Dynamic Animation and Control Environment

Ari Shapiro
University of California, Los Angeles
ashapiro@cs.ucla.edu

Petros Faloutsos
University of California, Los Angeles
pfal@cs.ucla.edu

Victor Ng-Thow-Hing
Honda Research Institute, USA
vngthowhing@honda.hra.com

Abstract

We introduce the Dynamic Animation and Control Environment (DANCE) as a publicly available simulation platform for research and teaching. DANCE is an open and extensible simulation framework and rapid prototyping environment for computer animation. The main focus of the DANCE platform is the development of physically-based controllers for articulated figures. In this paper we (a) present the architecture and potential applications of DANCE as a research tool, and (b) discuss lessons learned in developing a large framework for animation.

Key words: Dynamic animation, graphics system.

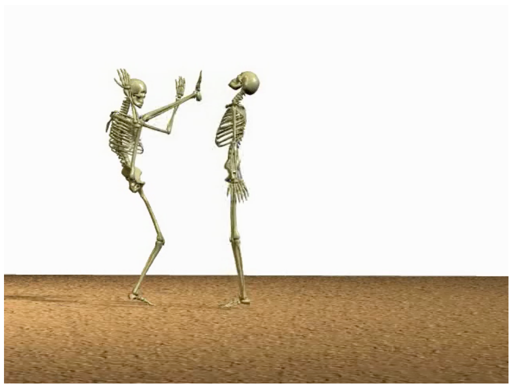

Figure 1: Two skeletons using different control methods.The left skeleton is key-framed, while the right character responds to physical forces derived from the kinematic motion. Both characters can simultaneously coexist and interact under DANCE.

1 Motivation

Physics-based animation research has a high barrier of entry. Research groups must dedicate resources towards building graphical tools and systems as a prerequisite to new research. Most systems developed by the dynamic animation community are not shared, requiring nearly every research group to re-engineer the same set of tools. Openly available tool sets usually take the form of libraries, such as RAPID [10] for collision detection or Open Dynamics Engine [28] for dynamic simulation. However, these disparate libraries address only parts of what are needed to properly implement physical simulation with articulated figures. In addition, they often have overlapping aspects, resulting in duplicated code that needs to be reconciled. Many individual libraries use different code bases for primitive modeling types, such as vectors and matrices, resulting in heavyweight conversions between function calls. Developing a framework for dynamic character simulation requires a great deal of engineering work in order to generate a reusable system. In addition, the various problem areas that have been neatly segmented by the research community, such as the separation of collision detection from collision resolution, must be unified in practice. We have found it difficult to cleanly separate these and other areas as well as the theories would suggest.

We have developed the Dynamic Animation and Control Environment (DANCE) as an open framework for computer animation research. DANCE's primary focus is the development of simulations and dynamic controllers. This is in contrast to many other animation systems which are oriented towards geometric modeling and kinematic animation.

1.1 Contributions

DANCE offers solutions to the problems of flexible reuse of controllers, actuators and interactive participation in 3-D physically based animation. In particular, DANCE has built-in support for physical simulation of articulated figures. In addition, DANCE provides basic modeling capabilities and support for kinematic animation such as motion capture and key framing.

DANCE demonstrates a powerful, open system model that permits a wide variety of different applications to be built with a common fundamental core and allows communication with external programs that can offer specialized functionality. The design of the base classes in DANCE unifies the large amount of specialized controllers and actuators that have been developed with a standard interface so that they can be shared in a common physical environment.

The power of an open plug-in architecture lies in the ability for a community of developers to work together to rapidly build a very complex system made up of relatively simple parts. As plug-ins can be selectively included into the main system, DANCE can be a useful research tool that enables various aspects of a system to be isolated for study.

Section 2 discusses DANCE's relation to other animation systems. Section 3 discusses the a number of design issues that were considered in constructing DANCE. Section 4 highlights the key base class components, called primitive plug-ins, of DANCE's architecture and outlines how some of them were used to create interesting subclasses. Section 5 presents a set of examples that illustrate the flexibility of our system with a focus on the development of dynamic controllers. In Section 6 we discuss limitations and restrictions of the system. In Section 7 we discuss lessons that we learned while developing this system and present recommendations for building similar systems in the future. Finally, Section 8 discusses possible research directions to extend DANCE or that DANCE can be applied to.

2 Related Work

Many tools, libraries and systems have been designed for the purpose of creating animations. Commercial systems such as Maya [1] and 3D Studio Max [5], focus on an interactive interface for modeling and animation. These tools are primarily oriented towards kinematic animation, such as key-framing. They incorporate dynamics in the form of particle systems and passive dynamics for rigid bodies by extending their modeling and kinematic animation systems. Our system differs from these in that it is tailored towards physics-based animation and control, rather than modeling or kinematic animation. The control architecture of DANCE enables active dynamics and interactive control during physical simulation, rather than passive dynamics and simple "rag doll" effects. DANCE also provides modeling and kinematic functionality, but does so as an extension of its dynamic capabilities, rather than vice-versa. Houdini [29] uses procedural networks to produce simulation and modeling results. DANCE's components are of a larger granularity than those provided by Houdini and it does not utilize a procedural network paradigm.

Endorphin [20] is a commercial simulation tool that provides basic dynamic control strategies for humanoid characters during physical simulation. Endorphin provides a set of behaviors for their dynamic characters and a simulation framework for developing character animation under physical simulation. We provide a research framework for designing controllers for dynamic characters with any topology or complexity.

Breve [16] is a simulation environment meant for the development of artificial life in a physically simulated world. It uses a scripting language that allows control strategies and event-based reactions to the environment for large numbers of agents. DANCE differs in that it is designed to support robust articulated objects with complicated control strategies for small number of agents.

Other documented animation systems exist but are not openly available for use. Industrial Light & Magic [15] created a dynamics animation system suitable based on a spring-mass model. Their solver is based on a second order Verlet integration.

Many dynamics engines exists to assist the development of physics-based animation, both open systems like Open Dynamics Engine (ODE) [28], and commercial ones such as Havok [11], Vortex [3] and SD/Fast [13]. Our system provides a framework and API with which to incorporate any simulation engine. To date, five different simulators are currently implemented in the DANCE framework, including ODE, SD/Fast and others (see Section 4.1).

Aside from architectural differences or design paradigms, DANCE differs from many commercial systems in that the code is publically available for use and inspection by the entire research community. This enables greater control over research and development by the user, since fundamental properties of the system can be modified as necessary. This also enhances the ability to perform research by reducing barriers to entry such as cost or restrictive licensing agreements.

2.1 Controllers

An interesting area of research in physics-based animation is the design and construction of *controllers* that can compute the active forces and torques required to control a character, either as an articulated figure or a deformable object. Controllers have been created to produce a range of locomotive and non-locomotive tasks, such as swimming[30, 34], running[12], walking[17] and breathing [35]. Although controllers are inherently reusable, their actual implementations are often restricted and embedded in custom systems built by various research groups for their own specific purposes. Other experimental systems allowed only a limited number of different types of controllers to coexist [30]. The incorporation of new controllers into such systems often implied a large undertaking in code redesign and development. DANCE allows controllers to be built as separate programs that adhere to restricted, but general, interfaces using standard object-oriented design. The end result is that an animation can be constructed consisting of several controllers cooperating or competing with each other as in [8], [7] and [26].

A true physically based character could react to changing obstacles in its environment and produce unique motions that do not suffer from the repetitive, pre-fabricated motion common in existing games and virtual worlds. Current interactive applications store sequences of fixed motion trajectories in finite state machines that selectively play back different sequences of motion according to user events.

Systems have been introduced, such as the *MOTIVATE Intelligent Digital Actor System*[6], that attempt to synthesize motion dynamically and model high-level behaviors. However, we do not believe that any physical simulation is being performed at the lower levels of motion. Many existing strategies attempt to parameterize existing motion kinematically to adapt it to different situations. Such strategies do not offer true interactivity, as the parameterized motion is still inherently related to a limited set of pre-recorded actions.

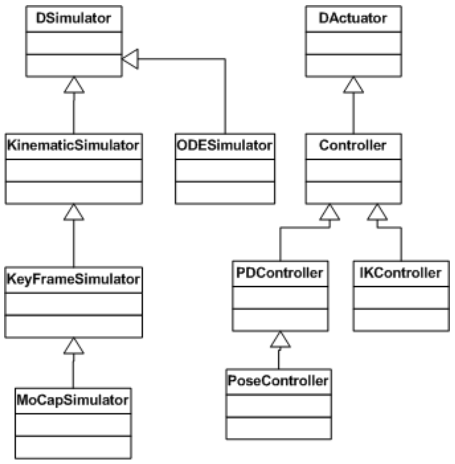

Figure 2: UML class diagram demonstrating the design of the dynamic simulators, kinematic simulators and controllers. The Open Dynamics Engine (ODE) is implemented using the DSimulator class. Kinematic animation can also be derived and used concurrently with dynamic controllers, such as a proportional-derivative (PD) controller. This enables simulation of mixed kinematic and dynamic control.

3 System Design

DANCE's design focus is on the creation of a modular and open physics-based animation system. In some cases, compromises had to be made to find a balance between competing requirements, like speed and data abstraction. Other times, seemingly contrasting goals like tight integration and modularity had elegant solutions that allowed both to be fulfilled.

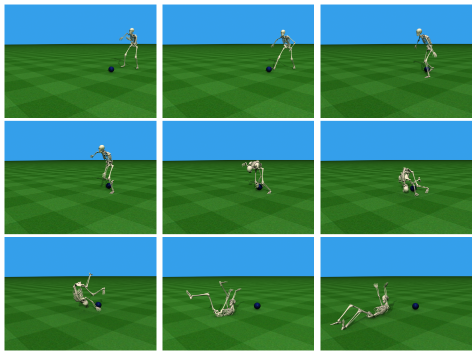

Figure 3: A dynamic character is programmed with dynamic controllers that allow it to roll and get up following an interaction with a physical object.

DANCE is structured using the principled application of object-oriented design and dynamically-linked objects (*plug-ins*). The core architecture maintains a simple set of base classes that it uses in its user-interface, display and physical simulation subsystems. We have purposely restricted the number of different primitives in order to have a relatively simple core, while leaving all the functionality and complexity within the plug-ins. For example, the interface to the *actuator* primitive has allowed us to serendipitously implement a wide set of features that include collision detection, ground models, interactive drag manipulators and deformable muscles as shown in Figure 5. These features can be selectively included in an animation, empowering the animator with a flexible set of tools that can be combined in different ways to produce animations.

Realizing that physically-based animation is only one possible means of creating motion, DANCE allows kinematically based controllers to handle traditional keyframed animation in a manner that is consistent and sensitive to the physical, dynamics-based environment. Furthermore, animation generated with DANCE can be exported directly into commercial animation and rendering systems such as Maya[1] or the RenderMan language[31] to take advantage of the advanced rendering and modeling capabilities that such facilities offer.

Although DANCE is primarily a programming tool, it can be used as an interactive modeling environment. It is a system that allows an articulated figure and its physical environment to be constructed with complete specification and manipulation of different joint types. Without leaving DANCE's environment, a system of equations of motion for the figure can be generated, followed by the resulting simulation which can be adjusted and viewed in real-time using interactive cameras and direct manipulation. DANCE supports a number of different input devices such as mice, joysticks and haptic devices. These devices can be used as a means of interactively manipulating a running simulation by providing real-time force and torque information.

4 Plug-in Primitives

DANCE is primarily designed as a system for the physics-based animation and control of articulated figures. The system is divided into components consisting of $C++$ classes.

Many other classes exist that are necessary for the proper execution of the DANCE environment, including those that provide the underlying *glue* that connects various components. For example, classes for management of the graphical interface, script interpreter and driver subsystem. The details for those classes will not be described in this paper as they appear in the source code and their details do not need to be understood in order to use the functionality of the system. This section discusses the most important classes of the system, that we refer to as the *plug-in primitives*.

There are several diagrams given in this paper that detail the structure of the relevant parts of the DANCE system. Figure 4 shows the general structure of the plug-in primitives, described below. Figure 7 shows the derivation of systems of hierarchical objects necessary for the development of complex articulated figures, such as human-like characters. Figure 2 shows the structure we used for the development of dynamic controllers.

The plug-in primitives, which provide the basic functionality of the system, are the *simulator*, *actuator*, *system*, *modifier* and *geometry* primitives. Each of these primitives is described in the sections below.

Our plug-in mechanism is based on the object-oriented facilities of the C++ programming language and the dynamic runtime linking provided by modern operating systems. DANCE exchanges information through the interfaces of the plug-in primitives, represented by classes, which can be sub-classed to the dynamically linked objects.

4.1 Simulator Class

The simulator is one of the most important components of a physics-based system as it performs the numerical integration to create motion over time. DANCE implements simulators as plug-ins so that they can be compiled and linked by the system at runtime.

To provide a uniform simulation interface for DANCE,

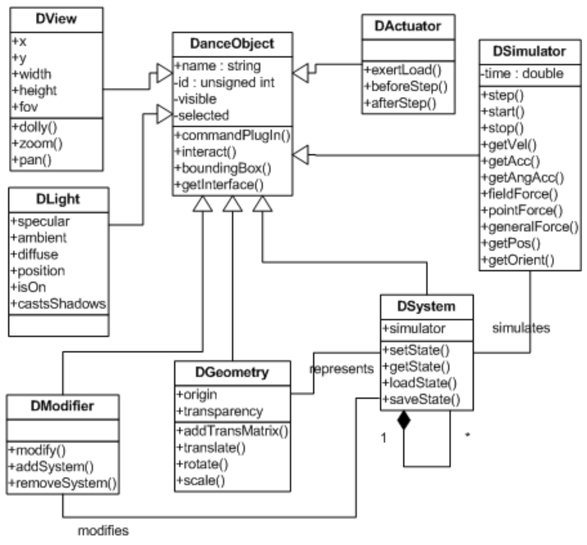

Figure 4: UML class diagram showing the DANCE plug-in primitives. For readability, important attributes and methods are shown to clarify the use of each class.

we bound the main simulation routines to methods of the simulator class so that the simulator could be encapsulated and associated with an individual System instance (described in Section 4.3) such as articulated objects. This allows DANCE to contain several different System instances, each with their own individual simulators. Of numerical importance, separate simulators can be applied in situations where there are large differences between the natural motion frequencies of several articulated objects. Objects with high frequency motion could use smaller integration timesteps, while objects with smoother motion would still be able to use larger timesteps. In contrast, if only a single, global simulator existed, the timestep must be set to the smallest value to ensure stability of the overall system.

To date, five different simulators are currently implemented in the DANCE framework as plug-ins: ODE, SD/Fast, ABDULA [9], a version of Baraff's simulator [2] and a version of Verlet integration [14].

These simulator implementations can solve the equations of motion for any plug-in that conforms to the primitive plug-in API. Other simulators can be added to simulate deformable objects or particle-based systems, such as fire, smoke or water.

4.2 Actuator Class

We define the *actuator class* to represent any source of internal or external loads onto a system. Actuators can apply forces and torques to all objects in a scene or to specific objects, link and joints. The actuator class has

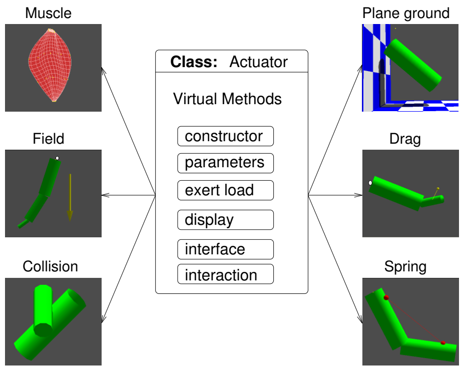

Figure 5: Actuator Class and some subclasses

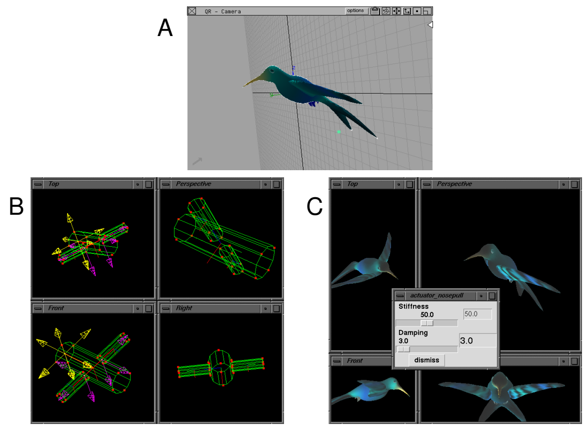

Figure 6: A hummingbird model can be modeled in Maya *in order to enhance the appearance of an underlying physical model.*

proved to be a versatile class that is capable of modeling a wide range of physical phenomena. Figure 5 shows the structure of the actuator base class.

Field actuators can be applied globally or on an individual object basis. We use the *field actuator* to simulate the effects of Newtonian gravity, wind forces or a water-based effects, such as in [34]. By implementing these forces as an actuator, we can selectively apply it to objects in the scene. The entire simulation environment can be controlled by turning these forces on and off both interactively and automatically based on temporal and contact-based events. Thus, users can remove or apply the physical effects of actuators on individual objects.

Damper actuators are global actuators that exert a viscous force or torque at the joints of an object to emulate the effects of joint friction or air resistance. They are mainly used to provide a more realistic physical environment that introduces gradual energy loss to the system. Without them, a moving passive object would never come to a complete rest because the physical simulation conserves all the kinetic and potential energy in the system.

To model periodic motions, we have built a PD-based *period actuator* that can provide sinusoidal control. We have used this actuator to create flapping wings in a hummingbird model shown in Figure 6. Frequency and amplitude of the control signal can be altered to create many different controllers for periodic locomotion. The details for such an actuator and their relationship to controllers are described in Section 5.

Traditionally, biomechanical studies have used linear actuators modeled as line segments to represent muscles in human motion studies [4]. As a result, forces are applied at single origin and insertion points at either end of the muscle. We have built a detailed *muscle actuator*, shown in Figure 8 that can apply forces over an area of an object's surface instead of a single point. The actuator has a deformable geometry that can be displayed and manipulated interactively, showing the flexibility of the actuator class.

4.3 System Class

Systems encapsulate objects that are simulated in the DANCE environment. Systems can be articulated objects, such as humanoid characters, or can represent hair or cloth. A system maintains a state and it is often associated with a simulator that models how the system's state changes over time.

An example of use of the system class is the generation of articulated figures for simulation of humanoid characters. We derive the ArticulatedObject class from the DSystem class for this purpose, Figure 7, described below. The ArticulatedObject allows dynamic simulation of hierarchical structure, while also providing a representation for kinematic animation, such as motion capture.

Articulated Object Class

The *ArticulatedObject* class provides the animatable objects or characters in DANCE. We define an articulated object to consist of one or more rigid body links held together by constraints. The *articulated object* class is actually a container class for *joint* and *link* class instances, providing useful methods for adding and removing links. A *link* represents the body segments of an object and can have their own *geometry* instances. The *geometry* class can be subclassed to create a variety of different visual representations such as cylinders or triangle lists. The *joint* class can represent any constraint between two links and is responsible for handling the display and manipulation of these constraints. DANCE supports a variety of translational and rotational joints.

4.4 Modifier Class

Modifiers are generic structures which alter systems in DANCE. A modifier could change a system by, for example, transforming its underlying geometry. Alternatively, a modifier could change the state of a system according to some parameters, such as time or number of objects in the scene. Whereas Systems are heavyweight objects that establish rules and structure for objects that exist in a simulation environment, Modifiers are lightweight structures that can be added and removed from the simulation without disrupting the basic operation of the underlying System. As an example, we create a plug-in that implements linear blend skinning as a modifier, which in turn manipulates the underlying geometry of an ArticulatedObject instance for better visual effect. The DANCE structure allows the user to subclass the basic linear blend skinning plug-in to create different interactive interfaces for skinning [19] or more complex character skinning algorithms, such as [18].

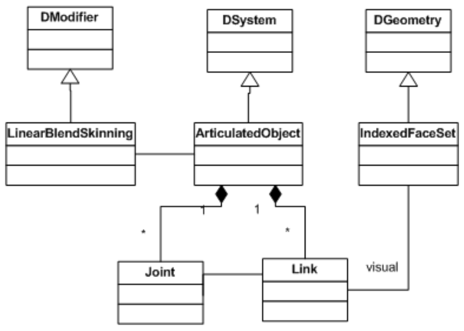

Figure 7: UML class diagram showing how articulated characters extend from the DSystem class. Links and joints are managed by the ArticulatedObject class. Character skinning is implemented by subclassing the DModifier, which changes the ArticulatedObject and underlying geometry.

4.5 Geometry Class

Geometry represents models, such as spheres, meshes and surfaces. Basic geometry can be designed using other modeling tools and subsequently imported into the DANCE environment.

5 Applications

While the plug-in primitives establish the architecture of the system, the power of the DANCE environment comes from the development of plug-ins that can be generated by extending the plug-in primitives.

We describe below a number of plug-ins that have been developed using the DANCE system. The design of the system allows us to to reuse modules developed for one application for a different purpose. For example, the development of a plug-in that represents gravity can be reused for any application that utilizes the primitive plug-in interfaces. Of special interest are plug-ins that enhance the ability of the user to utilize physics-based animation in a meaningful way.

Implementation of Controllers

The controller is a class of objects that can apply control loads to specified objects. Its purpose is to allow the user to implement simple or complex control techniques. It extends the actuator primitive plug-in to avoid restricting users to a specific type of controllers and also to avoid overloading the system for users that do not wish to use control.

We derive our controllers by refining the concept of an actuator. The actuator represents a muscle or external force independent of an object. Controllers, on the other hand, can be thought of as a brain, which in turn can affect a number of other actuators. Controllers can hierarchical in structure and use sensors and feedback to produce complex motions.

The controller implementation is very general and allows virtually any control paradigm that is possible to be implement by other systems, to be implemented as a plug-in for DANCE. For demonstration purposes we have implemented two different types of controllers namely *pose* and *kinematic* controllers.

Pose controllers have been used extensively as the basic control structure in a variety of applications [17, 7]. They are suited particularly for periodic motions and can be used with or without feedback. Typically, they are represented as a finite state machine (FSM) with time transitions. The states are snapshots of the character's motion which, in most cases, drive appropriate sets of proportional- derivative controllers. The latter transform the poses appropriately into spring and damping forces that act on the object.

Our version of *kinematic controllers* can prescribe motion for articulated objects using inverse dynamics methods similar to those employed in [36]. The user can specify functions of time that define the acceleration, velocity and positions of specific degrees of freedom of selected objects. Pure kinematic controllers that utilize inverse kinematics as well as other kinematic methods can also be created from the basic controller class.

5.1 Drag Actuators and Spring Actuators

Drag actuators and *spring actuators* allow an animator to interactively apply spring forces to objects. A spring actuator is defined by two endpoints p_1, p_2 which can be interactively attached to objects or at fixed locations anywhere in the scene. It is capable of automatically recognizing if its endpoints are attached on immobile or moving objects. It exerts spring and damping forces on the endpoints of the form:

$$\mathbf{f}_{p_1} = -\mathbf{f}_{p_2} = K(|\mathbf{l}| - l_0)\frac{\mathbf{l}}{||\mathbf{l}||} + D\dot{\mathbf{l}},$$

where K, D are stiffness and damping constants, l_0 is the spring's rest length, $l = p_2 - p_1$, the dots indicates a time derivative. The user can interactively control the stiffness and damping, the position of the endpoints, and the locations they are attached on any time during the simulation. Thus the spring actuator is a powerful tool that allows the user to position and manipulate objects in a physics based way. For example, using a number of spring actuators we can turn a system into a virtual puppet. The drag actuator is a spring actuator with one point always following the mouse pointer and the other attached on an object. Its rest length is by default zero, thus dragging the attached point towards the location of the pointer. It allows the user to quickly and easily apply a spring force on a point of an object by a simple mouse click.

A wide variety of animation applications involve objects interacting with a ground model. We avoid restricting the system to a specific type of ground model by implementing the latter as an actuator. The *plane ground actuator* implements a plane ground of arbitrary orientation. We have implemented a simple collision detection method based on monitor points and we use a penalty method to resolve the collisions. The user can interactively affect the stiffness, damping and friction constants any time during the animation to produce desired affects. Using our interface it is trivial to create arbitrary plane grounds on the fly with separately controlled parameters in order to construct more complex environments.

5.2 Dynamic Muscle Modeling

We have applied DANCE's open architecture to produce a set of wide range of different systems.

Figure 8 displays parts of an anatomically-based modeler for humanoid characters. Unlike previous models for anatomical animation[25, 32], DANCE was used to implement the muscles as actuators that are capable of exerting physical forces on the bone links they are attached to, directly creating motion in the articulated model. Constraint objects were used to establish point to point contacts between portions of the muscle actuator and the links. In each example, the user interfaces are customized towards the specific application, while still retaining a common core architecture. Note that the muscles were added to existing plug-ins that described an articulated character. We believe that the ability to share complex components is the most interesting feature of a framework like DANCE.

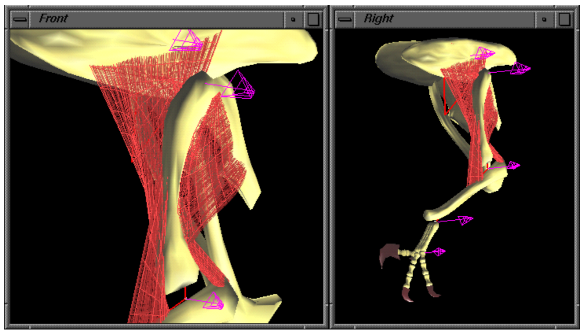

Figure 8: Biomechanical modeling and simulation in DANCE.

5.3 Hair Simulation

Figure 9 demonstrates interactive hair simulation. The hair model (derived from *system*) and the associated simulator are based on particle systems and dynamics. The face model is a separate plugin derived also from the *system* base class and is animated using a blendshape approach.

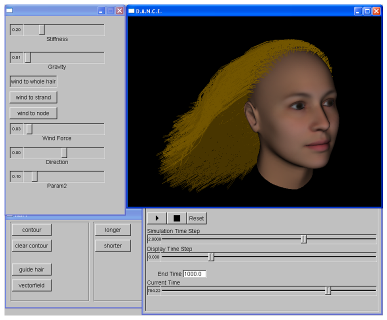

Figure 9: Hair simulation in DANCE using particle systems.

5.4 Speech and Facial Animation

By using blendshapes, extending the animation classes and adding voice synthesis libraries, we are able to synthesize speech and animate facial expressions within the DANCE environment, as shown in Figure 10.

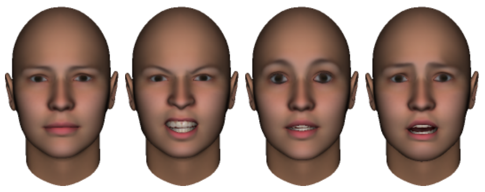

Figure 10: Expressive facial animation.

6 Limitations

Designing frameworks that provide the right level of granularity can be tricky. A highly structured system provides power to a programmer by allowing him to create complicated programs with small amounts of code. Highly structured systems, however, can suffer from too much rigidity, requiring the programmer to perform additional work when designing behavior that falls outside of the scope of the framework. On the other hand, too little structure can result in a disparate set of functions that is ignored by the programmer, who sees the framework as unnecessary and powerless. DANCE errs on the side of too little structure rather than too much, because we believe that an overly restrictive environment would discourage the use of the system as a research tool.

6.1 Metadata Efficiency

We designed DANCE to be flexible and modular. New elements can be designed as plug-ins and replace or extend existing functionality. However, this architectural flexibility comes with a cost in performance. The metadata required to generalize the interfaces is additional overhead that the system must handle. For example, a standalone version of cloth simulation ran nearly twice as fast as it was when integrated into DANCE. However, the cloth simulator integrated into DANCE can handle arbitrary cloth topology and arbitrary collision geometry, as well as being able to coexist alongside other simulators. Thus, our design preference for compatibility limits performance at the expense of compatibility.

6.2 Optimization Limitations

It is difficult to optimize component interfaces without adding some cost to the overall software system. This cost will either take the form of: 1) circumventing the interface by using the special capabilities of that component, or 2) adapting the general interface to the optimize component, forcing the other components to adhere to the optimized interfaces, thus complicating the system. Utilizing a more complicated interface reduces the effectiveness of the framework by increasing the amount of work necessary to integrate various components. Thus, our system is generally slower than most commercial systems, which are usually optimized for performance. Running dynamics engines directly through their interfaces will result in better performance than through DANCE, since the metadata of the system provides overhead that isn't necessary for all uses of the tool. For example, the ArticulatedObject, which describes a hierarchy of connected bodies, updates the local transformation matrices of all rigid bodies in the hierarchy at every timestep. This update is not necessary for every application yet is performed nonetheless for compatibility with other features of the system, such as controllers.

Loose vs. Tight Coupling

Certain plug-ins require a tight coupling with other plug-ins in order to work properly. For example, simulators need specific collision information in order to effect collision resolution on rigid bodies. Under our architecture, a simulator, a collision detector and a collision resolver are three separate plug-ins derived from DSimulator, DModifier and DActuator respectively. However, the interfaces between DSimulator and DActuator are not detailed enough in order to provide the proper functionality between the classes. Thus, instances of the simulator, collision detector and collision resolver override the framework by referring to the specific functionality of the other components. Future enhancements would either design better interfaces in order to properly separate the three plug-ins from each other, or combine all three aspects into one plug-in.

Code Management

The DANCE code base has been altered by over 20 different people over a period of 5 years. This results in a collection of different coding styles and design decisions as different principals have moved the code in various directions. As with any large system, maintenance of code and the elimination of bugs is an ongoing process.

7 Lessons Learned

Below we present some of our findings in building a large simulation system.

7.1 One Size Fits All Makes A Slow System

It is relatively easy to develop an animation system that can simultaneously handle the needs of different complex phenomena, such as rigid body simulation and fluid simulation. However, it is difficult to do so and have those aspects run quickly and efficiently. Computer graphics applications in particular must run quickly. To do so requires optimization of various parts of the system. This optimization is difficult to accomplish when the algorithm must accommodate disparate kinds of data and cannot make useful assumptions about the data. The original focus of the DANCE system of a simulation environment for developing controllers has not changed since its inception, but it has been adopted to accommodate a number of different simulation environments. These adapted environments run more slowly and present a less cohesive programming environment for the user than they would as standalone systems.

7.2 Development of Dynamic Controllers

Designing robust dynamic controllers is difficult. There is no universally accepted method for the development of dynamic controllers. Current methods of development include adapting algorithms from biomechanics or robotics, laborious hand coding of small actions or the use of machine learning techniques such as genetic algorithms or reinforcement learning. One of the design goals of DANCE was to create an environment where an inexperienced user could develop dynamic controllers without having to understand control theory, biomechanics and machine learning. We found it difficult to provide the vast number of resources required for such disparate methods for use by an end user without requiring the end user to have a large amount of specialized knowledge. Future research work in this area will probably need to describe a limited framework for developing specific types of dynamic control. DANCE remains a tool for programmers.

8 Conclusions and Future Work

With DANCE, we have built an open, extensible physically-based animation system that engages the animator to interactively direct a 3-D animation. The plug-in architecture allows a large library of diverse actuators and controllers to be implemented and integrated using a standard object-oriented interface. In addition, DANCE plugins extend the system to utilize kinematic animation and a variety of non-physically-based animation structures.

Controller designers can build their own actuators, unrestricted by any particular technique. Indeed, controllers can be built to perform prescribed motion to integrate pre-existing motion path trajectories such as those created with keyframe animation or spacetime constraint optimization [33]. Practitioners of physics-based animation can exchange controllers with each other to test out novel cooperative and competitive tasks. This area has not been explored very much in computer animation, but was enticingly hinted at in [27].

We use DANCE to build sophisticated, interactive 3-D environments, including the physical anatomical-based modeler shown in Figure 8. DANCE and its derivatives have been used in a number of different research projects, including [24, 8, 26, 21, 22, 23]. We openly and publicly distribute DANCE at http://www.magix.ucla.edu/dance/ in the hope of encouraging the further cooperation and sharing of controllers, actuators and other implementations. We see the DANCE platform as the basis for research into computer graphics in the areas of character animation, physically-based motion and control.

Acknowledgements

We would like to thank Joe Laszlo for many useful discussions and for porting earlier versions of DANCE on MacOS. We would also like to thank Michael Neff for testing early versions of the system and for his useful comments. Lucio Flores implemented simulator plugins. Albert Chu implemented the skinning module and interface. Gordon Cook contributed an inverse kinematics solver. Alice Wang and Michael Emmi produced the speech and facial animation images.

The work in this paper was partially supported by NSF under contract CCF-0429983. We would also like to thank Intel Corp., Microsoft Corp. and ATI Corp. for their generous support through equipment and software grants.

References

[1] Alias/Wavefront. Maya, 1997.

[2] David Baraff. *Physically Based Modeling: Principals and Practice*. SIGGRAPH Online Course Notes, 1997.

[3] CMLabs. Vortex, software. http://www.cm-labs.com, 2004.

[4] Scott L. Delp, J. Peter Loan, Melissa G. Hoy, Felix E. Zajac, Eric L. Topp, and Joseph M. Rosen. An interactive graphics-based model of the lower extremity to study orthopaedic surgical procedures. *IEEE Transactions on Biomedical Engineering*, 37(8):757–767, 1990.

[5] Discreet. 3d studio max, 2003.

[6] The Motion Factory. Motivate intelligent digital actor advantage. www.motion-factory.com, 1997.

[7] Petros Faloutsos, Michiel van de Panne, and Demetri Terzopoulos. Composable controllers for physics-based character animation. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 251–260, August 2001.

[8] Petros Faloutsos, Michiel van de Panne, and Demetri Terzopoulos. The virtual stuntman: Dynamic characters with a repertoire of autonomous motor skills. *Computers & Graphics*, 25(6):933–953, 2001.

[9] Frans Faure. Fast iterative refinement of articulated solid dynamics. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):268–276, 1999.

[10] S. Gottschalk, M. C. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[11] Havok. Havok 2. http://www.havok.com, 2004.

[12] J. K. Hodgins, W. L. Wooten, D. C. Brogan, and J. F. O'Brien. Animating human athletics. In *Computer*

Graphics (SIGGRAPH '95 Proceedings), pages 71–78, 1995.

[13] Michael G. Hollars, Dan E. Rosenthal, and Michael A. Sherman. Sd/fast. Symbolic Dynamics, Inc., 1991.

[14] Thomas Jakobsen. Advanced character physics - http://www.ioi.dk/homepages/thomasj/publications/gdc2001.htm.

[15] Zoran Kai-Alesi;, Marcus Nordenstam, and David Bullock. A practical dynamics system. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer animation*, pages 7–16. Eurographics Association, 2003.

[16] Jon Klein. Breve: A 3d environment for the simulation of decentralized systems and artificial life. In *Proceedings of the eighth international conference on Artificial life*, pages 329–334. MIT Press, 2003.

[17] Joseph Laszlo, Michiel van de Panne, and Eugene Fiume. Limit cycle control and its application to the animation of balancing and walking. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 155–162, 1996.

[18] Alex Mohr and Michael Gleicher. Building efficient, accurate character skins from examples. *ACM Transactions on Graphics*, 22(3):562–568, July 2003.

[19] Alex Mohr, Luke Tokheim, and Michael Gleicher. Direct manipulation of interactive character skins. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 27–30. ACM Press, 2003.

[20] Natural Motion. Endorphin. www.naturalmotion.org, 2003.

[21] Michael Neff and Eugene Fiume. Modeling tension and relaxation for computer animation. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 81–88. ACM Press, 2002.

[22] Michael Neff and Eugene Fiume. Aesthetic edits for character animation. In *2003 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 239–244, August 2003.

[23] Michael Neff and Eugene Fiume. Methods for exploring expressive stance. In *2004 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 49–58, July 2004.

[24] Victor Ng-Thow-Hing. *Anatomically-based models for physical and geometric reconstruction of humans and other animals*. PhD thesis, 2001.

[25] Ferdi Scheepers, Richard E. Parent, Wayne E. Carlson, and Stephen F. May. Anatomy-based modeling of the human musculature. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 163–172, 1997.

[26] Ari Shapiro, Frederic H. Pighin, and Petros Faloutsos. Hybrid control for interactive character animation. In *11th Pacific Conference on Computer Graphics and Applications*, pages 455–461, 2003.

[27] Karl Sims. Evolving virtual creatures. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, volume 28, pages 15–22, July 1994.

[28] Russell Smith. Open dynamics engine. http://opende.sourceforge.net, 2003.

[29] Side Effects Software. Houdini, 2000.

[30] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In Andrew Glassner, editor, *Computer Graphics (SIGGRAPH '94 Proceedings)*, Computer Graphics Proceedings, Annual Conference Series, pages 43–50. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[31] Steve Upstill. *The Renderman Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Publishing Company, New York, 1989.

[32] Jane Wilhelms and Allen Van Gelder. Anatomically based modeling. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 173–180, 1997.

[33] Andrew Witkin and Michael Kass. Spacetime constraints. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 159–168, August 1988.

[34] Po-Feng Yang, Joe Laszlo, and Karan Singh. Layered dynamic control for interactive character swimming. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 39–47. ACM Press, 2004.

[35] Victor B. Zordan, Bhrigu Celly, Bill Chiu, and Paul C. DiLorenzo. Breathe easy: Model and control of simulated respiration for animation. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 29–37. ACM Press, 2004.

[36] Victor B. Zordan and Jessica K. Hodgins. Motion capture-driven simulations that hit and react. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 89–96. ACM Press, 2002.