Dynamic Animation and Control Environment (Dance)

A User/Programming manual

version 1.0

by

Victor Ng-Thow-Hing
Petros Faloutsos

University of Toronto
Department of Computer Science

September 25, 2000

Contents

1 Introduction

Welcome to the DANCE (Dynamic Animation and Control Environment) system. DANCE is an open architecture for creating physics-based environments for simulation, computer animation, or just plain fun. As the area of physics-based animation is always changing and subject to improvements, the software architecture of DANCE was deliberately kept simple and general. We identified what we thought were the most common features of the majority of physics-based applications in existence and abstracted these into a set of key categories. Using object-oriented design, each category was implemented as an abstract class. For extensibility, the abstract classes can be subclassed to create specific instances through a plug-in mechanism. DANCE can be thought of as a general operating system for physics-based environments. It provides basic functionality that is common to most systems, while allowing you to implement functionality that is specific to your own needs.

Plug-ins or more formally, dynamically-loaded objects, have already been used successfully in both commercial and non-commercial software, such as *3D Studio Max*, *Maya*, *GIMP*, and *Photoshop*. Through this technique, we have transformed the basic core DANCE system into more complex systems, such as a physics-based control environment for virtual stuntmen and an anatomy-based modeller for biomechanical simulation of musculoskeletal systems. In addition to allowing you to extend the DANCE core system, plug-ins are a useful way to exchange features with fellow users of DANCE. From a research point of view, components can be compared with one another for performance. For example, research in control synthesis can be facilitated by allowing a standard articulated object to be shared and tested with different controllers in the same environment, with the same simulators. This is important because the type of integration method used can affect the performance of a controller and the subsequent motion produced.

In summary, we hope that DANCE will be used by the community of computer animators, physicists, roboticists, biomechanists or anyone who has the desire to experiment with physical simulation. DANCE allows you to start playing with simulations or test out new ideas without needing to devote a large amount of time building up the necessary infrastructure.

1.1 License

DANCE consists of a large amount of code written by its authors, Victor Ng-Thow-Hing and Petros Faloutsos. However, we owe a great deal to the efforts of countless others who were generous enough to freely release their code. Since a fundamental requirement of DANCE is the ability for it to run on multiple platforms, we chose software libraries that could be used in the major operating systems: Windows, Unix and MacOS. Full details on the licensing conditions of every package can be found in the $DANCE_DIR/doc/Licenses directory.[1] However, we list some of the important software contributions to DANCE by various people:

[1] $DANCE_DIR refers to the environment variable that contains the path to the DANCE software installation.

Arcball In the book *Graphics Gems IV*, Ken Shoemake introduced the intuitive Arcball technique for rotating camera views. The original sample code provided utilized the venerable IrisGL graphics API. This code was modified to work with the standardized OpenGL. In addition, new functions were added that allow the camera matrix to be initialized to an arbitrary rotation matrix.

f2c Several numerical routines used by various plug-ins in DANCE were converted from many of the excellent fortran libraries available publically. They were converted to the C language using *f2c* which was created by AT&T, Lucent Technologies and Bellcore.

GLUT This is an excellent cross-platform library for providing window management and event-handling functionality. DANCE uses GLUT to create multiple views of the 3-D scenes it creates. Mouse and keyboard events are also handled by the GLUT callback mechanism. The main event loop of DANCE is handled by the **glutMainLoop** function. Mark Kilgard is the original author of GLUT. The version of the GLUT library used in DANCE has been modified slightly to handle shared OpenGL display lists amongst several windows.

Minpack The Minpack library provides useful routines for unconstrained optimization. Several plug-ins use this library created by Jorge Moré, Burt Garbow, and Ken Hillstrom at the Argonne National Laboratory.

OpenGL 3-D graphics are handled by the OpenGL API, including selected functions from the GLU library. OpenGL was chosen over other APIs because of the cross-platform requirements of our design.

RAPID The ArticulatedObject plug-in allows geometry from the IndexedFaceSet geometry plug-in to be used to represent limb segments or bones. To perform collision detection on this geometry, we used the excellent RAPID library to quickly perform collision detection and proximity tests. RAPID's authors can be contacted through S. Gottschalk from the Department of Computer Science at Chapel Hill.

Tcl/Tk The scripting engine provided by the DANCE core is based on the Tcl language provided by Scriptics Corporation. In addition, the choice of Tcl enables graphical user interfaces to be created with Tk, which is an extension of Tcl. This enables practitioners to customize their user interfaces for individual applications by keeping the user interface generation code outside of the DANCE core.

TkCon For command-line script entry, we use the enhanced Tk Console written by Jeffrey Hobbs (with contributions from numerous others).

VCOLLIDE To track multiple body collisions, the Chapel Hill team from the Department of Computer Science created VCOLLIDE which works with RAPID. The ArticulatedObject plug-in uses VCOLLIDE to keep track of all the geometry assigned to the links of every articulated object in DANCE.

VrmlView Within the default user interface for DANCE, we use VRMLview, created by Systems in Motion for previewing VRML models for the Geometry plug-in.

1.2 Features

The power of DANCE is located in its plug-ins. The purpose of the DANCE core is to offer infrastructure that is intended to reduce the amount of development time for key features that developers want to implement.

Features of the DANCE core include:

- Recognition of the following system components: systems, simulators, actuators, geometry, and views
- Object classes for the management of major system components, including plug-in maintenance
- Tcl-based scripting language for accessing functionality within the DANCE core, enabling automation of tasks with the full command set of Tcl
- Powerful unix-like command shell (TkCon) for a command-line interface
- Ability to open multiple views of the same 3-D scene, including perspective and top, front, right orthographic views

- Access to a Proximity class that provides closest feature proximity tests to arbitrary 3-D points in the scene
- Direct manipulation interface for modifying simulators, systems, and actuators in DANCE

A representative sample of features of several plug-ins include:

- ArticulatedObject system plug-in provides a body-joint representation for articulated systems. Various joints such as ball, pin, gimbal and sliding joints are implemented. Direct forward kinematics of joints are provided. The links of the articulated object can be assigned geometry pulgins. Some limited collision detection and resolution during kinematic manipulation is implemented.
- An SdfastSimul plug-in allows SD/FAST(tm)-created objects to be dynamically linked into DANCE. SD/FAST is an equations of motion compiler by Symbolic Dynamics, Inc., and is used by several research and industrial labs.
- The IndexedFaceSet plug-in provides input filters for VRML97, ply, and OpenInventor indexed face set file formats for polygonal meshes. Routines are provided for computing the centre of mass and inertia tensors from the geometry, given assumptions about the density properties. The geometry can be re-transformed into a reference frame that coincides with the principle axes of the inertia tensor, producing a diagonal inertia tensor matrix with zero products of inertia. The geometry can be instanced to represent the links of articulated objects.
- Collision detection for IndexedFaceSet geometry and articulated objects is provided by the RAPID and VCOLLIDE libraries mentioned in Section 1.1.
- Actuators are provided for modelling force fields, a piece-wise flat ground and pose control.
- A sample inverse kinematics (IK) actuator is provided for IK control of articulated objects.

1.3 Resources

The official DANCE web page is: www.dgp.toronto.edu/dgp/software/dance. The latest version of DANCE can be downloaded from this web page. Currently, there is no mailing list.

2 Installation

The DANCE system is packaged as a single archived file which expands into a directory structure, complete with standard subdirectory names. Once you have expanded the archive into a given path location, you must then set a special DANCE_DIR environment variable to point to that location. The directory structure of the DANCE distribution is shown in Figure 1. The first two subsections describe how to install a *development* environment for DANCE on Windows-based and Unix-based operating systems respectively. The third Subsection 2.3 describes how to install a run-only version of DANCE.

2.1 Expanding on Windows 95/98/NT/2000

Use the danceWinXX.zip file archive, where XX refers to a specific version number. Using a Windows-based file unarchiver like WinZip, you can expand the file into a desired target destination $<$ *pathname* $>$. After expanding, the DANCE distribution should be located in: $<$ *pathname* $>$/dance.

To set the environment variable, there are two possibilities.

Windows NT From the Start → Settings → Control Panel window, click on the System applet. Choose the environment tab and create a new DANCE_DIR environment variable with the value set to $<$ *pathname* $>$/dance. NOTE: It is *very* important to use forward slashes in the entire path of DANCE_DIR because dance uses Tcl/Tk as its scripting environment. Several operations in Tcl assume that the forward slash convention is used to separate out directory names in Tcl and may fail if the Windows backslash convention is used instead. This also ensures consistency of pathnames across multiple operating systems. You can set DANCE_DIR either as a system variable or user variable depending on whether you want to share a common dance distribution with all users on the same computer or have each user have their own DANCE distribution directory.

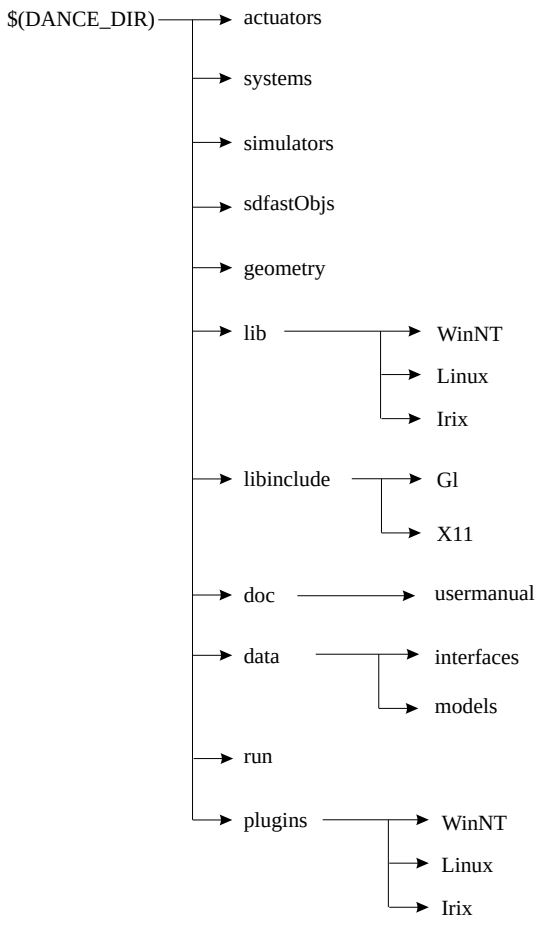

Figure 1: The DANCE directory structure

Windows 98 From a dos command-line or compatible shell window, you can set the DANCE_DIR as follows:

set DANCE_DIR=< *pathname* >/dance

This line can also be placed in the **autoexec.bat** file for automatic initialization during boot-up, as would be desired for Windows 98.

Finally, the PATH environment variable should be set to include the DANCE distribution directory:

set PATH=%PATH%;< *pathname* > \dance\bin.

As an example, if I unarchived DANCE into D:\Work, I would set DANCE_DIR to **D:/Work/dance** and the PATH would include **D:\Work \dance\bin**. NOTE: DANCE_DIR uses forwardslashes on all platforms.

2.2 Expanding on Unix-based operating systems

The procedure for Unix-based operating systems (so far tested on Linux and Irix), is similar to Windows. Simply unarchive the appropriate OS distribution in the destination directory:

gunzip danceUnixXX.tar.gz

where XX refers to the version number. The DANCE_DIR environment variable should be set using the method of whichever shell you are using.

For example, one could use

setenv DANCE_DIR < *pathname* >/dance

or

set DANCE_DIR=< *pathname* >/dance

In addition, set the LD_LIBRARY_PATH variable to include $DANCE_DIR/lib/$OS where $OS is either "Linux" or "Irix" depending on the platfrom you use. For example,

setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${DANCE_DIR}/lib/Linux

or

setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${DANCE_DIR}/lib/Irix

Lastly, set your path variable to include < *pathname* >/dance/bin. For some systems you may have to do a "rehash" so that the shell can find the dance executable. "Rehash" updates the hash table that some shells use to quickly locate applications.

2.3 Run-only Version

The Run-only version is a restricted distribution that contains only the files necessary to operate the DANCE application and its installed plug-ins. It is not possible to develop with DANCE, as the source directories for the core program and the plug-ins, the include files, and the link libraries are all not included. This distribution is useful if you want to distribute DANCE only for demonstration purposes. In the $DANCE_DIR/scripts directory, you will find a shell-script called **CreateRunTime**. This script contains the unix shell commands needed to create a run-time version of DANCE. The script should be run from the desired destination directory where you would like to install dance and the required subdirectories. This script will work in a Windows environment if the Cygnus tools are installed. Alternatively, you can replace the unix commands with their DOS equivalents (for example, replace cp with copy) to create a batch file which does the equivalent operations.

3 Compiling DANCE

Before compiling DANCE make sure you have properly set DANCE_DIR to <pathname>/dance.

3.1 Unix platforms

To compile DANCE, type "make linux" or "make irix" depending on your operating system at the root directory $(DANCE_DIR). If you get make errors make sure that your operating system is compatible to Redhat 6.xxx or Irix 6.xx.

3.2 Windows platforms

We provide project files for MS visual studio version 6. Load the work space dance.dsw and make all the projects. Load $(DANCE_DIR)/dance.dsw and make all the projects.

4 Running the demos

Under UNIX: If you have set the path properly you should be able to start dance by typing "dance<Enter>" in any window.
Under Windows: Make "dance" the active project and use the execute command of visual studio. If for some reason it is not already defined, define bin/dance.exe as the main executable through Project → Settings.

You can run a first example by opening the following session $(DANCE_DIR)/run/pend/inp.tcl You can do that in two ways:

```
GUI:
     File -> Open Session
command console using the "<" command:
     <prompt> < run/pend/inp.tcl
```

In the unix version the program will pause while compiling the dynamic simulator provided by Symbolic Dynamics Inc. The compiled simulator is automatically linked in and the system and then the program will continue normally. For the windows version, there is no automatic compilation of simulator code, so you have to build the appropriate projects in visual studio.

If everything loads successfully you should see a two link dynamic pendulum swinging while it emits kinematic particles form its base. Use the left mouse button to rotate the screen the middle button to move the camera and the right button to zoom in and out. If you have a two button mouse then the right button moves the camera if you hold down the ¡Shift¿ key.

Check the file run/pend/README.txt for more information on the demo.

5 DANCE Overview

The main aim in the design of DANCE was to create a system with a central part that implements most of the common tasks that animation systems need to implement, and that offers a complete set of APIs through which programmers can implement any special functionality they desire. We call the central part the *core* of DANCE. The core provides a number of classes, called core classes, that provide the neccessary APIs. Any other module, is a subclass of one of the core classes and it is implemented as a dynamically linked library (plugin). The next section describes the dance classes in more detail.

5.1 DANCE classes

The DANCE main class hierarchy is depicted in Figure 2. `DanceObject` is at the top of the hierarchy. It provides the general functionality that any object within dance should have. Class `Plugin` provides the functionality that all dynamically linked objects need in order to be recognized and properly loaded into a running DANCE session. `DanceObjectList` implements a linked-list structure for `DanceObject` instances. The rest are the core classes of DANCE:

- `DSystem.` It refers to virtual objects and creatures that can populate a virtual environment.
- `DSimulator.` Simulators compute the state of systems in time.
- `DActuator.` It is the base class for actuators, controllers and anything that can apply forces or otherwise affect the state of a system.
- `DGeometry.` It is the base class for objects that implement geometric representations.

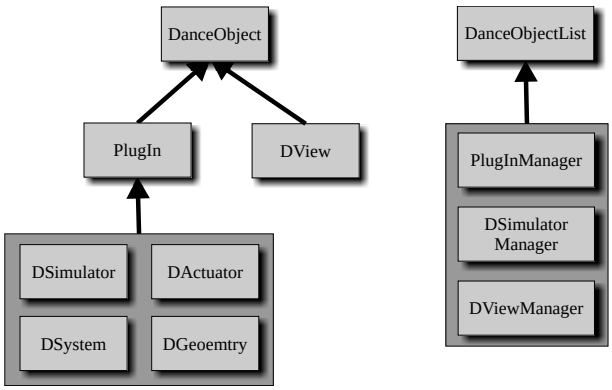

Figure 2: DANCE class hierarchy

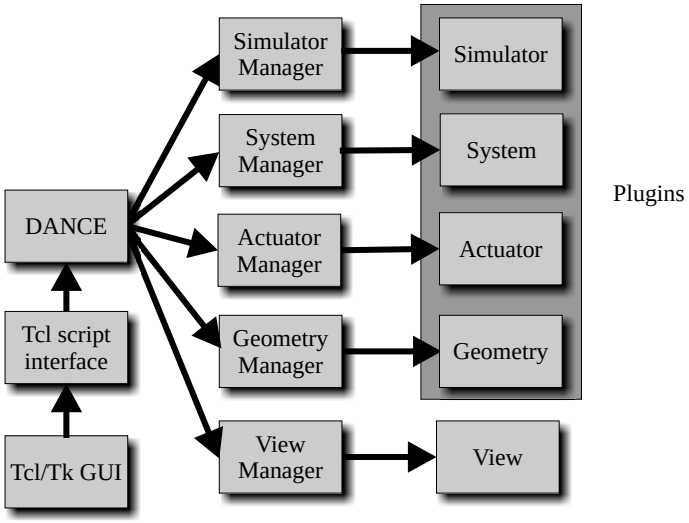

Figure 3: DANCE architecture

- `DView.` Views are windows to the virtual world.
- `DLight.` Lights can be used to illuminate a scene.

Note that only the four first classes can be subclassed in plugins. We will present these classes in detail later on.

5.2 DANCE modules

Figure 3 shows an overview of DANCE's architecture and its modules. The driving module is *dance* implemented by the `dance` static class. It provides the main loop unless views are present in which case it uses the Glut main loop. The interface, is implemented by the `danceTcl` static class and it is based on the Tcl/Tk library, see Section 6.

5.3 Plugins

DANCE supports the dynamic linking of plugins, which are kept in a list of type `PlugInManager: public DanceObjectList` called `AllPlugIns`. This list performs booking operations for all plugins. Each plugin belongs also to a list according to its base type. Theses lists are: `dance::AllActuators, dance::AllSystems, dance::AllLights, dance::AllGeometry, dance:AllViews, dance::AllSimulators`. These lists are exported and they can be accessed directly by plugin code.

5.4 Event Queues

There are two event queues in DANCE. The Tcl/Tk queue and the Glut queue. Both Tcl/Tk and Glut provide main loop functions that have to be given control of the appilcation. The co-existence of the two event queues is implemented as follows. DANCE starts by giving control to the Tcl/Tk main loop via the `danceTcl::InitTclTk` method which calls `Tcl_Main` or `Tk_Main`. `Tcl_Main` is called if the "-n" batch option (no GUI or graphic windows), is given when starting DANCE, otherwise `Tk_Main` is called. `Tcl/Tk_Main` has control until the first `View` is created. `View::Create` sets a timer event that passes control to the Glut main function via the `dance::MainLoop` method. The Glut main loop calls repeatedly the `dance::idleCB` method which is the main loop function of DANCE. This function also process Tcl/Tk events via the `Tcl/Tk_DoOneEvent` method.

6 DANCE's user interface

DANCE's interface is based on Tcl/Tk. Class `danceTcl` is responsible for the connection between DANCE and Tcl/Tk. It instantiates a Tcl interpreter using `Tcl_Init`. If DANCE is started without the "-n"[2] option, then Tk functionality is added using `Tk_Init` and the file "${DANCE_DIR}/bin/danceshell.tcl" is loaded to create the top level GUI. By running scripts that contain Tk widget commands the user can create compex user interfaces. Apart from `Tk_Init` and `Tk_DoOneEvent`, the core part of DANCE doesn't contain Tk commands. In other words, both the DANCE command/input mechanism and its GUI are fully scripted. DANCE itself understands only Tcl commands and is unaware of the GUI's presence. This design allows DANCE to run the same way both in batch background or interactive mode. Section 7.2.1 explains how plugins can use the interface mechanism and create their own GUIs.

7 Writing Plugins

DANCE supports four different types of plugins: simulators (class `DSimulator`), systems (class `DSystem`), actuators (class `DActuator`), and geometries (class `DGeometries`). However, all plugins share some common functionality which is realized by the class `PlugIn`.

7.1 Class `PlugIn`

A common problem when working with plugins is how to get access to the constructor of the subclass when the name of the subclass is not available to the core code. In DANCE the solution is provided by the plugin API:

```
class DLLENTRY PlugIn : public DanceObject {
public:
        virtual PlugIn *create(Tcl_Interp *interp, int argc, char **argv) {return NULL;};
        virtual int commandPlugIn(ClientData clientData, Tcl_Interp
                        *interp, int argc, char **argv) { return TCL_OK;};
        virtual int initInterface(void) ;
        // Appends contents to filename.
        virtual void printFile(char *filename) { return; };
protected:
        Tcl_Interp *interpreter;
};

// Factory function for creating instances of plugins.
extern "C" DLLEXPORT PlugIn *Proxy(void);
```

Every plugin implements a static function called `Proxy` that returns a base class pointer (`PlugIn`) that points to a subclass dummy object. For example, if we have a plugin class "`class MyPlugin : public Actuator`" then `Proxy` would look like:

```
PlugIn *Proxy(void) { return new MyPlugin ;} ;
```

[2]This options is used when no graphical display is desired, and it is only available for the unix version.

When a plugin is loaded, the core code retrieves `Proxy` and uses it to obtain a base class pointer to an object of type `MyPlugin`. Through that object the core code can call "`create`" passing it the command line arguments that the user has provided. The dummy object is stored in the system so that more plugins of the same type can be instantiated at the user's command. However it is not added in the normal plugin lists such as `dance::AllPlugIns`. It is advisable that plugins provide a default constructor that can create a dummy instance which doesn't use much memory.

Method `printFile` is used when the user wishes to save a DANCE session. Note that despite accepting a file name it must *append* its output to the given file. Normally, we would have made `printFile` accept a file pointer but it seems that Microsoft Windows have problem with file pointers being shared with dynamically linked code.

Finally all plugins must include defs.h, dance.h, and danceTcl.h before anything else. These files are in ${DANCE_DIR}/src.

7.2 Plugins and user interface

The interface of DANCE is based on Tcl/Tk command line interpreter and Tk widgets. An author of a plugin needs to be concerned with three issues concerning the user interface and his/her plugin,

- How to provide access to the plugin from the user interface.
- How to provide a graphical user interface (GUI).
- How to access the command line interpreter from his/her code. For example how to print messages, execute script commands etc.

The next sections explain these issues.

7.2.1 Command line access to plugin code

Virtual method `commandPlugIn` provides a hook to the interface. It is the function that is called when the user issues the following at the command line:

```
<base type of plugin> <name of plugin> arg1 arg2 ... argN
```

For example, assume that we have a plugin of type `FieldActuator` and base type `DActuator` called "gravity". Then the command

```
< dance> actuator gravity magnitude -9.8
```

results in the function `commandPlugIn` being called with the following arguments:

```
clientData = NULL
interp = danceTcl::Interpreter
argc = 2
argv = {``magnitude'', ``-9.8''}
```

7.2.2 Graphical user interface

A plugin can have a GUI implemented by a Tcl/Tk script. This script can be executed explicitly by the user, or automatically at start time. For the latter, the script must reside in "${DANCE_DIR}/data/interfaces". DANCE read and execute all files found in this directory that have a ".tcl" extension. If the script contains a function

```
proc {editForceActuator} {name type} { ... }
```

then this function will be called when the user invokes Edit → <Plugin type> → Edit from the main GUI.

Allternatively, you can override the function `int PlugIn::initInterface(void)`. This function is called when a plugin is instantiated. This function can be used to execute Tcl/Tk scripts and link variables of the interface with variables of the plugin.

7.2.3 Plugin access to the interpreter

Plugins access the interpreter using the following static methods of class `danceTcl`:

- `static void OutputMessage(char *format, ...)`. Works like `printf` only its output goes to the interpreter's standard output. It is implemented using the `puts` Tcl function.
- `static void OutputResult(char *format, ...)`. Works like `OutputMessage` only its output is returned as the result of the associated interpreter command. That means that the the output of `OutputResult` can be used in scripts. It is implemented using `Tcl_AppendResult`.
- `static void OutputListElement(char *format, ...)`. Works as the two previous function only its output is inserted in the current interpreter list. It uses `Tcl_AppendElement`.
- `static void ExecuteCommand(char *command)`. Executes a command in the interpreter using `Tcl_Eval`.

7.3 Display and Direct manipulation

Plugins that have a visual representation must override the following two virtual functions defined in `DanceObject` class:

- `void output(int mode)`. This is the method that draws the plugin in the view. The bits of parameter `mode` defines whether we use shaded, wireframe,shadow, or selection mode. The value of mode is discussed in detail in Section 11.
- `BoundingBox *calcBoundingBox(BoundingBox *)`. This method returns a bounding box that DANCE can use to automatically adjust the view volume of a given view in order to fit the plugin's graphical representation.

Class `Bounding Box` is defined as

```
class DLLENTRY BoundingBox {
public:
        BoundingBox();

        BoundingBox *merge(BoundingBox *, BoundingBox *);
        void copy(BoundingBox *);
        void display(float r=0.0, float g=0.7, float b=0.0,float a=1.0);
        void update(double transMat[4][4]);
        void scale(double x, double y, double z);
        int update(Vector point, int init=0);

        void getDimensions(double dim[3]);
        int isEmpty();

        // Actual coordinates of box.
        double xMin, yMin, zMin ;
        double xMax, yMax, zMax ;
} ;
```

The author of a plugin only has to set the six values that form the bounding box.

Plugins displayed in view windows can be manipulated directly through the Glut indexGlut and OpenGL selection and picking mechanisms. The `DanceObject` class provides the API which allows plugins to access mouse events. The virtual functions that make up the direct manipulation API are

- `int InteractStart(Event *event)`. Called when the mouse is over a plugin and a button is pressed down.
- `void Interact(Event *event)`. Called when the mouse is being moved with a button pressed down.
- `int InteractEnd(Event *event)`. Called when the button is released.
- `int PassiveMotionCB(DView *view, int x, int y)` Called when the mouse is moved with no button pressed.

- `int KeyboardCB( unsigned char key, int x, int y )`. Called when a keyboard key is pressed and the mouse is in a view window. The arguments are the ASCII code of the key and the position of the cursor in the view window.

`Event *event` contains all the information of the mouse event such as the associated view window, the button involved, the button's state, the number of objects picked if OpenGL picking was activated and the pick buffer.

```
class Event {
public:
        DView *Window;
        int winDiffX, winDiffY;  // when mouse motions is involved
                                 // holds the difference in motion between this
                                 // and the previous event.
        int winX, winY;          // position of the mouse
        int winWidth, winHeight; // window dimensions

        double winBasisX[3], winBasisY[3], winBasisZ[3]; // the coordinate system
                                                         // of Window

        int buttonID;
        int buttonState;

        int pickNumber;    // number of picked items
        int pickItem[256]; // pick buffer

        inline void setWindow(DView *win, int x,int y, int width, int height)
                { Window = win; winX = x; winY = y;
                   winWidth = width; winHeight = height; };
        inline void setWindowDiffs(int diffx, int diffy)
                { winDiffX = diffx; winDiffY = diffy;};
        void setWindowBasis(double basisX[3], double basisY[3],double basisZ[3])
                { memcpy(winBasisX,basisX,3*sizeof(double));
                  memcpy(winBasisY,basisY,3*sizeof(double));
                  memcpy(winBasisZ,basisZ,3*sizeof(double)); };
        inline void setButtons(int ID, int state)
                { buttonID = ID; buttonState = state; };
        void setSelected(int number, int item[256])
                { pickNumber= number; memcpy(pickItem,item,256*sizeof(int));};
        int getSelected(int **item)
                { *item = pickItem; return (pickNumber); };
};
```

Plugins that wish to support direct manipulation must override these functions and have a visual representation by overriding the `output` method.

7.4 Loading Plugins

Plugins must reside in the directory "${DANCE_DIR}/plugins/${OS}" where ${OS} is one of (WinNT,Linux,Irix). Then they can be loaded using the command "instance"

```
< dance> instance <plugin type> <name> arg1 arg2 ... argN
for example
< dance> instance FieldActuator gravity
```

When instance is invoked DANCE loads the plugin, retrieves and executes `Proxy` which returns an object of the plugin's type. That object calls `PlugIn::create` and passes it the command line arguments "arg1 ... argN". In turn, `create` calls the constructor of the plugins and passes it the appropriate arguments.

8 Systems

Systems refers to entities that a user may want to animate or otherwise include in a scene. Their motion is usually the target of an animation. Examples of systems are articulated figures, particle systems, mass-spring models etc. Systems must follow the plugin API as described in Section 7 and the system API provided by the `DSystem` class. The most relevant part of this class is

```
        class DLLENTRY DSystem : public PlugIn {
        public:
                // If TRUE the system is being simulated
                int IsSimul(void) { return isSimul ;} ;

                virtual DSystem *GetGroup(int index) ;
                virtual DSystem *GetGroup(char *id) ;
                virtual int GetNumGroups(void) ;
                virtual double GetGroupMass(int id) ;
                /** returns the index of the first controllable dof in the state vector */
                virtual int GetIndxFirstConDef(void)  return 0 ;  ;

                /** returns a pointer to monitor points */
                virtual MonitorPoints *GetMonitorPoints(void) ;

/** called by the simulator */
                void InitSimulationBase(double time, DSimuator *sim)
                       { isSimul = TRUE ; InitSimulation(time, sim) ;} ;
                virtual int InitSimulation(double time, DSimulator *sim) ;
                void BeforeSimStepBase(double time, DSimulator *sim)
                       { m_events.process(interpreter, time); BeforeSimeStep(time, sim);} ;
                virtual int BeforeSimStep(double time, DSimulator *sim) ;
                void AfterSimStepBase(double time, DSimulator *sim)
                       { AfterSimStep(time, sim) ; } ;
                virtual int AfterSimStep(double time, DSimulator *sim) ;
                void AtEndSimulBase(double time, DSimulator *sim)
                       { isSimul = FALSE; AtEndSimul(time, sim);} ;
                virtual int AtEndSimul(double time, DSimulator *sim) ;

                DSimulator *simulator ;
                int isSimul ; // shows if the system is under simulation
                SimulatorEvents m_events ;
} ;
```

Systems can have groups, for example in the case of an articulated figure groups could refer to links or joints. The groups are of type `DSystem` as well and therefore can have groups of their own. `DSimulator *simulator` is a pointer to the simulator associated with the object. It is set by the `DSimulator::AddSystem` method. The flag `isSimul` shows whether the object is currently simulating or not. Functions `BeforeSimStepBase, AfterSimStepBase, InitSimulationBase, AtEndSimulationBase` must be called by the simulator at the appropriate time. In particular:

- `InitSimulationBase.` Must be called at the beginning of a simulation session.
- `AtEndSimulBase.` Must be called at the end of a simulation session.
- `BeforeSimStepBase.` Must be called before each step of the simulation.
- `AfterTimeStepBase.` Must be called after each step of the simulation.

At the current implementation the `DSimulator` base class makes these calls at the proper points. The field `SimulatorEvents m_events` holds commands that will be executed by the main interpreter at specific times during the simulation. Any

command that can be given at the DANCE interpreter can be registered as an event. For example, to do a directory listing at time 1.5secs during the simulation of system "man"

```
<dance> system man simul_event add 1.5 "ls"
```

Events are automatically removed after being executed.

`MonitorPoints` class implements a simple structure that keeps a number of points that reside on the system for collision detection purposes. The definition of the class is

```
class DLLENTRY MonitorPoints {
public:
  MonitorPoints() ;
  ~MonitorPoints() ;
  int Allocate(int n) ;
  void Deallocate(void) ;

  int m_NumPoints ;
  Vector *m_Point ;
  Vector *m_PrevPos ;

  double *m_PrevPosTimeStamp ;
  int *m_InCollision ;
} ;
```

We usually cosider the following: m_Points are the monitor points in local coordinates, m_PrevPos is a previous position of monitor points in world coordinates and m_InCollision shows whether the points are involved in a collision or not. However, developers can use these structures any way they want. Systems typically have a geometry, inherited from the `DanceObject class`, which normally provides the monitor points, see Section 11. It is the responsibility of the author of a system class to mark all the monitor points as not being in collision at the start of every simulation. That is the field `m_InCollision` has to be set to `FALSE` for each collision point typically by DSystem::InitSimulation().

9 Actuators

Actuators are entities that can be applied to a system or other actuators to produce some effect. They usually tend to affect the state of a system in some way. For example, field forces, kinematic or dynamic controllers, biomechanical muscles and collision resolution can be implemented as actuators. Actuators inherit from the classes `DanceObject` and `PlugIn`. The API is provided by the `DActuator` class

```
class DLLENTRY DActuator: public PlugIn {
        int applyAllObjects;
        AppliedObject *applyList;

public:
        int controlled;

        int IsInApplyList(DanceObject *sys) ;
        void removeAppliedObject(DanceObject *sys) ;
        inline void setAppliedObject(DanceObject *sys) ;
        inline void setAppliedAllObjects(void) {applyAllObjects = 1;};

        // All virtual functions must be called by the simulator at
        // the appropriate place
        virtual void ExertLoad(DSystem *sys, double time, double dt,
                double *state, double *dstate) ;
        virtual void InitSimul(DSystem *sys, double time) ;
```

```
        virtual void BeforeStep(DSystem *sys, double time) ;
        virtual void AfterStep(DSystem *sys, double time) ;
        virtual void AtEndSimul(DSystem *sys, double time) ;

        // from DanceObject class
        virtual int IdleCB(void) ;
};
```

The base class maintain a list objects that the actuator is applied to. These objects include systems or other actuators. An actuator is applied to an object through the command interface as follows

```
< dance> actuator <actuator name> apply <system/actuator name>
for example
< dance> actuator gravity apply man
```

The function `IsInApplyList` can be used to check whether an object belongs in the list of applied objects of the actuator. If an actuator is controller by another actuator then the field `controlled` is set to `TRUE`. Method `ExertLoad` provides the hook to the simulator. It is the method that simulators call when they need to compute the forces that are acting on a system. The parameters passed to these method are the system under simulation, the current time, the state and its derivative. This function will be called for every actuator because simulators do not know which actuators are applied to the system they are simulating. Therefore each implementation of `ExertLoad` must check, using method `IsInApplyList` whether the first parameter is a system that belongs to the actuator's list of applied systems. The same applies to the rest of the virtual functions:

- `InitSimul.` Called when the simulation begins.
- `BeforeStep.` Called before the simulator takes a step.
- `AfterStep.` Called after the simulator takes a step.
- `AtEndSimul.` Called when a simulation session finishes.

Method `IdleCB` is inherited from class `DanceObject` and it provides a hook to the main loop. It returns 1 if the actuator did something that needs a redrawing of the views, 0 otherwise.

10 Simulators

Simulators update the state of systems in time. They can be dynamic or kinematic. Generally, the implementation of a simulator may be closely tied to the state representation of a system. For example, the nature of the dynamics of a system will influence the choice of numerical integration technique for simulation. Simulators can control a number of different systems. Each simulator has a list of systems that are under its control.

10.1 Main simulator API

The simulator API provides a number of overloaded methods in an attempt to satisfy the needs of different simulators. A programmer does not have to implement all those functions. In this section we will describe the part of the simulator API, provided by class `DSimulator`, that must be overriden so that a simulator can function properly within the framework of DANCE. This part is shown below:

```
class DSimulator : public PlugIn {
        virtual int SetTime(double currentTime) ;
        virtual double GetTime(void) ;
        // Advance state in time function
        virtual int Step(DSystem *sys, double destinationTime) ;
        // initialization function for every run of the simulation
        virtual int Start(double time) ;
```

```
        // clean up after simulation is stopped
        virtual int Stop(void) ;
};
```

Method `Step` is the most important one. It updates the state of the simulated system from the current time to a time no greater than `destinationTime`. If the plugin doesn't advance the time to `destinationTime` then the core code will call the plugin's `Step` function repeatedly until the desired time is reached. This allows every plugin simulator to have its own timestep, while the core can control what happens before or after every step. The implementation of `StepBase` shown in Section 10.6 will demonstrate better how this works.

10.2 Simulator and system interaction

The way simulators take control of systems is implemented by the base class DSimulator. The corresponding interface commands are discussed in Section 12.8. If the designer of the simulator wishes to bypass the interface mechanism then he/she should make sure that the method `DSimulator::AddSystem` is called at some point to update the list of applied systems.

If the simulator needs to initialize its state from the systems data structure the method `DSimulator::SetState FromObjectState` can be used. This method is called at the beginning of a simulation as shown in Section 10.6. Similarly, if the system's state must be updated given the new state computed by the simulator the method `DSimulator::SetObject StateFromState` can be overriden. Alternatively the designer of a simulator can create his/her own methods equivalent to the above as long as they are called at an appropriate place somewhere in the code.

10.3 Applying forces

The simulators must also provide hooks to actuators so that actuators can apply forces to the systems they control. However, the designer of a simulator is free to include all kinds of control and actuation within the simulator. We believe that pushing the control to actuator results in more modular designs and allows for code reuse. In any case, if a system is controlled by external (to the simulator) actuators, then the simulator must override a number of force methods provided by the `DSimulator` API:

```
// Force functions
virtual void FieldForce(double *force) ;
// force vector and point relative to the given part of the object
virtual void PointForce(int group, double *point, double *force) ;
// point and force in world coordinates
virtual void PointForce(DSystem *sys, double *pointWorldCoord, double *force) ;
// the most general form of point force
virtual void PointForce(int argc, char *argv[]) ;

virtual void BodyTorque(int group, double *torque) ;
virtual void GeneralizedForce(int coord, double force) ;
virtual void GeneralizedForce(int *index, double *force, int size)  ;
virtual void GeneralizedForce(double *Q) ;
virtual void GeneralizedForce(int group, int subgroup, double force) ;
```

The comments about the arguments of the above methods are an indication of how we have used them in our simulators and actuators. Someone who designs both the actuators and the simulator is free to use them anyway he/she wants.

10.4 Simulator and actuator interaction

Actuators typically override the `DActuator::ExertLoad` method in order to apply forces to a system during its simulation. It is the responsibility of the plugin simulator to call this method for every system they control. For convenience we provide the `DSimulator::ApplyActuatorForces()` that does this on a per system basis,

```
void DSimulator::ApplyActuatorForces(DSystem *sys, double time, double dt, double *state,
   double *dstate)
{
    for (int i=0; i < dance::AllActuators->size(); i++) {
        DActuator *wact = (DActuator *)dance::AllActuators->get(i);
        if (wact->controlled == 0) wact->ExertLoad(sys, time,dt,state,dstate);
    }
}
```

10.5 Providing information

Actuators such as controllers may need various quantities related to the systems they control. For example they may need the position in world coordionates of a point in the system or its velocity. The `DSimulator` API provides a number of methods that simulators can override in order to communicate information to associated actuators:

```
virtual void GetVel(DSystem *sys, double *localPoint, double *vel)  ;
virtual void GetVel(int group, double *point, double *vel) ;
virtual void GetAcc(int group, double *point, double *acc) ;
virtual int GetAcc(DSystem *sys, double *localPoint, double *acc) ;
virtual int GetPosition(DSystem *sys, double *localPoint, double *position) ;
virtual int GetPosition(int group, double *localPoint, double *position) ;
virtual int GetOrientation(DSystem *sourceSystem, double *sourceVec,
             DSystem *targetSystem, double *targetVec) ;
virtual int GetOrientation(int sourceGroup, double *localVec, int targetGroup,
             double *rotated) ;
 virtual int Transform(int argc, char **argv) ; // generic
```

10.6 Base class tasks

In this section we present some important functionality provided by the base class `DSimulator`. Note that the methods presented below are part of the core and they are only presented here so that the reader gets a better picture of how things work.

Systems and actuators need to be initialized before a simulation starts and they may need to perform a variety of tasks before or after each simulation step. The base classes `DSystem` and `DActuator` provide a number of methods that plugins can override to perform initialization, clean up and pre/post step tasks. The base class `DSimulator` ensures that these methods are called properly by implementing the following base methods which are called by the simulator manager of DANCE.

```
// Base class initialization of a simulator
int DSimulator::StartBase(double time)
{
    // first Initialize the plugin simulator
    int res = Start(time) ;

    // if the author of the plugins wishes so, suppres
    // the base class function and return immediately.
    if( m_ignoreBaseFlag == TRUE ) return res ;

    // Initialize all systems controlled by this simulator
    // and for each of these systems initialize all actuators.
    for( int s = 0 ; s < m_systems.size() ; s++ )
    {
        DSystem *sys = (DSystem *) m_systems.get(s) ;

        // Initialize the system for simulation
```

```
        if( m_initSystemsFlag == TRUE ) sys->InitSimulationBase(time, this) ;
        // Set the internal simulator state from the current state of the object
        SetStateFromObjectState(sys) ;
        // Initialize the actuators and pass them the current system
        // unless the author of the plugin has specified otherwise
        if( m_initActuatorsFlag == TRUE )
        {
            for (int i=0; i < dance::AllActuators->size(); i++) {
                DActuator *wact = (DActuator *)dance::AllActuators->get(i);
                wact->InitSimul(sys, time) ;
            }
        }
    }

    return res ;
}

// Base class stop method
int DSimulator::StopBase(void)
{
    // call the plugin simulator's Stop method
    int res = Stop() ;

    // if the author of the plugin wishes ignore the base code
    // and return immediately
    if( m_ignoreBaseFlag == TRUE )  return res ;


    // Call the AtEndSimul method of every system controlled by this
    // simulator
    for( int s = 0 ; s < m_systems.size() ; s++ )
    {
        DSystem *sys = (DSystem *) m_systems.get(s) ;
        sys->AtEndSimul(GetTime(), this) ;
    }
    return res ;

}

void DSimulator::StepBase(double destTime)
{
    int i ;
    double t = GetTime() ;

    // if the author of the plugin wishes to ignore the base code
    // call the Step method for each system and return immediately.
    if( m_ignoreBaseFlag == TRUE )
    {
        for( int s = 0 ; s < m_systems.size() ; s++ )
            Step((DSystem *) m_systems.get(s), destTime) ;
        return ;
    }
```

```
    // Advance the state of each object until we reach
    // the time specified by the calling routine.
    while(GetTime() < destTime )
    {
        for( int s = 0 ; s < m_systems.size() ; s++ )
        {
            DSystem *sys = (DSystem *) m_systems.get(s) ;
            sys->BeforeSimStepBase(t, this) ;
            for (i = 0; i < dance::AllActuators->size(); i++) {
                DActuator *act = (DActuator *)dance::AllActuators->get(i);
                act->BeforeStep(sys, t) ;
            }
            // advance the state of the system no further than destTime.
            Step(sys, destTime) ;
            // update the system's state if applicable
    SetObjectStateFromState(sys) ;
            sys->AfterSimStepBase(t, this) ;
            for (i = 0; i < dance::AllActuators->size(); i++) {
                DActuator *act = (DActuator *)dance::AllActuators->get(i);
                act->AfterStep(sys, t) ;
            }
        }
    }
    return ;
}
```

The author of a plugin can set a couple of flags that the prevent the base code from running. Setting `DSimulator::m_ignoreBase Flag` to `TRUE` prevents any base code from running. In that case, `StartBase, StepBase, StopBase` become equivalent to `Start, Step, Stop` respectively. Setting the flag `m_initSystemsFlag` and `m_initActuatorsFlag` to `FALSE` prevents systems and actuators respectively from initializing.

10.7 Simulation Manager

The simulation manager controls the general functionality and parameters of a simulation. In particular it supports the following commands:

- `Start.` Starts the simulation for all simulators. Calls the `StartBase` method and sets its state to SM_RUNNING. `StartBase` calls the `Start` method of all simulators.
- `Pause.` Pauses the simulation and sets the state to SM_PAUSED.
- `Cont.` Continues the simulation after a pause or works as `Start` if the simulation was not running. Sets the state to SM_RUNNING.
- `Stop.` Stops the simulation and calls `BaseStop` which calls `Stop` for all simulators so that they can do clean up or completion operations. Sets the state to SM_NOT_RUNNING.

Several options can be set to customize a simulation run. The general syntax is:

```
< dance > simul [-setDislayTimeStep <dt>]
                [-setSimulationTimeStep <dt>]
                [-setEndTime <end>]
                [-setCurrentTime <t>]
                [-setAutoMake on/off]
                [-block on/off]
                [start|stop|pause|cont]
```

The options work as follows:

- setDisplayTimeStep. Sets the time interval in seconds between subsequent scene redraws. A typical value is 0.03. Note: this is not wall clock time, it refers to simulation time. For example, a display time step of 0.03 seconds means that the system will redraw the scene every time the simulators advance 0.03 simulation seconds. The actual wall clock time might be less or more than 0.03 seconds depending on how a simulation seconds compares to an actual second.
- setSimulationTimeStep. Sets the time interval that every simulator is requested to move forward in time at each iteration. The smaller this variable is the closer the simulators are to being synchronized. Note: individual simulators can have their own time step which should be less or equal to the simulator manager's time step.
- setCurrentTime. Sets the current time. It is usually used to set the starting time to the desired value.
- setEndTime. Sets the time at which the simulation ends.
- setAutoMake. If set to on then when a *start* command is invoked the simulator manager will loop through all the systems and invoke their *BuildAndLinkSimulator* method. Systems can implement this method to build, compile and link a default simulator.
- block. If set to on, the simulation runs without executing Tcl/Tk or Glut events. This is neccessary for running batch jobs because of the way Tcl/Tk executes script files. However, this option doesn't affect the display update.

10.8 Using many simulators

DANCE supports multiple simulators in a layered fashion. Simulators take a step in sequence. The magnitude of this step is controlled by the simulator manager, see Section 12.9. Layered methods have problems when systems controlled by different simulators interact. Since simulators are invoked one after the other, interacting systems are considered at different points in time. For DANCE, this time difference is equal to the simulator manager's simulation time step. Reducing the time step of the simulator manager reduces but doesn't solve the problem. For many applications the layered method produces acceptable results. In future versions of DANCE we may solve this problem in a better way.

11 Geometries

Class `DGeoemtry` provides the API for visual and geometric representation of objects,

```
class DLLENTRY DGeometry: public PlugIn {
protected:
        virtual int createMonitorPoints(int numPoints)  return 0 ;  ;
public:
        RAPID_model *m_RAPIDmodel;

        // returns a triangle that corresponds to rapid id
        virtual int GetRapidTriangle(int id, Vector v1, Vector v2, Vector v3)  ;
        virtual int PrepareCollision(void) ;

        void getBounds(double dim[3]);
        inline void setTransparency(float transparency) m_Transparency = transparency;;
        inline void setColor(float r, float g, float b, float a) ;

        virtual void display(int style=0) = 0 ;
        void displayBoundingBox(float r=0.0, float g=1.0, float b=0.0, float a=1.0);
        virtual BoundingBox *calcBoundingBox(BoundingBox *box=NULL) ;

        virtual void calcInerTensor(double inerTensor[3][3], double mass) ;
        virtual void computeMoments(double den, double *mass, double inerTensor[3][3]) ;
```

```
        virtual void getEndEffector(Vector p) ;

        virtual int commandPlugIn(ClientData clientData, Tcl_Interp
                        *interp, int argc, char **argv) ;

        virtual void Center(void) ;
        virtual void Rotate(char *axis, double degrees, int center = 0) ;
        virtual void Translate(double x, double y, double z) ;
        virtual void Scale(double sx, double sy, double sz, int center = 0) ;
        virtual void SetOrigin(Vector or) ;

        // Monitor point functions
        int assignMonitorPoints(int npoints) ;
        int assignMonitorPoints(int n, Vector *p) ;
        void getMonitorPoints(double (*m)[4][4], Vector *pts) ;
        int getMonitorPoints(Vector **pts) ;
        int getMonitorPointsPrevPos(Vector **pts) ;
        int getNumMonitorPoints(void) ;
        void getMonitorPointsPrevPosition(double (*m)[4][4], Vector *pts) ;

        MonitorPoints * getMonitorStruct(void) ;
    } ;
```

Method `display` draws the geometry according to the given style. Parameter `style` is a bitwise logical "OR" of the following symbolic constants

- `LDISPLAY_WIRE.` Specifies wireframe rendering.
- `LDISPLAY_MONITOR.` Indicates that we should draw monitor points as well.
- `LDISPLAY_SHADOW.` Indicates that we are rendering shadows.
- `LDISPLAY_SOLID.` Indicates that solid (filled) rendering.
- `LDISPLAY_SELECTION.`Indicates that we are rendering for selection purposes.

For example, a value of '`LDISPLAY_SOLID | LDISPLAY_SHADOW` for `style` must result in the rendering of solid shadows.

The API provides support for collision detection. In the current implementation there are two different ways that can help to detect collisions. *Monitor points* and *RAPID*[3]. *Monitor points* are usually given in the geometry coordinate system and their position can be used to detect collision between objects. *Rapid* is a robust and powerful library based on triangles and oriented-bounding-boxes. We use monitor points in a flat ground actuator to detect collisions between objects and flat ground patches. For object to object collisions we have created an actuator that handles collisions based on the RAPID library. Note: that the `DSystem` class described in Section 8 has a method to return monitor points. It is the responsibility of a system's author to ensure that these methods are coordinated. For example, he/she may want to return the geometries monitor points when geometry is present.

12 DANCEscript commands

DANCE interface is primarily a Tcl/Tk interpreter. The core code of DANCE is not aware of the graphical user interface that can be built using Tk script commands. This way the GUI of DANCE is highly customizable. Users can create their own GUIs for DANCE itself or particular plugins on the fly. This functionality was one of the main reasons why we chose Tcl/Tk for DANCE's interface.

Figure 4 shows the opening graphical user interface implemented by bin/danceUI.tcl. The main console can be started by chosing `Tools`→`Console` while the main simulation panel can be invoked by chosing `Tools`→`Simul Panel`.

[3]Information on RAPID is available at http://www.cs.unc.edu/ geom/OBB/OBBT.html.

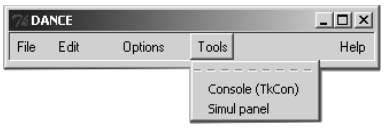

Figure 4: DANCE opening interface.

The user can run scripts in three dirrerent ways. Through the GUI: `File` → `Open Session`. Through the script interface using either "source", the built in Tcl/Tk command, or "<" which a DANCE wrapper for "source" that performs synchronization between the GUI and DANCE. It is recommended to use the GUI or "<".

The following is a limited reference guide for various DANCE commands:

12.1 plugin

syntax: `plugin <Plugin Type>`
Loads the code of a plugin without creating any instances. This command is useful when a plugin depends on another one, for example it inherits from it.

12.2 instance

syntax: `instance <Plugin Name> <name> args`
Creates instances of plugins. For example:

```
> instance FieldActuator gravity magnitude 0 -9.8 0
```

creates a plugin called `gravity` of type `FieldActuator` and passes it the arguments "magnitude", "0", "-9.8", "0".

The plugin code must reside in the appropriate plugin directory depending on the operating system of your machine.

12.3 show

syntax: `show system|actuator|simulator|geometry|light`
Shows the names of the objects of the given category that are in the system.

12.4 view

syntax: `view <view name>` *commands*
Available commands are:

`fitview`:
Recalculates the view volume based on the bounding boxes of the objects present in the system.

`create persp|top|right|front <x> <y> <width> <height>`
Creates a view at point `<x,y>` with dimenstion `<width>` × `<height>` which is either a perspectiv (persp) or one of the orthographic views.

`background <r> <g> <b> <a>`
Changes the background color to the given red, green, blue, alpha value.

`projtype persp|top|right|font`
Changes the type of the view

`dump [-n<numberOfFirstFrame>|-f<filename>|-d<directory>|start]|stop`
This command is used to create a series of frames in PPM raw format. The `start` command results in a frame being saved in ppm raw format every time the view redisplays. It produces a sequence of frames properly numbered. If no "-f" option is used the prefix defaults to "f". For example the command:

```
view test dump -dMyDirectory -fkey start
```

will save frames as follows: MyDirectory/key1.ppm MyDirectory/key2.ppm .....
The `stop` command stops the saving of frames. `geometry`

Returns a string that defines the geometry of the view.

`refresh`
Makes Glut redraw the scene.

`lights on/off`
Turns lighting on or off.

`solids on|off`
Turns solids on or off.

`shadows on|off`
Turns shadows on or off.

`clipsBounds <left> <right> <bottom> <top> <near> <far>`
Sets the view volume clipping planes.

12.5 system

syntax: `system <system name> args`
Passes "args" to the `commandPlugin` method of the given system. For example:

```
> system humanModel set mass 70
```

12.6 actuator

syntax: `actuator <actuator name> args`
Passes "args" to the `commandPlugin` method of the given actuator. For example:

```
> actuator gravity magnitude 0 -9.8 0
```

In addition every actuator through the base class supports the following command:

```
> actuator <actuator name> apply <object name>|all
```

where `<object name>` is the name of an actuator, system, light or view. If 'all' is specified then the actuator is applied to all objects.

12.7 geometry

syntax: `geometry <geometry name> args`
Passes "args" to the `commandPlugin` method of the given geometry. For example:

```
> geometry hipGeometry export_obj hip.obj
```

12.8 simulator

syntax: `simulator <simulator name> args`
Passes "args" to the `commandPlugin` method of the given simulator. For example:

```
> simulator humanSimul save state human.state
```

In addition every actuator through the base class supports the following commands:

```
> simulator <simulator name> apply <system name>
> simulator <simulator name> remove <system name>
```

where `<system name>` is the name of a system. Using these commands a simulator takes or loses control of a given system.

12.9 simul

This command controls the simulator manager.

```
syntax: simul [-setDislayTimeStep <dt>]
              [-setSimulationTimeStep <dt>]
              [-setEndTime <end>]
              [-setCurrentTime <t>]
              [-setAutoMake on/off]
              [-block on/off]
              [start|stop|pause|cont]
```

The parameters have been explained in Section 10.7.

12.10 light

```
syntax: light [position x y z w|
               diffuse|ambient|specular r g b a]
```

The four parameter position follows the OpenGL convention. The four color parameters correspond to the red, green blue, and alpha channels respectively.

13 Troubleshooting

This is a table of various problems that may arise while compiling or running plugins.

Problem	**Posssible Solution**
Core classes multiply defined	Include `src/defs.h` at the top of your file
The system cannot find the Tcl/Tk libraries	Set variable LD_LIBRARY_PATH to include $DANCE_DIR/Linux (or Irix)
I have added $DANCE_DIR to my path but the system cannot locate dance	Some shells require a "rehash" before they can locate applications that reside in a newly added directory
Cannot load a plugin	(a) The shared library must have the same name as the plugin class (b) The plugins depend on another plugin which must be loaded first (c) The plugin must be in "${DANCE_DIR}/plugins/$OS/ /<PluginName>.[dll/so]"
Inconsistent dll declaration	Make sure "defs.h" is the first included file
Simul button doesn't work	Make sure every system is associated with a simulator
Cannot move the camera	Go to camera edit mode: Edit → Interact → camera
The objects appear clipped out	Recalculate the view volume by executing `view <viewname> fitview`
Object are still clipped out	Make sure you provide bounding boxes for your objects
Compilation complains about TkMain	Make sure you are using the Tcl/Tk include files and libraries provided by DANCE.

14 Particle system example

This is an example of a simple particle system that we have implemented which can be found in the DANCE distribution, "${DANCE_DIR}/systems/ParticlePointSystem". The header file is as follows

File: *systems/ParticlePointSystem/ParticlePointSystem.h*

```
#ifndef ParticlePointSystem_h
#define ParticlePointSystem_h 1

#include "defs.h"
#include "DSystem.h"
#include "ParticlePoint.h"
#include "ParticlePointList.h"

class BoundingBox ;

class DLLEXPORT ParticlePointSystem : public DSystem {
public:
  ParticlePointSystem() ;
  ~ParticlePointSystem() { return ;}  ;
```

```
  // override virtual methods of class PlugIn
  PlugIn *create(Tcl_Interp *interp, int argc, char **argv) ;
  int commandPlugIn(ClientData clientData, Tcl_Interp
            *interp, int argc, char **argv) ;

  // overide virtual functions of class DanceObject
  void output(int mode);
  int GetStateSize(void) { return 0 ; } ;
  void GetState(double *state) { return ;} ;
  BoundingBox *calcBoundingBox(BoundingBox *b) ;

  // overrides the one in class DSystem
  int BeforeSimStep(double time) ;

  // stuff private to ParticlePointSystem class
  double m_lifeTime ;
  double m_birthRate ;                // particles per second
  ParticlePointList m_plist ;
  double m_posOffset[3] ;
  double m_initVel[3] ;
  double m_zMin ;
  double m_zMax ;
  double m_xMax ;
  double m_xMin ;
  double m_y ;
  double m_rotX ;
  double m_rotZ ;
  double m_length2 ;

  int ParticleGenerator(double time, double dt) ;
} ;
#endif
```

Note that all plugins must include the following files found in ${DANCE_DIR}/src: defs.h, dance.h danceTcl.h. In addition in the source file, "${DANCE_DIR}/systems/ParticlePointSystem/ParticlePointSystem.cxx" the following lines of code are neccessary

```
PlugIn *Proxy(void) { return (new ParticlePointSystem) ;}

PlugIn *ParticlePointSystem::create(Tcl_Interp *interp, int argc, char **argv) {
        ParticlePointSystem *f = new ParticlePointSystem ;
        if( f == NULL ) {
                danceTcl::OutputMessage("Cannot allocate memory!") ;
                return NULL ;
        }
        f->interpreter = interp ;
        return f ;
}
```

The simulator for this system is a simple simulator that applies gravity forces to the particles. Its header file is

FILE: *${DANCE_DIR}/systems/ParticlePointSimul/ParticlePointSimul.h*

```
   #ifndef ParticlePointSimul_h
   #define ParticlePointSimul_h 1
```

```
        #include "defs.h"
        #include <stdio.h>
        #include "DSimulator.h"

        class DSystem ;

        class DLLEXPORT ParticlePointSimul : public DSimulator {
        public:
             ParticlePointSimul() ;
             ParticlePointSimul(DSystem *) ;
             ~ParticlePointSimul() { return ; }  ;

             // overide virtual functions of PlugIn class
             PlugIn *create(Tcl_Interp *interp, int argc, char **argv) ;
             int commandPlugIn(ClientData clientData, Tcl_Interp
                         *interp, int argc, char **argv) ;

             // overide virtual functions of DSimulator class
             int Start(double time) ;
             int Step(double destTime) ;

        private:
             int step(double destTime) ;

             DSystem *m_sys ;
             double m_dt ;
             double m_time ;

             double m_ythreshold ;
             double m_fadeFactor ;
    } ;
#endif
```

Since the simulator is a plugin, its source code must include the following

```
        PlugIn *Proxy(void) { return (new ParticlePointSimul) ;}

        PlugIn *ParticlePointSimul::create(Tcl_Interp *interp, int argc, char **argv)
        {
                if( argc != 1  ) {
                        danceTcl::OutputMessage("usage: instance <simulator type> "
                        "<simulator name> <object name>") ;
                        return NULL ;
                }

                DSystem *ao = (DSystem *) dance::AllSystems->get(argv[0]) ;
                if( ao == NULL ) {
                        danceTcl::OutputMessage("No such object: %s", argv[0]) ;
                        return NULL ;
                }
                ParticlePointSimul *f = new ParticlePointSimul(ao) ;

                if( f == NULL ) {
                        danceTcl::OutputMessage("Cannot allocate memory!") ;
```

```
                return NULL ;
        }

        f->interpreter = interp ;
        return f ;
    }
```

The demo shows the particles bouncing off the ground. The height of the ground is hardcoded in the particle simulator which reflects the velocity of the particles when a collision occurs.

15 Sample script

This is a sample tcl script with comments that explain what each line does.

```
# load a geometry of type IndexedFaceSet called cube using the model file .../cube.vrml
instance IndexedFaceSet cube $env(DANCE_DIR)/data/models/cube.vrml

# load an actuator of type PlaneGround and call it ground
instance PlaneGround ground
# set the stiffness damping and friction parameter for the ground
actuator ground set model 27424 805 0.4
# set the geometry of the ground patch by giving the coordinates of the four corners
actuator ground set geometry -6.0 -6.0 -3.4 -6.0 -6.0 2.4 6.0 -6.0 2.4 6.0 -6.0 -3.4 10 10

# instantiate a 640x480 perspective view positioned at (100,100) and call it view1
instance view view1 persp 100 100 640 480
# set various parameters for the view1
view view1 lights on
view view1 solids on
view view1 shadows on

# instantiate a light and call it light1
instance light light1
# set various parameters for the light
light light1 position 0.2 2.5 1 0
light light1 ambient 0.2 0.2 0.2 1.0
light light1 diffuse 0.5 0.5 0.5 1.0
light light1 specular 0.5 0.5 0.5 1.0

# recalculate the view volume to fit all objects properly
view view1 fitview
#redraw the scene
view view1 refresh
```

16 Future Directions

DANCE is an ongoing work in progress and the majority of its potential will be realized by the power of its plug-ins. In that sense, the quality of the development community for DANCE plug-ins will determine its long-term success.

However, there are several specific areas we would like to improve or fill:

- The *Simulator* abstract class can handle different implementations of integrators for equations of motions of systems. Currently, we heavily use a plug-in based on SD/FAST for articulated objects. This limits DANCE's usefulness because

SD/FAST is a commercial product with a restrictive distribution license. We would like to implement a new publicly, distributable simulator for articulated objects.

- Constraints between system variables can be handled with DANCE, but is currently the responsiblity of the plug-in developer to provide and design the interface for such a mechanism. It would be advantageous to enforce a standard method of sharing system variables so that constraints between these variables can be formulated and maintained in a universal way amongst different system instances.
- The design of DANCE does not preclude a multi-threaded execution flow for the various simulators and systems it controls. Implementing a multi-threaded execution model within the DANCE core would help the scalability of DANCE to more complex physical environments with more interacting systems.

Index