UniGrafix

This appendix describes the new UniGrafix syntax developed in this report for time-varying objects using the compiled-code paradigm for parsing, code-generation and updating.

All of the original BUMP extension syntax is removed from the UG framework. The syntax below details this improvement with the major gain being the use of C for the dollar sign expressions. The basic change from the BUMP system extensions to the new format is the content of a dollar sign expression, which now must be a valid syntax C expression or function call. It is important to note that the first pass parser, `dynUG`, will not check the correctness of the expression C code. In addition, dollar sign expressions are currently valid only in vertex statements, color statements, and the transform elements of instance statements.

The use of any of the `(meval…)` type commands in their original form: `vardef`, `meval`, `mexecute`, `mapply`, `move`, and `mdef` is no longer permitted. The actual `meval` and `vardef` constructs still exist in the language and can be used with the new C script functionality. In addition, the functionality of `mexecute` can also be simulated using the new expression syntax by creating invisible geometry which executes shell scripts through the C function call `system()`. Finally, the statements of `mapply`, `move` and `mdef` are no longer supported as statements in the language. However, the notion of a defined motion and applying it to different objects can be created implicitly by writing generic C movement functions and having multiple object reference them. These last elements are specifically omitted since files incorporating these features did not appear in the WALKTHRU environment.

The syntax of the new statements annotated with comments appears below. The elements in bold represent the actual keywords of the UniGrafix language. Elements appearing in square brackets are optional.

Vardef statement:

The `(vardef…)` statement from the bump extension has been replaced with a similar variable declaration mechanism, also called `vardef`. The syntax structure and functional elements of this expression are taken directly from the GLIDE implementation [].

The syntax of this statement is as follows:

```
vardef variable_name $ variable_type $ $ assignment_expression $ ;
```

The value of `variable_name` is not placed within dollar sign characters and can be any valid C identifier. The identifier is the same string used to reference this variable in other kinematic expressions appearing elsewhere in the file. The `variable_type` expression represents the C type of the user-defined variable being created by this statement. Any valid C type may be used, such as `int`, `float`, `double` and `char`, or even a pointer, but beware that the end product of a dollar sign expression appearing in any of the above statements is always cast to a float (see the example of section 3, Figure 3 in the text). Finally, the `assignment_expression` must be a valid C assignment statement, of the form " `variable_name = a valid c expression` ". Both of the final two elements of this statement type must be surrounded by dollar signs.

The effect of a `vardef` is to create a variable declaration of the same name with the specified type in the generated C file, `ug.dynam.c`. In addition, the assignment expression component of the statement is placed within the function `user_variable_update()`, also appearing in `ug.dynam.c`. This function `user_variable_update()` gets executed every frame of the animation and the effect of this is to consecutively update all of the user-defined variables once for every frame before subsequent expression evaluation.

Example of a `vardef` statement:

```
vardef t $ int $ $ t = (int)FRAME $;
```

This particular `vardef` statement produces the following C code:

```
static int t;

t = (int) FRAME ;
```

The first line generated is the global declaration of the user defined variable, and the second line of code created is the assignment statement which is placed within the function `user_variable_update()`. No checking is performed, so if the `vardef` statement's `assignment_expression` had been only " `= (int)FRAME` " then precisely this would have been output causing a syntax error during the compilation step.

Vertex statement:

```
v name x_coord y_coord z_coord [w_coord] [color-name] ;
```

The values for `x_coord`, `y_coord`, and `z_coord` may be kinematic (dollar sign) expressions. Currently, `w_coord, name` and `color-name` are fixed to be static values.

Example of a `vertex` statement:

```
v vertex1 $ FRAME * 2 $  12.7  $position * fsinf(angle) $ red ;
```

This statement produces the following C code in one of the various `cases` of the `switch` statement appearing in the function `UG_DYNAM_FUNC0`:

```
case 1:
    floatval[0] = (float)(FRAME  * 2 );
    floatval[1] = (float)( 12.7 );
    floatval[2] = (float)( position * fsinf(angle) );
    break;
```

Elements to notice about the example are the use of `FRAME`, a predefined variable, and the presence of two other variables `position` and `angle`. These other two variables must be defined using the `vardef` construct mentioned below, or else a compilation error will occur later in the viewing/parsing process. Furthermore, there is no checking at all by this program or by the C compiler as to the 'correctness' of these statements. In other words if the `position` variable is declared to be a `char` then its multiplication with the `fsinf` function is meaningless and the value in the example assigned to the z coordinate of the vertex would be useless.

Instance statement:

```
i [name] ( def_name [material_name [light_name]] [tforms] ) ;
```

Currently, the new implementation of instance statements permits only the `tforms` element (i.e. the transformations) to be a time-varying expression. The types of transformations allowed are the same as in the original UniGrafix language with the exception of the arbitrary matrix statements, which are not implemented. In addition, the floating point expressions of these transform elements can be C code as in the vertex statements mentioned above. As an example consider a moving cube:

Example of an `instance` statement:

```
i moving_cube ( cube  –tx $TIME * 2$  -ty $TIME * (-2)$  ) ;
```

This instance of a cube moves in the positive x direction and in the negative y direction as time progressed. The C code generated by the parser is shown below:

```
case 1:
    floatval[0] = (float)( TIME  * 2  );
    floatval[1] = (float)( 0.0 );
    floatval[2] = (float)( 0.0 );
    break;
```

```
case 2:
   floatval[0] = (float)( TIME * (-2) );
   floatval[1] = (float)( 0.0 );
   floatval[2] = (float)( 0.0 );
   break;
```

The only element of interest is that each and every transform of a time-varying instance statement will be converted into a case statement of its own.

Color statement:

```
c name lightness [hue [saturation [translucency ] ] ] ;
```

The current implementation only supports time-varying color statements defined in the LHS system []. For this statement, the three variables `lightness`, `hue`, and `saturation` may be dollar sign expressions, while the `name` and the `translucency` must be fixed. In the LHS system the `lightness` of the color corresponds to its intensity (how far it is from black, zero `lightness`), the `hue` corresponds to the color of the value (measured in degrees around a dual hex-cone), while `saturation` is how much color of the appropriate `hue` is present (distance from a gray value at zero `saturation`). This functions identically to a `vertex` statement except that all but the first floating point value may be omitted. Likewise the C code generated is identical in format to the code generated by a `vertex` statement.

Cinclude statement:

This new statement type has been added to facilitate the inclusion of C code functions into the file generated (`ug.dynam.c`) from a time-varying UniGrafix file. The new statement type is conceptually identical to the UG equivalent statement `include` and simply includes the named C file in the generated code file `ug.dynam.c`. This is achieved by placing a C include statement in the generated file with the appropriate filename.

The syntax is as follows:

```
cinclude file_name ;
```

This in turn generates the following C code in ug.dynam.c:

```
#include “file_name”;
```

The author of the UniGrafix file is responsible for placing this C file in a location that will enable the C compiler to find it based upon the string given as the " `file_name` ".

Omitted Statements:

A number of items were left out of the new version of the time-varying UniGrafix syntax, among them were the original bump elements concerning defined motion, as well as kinematic camera, light and array statements. Finally, dollar sign expressions may only be placed within the floating point values of the above statements. It is conceivable that a user might want to alter the names and other strings in a UniGrafix file as time varies. This was intentionally left out since its usefulness was unclear and it would have required considerably more work than this current work.

Predefined Variables:

A number of new variables are provided to the author of new kinematic scenes. Most of these are elements, which GLIDE incorporated into its system and allow simple user interaction and time-dependent behavior.

They are listed below with a short description of what they represent:

`Variable Name`	`C Type`	`Description`
`FRAME`	int	the current frame number for an the animation
`TIME`	float	total time in seconds since the beginning of the animation
`MOUSE_X`	int	x coordinate of the mouse
`MOUSE_Y`	int	y coordinate of the mouse
`RESOLUTION_X`	int	viewing window's x resolution
`RESOLUTION_Y`	int	viewing window's y resolution

These variables maybe used in any dollar sign expressions that represent floating point numbers. They are all declared to be `static float`. User-defined variables should not be created with the same name, since this will cause compilation time errors.

Appendix B: Example time-varying geometry files

Original Cubogear File

Original BUMP extended UniGrafix file of the Cubogear, file: old_cubogear.ug

```
{MOVING cubocta gear on a pedestal for 5th floor machineroom}
{at x=2016 y-1632 z= 981 with center of gear at 981+75=1056}

c CUBOGEARyel .9 60 1.0;
c CUBOGEARgrn .9 120 1.0;
c CUBOGEARblu .9 240 1.0;
c CUBOGEARred .9 0 1.0;
c CUBOGEARped 0.7 0.0 0.0 ;

include wb.def;
include ws.def;
include ped.def;
i p (CUBOGEARped CUBOGEARped -tx 1800 -ty 1632 -tz 981);

def temp1;
(meval
 i g1 (CUBOGEARwb CUBOGEARgrn -tz  24.39112079 -rz $18*t$ -rx 90 -tx 1800 -ty
1632 -tz 1056);
 i g2 (CUBOGEARwb CUBOGEARgrn -tz  24.39112079 -rz $18*t$ -rx -90 -tx 1800 -
ty 1632 -tz 1056);
 i g3 (CUBOGEARwb CUBOGEARred -tz  24.39112079 -rz $18*t$ -rz 45 -tx 1800 -ty
1632 -tz 1056);
 i g4 (CUBOGEARwb CUBOGEARblu -tz  24.39112079 -rz $18*t$ -rz 45 -ry 90 -tx
1800 -ty 1632 -tz 1056);
 i g5 (CUBOGEARwb CUBOGEARred -tz  24.39112079 -rz $18*t$ -rz 45 -ry 180 -tx
1800 -ty 1632 -tz 1056);
 i g6 (CUBOGEARwb CUBOGEARblu -tz  24.39112079 -rz $18*t$ -rz 45 -ry 270 -tx
1800 -ty 1632 -tz 1056);
)
end;

def temp2;
(meval
 i G1 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx 54.73333333
-rz 45 -tx 1800 -ty 1632 -tz 1056);
 i G2 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx -
54.73333333 -rz 45 -tx 1800 -ty 1632 -tz 1056);
 i G3 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx 54.73333333
-rz 45 -ry 90 -tx 1800 -ty 1632 -tz 1056);
 i G4 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx -
54.73333333 -rz 45 -ry 90 -tx 1800 -ty 1632 -tz 1056);
```

```
 i G5 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx 54.73333333
-rz 45 -ry 180 -tx 1800 -ty 1632 -tz 1056);
 i G6 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx -
54.73333333 -rz 45 -ry 180 -tx 1800 -ty 1632 -tz 1056);
 i G7 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx 54.73333333
-rz 45 -ry 270 -tx 1800 -ty 1632 -tz 1056);
 i G8 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx -
54.73333333 -rz 45 -ry 270 -tx 1800 -ty 1632 -tz 1056);
)
end;

def temp;
i (temp1);
i (temp2);
end;

i (temp);
```

New Cubogear File

New extended UniGrafix file describing the Cubogear sculpture, file: new_cubogear.ug

```
{MOVING cubocta gear on a pedestal for 5th floor machineroom}
{at x=2016 y-1632 z= 981 with center of gear at 981+75=1056}

c CUBOGEARyel .9 60 1.0;
c CUBOGEARgrn .9 120 1.0;
c CUBOGEARblu .9 240 1.0;
c CUBOGEARred .9 0 1.0;
c CUBOGEARped 0.7 0.0 0.0 ;

vardef t $ int $ $ t = (int)FRAME $ ;

include wb.def;
include ws.def;
include ped.def;

i p (CUBOGEARped CUBOGEARped -tx 1800 -ty 1632 -tz 981);
i g1 (CUBOGEARwb CUBOGEARgrn -tz  24.39112079 -rz $18*t$ -rx 90 -tx 1800 -ty
1632 -tz 1056);
i g2 (CUBOGEARwb CUBOGEARgrn -tz  24.39112079 -rz $18*t$ -rx -90 -tx 1800 -ty
1632 -tz 1056);
i g3 (CUBOGEARwb CUBOGEARred -tz  24.39112079 -rz $18*t$ -rz 45 -tx 1800 -ty
1632 -tz 1056);
i g4 (CUBOGEARwb CUBOGEARblu -tz  24.39112079 -rz $18*t$ -rz 45 -ry 90 -tx
1800 -ty 1632 -tz 1056);
i g5 (CUBOGEARwb CUBOGEARred -tz  24.39112079 -rz $18*t$ -rz 45 -ry 180 -tx
1800 -ty 1632 -tz 1056);
```

```
i g6 (CUBOGEARwb CUBOGEARblu -tz  24.39112079 -rz $18*t$ -rz 45 -ry 270 -tx
1800 -ty 1632 -tz 1056);

i G1 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx 54.73333333
-rz 45 -tx 1800 -ty 1632 -tz 1056);
i G2 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx -54.73333333
-rz 45 -tx 1800 -ty 1632 -tz 1056);
i G3 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx 54.73333333
-rz 45 -ry 90 -tx 1800 -ty 1632 -tz 1056);
i G4 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx -54.73333333
-rz 45 -ry 90 -tx 1800 -ty 1632 -tz 1056);
i G5 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx 54.73333333
-rz 45 -ry 180 -tx 1800 -ty 1632 -tz 1056);
i G6 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx -54.73333333
-rz 45 -ry 180 -tx 1800 -ty 1632 -tz 1056);
i G7 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx 54.73333333
-rz 45 -ry 270 -tx 1800 -ty 1632 -tz 1056);
i G8 (CUBOGEARws CUBOGEARyel -tz 29.56724974 -rz $15 - 36*t$ -rx -54.73333333
-rz 45 -ry 270 -tx 1800 -ty 1632 -tz 1056);
```

Original Juggling File

Original UniGrafix Juggling Scene (before the C pre-processor), file: original_juggling.ug

```
(vardef a t*0.1 0 6)

#define h 2
#define w 2
#define d 1.2
#define BALL_RAD 0.5
#define WRAP6(t) (((t)<6)?((t)):((t)-6))
#define LINEAR(t,x0,x1) (((t)*(x1) + (1-(t))*(x0)))
#define PARABOLAX(t,x0,x2) (0.5 * ((t)*(x2) + (2-(t))*(x0)))
#define PARABOLAY(t,y) ((y)*(1-((t)-1)*((t)-1)))
#define H0(t) (1 + (t)*(t)*(2*(t) - 3))
#define H0p(t) ((t)*(1 + (t)*((t)-2)))
#define H1(t) ((t)*(t)*(3-2*(t)))
#define H1p(t) ((t)*(t)*((t)-1))

#define HERMITE(t,h0,h0p,h1,h1p)
((h0)*H0(t)+(h0p)*H0p(t)+(h1)*H1(t)+(h1p)*H1p(t))

#define TIME6(t,x0,x1,x2,x3,x4,x5)
(((t)<1)?(x0):(((t)<2)?(x1):(((t)<3)?(x2):(((t)<4)?(x3):(((t)<5)?(x4):(((t)<=
6)?(x5):(0)))))))
```

```
#define BALLX(t) TIME6(t,PARABOLAX(t,0,w),PARABOLAX(t,0,w),HERMITE(t-
2,w,w/2,w-d,-w/2),PARABOLAX(t-3,w-d,-d),PARABOLAX(t-3,w-d,-d),HERMITE(t-5,-
d,-w/2,0,w/2))

#define BALLY(t) TIME6(t,PARABOLAY(t,h),PARABOLAY(t,h),HERMITE(t-2,0,-
2*h,0,2*h),PARABOLAY(t-3,h),PARABOLAY(t-3,h),HERMITE(t-5,0,-2*h,0,2*h))

c ball_color 1.0 0 1.0;
c ball_color2 1.0 100 1.0;
c wall_color 1.0 240 1.0;

def ball;
include ball.ug;
end;

def back;
v v0 0 0 0;
v v1 1 0 0;
v v2 1 1 0;
v v3 0 1 0;
f f0 (v0 v1 v2 v3) wall_color;
end;

def all;
(meval
i (back -sx $ w+3*d $ -sy $ 2*h $ -tz -1 -ty $ -h/2 $ -tx $ -2*d $ );
i (ball -sa BALL_RAD -tx $ BALLX(WRAP6(a)) $ -ty $ BALLY(WRAP6(a)) $ );
i (ball -sa BALL_RAD -tx $ BALLX(WRAP6(a+2)) $ -ty $ BALLY(WRAP6(a+2)) $ );
i (ball -sa BALL_RAD -tx $ BALLX(WRAP6(a+4)) $ -ty $ BALLY(WRAP6(a+4)) $ );
)
end;

i (all) ;
```

Preprocessed Juggling File

Original UniGrafix Juggling Scene after the C pre-processor, file: post_cpp_juggling.ug

```
(vardef a t*0.1 0 6)

{This file was created  from the file original_juggling.ug using the C
preprocessor}

c ball_color 1.0 0 1.0;
c ball_color2 1.0 100 1.0;
c wall_color 1.0 240 1.0;

def ball;
include ball.ug;
```

```
end;

def back;
v v0 0 0 0;
v v1 1 0 0;
v v2 1 1 0;
v v3 0 1 0;
f f0 (v0 v1 v2 v3) wall_color;
end;

def all;
(meval
i (back -sx $ 2+3*1.2 $ -sy $ 2*2 $ -tz -1 -ty $ -2/2 $ -tx $ -2*1.2 $ );
i (ball -sa 0.5 -tx $   ((( (((a)<6)?((a)):((a)-6)))<1)?( (0.5 * ((
(((a)<6)?((a)):((a)-6)))*(2) + (2-( (((a)<6)?((a)):((a)-6))))*(0)))):(((
(((a)<6)?((a)):((a)-6)))<2)?( (0.5 * (( (((a)<6)?((a)):((a)-6)))*(2) + (2-(
(((a)<6)?((a)):((a)-6))))*(0)))):((( (((a)<6)?((a)):((a)-6)))<3)?( ((2)* (1 +
( (((a)<6)?((a)):((a)-6))-2)*( (((a)<6)?((a)):((a)-6))-2)*(2*(
(((a)<6)?((a)):((a)-6))-2) - 3))+(2/2)* (( (((a)<6)?((a)):((a)-6))-2)*(1 + (
(((a)<6)?((a)):((a)-6))-2)*(( (((a)<6)?((a)):((a)-6))-2)-2)))+(2-1.2)* ((
(((a)<6)?((a)):((a)-6))-2)*( (((a)<6)?((a)):((a)-6))-2)*(3-2*(
(((a)<6)?((a)):((a)-6))-2)))+(-2/2)* (( (((a)<6)?((a)):((a)-6))-2)*(
(((a)<6)?((a)):((a)-6))-2)*(( (((a)<6)?((a)):((a)-6))-2)-1)))):(((
(((a)<6)?((a)):((a)-6)))<4)?( (0.5 * (( (((a)<6)?((a)):((a)-6))-3)*(-1.2) +
(2-( (((a)<6)?((a)):((a)-6))-3))*(2-1.2)))):((( (((a)<6)?((a)):((a)-6)))<5)?(
(0.5 * (( (((a)<6)?((a)):((a)-6))-3)*(-1.2) + (2-( (((a)<6)?((a)):((a)-6))-
3))*(2-1.2)))):((( (((a)<6)?((a)):((a)-6)))<=6)?( ((-1.2)* (1 + (
(((a)<6)?((a)):((a)-6))-5)*( (((a)<6)?((a)):((a)-6))-5)*(2*(
(((a)<6)?((a)):((a)-6))-5) - 3))+(-2/2)* (( (((a)<6)?((a)):((a)-6))-5)*(1 + (
(((a)<6)?((a)):((a)-6))-5)*(( (((a)<6)?((a)):((a)-6))-5)-2)))+(0)* ((
(((a)<6)?((a)):((a)-6))-5)*( (((a)<6)?((a)):((a)-6))-5)*(3-2*(
(((a)<6)?((a)):((a)-6))-5)))+(2/2)* (( (((a)<6)?((a)):((a)-6))-5)*(
(((a)<6)?((a)):((a)-6))-5)*(( (((a)<6)?((a)):((a)-6))-5)-1)))):(0))))))) $ -
ty $   ((( (((a)<6)?((a)):((a)-6)))<1)?( ((2)*(1-(( (((a)<6)?((a)):((a)-6)))-
1)*(( (((a)<6)?((a)):((a)-6)))-1)))):((( (((a)<6)?((a)):((a)-6)))<2)?(
((2)*(1-(( (((a)<6)?((a)):((a)-6)))-1)*(( (((a)<6)?((a)):((a)-6)))-1)))):(((
(((a)<6)?((a)):((a)-6)))<3)?( ((0)* (1 + ( (((a)<6)?((a)):((a)-6))-2)*(
(((a)<6)?((a)):((a)-6))-2)*(2*( (((a)<6)?((a)):((a)-6))-2) - 3))+(-2*2)* ((
(((a)<6)?((a)):((a)-6))-2)*(1 + ( (((a)<6)?((a)):((a)-6))-2)*((
(((a)<6)?((a)):((a)-6))-2)-2)))+(0)* (( (((a)<6)?((a)):((a)-6))-2)*(
(((a)<6)?((a)):((a)-6))-2)*(3-2*( (((a)<6)?((a)):((a)-6))-2)))+(2*2)* ((
(((a)<6)?((a)):((a)-6))-2)*( (((a)<6)?((a)):((a)-6))-2)*((
(((a)<6)?((a)):((a)-6))-2)-1)))):((( (((a)<6)?((a)):((a)-6)))<4)?( ((2)*(1-((
(((a)<6)?((a)):((a)-6))-3)-1)*(( (((a)<6)?((a)):((a)-6))-3)-1)))):(((
(((a)<6)?((a)):((a)-6)))<5)?( ((2)*(1-(( (((a)<6)?((a)):((a)-6))-3)-1)*((
(((a)<6)?((a)):((a)-6))-3)-1)))):((( (((a)<6)?((a)):((a)-6)))<=6)?( ((0)* (1
+ ( (((a)<6)?((a)):((a)-6))-5)*( (((a)<6)?((a)):((a)-6))-5)*(2*(
(((a)<6)?((a)):((a)-6))-5) - 3))+(-2*2)* (( (((a)<6)?((a)):((a)-6))-5)*(1 + (
(((a)<6)?((a)):((a)-6))-5)*(( (((a)<6)?((a)):((a)-6))-5)-2)))+(0)* ((
(((a)<6)?((a)):((a)-6))-5)*( (((a)<6)?((a)):((a)-6))-5)*(3-2*(
(((a)<6)?((a)):((a)-6))-5)))+(2*2)* (( (((a)<6)?((a)):((a)-6))-5)*(
(((a)<6)?((a)):((a)-6))-5)*(( (((a)<6)?((a)):((a)-6))-5)-1)))):(0))))))) $ );
```

```
i (ball -sa 0.5 -tx $   ((( (((a+2)<6)?((a+2)):((a+2)-6)))<1)?( (0.5 * ((
(((a+2)<6)?((a+2)):((a+2)-6)))*(2) + (2-( (((a+2)<6)?((a+2)):((a+2)-
6))))*(0)))):((( (((a+2)<6)?((a+2)):((a+2)-6)))<2)?( (0.5 * ((
(((a+2)<6)?((a+2)):((a+2)-6)))*(2) + (2-( (((a+2)<6)?((a+2)):((a+2)-
6))))*(0)))):((( (((a+2)<6)?((a+2)):((a+2)-6)))<3)?( ((2)* (1 + (
(((a+2)<6)?((a+2)):((a+2)-6))-2)*( (((a+2)<6)?((a+2)):((a+2)-6))-2)*(2*(
(((a+2)<6)?((a+2)):((a+2)-6))-2) - 3))+(2/2)* (( (((a+2)<6)?((a+2)):((a+2)-
6))-2)*(1 + ( (((a+2)<6)?((a+2)):((a+2)-6))-2)*(( (((a+2)<6)?((a+2)):((a+2)-
6))-2)-2)))+(2-1.2)* (( (((a+2)<6)?((a+2)):((a+2)-6))-2)*(
(((a+2)<6)?((a+2)):((a+2)-6))-2)*(3-2*( (((a+2)<6)?((a+2)):((a+2)-6))-2)))+(-
2/2)* (( (((a+2)<6)?((a+2)):((a+2)-6))-2)*( (((a+2)<6)?((a+2)):((a+2)-6))-
2)*(( (((a+2)<6)?((a+2)):((a+2)-6))-2)-1)))):((( (((a+2)<6)?((a+2)):((a+2)-
6)))<4)?( (0.5 * (( (((a+2)<6)?((a+2)):((a+2)-6))-3)*(-1.2) + (2-(
(((a+2)<6)?((a+2)):((a+2)-6))-3))*(2-1.2)))):((( (((a+2)<6)?((a+2)):((a+2)-
6)))<5)?( (0.5 * (( (((a+2)<6)?((a+2)):((a+2)-6))-3)*(-1.2) + (2-(
(((a+2)<6)?((a+2)):((a+2)-6))-3))*(2-1.2)))):((( (((a+2)<6)?((a+2)):((a+2)-
6)))<=6)?( ((-1.2)* (1 + ( (((a+2)<6)?((a+2)):((a+2)-6))-5)*(
(((a+2)<6)?((a+2)):((a+2)-6))-5)*(2*( (((a+2)<6)?((a+2)):((a+2)-6))-5) -
3))+(-2/2)* (( (((a+2)<6)?((a+2)):((a+2)-6))-5)*(1 + (
(((a+2)<6)?((a+2)):((a+2)-6))-5)*(( (((a+2)<6)?((a+2)):((a+2)-6))-5)-
2)))+(0)* (( (((a+2)<6)?((a+2)):((a+2)-6))-5)*( (((a+2)<6)?((a+2)):((a+2)-
6))-5)*(3-2*( (((a+2)<6)?((a+2)):((a+2)-6))-5)))+(2/2)* ((
(((a+2)<6)?((a+2)):((a+2)-6))-5)*( (((a+2)<6)?((a+2)):((a+2)-6))-5)*((
(((a+2)<6)?((a+2)):((a+2)-6))-5)-1)))):(0))))))) $ -ty $   (((
(((a+2)<6)?((a+2)):((a+2)-6)))<1)?( ((2)*(1-(( (((a+2)<6)?((a+2)):((a+2)-
6)))-1)*(( (((a+2)<6)?((a+2)):((a+2)-6)))-1)))):(((
(((a+2)<6)?((a+2)):((a+2)-6)))<2)?( ((2)*(1-(( (((a+2)<6)?((a+2)):((a+2)-
6)))-1)*(( (((a+2)<6)?((a+2)):((a+2)-6)))-1)))):(((
(((a+2)<6)?((a+2)):((a+2)-6)))<3)?( ((0)* (1 + ( (((a+2)<6)?((a+2)):((a+2)-
6))-2)*( (((a+2)<6)?((a+2)):((a+2)-6))-2)*(2*( (((a+2)<6)?((a+2)):((a+2)-6))-
2) - 3))+(-2*2)* (( (((a+2)<6)?((a+2)):((a+2)-6))-2)*(1 + (
(((a+2)<6)?((a+2)):((a+2)-6))-2)*(( (((a+2)<6)?((a+2)):((a+2)-6))-2)-
2)))+(0)* (( (((a+2)<6)?((a+2)):((a+2)-6))-2)*( (((a+2)<6)?((a+2)):((a+2)-
6))-2)*(3-2*( (((a+2)<6)?((a+2)):((a+2)-6))-2)))+(2*2)* ((
(((a+2)<6)?((a+2)):((a+2)-6))-2)*( (((a+2)<6)?((a+2)):((a+2)-6))-2)*((
(((a+2)<6)?((a+2)):((a+2)-6))-2)-1)))):((( (((a+2)<6)?((a+2)):((a+2)-
6)))<4)?( ((2)*(1-(( (((a+2)<6)?((a+2)):((a+2)-6))-3)-1)*((
(((a+2)<6)?((a+2)):((a+2)-6))-3)-1)))):((( (((a+2)<6)?((a+2)):((a+2)-
6)))<5)?( ((2)*(1-(( (((a+2)<6)?((a+2)):((a+2)-6))-3)-1)*((
(((a+2)<6)?((a+2)):((a+2)-6))-3)-1)))):((( (((a+2)<6)?((a+2)):((a+2)-
6)))<=6)?( ((0)* (1 + ( (((a+2)<6)?((a+2)):((a+2)-6))-5)*(
(((a+2)<6)?((a+2)):((a+2)-6))-5)*(2*( (((a+2)<6)?((a+2)):((a+2)-6))-5) -
3))+(-2*2)* (( (((a+2)<6)?((a+2)):((a+2)-6))-5)*(1 + (
(((a+2)<6)?((a+2)):((a+2)-6))-5)*(( (((a+2)<6)?((a+2)):((a+2)-6))-5)-
2)))+(0)* (( (((a+2)<6)?((a+2)):((a+2)-6))-5)*( (((a+2)<6)?((a+2)):((a+2)-
6))-5)*(3-2*( (((a+2)<6)?((a+2)):((a+2)-6))-5)))+(2*2)* ((
(((a+2)<6)?((a+2)):((a+2)-6))-5)*( (((a+2)<6)?((a+2)):((a+2)-6))-5)*((
(((a+2)<6)?((a+2)):((a+2)-6))-5)-1)))):(0))))))) $ );
i (ball -sa 0.5 -tx $   ((( (((a+4)<6)?((a+4)):((a+4)-6)))<1)?( (0.5 * ((
(((a+4)<6)?((a+4)):((a+4)-6)))*(2) + (2-( (((a+4)<6)?((a+4)):((a+4)-
6))))*(0)))):((( (((a+4)<6)?((a+4)):((a+4)-6)))<2)?( (0.5 * ((
(((a+4)<6)?((a+4)):((a+4)-6)))*(2) + (2-( (((a+4)<6)?((a+4)):((a+4)-
```

```
6))))*(0)))):((( (((a+4)<6)?((a+4)):((a+4)-6)))<3)?( ((2)* (1 + (
(((a+4)<6)?((a+4)):((a+4)-6))-2)*( (((a+4)<6)?((a+4)):((a+4)-6))-2)*(2*(
(((a+4)<6)?((a+4)):((a+4)-6))-2) - 3))+(2/2)* (( (((a+4)<6)?((a+4)):((a+4)-
6))-2)*(1 + ( (((a+4)<6)?((a+4)):((a+4)-6))-2)*(( (((a+4)<6)?((a+4)):((a+4)-
6))-2)-2)))+(2-1.2)* (( (((a+4)<6)?((a+4)):((a+4)-6))-2)*(
(((a+4)<6)?((a+4)):((a+4)-6))-2)*(3-2*( (((a+4)<6)?((a+4)):((a+4)-6))-2)))+(-
2/2)* (( (((a+4)<6)?((a+4)):((a+4)-6))-2)*( (((a+4)<6)?((a+4)):((a+4)-6))-
2)*(( (((a+4)<6)?((a+4)):((a+4)-6))-2)-1)))):((( (((a+4)<6)?((a+4)):((a+4)-
6)))<4)?( (0.5 * (( (((a+4)<6)?((a+4)):((a+4)-6))-3)*(-1.2) + (2-(
(((a+4)<6)?((a+4)):((a+4)-6))-3))*(2-1.2)))):((( (((a+4)<6)?((a+4)):((a+4)-
6)))<5)?( (0.5 * (( (((a+4)<6)?((a+4)):((a+4)-6))-3)*(-1.2) + (2-(
(((a+4)<6)?((a+4)):((a+4)-6))-3))*(2-1.2)))):((( (((a+4)<6)?((a+4)):((a+4)-
6)))<=6)?( ((-1.2)* (1 + ( (((a+4)<6)?((a+4)):((a+4)-6))-5)*(
(((a+4)<6)?((a+4)):((a+4)-6))-5)*(2*( (((a+4)<6)?((a+4)):((a+4)-6))-5) -
3))+(-2/2)* (( (((a+4)<6)?((a+4)):((a+4)-6))-5)*(1 + (
(((a+4)<6)?((a+4)):((a+4)-6))-5)*(( (((a+4)<6)?((a+4)):((a+4)-6))-5)-
2)))+(0)* (( (((a+4)<6)?((a+4)):((a+4)-6))-5)*( (((a+4)<6)?((a+4)):((a+4)-
6))-5)*(3-2*( (((a+4)<6)?((a+4)):((a+4)-6))-5)))+(2/2)* ((
(((a+4)<6)?((a+4)):((a+4)-6))-5)*( (((a+4)<6)?((a+4)):((a+4)-6))-5)*((
(((a+4)<6)?((a+4)):((a+4)-6))-5)-1)))):(0))))))) $ -ty $   (((
(((a+4)<6)?((a+4)):((a+4)-6)))<1)?( ((2)*(1-(( (((a+4)<6)?((a+4)):((a+4)-
6)))-1)*(( (((a+4)<6)?((a+4)):((a+4)-6)))-1)))):(((
(((a+4)<6)?((a+4)):((a+4)-6)))<2)?( ((2)*(1-(( (((a+4)<6)?((a+4)):((a+4)-
6)))-1)*(( (((a+4)<6)?((a+4)):((a+4)-6)))-1)))):(((
(((a+4)<6)?((a+4)):((a+4)-6)))<3)?( ((0)* (1 + ( (((a+4)<6)?((a+4)):((a+4)-
6))-2)*( (((a+4)<6)?((a+4)):((a+4)-6))-2)*(2*( (((a+4)<6)?((a+4)):((a+4)-6))-
2) - 3))+(-2*2)* (( (((a+4)<6)?((a+4)):((a+4)-6))-2)*(1 + (
(((a+4)<6)?((a+4)):((a+4)-6))-2)*(( (((a+4)<6)?((a+4)):((a+4)-6))-2)-
2)))+(0)* (( (((a+4)<6)?((a+4)):((a+4)-6))-2)*( (((a+4)<6)?((a+4)):((a+4)-
6))-2)*(3-2*( (((a+4)<6)?((a+4)):((a+4)-6))-2)))+(2*2)* ((
(((a+4)<6)?((a+4)):((a+4)-6))-2)*( (((a+4)<6)?((a+4)):((a+4)-6))-2)*((
(((a+4)<6)?((a+4)):((a+4)-6))-2)-1)))):((( (((a+4)<6)?((a+4)):((a+4)-
6)))<4)?( ((2)*(1-(( (((a+4)<6)?((a+4)):((a+4)-6))-3)-1)*((
(((a+4)<6)?((a+4)):((a+4)-6))-3)-1)))):((( (((a+4)<6)?((a+4)):((a+4)-
6)))<5)?( ((2)*(1-(( (((a+4)<6)?((a+4)):((a+4)-6))-3)-1)*((
(((a+4)<6)?((a+4)):((a+4)-6))-3)-1)))):((( (((a+4)<6)?((a+4)):((a+4)-
6)))<=6)?( ((0)* (1 + ( (((a+4)<6)?((a+4)):((a+4)-6))-5)*(
(((a+4)<6)?((a+4)):((a+4)-6))-5)*(2*( (((a+4)<6)?((a+4)):((a+4)-6))-5) -
3))+(-2*2)* (( (((a+4)<6)?((a+4)):((a+4)-6))-5)*(1 + (
(((a+4)<6)?((a+4)):((a+4)-6))-5)*(( (((a+4)<6)?((a+4)):((a+4)-6))-5)-
2)))+(0)* (( (((a+4)<6)?((a+4)):((a+4)-6))-5)*( (((a+4)<6)?((a+4)):((a+4)-
6))-5)*(3-2*( (((a+4)<6)?((a+4)):((a+4)-6))-5)))+(2*2)* ((
(((a+4)<6)?((a+4)):((a+4)-6))-5)*( (((a+4)<6)?((a+4)):((a+4)-6))-5)*((
(((a+4)<6)?((a+4)):((a+4)-6))-5)-1)))):(0))))))) $ );
)
end;

i (all);
```

New Juggling File

Juggling scene using the new extended syntax system, file: new_juggling.ug

```
vardef q $float$ $ q = (float)((0.01)*FRAME) $;

cinclude jug.c;

c ball_color 1.0 0 1.0;
c ball_color2 1.0 100 1.0;
c wall_color 1.0 240 1.0;

def object;
   include ball.ug;
end;

def back;
v v0 0 0 0;
v v1 1 0 0;
v v2 1 1 0;
v v3 0 1 0;
f f0 (v0 v1 v2 v3) wall_color;
end;

def temp;
i (back -sx $ 2+3*1.2 $ -sy $ 2*2 $ -tz -1 -ty $ -2/2 $ -tx $ -2*1.2 $ );
i (object -sa 0.5 -tx $ BALLX(WRAP6(q)) $ -ty $ BALLY(WRAP6(q)) $ );
i (object -sa 0.5 -tx $ BALLX(WRAP6(q+2)) $ -ty $ BALLY(WRAP6(q+2)) $ );
i (object -sa 0.5 -tx $ BALLX(WRAP6(q+4)) $ -ty $ BALLY(WRAP6(q+4)) $ );
end;

i (temp);
```

C Code File for New Juggling Example

Included C file for the new juggling scene (defines all referenced functions), file: jug.c

```
/* c file for use in the juggler routine needs to be included in the dynamic
c file */

#define h 2
#define w 2
#define d 1.2

float WRAP6( float t)
{  return (((t)<6)?((t)):((t)-6));  }
```

```
float LINEAR(float t, float x0, float x1)
{  return((t)*(x1) + (1-(t))*(x0));  }

float PARABOLAX(float t, float x0, float x2)
{  return((0.5 * ((t)*(x2) + (2-(t))*(x0))));  }

float PARABOLAY(float t, float y)
{  return (((y)*(1-((t)-1)*((t)-1))));  }

float H0(float t)
{  return (1 + (t)*(t)*(2*(t) - 3));  }

float H0p(float t)
{  return((t)*(1 + (t)*((t)-2)));  }

float H1(float t)
{  return ((t)*(t)*(3-2*(t)));  }

float H1p(float t)
{  return((t)*(t)*((t)-1));  }

float HERMITE(float t, float h0, float h0p, float h1, float h1p)
{  return((h0)*H0(t)+(h0p)*H0p(t)+(h1)*H1(t)+(h1p)*H1p(t));  }

float TIME6(float t, float x0, float x1, float x2, float x3,
                                       float x4, float x5)
{ return (((t)<1)?(x0):(((t)<2)?(x1):(((t)<3)?(x2):(((t)<4)?(x3)
                  :(((t)<5)?(x4):(((t)<=6)?(x5):(0)))))));           }

float BALLX( float t)
{
   float ret_value;
   ret_value = TIME6(t,PARABOLAX(t,0,w),PARABOLAX(t,0,w),
                 HERMITE(t-2,w,w/2,w-d,-w/2),
                 PARABOLAX(t-3,w-d,-d),PARABOLAX(t-3,w-d,-d),
                 HERMITE(t-5,-d,-w/2,0,w/2));
  return ret_value;
}

float BALLY( float t)
{
   float ret_value;
   ret_value = TIME6(t,PARABOLAY(t,h),PARABOLAY(t,h),
                  HERMITE(t-2,0,-2*h,0,2*h),PARABOLAY(t-3,h),
                  PARABOLAY(t-3,h),HERMITE(t-5,0,-2*h,0,2*h));
  return ret_value;
}
```

Gravity Simulation New Kinematic UniGrafix File

Gravity simulation scene description in the new extended system, file: grav.ug

```
{Simulation of four objects moving underneath a psuedo gravity force within
the walls of a cube environment which repel the objects from the boundary}

{Walls of a cube}
v w_v1      10  10  10;
v w_v2     -10  10  10;
v w_v3     -10 -10  10;
v w_v4      10 -10  10;
v w_v5      10  10 -10;
v w_v6     -10  10 -10;
v w_v7     -10 -10 -10;
v w_v8      10 -10 -10;

f top (w_v4 w_v3 w_v2 w_v1);
f front (w_v5 w_v8 w_v4 w_v1);
f back (w_v6 w_v2 w_v3 w_v7);
f bottom (w_v6 w_v7 w_v8 w_v5);
f right (w_v5 w_v1 w_v2 w_v6);
f left (w_v8 w_v7 w_v3 w_v4);

c c1 0.5 45 1;
c c2 0.5 95 1;
c c3 0.5 145 1;

{generic dynamic object}
def sphere;
    include ball.ug;
end;

{ C include statement for the psuedo gravity simulation }
cinclude grav.c;

{ within the two objects below the array of variables objs is defined in the
C file grav.c which is included }

i object1 (sphere
                   -tx $objs[0]->c_of_mass->x$
                   -ty $objs[0]->c_of_mass->y$
                   -tz $objs[0]->c_of_mass->z$);

i object2 (sphere -sa 0.5
                   -tx $objs[1]->c_of_mass->x$
                   -ty $objs[1]->c_of_mass->y$
                   -tz $objs[1]->c_of_mass->z$);

i object3 (sphere -sa 0.5
                   -tx $objs[2]->c_of_mass->x$
```

```
                   -ty $objs[2]->c_of_mass->y$
                   -tz $objs[2]->c_of_mass->z$);

i object4 (sphere -sa 0.5
                   -tx $objs[3]->c_of_mass->x$
                   -ty $objs[3]->c_of_mass->y$
                   -tz $objs[3]->c_of_mass->z$);
```

C Code Control File for Gravity Simulation

Included C code file in the above (performs the simulation), file: grav.c

This file performs the simulation of the various objects and the gravitational forces interacting among the objects. It provides the new compiled-code UniGrafix file with a link to the simulation's data structures via an array of structure pointers (`objs`). The structure `objs` takes an `ID` as an index and returns the updated 3d coordinates of that particular (`ID`) object for the current value of time.

Note: in order to actually view this file the generated C file `ug.dynam.c` must be modified such that the function `initialize_all()` gets called in the `initialize_dynamics()` routine and that the `find_all_positions()` function gets called in the `update_global_variables()` function. See the future work section for additions that would remove this part of the process.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define sq(arg) ((arg)*(arg))
#define G_grav_constant  100.0 /* used to be 100.0 */
#define Van_constant     5.0   /* used for almost inter-penetrating object*/
#define IMPACT           30.0  /* used if distance is two small */
#define cube(arg) ((arg)*(arg)*(arg))
#define TRUE 1
#define FALSE 0

typedef struct {
  double x;  double y;  double z;
} vector;

typedef struct {
  vector *c_of_mass;   /* center of mass of the object */
  vector *vel;
  double mass;
  int    moving_obj;
} object;
```

```
typedef struct {
  vector *normal;      /* normal to the plane of the wall */
  vector *c_of_mass;   /* center of mass of the wall  */
  double mass;
} wall;

/* global variables */
vector *cur_force;   vector *acc;   vector *dif_vec;   vector *force;

double dt;  /* size of the time stamp size */

wall **walls;
object **objs;

#define NUM_WALLS 6
#define NUM_OBJECTS 4
int do_initialize = 1;

void error (char *string) {  printf("An Error Has Occurred: %s \n");  }

void initialize_global_vectors(void) {
  cur_force = (vector *)malloc(sizeof(vector));
  acc       = (vector *)malloc(sizeof(vector));
  dif_vec   = (vector *)malloc(sizeof(vector));
  force     = (vector *)malloc(sizeof(vector));
}

void initialize_walls(void) {
  int i;
  walls = (wall **)malloc(sizeof(wall *)*NUM_WALLS);
  if (walls == NULL) { error("unable to malloc all walls, exiting");
exit(0);}
  for (i = 0; i < NUM_WALLS; i++) {
    walls[i] = (wall *)malloc(sizeof(wall));
    if (walls[i] == NULL) { error("unable to malloc a wall, exiting");
exit(0);}
    walls[i]->normal = (vector *)malloc(sizeof(vector));
      if (walls[i]->normal == NULL) { error("unable to malloc a wall normal,
exiting"); exit(0);}
    walls[i]->c_of_mass = (vector *)malloc(sizeof(vector));
    if (walls[i]->c_of_mass == NULL) { error("unable to malloc a wall center
of mass, exiting"); exit(0);}
  }
}

void initialize_objects(void) {
  int i;
  objs = (object **)malloc(sizeof(object *)*NUM_OBJECTS);
  if (objs == NULL) { error("unable to malloc all objects, exiting");
exit(0);}
  for (i = 0; i < NUM_OBJECTS; i++) {
    objs[i] = (object *)malloc(sizeof(object));
```

```
    if (objs[i] == NULL) { error("unable to malloc an object, exiting");
exit(0);}
    objs[i]->c_of_mass = (vector *)malloc(sizeof(vector));
    if (objs[i]->c_of_mass == NULL) { error("unable to malloc an object
center of mass, exiting"); exit(0);}
    objs[i]->vel = (vector *)malloc(sizeof(vector));
    if (objs[i]->vel == NULL) { error("unable to malloc an object velocity,
exiting"); exit(0);}
  }
}

void load_walls(void) {
  walls[0]->normal->x = 1.0;            walls[1]->normal->x = -1.0;
  walls[0]->normal->y = 0.0;            walls[1]->normal->y = 0.0;
  walls[0]->normal->z = 0.0;            walls[1]->normal->z = 0.0;
  walls[0]->c_of_mass->x = -10.0;       walls[1]->c_of_mass->x = 10.0;
  walls[0]->c_of_mass->y = 0.0;         walls[1]->c_of_mass->y = 0.0;
  walls[0]->c_of_mass->z = 0.0;         walls[1]->c_of_mass->z = 0.0;
  walls[0]->mass = 5.0;                 walls[1]->mass = 5.0;

  walls[2]->normal->x = 0.0;            walls[3]->normal->x = 0.0;
  walls[2]->normal->y = 1.0;            walls[3]->normal->y = -1.0;
  walls[2]->normal->z = 0.0;            walls[3]->normal->z = 0.0;
  walls[2]->c_of_mass->x = 0.0;         walls[3]->c_of_mass->x = 0.0;
  walls[2]->c_of_mass->y = -10.0;       walls[3]->c_of_mass->y = 10.0;
  walls[2]->c_of_mass->z = 0.0;         walls[3]->c_of_mass->z = 0.0;
  walls[2]->mass = 5.0;                 walls[3]->mass = 5.0;

  walls[4]->normal->x = 0.0;            walls[5]->normal->x = 0.0;
  walls[4]->normal->y = 0.0;            walls[5]->normal->y = 0.0;
  walls[4]->normal->z = -1.0;           walls[5]->normal->z = 1.0;
  walls[4]->c_of_mass->x = 0.0;         walls[5]->c_of_mass->x = 0.0;
  walls[4]->c_of_mass->y = 0.0;         walls[5]->c_of_mass->y = 0.0;
  walls[4]->c_of_mass->z = 10.0;        walls[5]->c_of_mass->z = -10.0;
  walls[4]->mass = 5.0;                 walls[5]->mass = 5.0;
}

void load_objects(void) {
  /* currently the number of objects is only four 0 - 3*/
  objs[0]->c_of_mass->x = 0.0;          objs[1]->c_of_mass->x = 70.0;
  objs[0]->c_of_mass->y = 0.0;          objs[1]->c_of_mass->y = 0.0;
  objs[0]->c_of_mass->z = 0.0;          objs[1]->c_of_mass->z = 0.0;
  objs[0]->vel->x = 0.0;                objs[1]->vel->x = 0.0;
  objs[0]->vel->y = 0.0;                objs[1]->vel->y = 100.0;
  objs[0]->vel->z = 0.0;                objs[1]->vel->z = 0.0;
  objs[0]->mass = 100.0;                objs[1]->mass = 1.0;
  objs[0]->moving_obj = FALSE;          objs[1]->moving_obj = TRUE;

  objs[2]->c_of_mass->x = 73.0;         objs[3]->c_of_mass->x = 0.0;
  objs[2]->c_of_mass->y = 0.0;          objs[3]->c_of_mass->y = 12.0;
  objs[2]->c_of_mass->z = 0.0;          objs[3]->c_of_mass->z = 0.0;
  objs[2]->vel->x = 0.0;                objs[3]->vel->x = 10.0;
```

```
  objs[2]->vel->y = 118.0;              objs[3]->vel->y = 0.0;
  objs[2]->vel->z = 0.0;                objs[3]->vel->z = 0.0;
  objs[2]->mass = 0.5;                  objs[3]->mass = 1.0;
  objs[2]->moving_obj = TRUE;           objs[3]->moving_obj = TRUE;
}

void initialize_all(void) {
  if (do_initialize) {
    initialize_global_vectors();
    dt = 0.01;
    if (NUM_WALLS > 0) {
      initialize_walls();
      load_walls();
    } else {
      printf("Warning the value for the number of walls was zero \n");
    }
    if (NUM_OBJECTS > 0) {
      initialize_objects();
      load_objects();
    } else {
      printf("Warning the value for the number of objects was zero \n");
    }
    do_initialize = 0;
  }
}


double find_distance (vector *v1, vector *v2){
  double distance;
  distance = sqrt(sq(v1->x - v2->x) + sq(v1->y - v2->y) + sq(v1->z - v2->z));
  return distance;
}

double find_dist_wall_obj (object *obj, wall *wall1) {
  /* REQUIRES THE VECTOR dif_vec to be global */
  double mag;
  double distance;
  double dot_product;
  dif_vec->x = (wall1->c_of_mass->x) - (obj->c_of_mass->x);
  dif_vec->y = (wall1->c_of_mass->y) - (obj->c_of_mass->y);
  dif_vec->z = (wall1->c_of_mass->z) - (obj->c_of_mass->z);
  mag = sqrt(sq(wall1->normal->x) + sq(wall1->normal->y)
                          + sq(wall1->normal->z));
  dot_product = fabs((wall1->normal->x)*(dif_vec->x) +
           (wall1->normal->y)*(dif_vec->y) + (wall1->normal->z)*(dif_vec->z));
  distance = dot_product/mag;
  return distance;
}

vector *find_force_objs (object *obj1, object *obj2) {
  double mag;
  double distance;
```

```
  double dir_x;
  double dir_y;
  double dir_z;
  distance = find_distance(obj1->c_of_mass, obj2->c_of_mass);
  if (distance <= 0.2) {
    mag = IMPACT;
    printf("an impact occurred between two objects \n");
  } else if (distance <= 0.6) {
    mag = (-1.0)*(Van_constant/ distance);
    printf("objects too close using repuslive force \n");
  } else {
    mag = ((obj1->mass)*(obj2->mass)*G_grav_constant)/ (sq(distance));
  }
  dir_x = obj2->c_of_mass->x - obj1->c_of_mass->x;
  dir_y = obj2->c_of_mass->y - obj1->c_of_mass->y;
  dir_z = obj2->c_of_mass->z - obj1->c_of_mass->z;
  force->x = dir_x*mag; force->y = dir_y*mag; force->z = dir_z*mag;
  return force;
}

vector *find_force_obj_wall (object *obj, wall *wall1) {
  double distance;
  double mag;
  distance = find_dist_wall_obj(obj, wall1);
  if (distance <= 0.1) {
    mag = IMPACT;
    printf("an impact occurred between the wall and an object !!! \n");
  } else {
    mag = ((obj->mass)*(wall1->mass)*G_grav_constant)/ (sq(distance));
  }
  /* the direction is in the normal of the wall */
  force->x = (wall1->normal->x)*mag;
  force->y = (wall1->normal->y)*mag;
  force->z = (wall1->normal->z)*mag;
  return force;
}

void update_object (object *obj) {
  int i;
  vector *new_force;
  cur_force->x = 0.0;  cur_force->y = 0.0;  cur_force->z = 0.0;
  for (i = 0; i < NUM_WALLS; i++) {
    new_force = find_force_obj_wall (obj, walls[i]);
    cur_force->x = cur_force->x + new_force->x;
    cur_force->y = cur_force->y + new_force->y;
    cur_force->z = cur_force->z + new_force->z;
  }
  for (i = 0; i < NUM_OBJECTS; i++) {
    if (obj != objs[i]) {
      new_force = find_force_objs (obj, objs[i]);
      cur_force->x = cur_force->x + new_force->x;
      cur_force->y = cur_force->y + new_force->y;
```

```
      cur_force->z = cur_force->z + new_force->z;
    }
  }
  acc->x = (cur_force->x)/(obj->mass);
  acc->y = (cur_force->y)/(obj->mass);
  acc->z = (cur_force->z)/(obj->mass);
  obj->c_of_mass->x = (0.5)*(acc->x)*sq(dt) +
                      (obj->vel->x)*dt + obj->c_of_mass->x;
  obj->c_of_mass->y = (0.5)*(acc->y)*sq(dt) +
                      (obj->vel->y)*dt + obj->c_of_mass->y;
  obj->c_of_mass->z = (0.5)*(acc->z)*sq(dt) +
                      (obj->vel->z)*dt + obj->c_of_mass->z;
  obj->vel->x = (acc->x)*dt + obj->vel->x;
  obj->vel->y = (acc->y)*dt + obj->vel->y;
  obj->vel->z = (acc->z)*dt + obj->vel->z;
}

void find_all_positions (void) {
  int i;
  for (i = 0; i < NUM_OBJECTS; i++) {
    if(objs[i]->moving_obj == TRUE) {
      update_object( objs[i] );
    }
  }
}
```

Appendix C: Locations of Demo Files and Other Related Example Files

A copy of this file can be found on-line at: http://www.cs.berkeley.edu/~ug/WALK-KIN/AppendixC.html

1- Tutorial for UGMovie application and compilation script: (Section 2 and 3)

   http://www.cs.berkeley.edu/~ug/WALK-KIN/ugmovie.html

2- Location for the two kinematic sculptures `hingebug` and `cubogear` (Section 4).

   http://www.cs.berkeley.edu/~ug/WALK-KIN/kinematic/sculptures.html

3- Location for the kinematic gravity simulation example (Section 4), also see Appendix B.

   http://www.cs.berkeley.edu/~ug/WALK-KIN/kinematic/gravity.html

4- Location for the multi-cube example (Section 4).

   http://www.cs.berkeley.edu/~ug/WALK-KIN/interactive/mcube.html

5- Interactive juggling sculpture object example (Section 5), also see Appendix B.

   http://www.cs.berkeley.edu/~ug/WALK-KIN/interactive/juggling.html

6- Re-implementation of BUMP sculpture objects as interactive elements (Section 5).

   http://www.cs.berkeley.edu/~ug/WALK-KIN/interactive/sculptures.html

7- Interactive video `viewscreen` object example and notes (Section 5).

   http://www.cs.berkeley.edu/~ug/WALK-KIN/interactive/viewscreen.html

8- WALK-KIN extension notes and `wkadd` tutorial (Section 4).

   http://www.cs.berkeley.edu/~ug/WALK-KIN/walk_kin.html