An Integrated Graphical Simulation Platform

David Zeltzer, Steve Pieper and David J. Sturman
Computer Graphics and Animation Group
The Media Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

This paper describes an ***integrated graphical simulation platform*** (IGSP) which provides a framework for constructing interactive simulations, specifically those oriented towards ***task level animation*** — that is, animation for which the user specifies the tasks to be performed and the system determines the correct sequence of events and selection of tools to use in accomplishing that task. Our prototype system, which we call bolio, allows diverse applications to interact within a run-time environment, displaying their results on a common 3D graphics platform. Here we describe aspects of bolio's design that allow applications to interact through a common network of constraints, including an integrated suite of tools that simulate kinematic, dynamic, and event-driven processes in virtual worlds. In addition, the IGSP architecture allows us to easily integrate gestural input from a new device — the DataGlove — that allows non-expert users to manipulate virtual objects directly in these microworlds.

KEYWORDS: computer animation, simulation, constraints, interaction techniques.

1. Introduction

Previous analyses of interactive, 3D graphics systems have focused on the specification of graphical primitives and viewing parameters in terms of the programming interface — e.g., ACM Core, GKS, PHIGS+; the specification and design of interactive user/computer dialogs — i.e., user interface management systems (UIMS)[12, 20, 27]; or various paradigms for "software testbeds" for developing rendering tools[5, 19, 24, 33]. In this paper we will present an analysis of the requirements of an ***integrated graphical simulation platform*** (IGSP) intended to support a suite of behavior modeling tools and applications for the purpose of making complex 3D simulation of agents and objects accessible to non-expert users. We call this ***task level animation***[39]. The requirements of a task level IGSP include those of the UIMS, for example, but require, in addition, consideration of mechanisms for managing interaction among simulated autonomous agents and physical processes. In view of recent interest in developing a variety of mechanisms for simulating the behaviors of agents and objects, e.g., "physically-based modeling" of non-rigid objects[30, 31], simulation of the dynamics of rigid (possibly articulated) bodies[13, 34], and modeling of complex behaviors[10, 14, 37] we think it is timely to begin discussion in the graphics community of the notion of an IGSP.

This work was supported in part by NHK (Japan Broadcasting Corp.), the Institute for Computer Literacy, and an equipment grant from Hewlett-Packard, Inc.

First we describe in general terms what we mean by an IGSP, and describe the procedural elements necessary to support a task level graphical simulation environment. We then present the *manus* constraint network, which, in our implementation, is the mechanism for associating procedural elements with objects, and for defining and satisfying invariant relationships (i.e., constraints) among simulated agents and objects. We conclude by describing tools we have implemented for interacting with the virtual worlds which our prototype system supports.

2. Requirements of an IGSP for Task Level Control

In earlier work[39], three modes for controlling the behavior of animated, simulated objects were identified: ***guiding, programming*** and ***task level,*** defined roughly as follows:

- ***guiding:*** explicit specification of behavior over time by the user;
- ***programming:*** specification of behavior in some programming notation, possibly a special-purpose animation language, e.g., ASAS[25], or MIRA[16].
- ***task level:*** implicit specification of behavior in terms of goals, events and constraints.

In general, guiding systems — e.g., *twixt*[11] and *bbop*[35] — are typified by the interactive specification of key transformations using various graphical input devices, usually accompanied by built-in procedural support (e.g., linear and cubic interpolation) for automatic in-betweening. While the distinction between the guiding mode and the other two is rather intuitive, the distinction between programming and task level control is less clear cut. Nevertheless, the notion is that at the programming level the user interacts with the system in terms of the abstractions provided by the virtual *machine*, i.e., the graphical and procedural primitives of the implementation language. At the task level, interaction is in terms of the virtual *world*, i.e., names and behaviors of simulated agents and objects, such as walking, grasping, or possibly complex goal-seeking activity. We would control a simulated human figure at the task level, for example, by telling it to "Walk to the door," rather than writing a script describing the needed motions in detail.

The implementation of a task level system requires what we call *adaptive motion* and an appropriate set of abstraction mechanisms[38, 39].

2.1. Adaptive Motion and Object Representation

Adaptive motion means simply that the motion of an object depends on information about the current state of the object and the current states of other objects and processes in the environment. This facility is fundamental to physically-based simulation and other animation techniques where the interaction between simulation processes determines the behavior of objects. In our IGSP implementation, object attributes and geometric representations are globally accessible through a set of standard functions, so that object state information is available to all simulation processes that require it.

Since we can't predict in advance what conceptual views of an object will be required in a given situation, an IGSP must abide by the *principle of least commitment* with respect to common, globally-accessible object representations. I.e., the representation should enforce no particular view of what an object is. Object representations in a simulation environment are in principle open-ended: we may wish to model things that are stationary, rigid and monolithic; as well as objects that are deformable, or that make up parts of more complex assemblies — including human, robot or animal figures — in which the sub-parts are interrelated in many ways. In addition, dynamic simulation imposes a set of time-varying interactions among objects.

As detailed below (section 3.2), we maintain a geometric representation for every object in bolio; other kinematic and dynamic information is computed from that data. A process which, for example, applies forces to objects based on collisions, may maintain private state information about those objects — e.g., saving relative velocity information in order to compute appropriate reaction forces. This representation paradigm is simple and economical, since global representations aren't encumbered by all the details that every simulation process might require, and because more expensive computations on object state are invoked only by processes that require them, at the time that data is required. While some computations may be duplicated at runtime by processes, for example, calculating world space bounding volumes, generality is gained by keeping global representations simple. In fact, applications may maintain their own internal representations of the objects and processes they model, but ultimately, all applications affect the virtual world through the primitive object representations.

2.2. Abstraction Mechanisms: Structural, Procedural and Functional

Some *structural abstraction* mechanism must be provided for describing the kinematic structure of objects — e.g., a transformation hierarchy for a jointed figure — and defining physical attributes — e.g., mass, stiffness, etc. Such a structural definition serves as a data structure with no functional specifications for how an object should move. While bolio maintains only a minimal geometric representation for objects, the *manus* constraint mechanism, described below, makes it possible to associate structural and dynamic properties, e.g., linkages and springs, as necessary with the geometric description. Application modules, moreover, such as *sa* for jointed-figure simulation, and *roach* for hexapod locomotion, may maintain their own internal structural definitions as needed. It is important to note, however, that no matter how complex these modules may be, they interact solely through the common structural abstractions which describe the objects in the terms appropriate to the processes being modeled.

Procedural abstractions are the mechanism for defining processes that control object motion. In bolio, at the guiding and task level, procedures are treated as built-ins. At the programming level, a consistent and modular function library is available so that programmers have access to the primitive operations for kinematics, dynamics, and constraint definition and satisfaction that bolio provides. This is important both for coding routines for controlling object motion, as well as for incorporating application modules into the IGSP.

Just as object representations are open-ended, it is not possible to specify in advance a complete set of procedures for a task level simulation platform. This is because we often don't know in advance the details of those processes which we may wish to visualize, and also because the representation and computation of many physical processes is not completely understood[1, 17]. Nevertheless, we can identify a useful set of procedures for supporting a wide range of processes. These include inverse kinematics, collision detection, path planning, and certain kinds of dynamic simulation, all of which are present in bolio. These procedural tools are the basis for composing behaviors of simulated objects. Recall that the distinction between the programmer level and task level simulation really forms a continuum: at the purely programming end of the spectrum we deal with the procedural primitives of the implementation language; at the task level we deal with the conceptual entities of the simulated model. In our view, while the distinction is not clear cut, we feel that if the procedural support tools are rich enough, we can achieve a qualitative change in our interaction with the virtual world.

Functional abstractions are the means with which we associate procedures with objects and object subassemblies, for the purpose of defining meaningful behaviors. For example, we can define a grasping and reaching behavior by controlling the motion of a kinematic chain using inverse kinematic procedures in a given context. This has the effect of simplifying the control of complex systems with many degrees of freedom. If we know the operation to be performed in a certain situation, we can apply functional constraints so that the system can be controlled with far fewer parameters to perform a specific task. In our IGSP implementation, we support functional abstraction in two ways:

- The *manus* constraint system provides a built-in mechanism for associating objects and processes in order to define behaviors.
- Application modules may have their own internal representations and functional abstraction mechanisms, can be freely incorporated in bolio, by means of both bolio's object and graphical primitives, and by the use of *manus* for communicating with other applications and bolio processes.

2.3. Forward Simulation and Key Event Simulation

Recent work has described methods for generating animation using *key events* over time in terms of forces, torques, dynamic and kinematic constraints[4, 15, 36]. As long-time Disney animator Frank Thomas has pointed out, the creation of conventional animation can proceed in two ways, either as *key frame* animation, which involves the well-known in-betweening technique; or as *straightahead* animation, in which the animator draws frame after frame of a sequence with only a general idea of the unfolding action[32].

While the analogy only loosely holds, we view computer animation and simulation in a similar two-fold way. In some cases, we may wish to visualize a known sequence of events, and here the appropriate paradigm is what we have called *key event* simulation, as addressed by the above-mentioned authors. At other times however, as in the simulation of the performance of a vehicle, or in studying the behavior of a simulated insect for the purpose of validating a hypothesized locomotion control technique, we may have little interest in imposing outcomes on a sequence — except perhaps through user interaction, as the simulation evolves. We call this view *forward simulation*, and it is characterized by incomplete knowledge of events and the need to observe *autonomous* behaviors and interactions among objects. The two views are certainly not mutually exclusive, since, for example, we may wish a simulated vehicle to follow a particular path and perform certain operations in the course of a simulation. That is, we want to define motor goals which result in some desired behavior, but in a forward simulation they take the form of motor control programs and feedback loops.

2.4. Mixed Methods for Behavior Control

One further issue needs to be addressed in the context of an IGSP, namely, the interaction of kinematic control and dynamic control. This, in particular, is a user interface issue, since we want the effects of user input to be consistent, and not violate the user's assumptions and expectations about the simulation.

- Dynamics/Dynamics
  If all processes in a simulation speak a common language, as it were, and always operate on objects in terms of forces and torques, the outcome is a purely physical simulation. If the numerical, computational and sampling issues are properly addressed, then the motion of objects will be based on physical principles with a degree of realism corresponding, for the most part, to the level of detail represented by the simulated objects and processes. In this case, user input, whether through conventional graphic input devices or other sources, should be transformed into appropriately scaled forces and torques.
- Kinematics/Kinematics
  In a strictly kinematic simulation there is no longer any commitment to modeling Newtonian physics. In such a situation, the intentions of the user are the dominant factor in determining the order of operations and the level of realism desired. For example, in assembling parts of a mechanism for the purposes of design validation and interference checking, the user may be interested only in rigid motion and collision detection, with little interest in constraining the motion of objects to be based on classical mechanics.
- Kinematics/Dynamics
  If the simulation involves a mixture of kinematic and dynamic operations — for example, combining kinematic gestural manipulations using the DataGlove with dynamically controlled objects — then the simulation no longer follows Newtonian physics. Realistic motion cannot be guaranteed, since kinematic operations may, for example, cause velocity discontinuities. In this case, the intentions of the user again become the deciding factor, and it becomes necessary to prioritize the order in which forces and torques are applied to objects in concert with strictly kinematic operations. Note that the emphasis here is on combining interactive kinematic input with dynamic simulation, unlike the work reported by Cohen and Isaacs[15], which described methods for combining *a priori* kinematic constraints with dynamic simulation.

Bolio at present is a kinematic/dynamic environment, in which kinematic gestural input is combined with forward dynamic simulation.

3. Bolio

Bolio is our prototype IGSP. It serves as a common base for the development of simulations by providing an input/simulate/draw loop into which new applications can be incorporated. Bolio provides an object level interface to a graphics environment, and is implemented in C on Hewlett-Packard 9000 workstations running the HP-UX operating system, a derivative of AT&T System V UNIX†.

3.1. User Interface: Commands and Scripts

Bolio maintains an internal string buffer and parsing mechanism, so that the user can freely intermix input from the keyboard, conventional graphical input devices, or from other sources such as the DataGlove (described below). Typically, in the bolio task level environment, the user is wearing a DataGlove, so that his or her interaction centers on occasional menu picks, direct manipulation of objects and invocation of processes via specific hand postures. This is in contrast to the developers interface, in which application and environment developers make heavy use of keyboard and mouse — often creating complex scripts for testing and debugging various simulation modules, or for defining virtual environments and agents. Interactive procedures using mouse and knob box are available to developers for setting parameters of objects in the virtual environments they create. Specific details of the bolio user interface have been described elsewhere[3], and will not be discussed further here.

3.2. Bolio Objects

The central data structure for the exchange of information between applications is the bolio object. It contains fields to describe attributes (name, transformation matrix, bounding box...) common to all objects appearing in the microworld. This is the only information needed by most bolio applications.

Some bolio operations such as reading and writing objects to disk files, and graphical display of objects require more type-specific information. This is available through the `description` field of the `bOBJECT` structure, a generic pointer to a specific object-type recognized by bolio. These structures (`bPOLYHEDRON`, `bCAMERA`, `bLIGHT`, `bDEPTHMAP`, etc.) contain data relevant only to their object type. The `bPOLYHEDRON` structure, for example, contains lists of the vertices, polygons, and edges which define the polyhedron. While we don't argue that all information required for potential interaction between simulated characters can be represented by the mainly geometric information in the `bOBJECT` structure, we have found that it works well for the types of simulations we have envisioned and implemented thus far.

Bolio uses a two stage file format for `bOBJECTs` (originally developed at Ohio State University) which reflects the distinction between the `bOBJECT` level and the `description` level[5]. Generic information about `bOBJECTs` is stored in files with a `.obj` extension. These are ASCII text files which contain keyword/value pairs to define fields such as the object's name, its initial transformation matrix, and the data type and source file of its description. Since this file is generally very short, the entire text is kept in memory. The `detail` keyword in the `.obj` file is followed by the name of a file from which to read the type-specific description of the object. The description is read by routines specific to that type. These routines build the appropriate data structure for the type. A pointer to this data structure is stored in the `description` field. Example detail file types include `.asc` for an ASCII representation of an object of type `bPOLYHEDRON`, `.det` for a binary description of an object of type `bPOLYHEDRON`, and `.dm` for an object of type `bDEPTHMAP`. This structure provides a simple and general method for representing a wide variety of object types.

To make the rendering of device independent graphical objects more efficient, they are compiled into a format specific to the hardware platform. This device specific data is stored in `drobjs` (drawing objects) structures. A `bOBJECT` structure contains a list of pointers to the `drobjs` which define the hardware calls needed to display it on the screen. These `drobjs` are created by the compile routine specific to the `bOBJECT` description and are in a format dependent on the type of output device to be used for rendering. Currently, a set of data structures is used which is specific to the Starbase graphics system running on the Hewlett-Packard 9000 series of workstations.

A view of the bolio world is generated according to the parameters stored in a camera object. Some of the view information (e.g. the view point, view normal and view up) is extracted from the information stored at the `bOBJECT` level, while other view parameters (such as the camera field of view) are stored in the `bCAMERA` referenced by the `description` structure. This arrangement makes it possible to use standard matrix inheritance constraints to attach the camera to any object in the scene, such as the eye of a simulated character, or the position of the DataGlove.

4. The *Manus* Constraint System

A constraint package was a key element of Sutherland's classic work, *Sketchpad*[29], and of Borning's *Thinglab*[2]. Our work with constraints is similar in spirit to both of these systems, although both of them were restricted to 2D graphics. All three systems incorporate rather general mechanisms for defining constraints and constraint satisfaction methods. However, the two earlier constraint systems incorporate an analysis stage, and Borning's work included two additional satisfaction techniques beyond one-pass solutions and relaxation.

Manus was developed initially to handle position and orientation constraints on the motion of rigid objects, and non-rigid motion of polygonal meshes. Thus, unlike the earlier *Sketchpad* and *Thinglab* systems, which were intended to satisfy multiple, interacting constraints encountered in geometric and mechanical design problems, bolio does not perform preliminary analysis of the constraint network. Since relaxation is time-consuming and may not converge, the purpose of this constraint planning step is to identify constraints that can be satisfied by simpler, direct means, so that relaxation is invoked only when necessary. However, bolio supports an interactive, time-varying virtual environment, perhaps with active agents whose behavior may not be known *a priori*. Thus,

†UNIX is a trademark of AT&T Bell Laboratories.

constraint satisfaction has to proceed in parallel with forward simulation, and a constraint pre-planning stage is not feasible.

The *manus* constraint network is composed of bOBJECTs in the bolio world connected by instances of constraints. Each instance of a constraint contains information specific to the objects it is connected to and pointers to the code necessary to process the constraint. Thus, constraint instances share procedures but maintain private copies of relevant data structures.

Each time a constraint instance connects to a bOBJECT which should trigger it, it adds a pointer to itself into the bOBJECTs *who-cares* list (part of the **constraints** structure). Later, when a constraint instance modifies the bOBJECTs, the bOBJECT notifies all constraint instances in its *who-cares* list. Those constraint instances then execute, modifying other bOBJECTs which trigger constraint instances in their *who-cares* list, etc. This process proceeds in an manner managed by the *manus_renormalize* function.

When a bOBJECT triggers constraint instances in its *who-cares* list it actually just puts a pointer to each instance on the end of a global *pending constraint instance list* (*pending queue*). The *manus_renormalize* function goes sequentially through the queue (in effect a breadth-first search of the constraint network) invoking constraint instances as they are pulled from the list. As constraint instances execute, objects they affect place new items at the end of the queue. This procedure continues until the queue is empty. We use several methods to ensure termination of this process:

- Carefully construct constraint networks so as to avoid loops.
- Allow a particular constraint instance to be placed on the queue only *once* per frame. This technique is used by the DataGlove constraints which only sample the external device once per frame.
- Program constraints such that their instances only modify their dependent bOBJECTs once per frame, treating all subsequent constraint invocations as interrupts of the forward simulation of the object's motion, which then instantaneously update the object's position and orientation. E.g., if the user catches a bouncing ball with the DataGlove, the glove constraint causes the simulation of the ball's motion to suspend; the ball is repositioned according to the glove constraint. Constraints are prioritized in the order in which they are invoked.
- If none of these methods prove flexible enough to handle the interactions of several constraints which all wish to control a single bOBJECT, the final resort is to express the constraints in terms of forces and let the solution evolve over time via forward simulation. This is used, for example, in the case of interpenetration prevention where several constraint instances may all influence the position of the same bOBJECT.

The *manus_renormalization* function is invoked at three points within each iteration of the bolio's main loop. Two special constraint structures exist for the start and end of each iteration, allowing events such as device sampling to be triggered by the start of a new frame or control of a video tape recorder to be signaled by the end of a frame. The renormalization operation is performed after each of these signals is sent. Renormalization is also performed after the command portion of the main loop, between the start and end of the frame.

An example *manus* operation (or *mop*) is the *link* constraint. This *mop* updates the size and position of a "link" object so that it appears to physically connect two other objects. The code which satisfies an instance of this constraint looks at the positions of the two objects and calculates an appropriate transformation matrix for the link object. When either of the two objects is moved, the *mop* is re-executed to properly transform the link object. The constraint instance contains a pointer to the code to calculate the appropriate transformation for the link object given the positions of the other two objects, and a structure containing pointers to the object to be used as a link, the two objects to be linked, and flag telling whether the linking transformation should be volume conserving.

5. Bolio Tools

Applications incorporated into the bolio system and optional libraries of constraints are called bolio tools. Examples of bolio tools include a camera manipulation and animation package, an autonomous hexapod (***roach***), an inverse kinematic constraint package, a set of dynamics simulation tools, the DataGlove device handlers, and a path-planning module. These all link into the main bolio system and communicate with each other through the constraint network.

At the UNIX shell level, a configuration file called BOLIOTOOLS contains a list of tool sets to be included in the version of bolio being compiled. The code for each tool is contained in a separate subdirectory of the bolio directory hierarchy. Each tool directory has a makefile that builds a library (UNIX archive file) containing that tool's code. For each included tool, bolio's compile script executes the makefile and reads three configuration files from the tool's directory; the **bopnames** file, which has a list of the main loop commands for the tool; the **mopnames** file, containing a list of the constraint commands for the tool; and the **usrlibs** file, which contains a list of other system libraries which must be linked with the tool library into the bolio executable. The first two files are used to construct branch tables for the main command command parser and to allow the constraint command parser to map input strings into function calls. The third file is used to add arguments to the command which links the executable bolio. In this way independent libraries and applications are kept separate and compiled in only as needed. Optimally, bolio would contain a run-time loader that linked in code libraries as they were needed.

5.1. Core Tools

Functions that are of general use to bolio users regardless of the particular bolio application are grouped together in a library of *core tools*, which are included in all linked versions of bolio. The current set of core tools includes a range of useful interactive graphics utility commands to manipulate object and viewport structures; e.g. using the mouse for moving and sizing viewports, changing camera parameters, transforming and changing colors of objects, or positioning lights, and a set of general purpose constraints such as object transformation hierarchies, bounding box representations of objects, and graphical links between objects (e.g. rubberband lines). As new users and developers enter the system, they tend to add new functions to the core tool library.

5.2. Dynamics Simulation

Bolio's dynamics simulation module was developed as part of a project to model the deformation behavior of human soft tissue, particularly in the facial region, taking into account muscle forces and the underlying skeletal structures[23].

The simulation module uses a network of data structures representing the physical properties of material samples in the tissue including mass, position and velocity. The material samples are connected by springs which apply forces to maintain rest distances. Other constraints on the material samples include gravity, which simply applies a constant downward force, and interpenetration prevention. The latter is accomplished by computing repulsive forces, triggered whenever a sample point enters a forbidden region. These regions can be defined by the boundaries of the microworld, by the bounding sphere of a bOBJECT, or by the surface of a bDEPTHMAP.

The primary *dynamic simulation* constraint attaches a structure to a bOBJECT which maintains material sample information. If there is no explicit modification of the bOBJECT by outside actions, such as the DataGlove or *roach* module, then the position and orientation of the bOBJECT is modified based on the current velocity, position, and forces generated by dynamic constraints. The *spring* constraint applies forces to two dynamic structures (attached to bOBJECTs) based on spring and damper constants, a rest length, relative separation and velocity between the two bodies, etc. The *gravity* constraint applies forces to objects based on a global gravitational constant. These forces, together with the interpenetration prevention forces, are integrated into new object positions by the *clock* constraint. The latter is fired by the *frame-start* event and runs a specified number of integration time steps. The independence of the rendering frame count and dynamic simulation time step allows animation of dynamic simulations to trade-off speed and accuracy, depending on the needs of the moment.

As described above, non-rigid objects are modeled as networks of these points and springs, which deform according to the influence of forces. simulation of rigid bodies and their interactions can be modeled with appropriate combinations of the various *dynamic simulation* constraints. Each iteration through the dynamics loop (or group of iterations) provides another frame in the animation. In one example, the hexapod pushes objects out of its path as it wanders the microworld landscape. In another example, the DataGlove can be used to pick up an assembly of balls and springs and whirl them through the virtual air.

5.3. Autonomous Hexapod

Our research on the simulation of human and animal locomotion includes a control structure for six-legged locomotion — called *roach* — based on biological research into the neural mechanisms found in insects and mammals[9, 18, 22]. The control system is based on coupled oscillators that coordinate the action of the legs to form appropriate gait patterns for a given speed. Within the overall pattern, simulated reflexes generate changes in state for each leg. The implementation uses inverse-kinematics to bend the "knees" correctly according to the current body location and desired foot position.

Communication between the bolio environment and the *roach* module is accomplished through two constraints. The first is *roachwalk*, an instance of which exists for each hexapod in the environment and is triggered by the *frame-start*. *Roachwalk* sets the transformation matrices defining the positions of the hexapod's constituent parts for the current frame and triggers any constraint instances which depend on those parts. The second constraint, *roachorient*, provides a communication path *from* the bolio environment *to* the *roach* modules through the objects representing the hexapod's body. Any time the hexapod's body is moved the *roachorient* constraint sends the new location of the hexapod body to the *roach* module, which updates its internal data structures accordingly. Thus, for example, if the DataGlove picks up the hexapod body, the *roach* module will get the new location and ensure that the hexapod's legs are moved also.

The results of simulated physical interactions also are propagated to the *roach* module through the *roachorient* constraint; the dynamic simulation tracks the position of the hexapod via the constraint network and modifies the position of the body in accordance with the forces that affect it. These forces include gravity, ground forces, and collision forces generated when the hexapod walks into a wall or other objects. In this way the hexapod becomes subject to any laws of dynamics within a bolio simulation. The hexapod can also affect the environment through the constraint structures, for example, pushing objects out of the way when a collision is detected.

Commands coming through bolio's standard input which begin with the word roach are passed to the *roach* module's input parser. Thus it is easy to mix *roach* initialization and control commands into scripts which set up and modify bolio environments.

5.4. Robot: Inverse Kinematics

A set of routines to perform inverse-kinematics on jointed figures are provided by the *robot* tool. The routines allow the specification of jointed figures using the

Denavit-Hartenberg (DH) description conventions which define kinematic linkages in terms of the relative transformations between neighboring joint-centered coordinate systems[6, 26]. This information is embodied in a Jacobian matrix, whose pseudo-inverse provides the joint angle velocities needed to achieve given end effector velocities[7, 21].

Robot is linked into bolio via the *ik* (inverse kinematics) constraint which operates on a list of bOBJECTs which will serve as joints for an articulated figure. The DH description of the figure is calculated based on the initial locations of the objects in the list. The first item on the list is used as the kinematic base, the last item is the used as the end effector, and the remaining items are used as the ordered set of intermediary joints. Any time the base object or end effector object is moved, the *robot* module calculates new positions and orientations for the intermediate joint bOBJECTs as if they were joined by rigid links. The links themselves do not appear unless constructed separately by, say, the *link* constraint.

Because the *ik* constraint is based on the end effector and base bOBJECTs of a kinematic linkage, the pose of a jointed figure can to be manipulated by any bolio tool (e.g. the DataGlove or dynamic simulation) that modifies the position of those bOBJECTs.

5.5. The DataGlove

We have developed a standard method of dealing with input to bolio using an event loop that triggers constraints on objects. This standardized method has made it extremely simple to incorporate novel devices into the system. The DataGlove is one such device.

For the past year we have been experimenting with a VPL DataGlove which records the position, orientation, and finger postures of a human hand[8, 28, 40]. It transmits this information through a serial communication line to the host computer up to 60 times per second, more than sufficient for our current needs.

The glove is incorporated into bolio via the *glovepoll* constraint. An instance of *glovepoll* is attached to the *frame-start* structure so that the DataGlove device driver is given a chance to execute every time bolio runs through its main loop. *Glovepoll's* code updates an internal structure (the `struct glovepoll_data`) containing the current DataGlove values; instances of other constraints that depend on the values in the *glovepoll_data* structure are triggered whenever those values are updated. One constraint dependent on the *glovepoll_data* structure is the *dghand* constraint, which transforms a set of objects so that their screen position matches the position and orientation information supplied by the DataGlove, thereby providing a screen echo of the hand.

The *glovepoll* constraint also checks the current finger-bend values against a user-defined posture table and sets a value in the `glovepoll_data` structure indicating the current hand posture. For example, when the *grasp* posture is recognized, the *grabber* constraint finds the nearest object (if it is within a threshold distance) and constrains it to track the location of the "grabbing" object (usually the index finger tip, whose position is being controlled by the DataGlove through the *dghand* constraint). Note that moving the "grabbed" object invokes any constraint instances dependent on its position. With this behavior, it is possible to use the DataGlove to interact with objects in the virtual world. In fact, any bolio tools which use position and orientation of objects as input (like the *robot* tool, or *dynamic simulation*) can make use of the glove as an input device.

Another glove constraint which depends on the sampled data in the *glovepoll_data* structure is the *glove_cursor* constraint. When this constraint detects the appropriate posture, it maps finger bends into cursor movement up and down the bolio menu. By making a hand movement, the glove wearer can cause this constraint to pass a button-press event to the menu code, and thereby select the menu item. With this constraint, the entire system can be controlled through posture and position input from the glove rather than the keyboard and mouse.

5.6. sa

sa is a figure animation system developed by Zeltzer in 1984 to describe and manipulate jointed figures via an event-driven simulation mechanism. It includes a special animation language for controlling jointed figures[37, 39]. *sa's* simulation mechanisms take into account the articulated structure of a walking figure, the current gait, and the support requirements to place the figure's limbs in space. It has successfully been used to animate a walking, jumping skeleton, George[38].

The task of incorporating *sa* into bolio involved replacing *sa's* graphics output routines with calls to bolio's graphics utilities, providing a way for bolio users to interact with *sa* through it's own animation language, and developing the appropriate constraint structures to permit interaction between bOBJECTs controlled by bolio and bOBJECTs controlled by *sa*. Replacing the graphics routines was a straightforward, easily accomplished task. Providing a method for users to access *sa's* animation language was handled, as with the *roach* module, by providing a keyword to bolio's command processor that would pass the remainder of the instruction to *sa's* internal command parser. Providing the constraint modules to permit smooth interaction between bolio's control of objects and *sa's* is underway. Currently the transformation matrices which define the positions of the figure's parts are used by *sa* to position the bOBJECTs which represents the parts. However *sa* receives no feedback from bolio through the constraint network. When the integration is complete, agents in the bolio environment will be able to fully interact with *sa* characters and vice-versa. For example, we will be able to use the DataGlove and inverse kinematics to shake hands a with virtual human figure. Future work will use *manus* to allow *sa* to take advantage of the inverse-kinematics and dynamics simulations in bolio.

6. The Roach and Glove Microworld Example

A more complete example may clarify how these constraints combine to produce an interactive simulation. The Roach and Glove demo is a set of bolio scripts and menus which make use of the most advanced features of the system. The world consists of a hexapod and various objects (cubes and soccer balls) on a grid floor. The hand of the user (who is wearing the DataGlove) floats in space among the objects. By forming the *grab* posture, a constraint is triggered that draws a red line from the current glove location to the nearest of the objects in the scene. This serves as a visual cue to indicate the nearest object to grab. If the user moves so that the glove touches an object the glove's *grabber* constraint causes the object to track the motion of the glove; stick to the glove as it were. This tracking continues until the user leaves the *grab* posture. Thus, the user can pick up and throw objects around in the virtual space.

A set of scripts can be invoked to link a group of objects together with dynamic springs. When the user *grabs* one of the objects in the group the others are realistically dragged along by the spring forces. Interestingly, on a fast machine the illusion is convincing enough that the user expects to feel the inertia of the objects, and the lack of tactile feedback is disturbingly apparent.

The hexapod randomly traverses the ground plane using simple collision detection and dynamic constraints to push obstructing objects from its path. When the user's hand is in the *follow* posture, a path is drawn from the roach to the the glove and the roach is redirected to walk along that path. The path is updated every frame that the user is in the *follow* posture, thereby allowing the user to lead the hexapod around the world. It is also possible to pick up and reposition the roach by using the *grab* posture.

Another set of scripts attaches the eyepoint to the hexapod body to achieve a 'roach-eye view' of the world. The local origin of the DataGlove is also attached to the local origin of the hexapod. The effect is that of riding the hexapod with one's hand extended in front, pointing out the direction of travel, grabbing and throwing obstacles out of the way.

7. Conclusion

In this paper we have suggested a conceptual framework for the integration of graphical simulation tools, some of which, e.g., dynamic simulation, are receiving increasing attention in the graphics community. We call this framework an *integrated graphical simulation platform* (IGSP), we have described the requirements for a task level IGSP, and we have described our prototype implementation, bolio.

We are taking a breadth-first approach in our development of bolio. While initially the virtual worlds we model may be sir^ple, at every stage of the development process we have a working system that increases in complexity as simulation components are added. Moreover, at each stage users can interact with virtual objects at the task level, often using gestural input from the DataGlove. This provides a rich environment for testing and refining new behavior control modules — taking full advantage of existing tools.

Future work will include increasing the complexity of virtual worlds by distributing simulation processes among a set of networked compute platforms, as well as research aimed at modeling the planning and problem solving behaviors of autonomous robotic and human agents.

8. Acknowledgments

Many people have contributed much time and effort to the development of bolio. Much of the initial implementation is due to Cliff Brett. Dave Chen provided the inverse kinematic tools, Mike McKenna wrote and adapted the hexapod simulation, and Peter Schroeder developed path planning and camera specification tools. Our Media Lab colleagues and many visitors provided invaluable comments and advice, as well as serving as willing guinea pigs and "demo-breakers".

References

1. R. Alexander and G. Goldspink, *Mechanics and Energetics of Animal Locomotion,* John Wiley & Sons, New York (1977).
2. A. Borning, "Thinglab – A Constraint-Oriented Simulation Laboratory," Tech. Report No. SSL-79-3, Xerox PARC, Palo Alto, CA (July 1979).
3. C. Brett, S. Pieper, and D. Zeltzer, "Putting It All Together: An Integrated Package for Viewing and Editing 3D Microworlds," *Proc. 4th Usenix Computer Graphics Workshop*, (October 1987).
4. L. Shapiro Brotman and A. Netravali, "Motion Interpolation by Optimal Control," *Computer Graphics* **22**(4) pp. 309-315 (August 1988). Proc. ACM SIGGRAPH 88.
5. F. C. Crow, "A More Flexible Image Generation Environment," *Computer Graphics* **16**(3) pp. 9-18 (July 1982). Proc. ACM SIGGRAPH 82.
6. J. Denavit and R. B. Hartenberg, "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices," *Journal of Applied Mechanics* **23** pp. 215-221 (June 1955).
7. A. Ferdman, "Robotics Techniques for Controlling Computer Animated Figures," M.S.V.S Thesis, Massachusetts Institute of Technology (August 1986).
8. J. D. Foley, "Interfaces for Advanced Computing," *Scientific American* **257**(4) pp. 126-135 (October 1987).
9. C. R. Gallistel, *The Organization of Action: A New Synthesis,* Lawrence Erlbaum Associates, Hillsdale, New Jersey (1980).
10. M. Girard and A.A. Maciejewski, "Computational Modeling for the Computer Animation of Legged Figures," *Computer Graphics* **19**(3) pp. 263-270 (July 1985). Proc. ACM SIGGRAPH 85.

11. J. E. Gomez, "Twixt: A 3-D Animation System," *Proc. Eurographics '84*, North-Holland, (September 1984).

12. M. Green, "The University of Alberta Interface Management System," *Computer Graphics* **19**(3) pp. 205-213 (July 1985). Proc. ACM SIGGRAPH 85.

13. J. K. Hahn, "Realistic Animation of Rigid Bodies," *Computer Graphics* **22**(4) pp. 299-308 (August 1988). Proc. ACM SIGGRAPH 88.

14. D. R. Haumann and R. E. Parent, "The Behavioral Test-bed: Obtaining Complex Behavior from Simple Rules," *The Visual Computer* **4**(6) pp. 332-347 (December 1988).

15. P. M. Isaacs and M. F. Cohen, "Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions and Inverse Dynamics," *Computer Graphics* **21**(4) pp. 215-224 (July 1987). Proc. ACM SIGGRAPH 87.

16. N. Magnenat-Thalman and D. Thalman, "The Use of High-Level 3-D Graphical Types in the Mira Animation System," *IEEE Computer Graphics and Applications* **3**(9) pp. 9-16 (Dec 1983).

17. M. T. Mason and J. K. Salisbury, Jr., *Robot Hands and the Mechanics of Manipulation,* The MIT Press, Cambridge, MA (1985).

18. M. McKenna and D. Zeltzer, "Dynamic Simulation for Autonomous Legged Locomotion," MIT Media Lab (January 1989). Submitted for publication.

19. T. Nadas and A. Fournier, "GRAPE: An Environment to Build Display Processes," *Computer Graphics* **21**(4) pp. 75-84 (July 1987). Proc. ACM SIGGRAPH 87.

20. D. R. Olsen, Jr., E. P. Dempsey, and R. Rogge, "Input/Output Linkage in a User Interface Management System," *Computer Graphics* **19**(3) pp. 191-197 (July 1985). Proc. ACM SIGGRAPH 85.

21. R. Paul, *Robot Manipulators: Mathematics, Programming, and Control,* MIT Press (1981).

22. K. Pearson, "The Control of Walking," *Scientific American* **235**(6) pp. 72-86 (December 1976).

23. S. D. Pieper, "More Than Skin Deep: Physical Modeling of Facial Tissue," S.M. Thesis, Massachusetts Institute of Technology (February 1989).

24. M. Potmesil and E. M. Hoffert, "FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes," *Computer Graphics* **21**(4) pp. 85-94 (July 1987). Proc. ACM SIGGRAPH 87.

25. C. W. Reynolds, "Computer Animation with Scripts and Actors," *Computer Graphics* **16**(3) pp. 289-296 (July 1982). Proc. ACM SIGGRAPH 81.

26. E. A. Ribble, "Synthesis of Human Skeletal Motion and the Design of a Special-Purpose Processor for Real-Time Animation of Human and Animal Figure Motion," M.S. Thesis, The Ohio State University (June 1982).

27. J. L. Sibert, W. D. Hurley, and T. W. Bleser, "An Object-Oriented User Interface Management System," *Computer Graphics* **20**(4) pp. 259-268 (August 1986). Proc. ACM SIGGRAPH 86.

28. D. Sturman, D. Zeltzer, and S. Pieper, "Hands On Interaction with Virtual Environments," MIT Media Lab (January 1989). Submitted for publication.

29. I. E. Sutherland, "Sketchpad: A Man-Machine Graphical Communication System," *Proc. AFIPS Spring Joint Computer Conf.* **23** pp. 329-346 (Spring 1963).

30. D. Terzopoulos, J. Platt, A. H. Barr, and K. Fleischer, *Elastically Deformable Models,* Proc. ACM SIGGRAPH 87, Anaheim, CA (July 1987).

31. D. Terzopoulos and K. Fleischer, "Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture," *Computer Graphics* **22**(4) pp. 269-278 (August 1988). Proc. ACM SIGGRAPH 88.

32. F. Thomas and O. Johnston, *Disney Animation: The Illusion of Life,* Abbeville Press, New York (1981).

33. T. Whitted and D. Weimer, "A Software Test-Bed for the Development of 3-D Raster Graphics Systems," *Computer Graphics* **15**(3) pp. 271-277 (August 1981). Proc. ACM SIGGRAPH 81.

34. J. Wilhelms, "Using Dynamic Analysis for Realistic Animation of Articulated Bodies ," *IEEE Computer Graphics and Applications* **7**(6) pp. 12-27 (June 1987).

35. L. Williams, "BBOP," *Course Notes, Seminar on Three-Dimensional Computer Animation,* (July 27, 1982). ACM SIGGRAPH 82.

36. A. Witkin and M. Kass, "Spacetime Constraints," *Computer Graphics* **22**(4) pp. 159-168 (August 1988). Proc. ACM SIGGRAPH 88.

37. D. Zeltzer, "Motor Control Techniques for Figure Animation," *IEEE Computer Graphics and Applications* **2**(9) pp. 53-59 (November 1982).

38. D. Zeltzer, "Representation and Control of Three Dimensional Computer Animated Figures," Ph.D. Thesis, Dept. of Computer and Information Science, Ohio State University (August 1984).

39. D. Zeltzer, "Towards an Integrated View of 3-D Computer Animation," *The Visual Computer* **1**(4) pp. 249-259 (December 1985).

40. T. G. Zimmerman, J. Lanier, C. Blanchard, S. Bryson, and Y. Harvill, "A Hand Gesture Interface Device," *Proc. CHI+GI 1987,* pp. 189-192 (April 5-9, 1987).