August 1994

Jack/TTES: A System for Production and Real-Time Playback of Human Figure Motion in a DIS Environment

John P. Granieri
University of Pennsylvania

Jack/TTES: A System for Production and Real-Time Playback of Human Figure Motion in a DIS Environment

Abstract

This document describes a modified *Jack* system for off-line motion production and on-line (real-time) motion playback to an external IRIS-Performer-based host rendering system. This work was done in partial fulfillment of Contract #N61339-94-C-0005 for the US Marine Corps through NAWCTSD (Naval Air Warfare Center, Training Systems Division). The work described herein was contributed by several of the members of the Center for Human Modeling and Simulation: John Granieri (Design/Engineering/Integration), Rama Bindiganavale (animator, posture transitions), Hanns-Oskar Poor (animator, posture transitions, Hyeongseok Ko (walking and running motion), Micheal Hollick (locomotion playback control), Bond-Jay Ting (body sculpting), Francisco Azoula (body sculptin, anthropometry), Pei-Hwa Ho (body normalization), Jonathan Crabtree (Performaer, TIPS file format), Xinmin Zhao (slaving), Zhongyang Feng (DIS logfile player), Welton Becket and Barry Reich (terrain reasoning and reactive agent control).

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-94-42.

Jack/TTES: A System for Production and Real-time Playback of Human Figure Motion in a DIS Environment

MS-CIS-94-42
HUMAN MODELING & SIMULATION LAB 65

John P. Granieri

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

August 1994

Jack/TTES: A System for Production and Real-time Playback of Human Figure Motion in a DIS Environment

John P. Granieri
Center for Human Modeling and Simulation
University of Pennsylvania
Philadelphia, PA 19104-6389, USA
granieri@graphics.cis.upenn.edu

August 5, 1994

1 Overview

This document describes a modified *Jack* system for off-line motion production and on-line (real-time) motion playback to an external IRIS-Performer-based host rendering system. This work was done in partial fulfillment of Contract #N61339-94-C-0005 for the US Marine Corps through NAWCTSD (Naval Air Warfare Center, Training Systems Division).

The work described herein was contributed by several of the members of the Center for Human Modeling and Simulation: **John Granieri** (Design / Engineering / Integration), **Rama Bindiganavale** (animator, posture transitions), **Hanns-Oskar Porr** (animator, posture transitions), **Hyeongseok Ko** (walking and running motion), **Micheal Hollick** (locomotion playback control), **Bond-Jay Ting** (body sculpting), **Fransisco Azoula** (body sculpting, anthropometry), **Pei-Hwa Ho** (body normalization), **Jonathan Crabtree** (Performer, TIPS file format), **Xinmin Zhao** (slaving), **Zhongyang Feng** (DIS logfile player), **Welton Becket** and **Barry Reich** (terrain reasoning and reactive agent control).

2 Description of *Jack*/TTES

The Team Tactical Engagement Simulator (TTES) system is being designed and built at NAWCTSD in Orlando, Florida. This system projects one or more soldiers into a virtual environment, where they may engage hostile forces. The hostiles throw stones and fire their weapons at the soldier. See Figure 1 for a schematic overview of the system components. For a full description of TTES refer to the proceedings of INCOMSS-94, where a presentation regarding TTES was made.

The soldier stands in front of a large projection screen, which is his view into the environment. He has a sensor on his head and one on his weapon. He locomotes through the environment by stepping on a resistive pad and controls direction of movement and field of gaze by turning his head. The soldier may also move off the movement pad, and the view frustum is updated accordingly based on his eye position (head-coupled display). This allows the soldier, for example, to crouch down to see under a parked vehicle, or to peek around the corner of a building while still affording himself the protection of the building.

Essentially, both the hostiles and the soldier can move around the environment and engage each other. The hostiles are controlled via a DIS stream of commands coming from a computer-generated forces (CGF) simulator. (The CGF system is currently under construction at the Institute for Simulation and Training, Orlando, FL. I think they are calling it "SAFDI"). The TTES filters and translates the DIS stream into a set of posture and command "tokens" that are passed to *Jack*. *Jack* then animates the human figures by transitioning from one posture to another, or locomoting in a cyclical posture change. *Jack* passes the joint angles back to TTES for animating in an IRIS Performer run-time articulated database of human geometry.

The TTES/*Jack* connection is made through two TCP/IP stream sockets (The first incarnation of the interface was done with shared-memory. This was dropped in favor of the flexibility of the socket interface - the machine *Jack* runs on and the machine TTES runs on don't have to be the same, although they can be).

TTES controls the global position of each human figure (*Jack* only moves the figures in its local coordinates), using DIS dead-reckoning algorithms and information about the terrain. The posture transitions are recorded in such a way that the direction of the face and gun are always in a known direction, so the human can be globally oriented correctly when it fires its weapon. TTES also creates the necessary DIS Entity State PDUs to represent the real soldier (mapping from sensor values into the small set of postures in the Entity State PDU), and sends them out over the net to the CGF system and other TTES stations that are participating in the exercise. TTES also performs the ballistics computation for firing the soldier's gun into the scene and determining if and where the hostile human figures get hit.

3 Description of Implementation

Below, I summarize the implementation details of modifications to *Jack* and the TTES stub program (we do not have an actual TTES trainer station) which mimics the communications interface and rendering of the real TTES program with *Jack*.

3.1 DIS Protocol

The information representing the human entity in the simulation is limited by what is stored in an Entity State Protocol Data Unit (PDU) in the DIS protocol. The information we are interested in from the ES PDU is shown in Figure 2. The human is always in one of the 4 postures, along with a weapon state. We only modeled the two values of the weapon state, ***deployed*** and ***firing***, and not *stowed* (which would represent the gun slung over the shoulder or something like that).

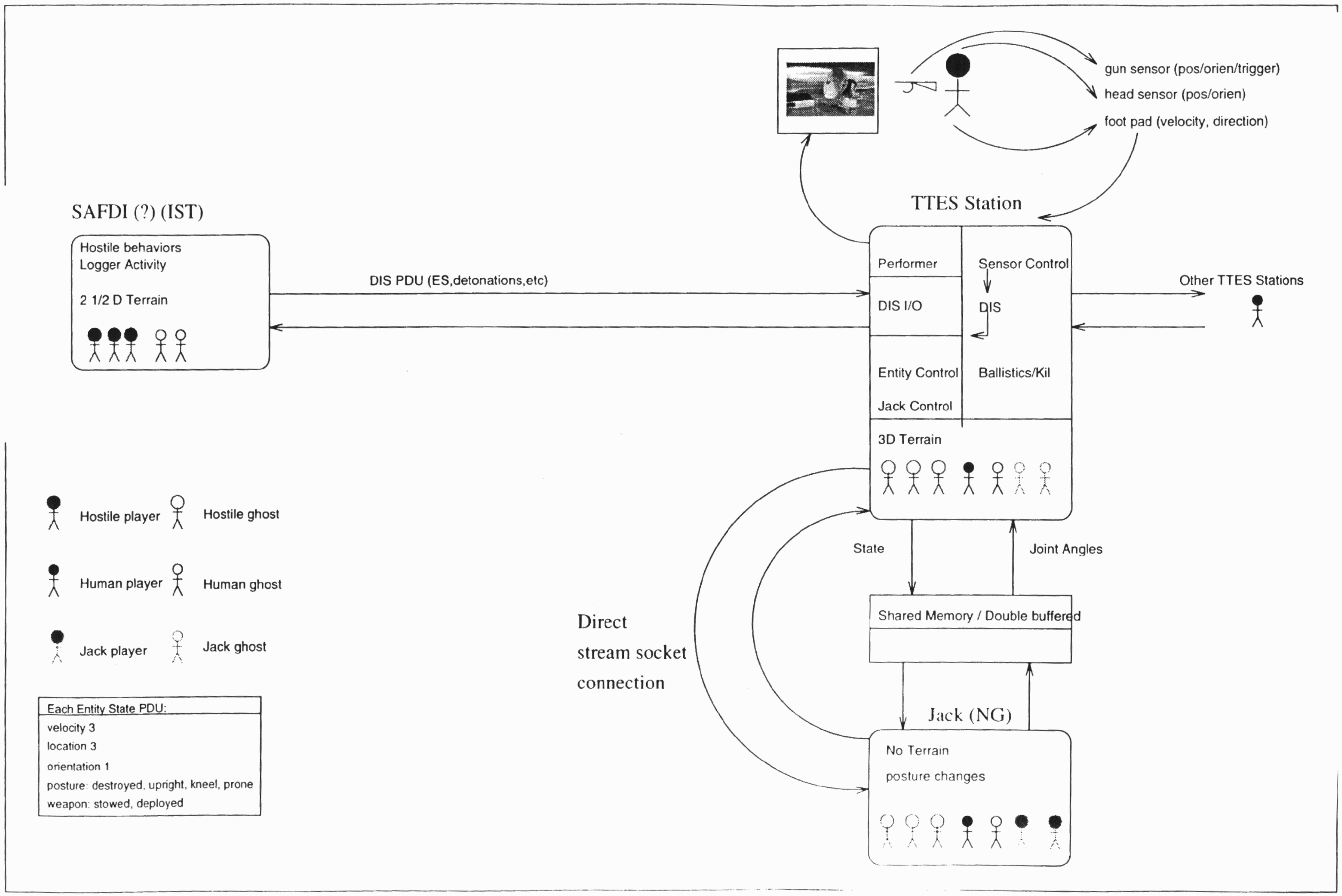
The DIS protocol also allows for upto three weapons on a soldier, but we only modeled one. Also, the protocol allows for a large library of weapon types to fit on the soldier (i.e. pistols, grenades, machine guns, blow pipes, knives, etc). We only modeled AK-47 and M16 machine guns.

3.2 Posture Transitions - Static

In the case where the magnitude of the velocity vector is zero (meaning the human is not moving), we want to transition between the possible ***static*** postures. We encoded the possible static states of the human in a *posture graph*, where the nodes represent static postures, and the directed arcs represent the animated transitions, or movement, from posture to posture.

The possible transitions between static postures are encoded in the posture graph of Figure 3. The actual postures are shown in Figure 4. Each directed transition from posture A to B has an associated motion file, which scripts the transition on the standard human figure at the origin of the *Jack* environment. For example, the transition between (Standing Stowed) and (Standing Firing) is in a file `ststow2stfire_motions.env`. For several transitions between A and B, we only author a directed transition from A to B, and to go from B to A, we simply play the transition backwards. Each motion file typically has about 10-15 primitive *Jack* motions to transition the human and the gun from one posture to another.

In general, a given posture transition was first studied from a video tape of a soldier, as supplied by NAWCTSD (the movie files are supplied on the tape). The goal was not so much as to completely recreate the motion in *Jack* but more to treat the video as a visual template for the movement. For instance, it served as an example of where the arms where in relation to the feet, or what direction the


gun sensor (pos/orien/trigger)
head sensor (pos/orien)
foot pad (velocity, direction)
TTES Station
SAFDI (?) (IST)
Hostile behaviors
Logger Activity
2 1/2 D Terrain
DIS PDU (ES,detonations,etc)
Performer
Sensor Control
DIS I/O
DIS
Entity Control
Ballistics/Kil
Jack Control
3D Terrain
Other TTES Stations
Hostile player
Hostile ghost
Human player
Human ghost
Jack player
Jack ghost
State
Joint Angles
Shared Memory / Double buffered
Direct
stream socket
connection
Jack (NG)
No Terrain
posture changes
Each Entity State PDU:
velocity 3
location 3
orientation 1
posture: destroyed, upright, kneel, prone
weapon: stowed, deployed

Field	Value	Units
Posture	*standing, kneeling, prone, destroyed*	
Weapon	*(Stowed), Deployed, Firing*	
Position	P_x, P_y, P_z	meters
Velocity	V_x, V_y, V_z	meters/second
Heading	*theta*	compass heading in degrees

Figure 2: Essential information from the Entity State PDU

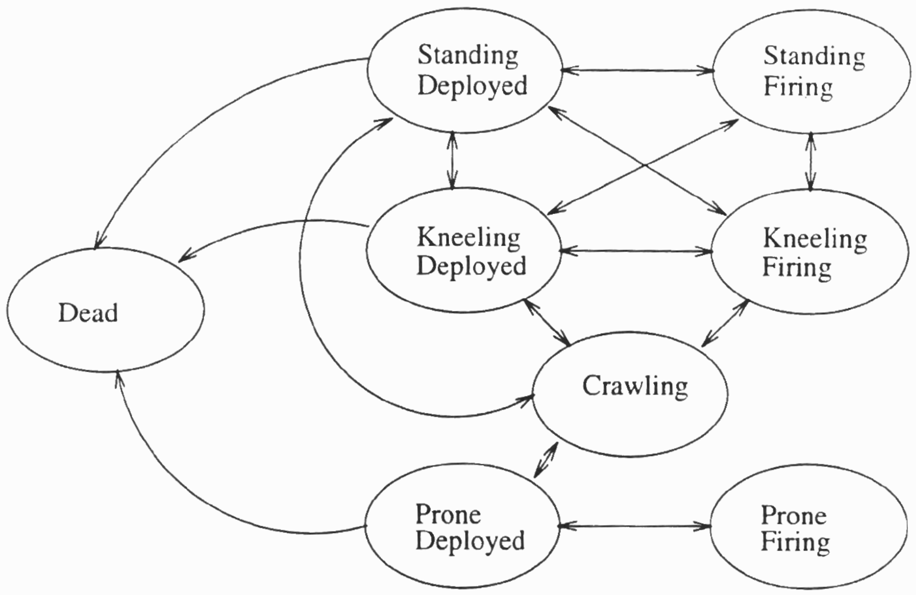


Figure 3: The posture transition graph

shoulder was tilted, or the general timing of the individual movements of the body in relation to each other.

Once the individual body segment movements were visually recognized and isolated, the animators set out to create corresponding movements in ***Jack*** using its goal-oriented motion facilities. To facilitate the creation, it was found helpful to display the figure and the respective motion from several angles on the screen, which helped in the precise placement of the limbs in 3D space.

There was a restriction that the ending posture of a posture transition had to be identical to the starting posture of the next posture transition. This was no problem when we created motions in the forward direction (e.g.. Stand to Kneel). But when we had to create the motions in the reverse direction (Kneel to Stand), it was very difficult to get the ending posture to a particular position (joint angles, displacements, torso position, etc had to coincide exactly) and still get "good" motions. This difficulty was overcome by creating mostly forward-moving motions, and then using the ability to play channelsets in reverse (see below), to get the corresponding reverse direction, as described above. In most cases, the resulting motions looked fine. In a few cases, the reverse motion was scripted explicitly for better results.

Also, we attempted to only animate those sequences which were absolutely neccessary. For example, we have no direct transition from Prone Firing to Kneeling Firing. The run-time system can find the shortest path (in time) between any two postures in the graph, and execute the sequence of transitions.

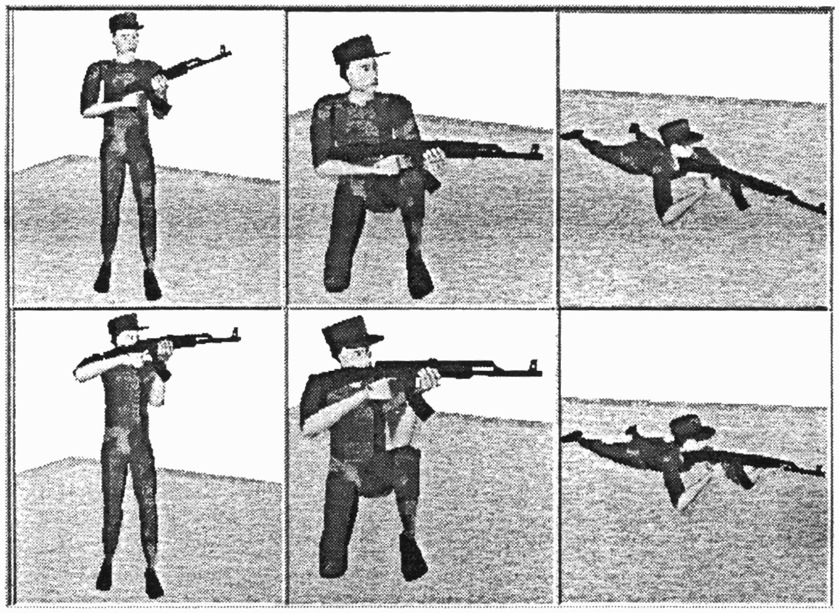


Figure 4: The postures a soldier can take in DIS

3.3 Posture Transitions - Dynamic/Cyclic

When the magnitude of the velocity vector is not equal to zero, the soldier is moving (either forwards or backwards, depending on the difference between the heading and the direction of the velocity vector) by either locomoting (if appearance is upright) or crawling (if appearance is prone). When the soldier is moving, we call this a *dynamic* or *cyclical* posture transition.

The animations were generated by Hyeongseok Ko's walking system. He generated 6 strides for each type of walking (forward walking, backward walking, running): left and right starting steps (that go from the base posture to the cyclic state), left and right ending steps (that go from the cyclic state to the base posture), and left and right cyclic steps. The crawling animation was generated manually, and is based on two animations - one that goes from the base (prone) posture to the cyclic state, and one complete cyclic motion. When crawling is ending the starting animation is played backwards to get back to the base posture.

Playback control of the animation frames is based on a simple state machine shown in Figure 5[1]. Walking is begun whenever the velocity of a figure goes from zero to a non-zero value. The heading and velocity vector are compared at this time to determine whether the figure is walking forwards or backwards. Once this determination is made, the appropriate animation information is referenced and used. The purpose of this initial step is to create a smooth transition from the base posture (standing deployed) to the walking animation.

When the first step is finished, the cyclic "walking" state is started. This state continues to generate normal walking motions until the velocity goes back to zero (at which time the ending step is used), or the velocity becomes greater than the walk/run transition value. If this occurs, and if the figure is walking forwards (there is no backwards running), the running animations are referenced and used. There is a single walk->run transition step that is played, then the "running" cyclic state is entered. The figure will continue to run until the velocity drops below the threshold, at which time the run->walk step will be used, and the "walking" state will be re-entered.

[1]Note that in this graph, the nodes are posture transitions that can loop, or states, and the arcs are *conditions* to transition between states

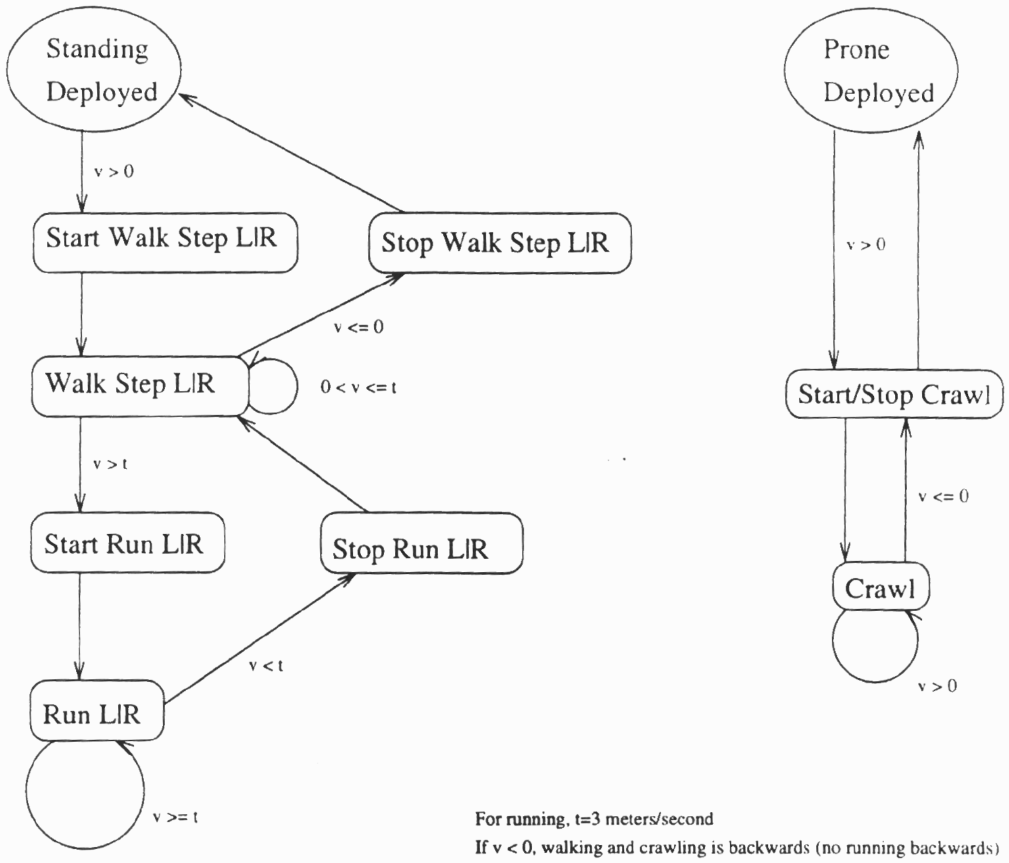


Figure 5: The walking/running/crawling state transition graph

3.4 Preparation for Motion Playback

Once the individual soldier motions/transitions were authored, they were recorded to *channelset* files and organized together into *posture graph* files. Then they are re-recorded onto lower-resolution soldier figures, and then loaded and bound for playback to the TTES program. I'll describe each phase, along with the corresponding *Jack* commands which do the work.

3.4.1 Recording to Channelsets

When you create a regular motion in *Jack* and then execute go, the interpolated frame motion is stored in a *channel*. A channel is storage for any time-varying parameter. For example, each joint in the environment has its own channel to store the angles . The channels are tightly bound to each object for which they store data. I added two new objects to *Jack* and Peabody, a *channelset* and a *sharedchannel*. A sharedchannel holds the same data as a channel, except it is not bound to a specific object in the environment. This allows it to share its data between several objects of the same type (here, the term *object* means either a joint(angles) or a figure(position)). A channelset organizes a set of sharedchannels together, giving them a name and a couple of other parameters. Channelsets are used to store the posture transitions and other motions described above. Each motion sequence, once it is interpolated in *Jack*, is saved as a channelset file (including only those channels from the environment that are part of the soldier and gun).

Once a channelset is created and loaded into *Jack* the sharedchannels can be *bound* to objects in the environment. Each sharedchannel can be bound to many objects (of the same type). For example, a sharedchannel containing joint angle data for the left knee can be bound to all figures (of the appropriate type) with a left knee. A bound channelset is called, of course, a *bound channelset*. The amount of memory required to bind a channelset is *very* small in comparison to the size of the data in a channelset.

A channelset is a Peabody construct, stored in an environment file. An example channelset is shown in Figure 6.

Both a channelset and a sharedchannel have a name associated with them, but it is arbitrary (it doesn't relate to any Peabody construct), but it should be descriptive (as in the above example). The Peabody fields of the channelset are:

size : This is the number of frames stored in each sharedchannel in the set.

count : The number of sharedchannels in the set.

fps : The frames-per-second that the set was saved at. From this, one can deduce the actual time represented in this set. For example, if size=90 and fps=60, then this set represents (90/60) or 1.5 seconds. This also alerts the sampling functions on the sampling frequency for the motion.

Sharedchannels must be declared within a channelset construct. There are no restrictions on the number or type of sharedchannels in a channelset. For TTES each channelset usually contains two figure channels (for the soldier and gun) and joint channels for the soldier. The fields of a sharedchannel are:

type : This is either "sharedfigure" or "sharedjoint" for now, denoting a figure channel (figure location) or a joint channel (joint angles).

object : In the case of a "sharedfigure" channel, this is a string which just holds the figure name of the figure from whence this channel was recorded, and is not used after created. In the case of a "sharedjoint" channel, it is the name of the joint, and will be used to locate a joint to bind to.

protofiletype : This is the name (without path prefix) of the figure file from which this object came. It is used when the channel is bound. This channel can only be bound to objects which have the same figure file name (this is a matching criteria for sharing).

```
channelset crawl2prstow.chset {
    size = 48;
    fps = 60;
    count = 70;
    sharedchannel soldier { /* figure position */
        type = "sharedfigure";
        protofiletype = "soldier_cam.fig";
        object = "soldier";
        frame[0] = ("lower_torso.proximal", trans(31.68cm,43.53cm,-31.05cm));
        frame[1] = ("lower_torso.proximal", trans(31.68cm,42.77cm,-31.05cm));
        ⋮
        frame[48] = ("lower_torso.proximal", trans(11.52cm,4.15cm,-26.49cm));
    }
    sharedchannel right_toes { /* joint angles */
        type = "sharedjoint";
        protofiletype = "soldier_cam.fig";
        object = "right_toes"; /* R(y) */
        frame[0] = (0);
        frame[1] = (0);
        ⋮
        frame[48] = (0);
    }
    ⋮
}
```

Figure 6: **An Example Channelset file**

frame[*i*] : This is the frame data, indexed per frame. For figures, it's a root site and a transform. For joints, it's the joint angles (in radians). The Peabody parser was modified to allow field names with indices.

Most of the channelsets for TTES were recorded at 60Hz, to provide for better motion sampling, and the ability to vary playback time (see posture graph files below).

The commands added to *Jack* for working with channelsets are the following:

`write_channelset` : This command will create a new channelset file, from currently defined (and interpolated) channels in the environment (i.e. after you've create some motions and typed **go**). The command prompts for an output file and a channelset name, followed be the begining and ending frame numbers (of current motions) to grab frames from. The nomenclature for channelset names are any valid Peabody identifier, without the "/" character. It also prompts for a fps and stride value. For example, if begin=100, end=220, fps=30, and stride=2, you'd be pulling 60 frames, or 2 seconds, of motion from frames {100, 102, 104, ... , 220}). It also prompts for a list of figures and the channelset will be created with all the channels (figure position and joint angles) from the list of figures. In the case of TTES we always included a soldier and a gun.

`load_channelset` : This command prompts for a channelset file (an .env file), and reads it. The channelset is built via the call-outs from the Peabody parser. The channelsets can, alternatively, just be loaded via the **`read_file`** command. The channelsets will have their names corresponding to what they were named when written.

`bind_channelset` : This command prompts for a channelset (from the list of loaded channelsets), and then for a new name for the bound channelset you are creating. It then proceeds to create a new bound channelset, prompting you for figures of the appropriate types (i.e. from the **protofiletype** fields of the sharedchannels). For example, if the channelset name you are binding is called "stand2crawl", and you are binding this to the 3rd soldier-gun pair, where the soldier's figure name is "soldier3", then a good name for the bound channelset would be "stand2crawl/soldier3".

`play_bound_channelset` : Once a bound channelset is created, you can play it. This command prompts for a bound channelset, as well as a direction (forwards or backwards) and a transition time (-1 means play it at the stored time), and then will play the motion.

`step_bound_channelset` : Similar to the above command, but allows you to single-step the frames, for debugging purposes.

`set_channelset_parameters` : This command sets the only global parameter concerning bound channelset playback: "yes" means play the bound channelset by traversing each frame, so you'll see the complete motion, as fast as it can be drawn (which most likely is less than 30Hz), or "no" means play the motion in real-time, sync'd to a real-time clock. This will skip frames, so the motion plays back in true "wall-clock" time. The setting should be "no" during playback to TTES.

`print_channelset_info` : This command will print information, at several levels of verbosity, about memory usage and contents of stored channelsets.

`reroot_figure_channel` : This is a utility command, which operates on a channel (not a sharedchannel or a bound shared channel). Its purpose is to re-root a figure position channel so all the root site references are the same. For example, in the motion data recorded from the walking algorithm, the figure root moves all over the place (the toes, heels, hips, pelvis), and we want to transform it so all references are to the pelvis (for performance, the soldier figures are never re-rooted during motion playback, although it is possible). The command simply prompts for a figure root site, then finds the corresponding channel, then plays it out, re-rooting to the new site, and re-recording over the old position data as it goes. This command can take a while to execute.

3.4.2 Posture Graph files

The channelsets can be organized into *posture graph* files, for easier loading, and optionally, for motion playback (only the static posture transitions make use of posture graphs; the walking and running control code only uses posture graph files as a convenience for storing channelset motions).

An example posture graph file (for the static posture transitions of the `DI.fig` (See Section 3.4.3 below), as shown in Figure 3, is shown in Figure 7.

The file is divided in 3 sections. The first section introduces the tokens for the nodes of the graph (the static posture). The second section just lists the channelset files for all motion, and assigns a number to each. The third section describes the actual directed arcs, or transitions, in the graph. Each line starts with the beginning and ending posture, followed by the number of the channelset to play. Following that is the direction of play on the channelset, and the time it should take for traversal. If you change the timing here, the playback system will sample the motions accordingly.

The following commands in *Jack* are for dealing with posture graphs. Note that there is currently no command for creating a posture graph file. They are usually created via the Universal Data Manipulator (a.k.a. gnu-emacs).

load_posture_graph : This command prompts for, then loads a posture graph file. By convention, the suffix for the posture graph file is **.graph**. It also asks for a name to give the posture graph. This name will be used later for reference. Once the actual graph is loaded, all the channelsets will be loaded also.

bind_posture_graph : This command will take a posture graph, and create a *bound posture graph* analogous to the channelsets and bound channelsets. It prompts for a posture graph, and a new name for the new graph. I usually just add a prefix like "soldier3/" to the posture graph name, so if the posture graph name was "ptrans", then the bound posture graph is "soldier3/ptrans".

It also traverses the channelsets and collects all the unique protofigfile fields, and prompts you to pick a figure of each kind. It then creates bound sharedchannels (bound channelset) for each channelset. The bound channels then are named something like "soldier3/ptrans/ststow2crawl" (i.e. they are prefixed by the bound posture graph name). The names are not so important, just as long as they make some kind of sense.

posture_change : This command lets you test out a posture graph. It prompts for a bound posture graph, and then one of the nodes of the posture graph. It will then search for a path from the current node to the target node in the posture graph (shortest path as defined by traversal time), and then execute the set of transitions. For example, if the bound posture graph is currently at STANDING_STOWED, and you request PRONE_FIRING, it will transition from STAND_STOWED to CRAWL to PRONE_STOWED then to PRONE_FIRING.

Jack uses three posture graph files for the TTES simulation: **soldier_low.graph** holds the static posture transition graph, **soldier_loco.graph** holds the walking and running transitions, and **soldier_crawl.graph** holds the crawling transitions. Note that the last two aren't proper posture graphs in the preceding sense, but are just used for convenience for storing and creating bound channelsets.

Note that when a posture transition is requested, the system will sample the pre-recorded motion at the frame rate frequency, so it is guaranteed to always play back in real time. For a 2 second posture transition recorded at 60fps, and a current frame rate of the image generator of 20fps, the playback system plays frames 0, 3, 6, ..., 120. It recomputes the elapsed time on every frame, in case the frame rate is not uniform.

3.4.3 Lower resolution body model

Because of frame-rate requirements and polygon-count restrictions, it was necessary to build a lower resolution human figure for use in the runtime TTES system. The low res soldier figure (`DI.fig`) has the following properties, compared to the regular, polybody human:

```
// First, the posture names and posture tokens
// (they must start at 0, and be consecutively numbered)
8 // number of posture states
0 STAND_STOWED
1 STAND_FIRE
2 KNEEL_STOWED
3 KNEEL_FIRE
4 PRONE_STOWED
5 PRONE_FIRE
6 CRAWL
7 DEAD
# marks end-of-postures
//
// Now, the posture transition tokens, and channelset filenames
// (they must start at 0, and be consecutively numbered)
//token filename
17 // number of transition files
0 ststow2stfire_low.chset.env
1 ststow2kstow_low.chset.env
2 ststow2crawl_low.chset.env
3 ststow2kfire_low.chset.env
:
14 prstow2dead_low.chset.env
15 prfire2dead_low.chset.env
16 stfire2kstow_low.chset.env
# marks end-of-postures-transitions
// Now, the actual transitions (arcs in the graph)
//
// start      end          channelset     playback   time
STAND_STOWED  STAND_FIRE        0           forward    0.8
STAND_FIRE    STAND_STOWED      0           backward   0.8
STAND_STOWED  KNEEL_STOWED      1           forward    1.6
KNEEL_STOWED  STAND_STOWED      1           backward   1.6
:
PRONE_FIRE    DEAD             15           forward    0.8
STAND_FIRE    KNEEL_STOWED     16           forward    1.6
KNEEL_STOWED  STAND_FIRE       16           backward   1.6
DEAD          STAND_STOWED      4           backward   10.0
# marks end-of-arcs
```

Figure 7: A Posture Graph file

	`soldier.fig`	`DI.fig`
polygons	2410	478
edges	4772	773
nodes	2510	327
segments	69	24
sites	180	147
joints	68	23
(DOFs)	134	50

The `DI.fig` emulates the polybody in most every detail, except that it has no fingers (fingers and palm are a single segment), no spine, no eyeballs, and no clavicle psurf (i.e., the clavicle is a virtual segment). The `DI.fig` link structure is the same as the polybody, except for hands (no fingers) and spine (the spine was replaced with two rotational joints, and one translational joint, to mimic the compression that the normal spine can do). The geometry of the segments of `DI.fig` were not normalized, making it un-scalable (anthropometrically)[2]. Currently, `DI.fig` has the dimensions of a 95th percentile male as defined in ANSUR 88, as given by SASS v2.2.1.

3.4.4 Slaving and re-recording

Because of the difference in internal joint structure between the `soldier.fig` and `DI.fig`, its motion cannot be controlled by the available human control routines in *Jack* (which all make assumptions about the structure of the human figure). Instead of controlling its motion directly, we use the existing commands to control the motion of the regular human (as described above) and map the motion on to the low resolution figure, DI.fig. We call this process *slaving*, because the high resolution figure is the *master*, and the low resolution figure is the *slave*.

We use *Jack*'s constraint system to do the slaving. Even though the two figures have different internal joint structures, their dimensions (e.g., length of arms, legs, etc.) are the same. Our goal is have the important landmark sites on both bodies match during the motion. Since from waist down the two figures have the same internal joint structure, we can simply copy the joint angles. From waist up, constraints are used to insure the motions of the two figures match. We create one constraint for each site to be matched. The important sites to be matched are (and the respective constraints):

Constraint	endeffector (on `DI.fig`)	goal (on `soldier.fig`)	joint chain (on `DI.fig`)
DI_belly	middle_torso.upper_torso	t12.DI_middletarget	waist - belly
DI_torso	upper_torso.distal	upper_torso.distal	belly - top of spine
DI_sight	bottom_head.sight	bottom_head.sight	top of spine - eyes
DI_leftpalm	left_palm.palmcenter	left_palm.palmcenter	elbows - palms
DI_rightplam	right_palm.palmcenter	right_palm.palmcenter	
DI_leftelbow	left_upper_arm.distal	left_upper_arm.distal	shoulders - elbows
DI_rightelbow	right_upper_arm.distal	right_upper_arm.distal	

Jack's constraint system works best if the initial configuration of the figure is close to the goal configuration. To give a good starting configuration for the constraint solver, we first copy the joint angles of the master to the slave (blending the 17 spine joints onto the 7 DOF torso of the slave). After copying the joint angles, the constraint solver is invoked to make sure that the important sites of two figures match during the motion. Because of geometry differences, in general we cannot expect all the sites to match exactly. In the case which we cannot match all the sites, we would prefer to match the most important sites as close as possible. In this application, the hands always hold a gun. So the matching of the hand motion is very important, otherwise the hands may go through the gun. Using the

[2]Pei-Hwa Ho is currently fixing this problem, by making some commands in *Jack* that open and automate the usually painful and error-prone normalization process

priority feature of *Jack's* constraint system, we can assign higher priority to the palm center matching constraints than others.

In summary, the slaving process consists of two steps:

1. Copy all the joint angles from the master to the slave.
2. Evaluate constraints to make sure that important sites (such as the palm_centers) of the two figures match up.

This *slaving* technique could be exploited in the future to allow us to define a variety of lower-resolution (than **soldier.fig**) figures, but still program and create motions for the regular human. Then we just create a unique slaving procedure for each new lower-resolution figure we've defined.

Once the channelsets have been recorded for **soldier.fig** (from the previous section), they are re-loaded into *Jack*, and played back on a standard soldier. A lower-resolution soldier figure is then slaved onto the regular soldier, and the resulting motions are saved for the lower-resolution figure.

The commands for creating a master-slave pair in *Jack* are:

create_master_slave_pair : This command prompts for a **soldier.fig** figure, and a **DI.fig** figure, and creates the master-slave pair. It launches a SimulationFunction which updates the slave on each iteration of AdvanceSimulation in *Jack*.

slave_parameters : This command allows you to set various parameters concerning the slave and the slaving procedure. Usually, it would be executed like:

```
slave_parameters("yes","yes","yes","yes",0.50,"yes",0.50,"yes","yes");
```

turn_slave_on and **turn_slave_off** : These commands toggle the slave updating on and off.

turn_behavior_constr_off : This command turns the behavioral constraints of the master (soldier) figure off. This is useful if you're slaving a master that is being driven by channelsets, where your just interested in the kinematic motion (i.e. the master is **not** being driven by motions, but by channelsets.).

create_channelset_motion : If you're creating slave motions for a master soldier that already has its channelsets loaded, you can make the master execute its channelset by creating motions with this command. Once the motions are interpolated, you'll have the channels ready for the slave to create the slaves channelsets.

Each channelset recorded in the first phase (on soldier.fig) is reloaded and played back on a master **soldier.fig**, while a **DI.fig** is slaved. This then gives us the channelsets for **DI.fig**, which are then saved back to new channelset files. Thus, the re-recording is accomplished.

So the final set of steps, from motion creation to playback are:

1. Create motion files for a soldier and gun (posture transitions, walking, running and crawling).
2. Record these motions to channelsets.
3. Build a posture graph file, to organize the channelsets logically.
4. Create a master-slave pair, between a soldier master and a DI slave.
5. Re-load all the channelsets for the soldier, and create a channelset motion for each channelset (the channelsets can be loaded via the posture graph file).
6. Re-interpolate the motions.
7. Write out new channelsets for the slave DI.

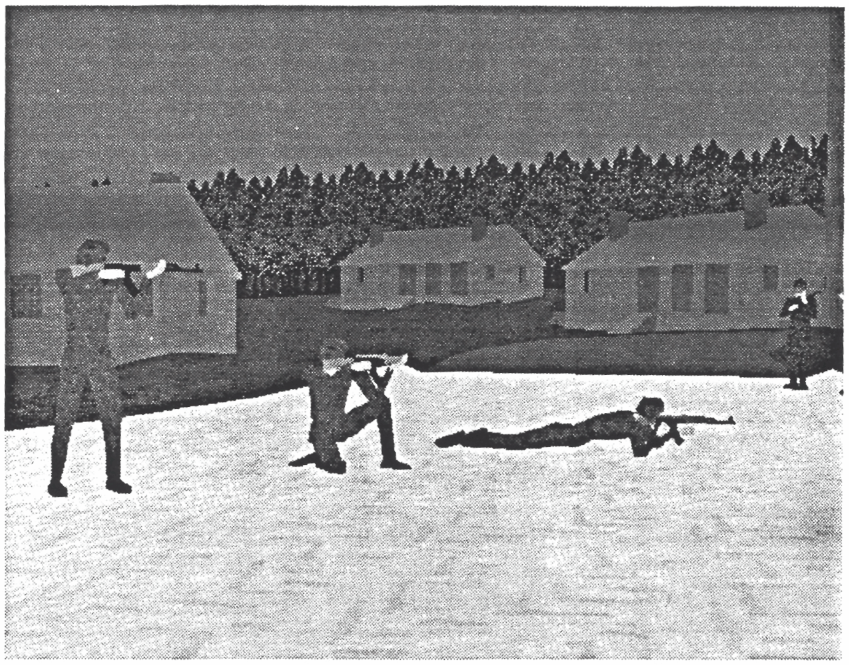


Figure 8: A View of Quantico village with several soldiers in different postures

3.5 TTESPerformer stub program

We built a TTES stub program, which allows us to load the Quantico village database, several soldiers, and mimic the control of the soldiers with simple keyboard commands (to emulate the commands coming from SAFDI and TTEScontrolled humans). It is called **`ttes-stub`**, and it is based on the **`pickfly`** demonstration program distributed with Performer 1.2. It is a fairly generic Performer application.

The Quantico village database is loaded via the standard Flight-format loaders. The human figures are stored in TIPS files, and loaded via the TIPS-format loader. The format of the run-time database generated for the humans is described in Section 7. See Figure 8 for a look at what the village looks like.

I will briefly describe the structure and functionality of the program here and in the next section. The coordinate transforms between the Peabody environment and the Performer environment are the following:

	Jack	Performer
Up vector	+Y	+Z
Zero heading	-Z	+Y
Distance units	cm	m

There are two main structures that are used in **`ttes-stub`** to connect with *Jack* : a JackProcess object (Figure 9) for making a connection to *Jack* and a JackSoldier object (Figure 10) for controlling the articulation. Note that in the code shown here, not all fields are shown, just the essential ones.

The declaration of the two classes is in **`jacksoldier.h`**. The definition is in **`jacksoldier.c++`**. The normal sequence of events would be:

```
    jack = new JackProc();
```

This creates a JackProc object and initializes all the internal fields. Then, the friendly and hostile soldiers are loaded via calls to LoadTIPSFile, and the appropriate fields in JackProc should be set. Then a *Jack* process must be launched.

```
class JackProc {
public:
    pfGroup *hostilesoldierclone; // the soldier group for cloning
    pfGroup *friendlysoldierclone; // the soldier group for cloning

    JackUpdateTable *updatetable;
    int njoints;
    int nfigs;
    JointInfo jointsinfo[MAX_JOINTS];
    FigureInfo figsinfo[MAX_FIGURES];

    int readsocket; // the fd were reading from
    int writesocket; // the fd were writing to
    int connected; // are we connected yet?
    int jackready; // is jack ready yet?
    int quitting; // everything stops...
    JackSoldier *table[JACK_MAX_SOLDIERS]; // a table of soldiers
    int tablesize; // how many are in there

    JackProc();
    virtual int send(int token, int size, void *jpk);
    virtual int launch(char *host, char *remotehost, char *exe, char *datadir);
    virtual int receive(char *message);
    virtual int initsocketconnection();
    virtual int start(char *soldierprefix);
    virtual int quit();
    friend void JackSocketReader(JackProc *jp);
    friend class JackSoldier;
};

extern JackProc *jack; // the jack process wrapper
```

Figure 9: JackProcess

```
class JackSoldier {
public:
    int soldiernum; // for internal reference
    JackProc *jp; // the Jack process controlling this soldier
    pfGroup *soldier; // the Performer clone
    pfDCS *jointptrs[MAX_JOINTS];
    pfDCS *figptrs[MAX_FIGURES];

    int sflag; // is there something in the socket buffer?
    int buffer; // consumer buffer to use...
    JackPack1 buf[2]; // jack update packet, double buffer
    int bufsize[2]; // size of each buffer

    JackSoldier(pfGroup *clone, JackProc *jp);
    virtual int receive(JackPack1 *jpk,int size); // fill a buffer
    virtual int update(); // update the Performer tree of this soldier
    // update the Jack side of this soldier
    virtual int request(int appear, int weapon, float velocity[3],
        float heading, float position[3], int immediate);
    // friendlies
    friend void JackSocketReader(JackProc *jp);
};
```

Figure 10: JackSoldier

```
jack->launch(<host>, <remotehost>, <exe>, <data directory>);
```

where <host> and <remotehost> are the machine names for the local machine and the machine *Jack* should run on (they can be the same, of course). The <exe> is the name of the *Jack* executable, in our case `jack-ntsc-g`, and <data directory> is the startup directory of *Jack* (where all the data files are). *Jack* is launched via a call to `rsh`. After *Jack* is launched, we establish the two sockets:

```
jack->initsocketconnection();
```

The above command opens one read and one write socket to *Jack* (*Jack* will be waiting for these connections). It also `sproc()`'s a process to read the incoming packets from *Jack*. Once the sockets are established, we can communicate with *Jack*. We send the startup commands to *Jack* via:

```
jack->start("soldier_low");
```

which just sends a JCL command to *Jack* `read_file("jack-ttes-soldier_low.jcl")`. This starts *Jack* initializing, which may take a while. Before sending any updates to *Jack* you have to wait until *Jack* is ready. This is signaled when the jack->jackready flag is set (usually takes about 30-40 seconds for *Jack* to get ready).

```
while (1) {
    if (jack->jackready)
        break;
    sleep(1);
}
```

Once *Jack* is ready, we can start adding soldiers, and receiving updates from *Jack*. For each soldier, first you create a new soldier:

```
JackSoldier *newsoldier = new JackSoldier(jack->hostilesoldierclone, jack);
```

where the first parameter is a Performer tree to clone for the soldier, and the second is the `jack` process pointer. An example implementation of the use of JackSoldier is in the file `simsoldier.c++`, which has a sample class called `SimSoldier`, which implements a kind of control for a soldier in the Performer environment, along with simple simulations (soldiers that walk in squares and circles). Once a soldier is created, you can send updates to *Jack* via:

```
newsoldier->request(<posture>, <weapon>, <velocity>, <heading>, <position>);
```

This sends the appropriate information to *Jack* so it can decide which set of joint angles (frame within a bound channelset) to playback.

3.6 Start-up in *Jack*

Jack is launched from within the TTES process via a call that looks like:

```
rsh <remotehost> (unsetenv REMOTEHOST; setenv DISPLAY :0;
    jack-ntsc-g -W 9,600,500,1000
    -M 9,600,450,480 -A 9,350,80,10,Courier7
    -E open_ttes_sockets(<host>)
    -E change_directory(<directory>)
```

which does a remote shell (so your `.rhosts` file should contain *<remotehost>* if it's different than the local) to launch *Jack*. The command arguments are: -W: this sets the window location; -M: sets message window location; -E: execute the command.

The first thing *Jack* does is open the read and write sockets to TTES. This is done via the `open_ttes_sockets(<hos` call. *Jack* acts as the server for the connection, so it blocks until TTES requests the sockets. The second command, `change_directory(<directory>)` will set the default directory to where the TTESrelated data files are (for *Jack*). This allows you to keep *Jack* and its data files in a separate tree from the rest of TTES.

Once the sockets are open, TTES will send the command `read_file("jack-ttes-soldier_low.jcl")`, which will force *Jack* to begin the initializing sequence. This file is below:

```
1   read_file("jack-ttes-soldier_low.env");
2   create_soldier_state("s0","Di2_0","AK_47_0");
3   create_soldier_state("s1","Di2_1","AK_47_1");
⋮
10  create_soldier_state("s9","Di2_9","AK_47_9");
11  set_channelset_parameters("no");
12  set_ttes_report_interval(10.00);
13  set_ttes_heading_offest(256.00,0.00,256.00);
14  set_ttes_soldier_throttle(0.04);
15  set_ttes_report_interval(30.00);
16  start_ttes_socket_simulation();
17  disable_graphics();
```

The first thing done (line 1) is to load the environment file containing the 10 soldier/gun figure pairs. Lines 2-10 create the ten *soldier states* (structures that hold all state info for the soldier). For example, `create_soldier_state("s0","Di2_0","AK_47_0")` creates soldier "s0", from the DI.fig figure Di2_0 and gun AK_47_0. The first soldier state create triggers the loading of all the posture graph files and associated channelset files. Also, the channels are bound to the soldiers at this point. Line 11 sets the playback of the channelset to real-time (i.e. frame skipping). Line 12 tells *Jack* to report status every 10 seconds, to the shell. You can set this to a higher value to remove the messages that appear. Line 13 sets the heading offsets associated with the posture transitions, walking, and crawling channelsets respectively. This was necessary, as the channelsets were recorded before we had the interface working, and before we realized we had a bad offset in there! (it was easier to add this command than to re-record all the channelsets). Line 14 sets a "throttle" value, in seconds. This actually slows the rate of update packets sent to TTES. This was necessary, as *Jack* running on its own processor can send out about 1000 updates a second, for 10 soldiers, and the max we'll need for 10 is about 300. This value, set at 0.04 seconds, or about 1/30th of a second, will stop *Jack* from sending out updates for the same soldier at greater than the inverse of the number (e.g. 1/30 means don't send more than 30 updates per second). If your image generator is never going to go over 15Hz, set this value to 1/15, or about .065. It saves on net traffic. Line 16 sends the "I'm ready!" message to TTES so it can proceed with the simulation. And finally, line 17 shuts off *Jack's* graphics. *Jack* will not redraw its windows, or spend any time doing anything graphical. At this point, *Jack* is ready to receive updates from TTES and send data back.

3.7 Motion Playback in *Jack* and TTES

Once the simulation is set up, TTES sends requests to *Jack* (via JackSoldier::request), and receives joint angle packets (via JackSoldier::receive). The format of the packets going from TTES to *Jack* is shown in Figure 11.

This is essentially the key DIS parameters for the soldier. The values are in Performer coordinates, and are transformed in *Jack* when they arrive. Currently, heading and position should be sent, but they are not reflected back in the update packets sent to TTES. This is by design, as TTES wants to set heading and position itself. Position is of the feet (i.e. ground level). The pseudo-code for the main soldier controller in *Jack* (which is run every iteration of the *Jack* main loop, for each soldier) is shown in Figure 12.

The update packets are sent out of *Jack* from within the motion controllers (which loosely correspond to the "**continue ...**" statements in the above code; the motion controllers actually are running 'concurrently', as SimulationFunctions in *Jack*). There are three different motion controllers: posture, walking/running, and crawling. Each one's job is to pick the correct frame from within a bound channelset, and this frame is formatted and sent to TTES as an update packet.

The format for joint angle update packets from *Jack* to TTES look like:

```
#define JACKTTES_DEAD 1
#define JACKTTES_UPRIGHT 2
#define JACKTTES_KNEEL 3
#define JACKTTES_PRONE 4
#define JACKTTES_WEAPON_STOWED 0
#define JACKTTES_WEAPON_DEPLOYED 1

struct JackPackIn {
    int soldierindex; // same on Jack and TTES side [0..9]
    int appearance; // one of above (JACKTTES_*) [1,2,3,4]
    int weaponstate; // one of above (JACKTTES_WEAPON_*) [0,1]
    float velocity[3]; // in meters/second
    float heading; // in degrees
    float position[3]; // in meters
    int immediate; // do we want this now?  ``snap to''
};
```

Figure 11: TTES Packets

soldierindex =	[0,9]
size of data array =	S
data[0]:	object index 1
data[1]:	dof 1
data[2]:	dof 2
data[3]:	object index 2
data[4]:	dof 1
data[5]:	dof 2
data[6]:	dof 3
data[7]:	object index 3
⋮	⋮
data[k]:	object index i
data[k+1]:	dof 1
data[k+2]:	object index i+1
⋮	⋮
data[S-3]:	item index n
data[S-2]:	dof 1
data[S-1]:	dof 2
data[S]:	dof 3

When the TIPS file is loaded into the Performer runtime database, the loader writes out a file of articulation parameters, and this information is used to build an *update table*, which in turn is used to interpret the update packet shown above. Both *Jack* and TTES use the same update table. Each entry in the table contains an index, an object name (for either a joint or a figure), and the number of degrees-of-freedom passed for this item. Therefore, the number of dofs per entry varies, usually 1 to 3 for joints, and 16 for figure position (the whole 4x4, but we'll cut this down later to 7: 3 for position, 4 for orientation).

This design allows us to transmit the minimum amount of information per update, to keep network

```
SoldierState::soldiermanager()
{
    if (a packet has arrived) {
        copy packet
        convert from TTES to Jack coordinates
    }
    compute velocity
    compute heading
    if (soldier is currently transitioning) {
        grab new heading
        continue posture transition
        return
    }
    if (a packet has arrived and
        (posture is different or weapon is different)) {
        if (soldier is locomoting)
            cancel the step
        select the goal posture node from posture graph
        start a new posture change (from current to goal)
        return
    }
    if ((velocity > 0) and (soldier is PRONE)) {
        continue crawling
        return
    }
    if ((velocity > 0) and (soldier is STANDING)) {
        continue either walking or running
        return
    }
    if (no motion or posture change, but a heading change) {
        send the last update packet with heading change
    }
}
```

Figure 12: Soldier manager's main loop

traffic at a minimum (a complete low-res soldier update is about 440 bytes), and affords us the ability to take advantage of frame-to-frame coherence in a motion, and not transmit data that's not changing (although we haven't tried that yet). The first incarnation of the *Jack*TTES interface was implemented via shared memory. We passed the entire joint transform (16 floats) for each joint. While this has the advantage of speed, it has the greater disadvantage that *Jack* needs to run on the same machine, and will not scale well to many soldiers.

4 Running the Demo

To run the demonstration executable, you should first unload the tape (tar format) into a directory, say `/usr/demos/ttes`. For convenience, define an environment variable to point here, for example,

```
% setenv TTES /usr/demos/ttes
```

The files will be in directories according to the organization in Section 6. Your environment should contain the necessary settings for running *Jack*.

4.1 Starting things up

To start the demo, run the following commands[3]:

```
% cd $TTES
% cd demo/performer
% ./ttes-stub -W 800 -H gumby -J $TTES/demo/jack/jack-ntsc-g -I
    $TTES/demo/jack -F $TTES/demo/perf_terrain urban.flt
```

The parameters are: -W: window size for the program; -H: host to run *Jack* on (can be local); -J. *Jack* executable; -I: directory to run *Jack* in; -F: Performer data file path; **urban.flt** is the Quantico village database. Once the program starts, *Jack* will be launched (you should see its screen come up). **ttes-stub** will load the village database, and finally show you a bird's eye view of the whole area. You should pick the "SOLDIER CAM" view from the "View" menu on the left side of the window. This places you on a tethered camera, attached to soldier 0, who is standing in the middle of the village. Soldier 1 is walking in a square, and soldier 2 is walking in a circle.

4.2 Keyboard commands

The **ttes-stub** program is derived from pickfly, so it has all the features of that program (see Performer 1.2 documentation). In addition, the following keyboard commands are added (these are to exercise the features of moving the soldier figures around) (Note: the current soldier starts at 0, and all commands are relative to the current soldier):

[3] All executables were compiled for Irix 4.0.5

1	Set posture to DEAD on current soldier.
2	Set posture to STANDING on current soldier.
3	Set posture to KNEELING on current soldier.
4	Set posture to PRONE on current soldier.
5	Set weapon state to DEPLOYED on current soldier.
6	Set weapon state to FIRING on current soldier.
n or N	Next soldier. Set current soldier to next soldier (i.e. if current soldier is 1, sets current soldier to 2.)
p or P	Previous soldier.
a	Add another soldier. This creates a new soldier, standing near the low office building. A maximum of 10 soldiers (0-9) can exist.
A	Add a *Jack*controlled soldier. This adds a new soldier that is controlled via *Jack* (reactive behaviors). This is experimental for now.
u or U	Move the tethered camera AWAY from soldier by +5 meters.
d or D	Move the tethered camera TOWARDS the soldier by -5 meters.
i or I	Rotate tethered camera +10 degrees about current soldier.
j or J	Rotate tethered camera -10 degrees about current soldier.
o or O	Move tethered camera to OVERHEAD view on current soldier.
Left, Right Arrows	Change heading -10 or +10 degrees on current soldier.
Up, Down Arrows	Change velocity +0.5 or -0.5 meters/second on current soldier.
?	Prints a list of keyboard commands to the stdout (console).

4.3 Taking a walk in the field...

When you first start up, you are looking at soldier 0. Start him walking slowly by pressing the [**Up Arrow**] key. Change his heading so he walks towards the open fields by pressing [**Left Arrow**]. You can swing around him by holding the [**i**] or [**j**] keys. Take a look at him from above by pressing [o]. Press [**i**] again until you are looking at him from the side. Now increase velocity with the [**Up**] arrow. When he exceeds 3 meters/second he'll start to run. The drop him back to zero by pressing [**Down Arrow**]. If you press [**Down Arrow**] again, he'll walk backwards. Bring him to a stop again. Now try pressing the keys [**2**], [**3**] **and** [**4**] to see him adopt various postures. Also use [**5**] **and** [**6**] to see him shoulder his rifle. If you press [**1**] he dies...press [**2**] to resuscitate him. Press [**4**] to make him go PRONE. Then give a little forward velocity, and he CRAWLS forward. Once you get the hang of it, try taking a couple of soldiers out into the field, using [**n**] and [**p**] to toggle between them. You can drop the soldier-tethered camera and just drive or fly around by choosing "Fly" or "Drive" from the "View" menu.

4.4 Looking at things in *Jack*...

While the simulation is going on in the Performer program, *Jack* is providing motion data, reacting to the changing states of the soldiers. You won't see anything happening in *Jack* as the graphics are disabled. If you enable graphics, you'll see 10 soldiers standing at the origin, and several may be walking in place, according to their state. However, what's going on inside *Jack* is the same as in Performer. We can see that by doing the following (while you have **ttes-stub** up and running). Go to the station *Jack* is running on. *Jack* will respond to your keyboard commands.

Enter the command **read_file(''jack-ttes-terrain.jcl'')**. This will turn off soldiers 5-9, and set *Jack* to show the soldiers in their correct global positions in space (1-to-1 with Performer). It also will read the file **urban_terrain.env**, which is the *Jack* equivalent to the **urban.flt** file[4]. You can

[4]If *Jack* doesn't load the file correctly, load it manually from the directory you installed it into, **$TTES/jack_terrain/urban_terrain.env$**

do `change_view` to move the camera around a bit, but you should see things just as they are on the Performer side of things.

5 Conclusions & Future Work

We have built and demonstrated a system for off-line production of motion sequences, together with a method for putting those motion sequences together (posture graphs and locomotion control) for real-time playback to a remote image generator. Also, we have created a system for animating the human motion associated with what can be expressed about a human figure in the DIS protocol.

There are many areas in which this work could advance or be improved. Some of our suggestions for immediate additional work are listed below:

- Posture Transitioning

  As the number of possible states for the human increases, the posture graphs should be replaced with a more procedural approach to changing posture. For the applications today on current workstations, the current technique balances performance and realism. NAWCTSD would like to control the human figures (both hostile synthetics and friendly avatars) with the same control scheme, based on sensor values from locations on the human figure. We will be investigating this in the near term.

- Production of posture transitions:

  In general, the process went smoothly, but there were several aspects that can be improved upon.

  - The video that was supplied to us showed the soldier only from one viewing angle. It would have been better to show the same movement from several different views (ultimately from the three orthogonal axes views, as they are used in a rotoscoping system). As it was, it was sometimes hard to tell where exactly a given limb would end up (e.g. it was blocked from view by another body part).
  - *Jack's* constraint system was very helpful in roughing out the movement. Yet, for this simulation we wanted to achieve a movement that looks as fluent as possible. This proved to be somewhat difficult under *Jack*. What would be desirable, would be a facility to "fine-tune" a motion better. Similarly, the algorithm used to drive the inverse kinematics sometimes produced unpredictable results when different goals (motions) affected the same kinematic chain.
  - As an ultimate improvement for this process would be a system that could create the motions directly from the video, without the use of any animators (automatic rotoscoping), or sample the motion using a body suit, or a set of sensors (e.g. Flock Of Birds) and recreate the motion using this data. We think this is the most promising solution.

- Generalize the cyclic posture changes

  The cyclic posture transition state machines are currently hard-coded for each cycle (walking, running, and crawling). The static posture transition state machine is general and data-driven. We should generalize the cyclical state machines, so they can be driven completely by data files (like the static postures). The key to doing this will be the specification of conditionals on which states change. These could be specified in cyclic posture graph data files as LISP expressions, and interpreted on-the-fly, during runtime.

- Some bells and whistles

  During the production of one of our posture transitions, a glitch was introduced in one of the transitions that caused the soldier to jerk backwards a bit when a transition started. The soldiers working with TTES saw this and liked it very much, thinking we were animating kickback from a firing rifle. We could formalize this a bit, and look at the event-type information in the DIS stream (weapon fire, explosions, etc) and create small (2-5 frame) sequences of motion to give the illusion the soldier is reacting to the event. This could prove a very inexpensive way of increasing the illusion of reality.

- Intelligent, Reactive Friendly soldiers:

  Another interesting extension of using *Jack* for TTES will be the incorporation of some work by Becket and Reich, which endows the soldier figures with the following abilities. These could be used to generate *Jack*-controlled soldiers to easily populate the world, with a simple programmatic control of their higher-level behaviors.

  Infrared: A simple infrared model is available. Each figure or segment in an environment can be tagged with a "heat" value between zero and one. A mode exists to display the environment using these heat values, interpolating from black to red, instead of the normal visible-light colors.

  Attraction: Attraction is used to guide an agent towards an object or agent, towards a global location, or towards a location relative to an object or another agent. For example, attraction might be used to have a friendly soldier try to stay 10 feet away, 20 degrees to the right of another agent as that agent moves around in the environment.

  Avoidance: Avoidance is used to keep an agent away from specific objects, locations, or other agents.

  Terrain Sensor: Terrain sensors detect the types of terrain surrounding an agent. A terrain sensor may be used to have an agent avoid certain types of terrain such as water, or to keep an agent on a path or road.

  Hostile Field-of-View Sensor: The hostile field-of-view sensor may be used to have an agent avoid the line of sight of one or more hostiles.

- Embed motion control in Performer

  The current implementation of the motion controllers in *Jack* rely mostly on stored motions. Therefore, it would be quite possible to remove them from *Jack* and embed directly in Performer Then, the *Jack* process would not need to be run during a TTES session, increasing performance of the system overall. If we use inverse-kinematics and constraints on sensor data (as is likely in the near-term), this would be more difficult, as it would entail embedding that part of *Jack* in Performer. Eventually, though, this is the goal.

6 Appendix A: Data and Code Files

This is the directory structure on the tar tape. Note: files with `.Z` suffix are compressed and should be uncompressed before using. Also, there are several symbolic links to cut down on duplications of files.

`./movies/*.mv` SGI movieplayer (Irix 5.2) files of posture transitions (as supplied by NAWCTSD). The video was broken up into 3 movie files.

`./share/motions/*` Motion files live here. Each motion has an associated frame 0 **`*.env`** file and a **`*_motions.env`** file. In all motion files, the regular (soldier_cam.fig) soldier is named soldier, the gun is named AK_47_color2 (don't ask), and the DI is called Di. The nomenclature for files is:

- st = standing
- k = kneeling

- pr = prone
- dead = dead
- fire = firing
- stow = stowed (actually deployed)

For example the motions from standing stowed to kneeling firing are **ststow2kfire.env**

./share/motions_chsets/* The channelsets, both for regular soldier, and low-res DI.fig. The nomenclature is similar to that of the motion files, with the additional **_low** specifying the low-res soldier, and **.chset.** marking this as a channelset file. Several JCL files here are also used to automate the motion->channelset and slaving procedures.

./share/perf_terrain/* These are the Flight-format geometry files for the Quantico village environment. The main file is **urban.flt**

./share/jack_terrain/* These are the same Quantico village environment, but converted to Peabody format. The main file is **urban.env** (unfortunately, no texture maps).

./share/data/* These are *Jack* data files, defining the various figures used, psurfs, and texture maps.

./src/common These are the shared files between *Jack* and TTES.

> **jackpack.h** Format for packets going from TTES to *Jack* and back.
>
> **updatetable.c++** Both sides use the updatetable object, which defines format of packets flowing from *Jack* to TTES.

./src/jack/ These are the files that make up *Jack*. These files were the only additional or changed files above and beyond the 5.8 libraries.

> **MotionControl.h** The locomotion/crawling motion control object.
>
> **channelset.c++** Channelset CMDs and definition.
>
> **channelset.h** Channelset declarations.
>
> **jack-ntsc-g** The *Jack* executable (symbolic links point here)
>
> **jack_channel.c++, control input2 motion** Slightly modified from the 5.8 libraries. These files don't have much to do with the TTES functionality.
>
> **jack_motioncontrol.c++** Definition of MotionControl (locomotion) controller.
>
> **jacksock.c++** Socket communications process, and CMDs for all TTES related commands.
>
> **menu.c++** Main menu definition for *Jack*.
>
> **pea_parse.c++, pea_parse.y** Modified Peabody parser to handle indexed fields in constructs.
>
> **peastuff.c++** Parser callouts for handling channelset and sharedchannel fields and constructs.
>
> **pgraph.c++** Definition of PostureGraph class.
>
> **postures.c++** Definition of BoundPostureGrapph, and CMDs for all posture commands.
>
> **postures.h** Declaration of Posture* classes and functions.
>
> **setmotion.c++** Definition of the channelset motion.
>
> **setmotion.h** Declaration of the new SetMotion type.
>
> **sharedchannel.c++** Definition of the new SharedChannel type (sub-class of Channel), and two sub-classes: SharedFigureChannel and SharedJointChannel.

sharedchannel.h Declaration of above.

slave.c++ Slaving CMDs and functions.

soldier.c++ Definition of SoldierState class, which wraps up all the other things, and manages interface with TTES.

soldier.h Declaration of above.

spitter.c++ The spitter, creates .tips files

ttes_crawl.c++ State machine for crawling.

ttes_global.c++ Global coordinates computation for soldiers

ttes_loco.c++ State machine for walking and running

ttes_testcmds.c++ Testing commands for use when no TTES process connected.

OPT/ttes-stub.OPT Actual executable file

cmdline frame gui keybd main object picking These are all just slight modifications from the pickfly original, as distributed with Performer 1.2.

jacksoldier.c++ Definition of the JackProc and JackSoldier classes.

jacksoldier.h Declarations for above.

jackupdate.c++ Some sample functions, which call the appropriate functions within other nodules, to start up *Jack* and create soldiers. You would have something like this in your own system.

pftips.c++ The TIPS loader

pftips.h Declarations for above.

pickfly.c Main routines for the interface.

socket.c++ Simple routines for making a socket connection with *Jack*.

soldiersim.c++ A sample implementation of a simulated soldier, with calls into the main functionality of JackSoldier. Use as a template for your own.

./src/tips11/ This is the modified TIPS loader for Performer 1.1. See the description in Section 7.3.

/demo/jack/ Files for *Jack* when running the demo. This is the directory in which *Jack* should execute during the demo. The subdirectory **./demo/jack/chsets** holds the pre-computed motions.

./demo/performer/ This is where you start up the demo.

7 Appendix: TIPS file format and Performer Loader

Roughly, a **.tips** file is a record of the information obtained in a traversal of the Peabody environment. For each figure in the environment (except the camera) a depth-first traversal of the figure's sites is performed, starting at the root site and following joint connections between sites. Use of this intermediate-style format avoids calling the Peabody parser from within the Performer application; we found development to be faster without the added complexity of modules linked with both the (extensive) *Jack* 5.8 libraries and the Performer 1.1 or 1.2 libraries. It is for similar reasons of speed and convenience of development and debugging that we chose a human-readable plain-text format. You can create a **.tips** file from within *Jack* be executing the command **spit("file.tips")** which will dump everything in the environment (except cameras) into **file.tips**.

7.1 TIPS file format specification

The following is an informal description of the structure of a `.tips` file. Newlines in each part of the description correspond to newlines in the file. Entries in the file are generally strings, integers, or floating point values, and the exact meaning of constructs in the description should be clear from the example `.tips` file in the next section.

```
[.TIPS FILE] ::=
            [number of figures] [COMMENT]
            [FIGURE DESCRIPTION] *repeated [number of figures] times*

[FIGURE DESCRIPTION] ::=
            [COMMENT]
            [figure name]
            [COMMENT]
            [figure filename]
            [COMMENT]
            [4x4 global position transform]
            [COMMENT]
            [4x4 root site inverse transform]
            [SEGMENT DESCRIPTION] *for the figure's root segment*

[SEGMENT DESCRIPTION] ::=
            [PSURF FLAG] [COMMENT]
            [GEOMETRY] *only if [PSURF FLAG] = 1*
            [number of sites] [COMMENT]
            [SITE] *repeated [number of sites] times*

[SITE] ::=
            [number of joints] [COMMENT]
            [COMMENT]
            [site name]
            [COMMENT]
            [4x4 site transform matrix]
            [JOINT] *repeated [number of joints] times*

[JOINT] ::=
            [ROOTJOINT FLAG] [COMMENT]
            [COMMENT]
            [4x4 joint transform matrix]
            [REVERSE JOINT FLAG] [COMMENT]
            [number of DOFS] [COMMENT]
            [DOF] *repeated [number of DOFS] times*
            [COMMENT]
            [joint name]
            [COMMENT]
            [OTHERSITE name]
            [COMMENT]
            [4x4 inverse of OTHERSITE transform]
            [SEGMENT DESCRIPTION] *for segment at other end of joint*

[DOF] ::=
            [DOF type] [COMMENT]

[GEOMETRY] ::=
            [number of attributes] [COMMENT]
            [ATTRIBUTE] *repeated [number of attributes] times*
            [number of nodes] [COMMENT]
            [node coordinate 3-vector] *repeated [number of nodes] times*
            [number of faces] [COMMENT]
            [FACE] *repeated [number of faces] times*

[ATTRIBUTE] ::=
            [ambient color 3-vector]
            [diffuse color 3-vector]
            [specular color 3-vector]
            [TEXTURE FLAG] [COMMENT]
            [texture filename] *only if [TEXTURE FLAG] = 1*

[FACE] ::=
            [attribute index number] [COMMENT]
            [number of vertices] [COMMENT]
            [face color RGB vector]
            [VERTEX] *repeated [number of vertices] times*

            [VERTEX] ::=
            [node index number]
            [texture coordinate 2-vector] *only if [TEXTURE FLAG] = 1*

[COMMENT] ::= any string of characters terminated by a new line
```

7.2 Example .tips file

The following **.tips** file (reproduced verbatim) represents a Peabody environment which contains a single unit cube as its only figure:

```
1 figures
FIGURE #0
cube
FIGURE #0
cube.fig
Global position matrix
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000
Root site inverse matrix
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000
1 (psurf flag)
```

```
1 (# attributes)
0.173333 0.125490 0.173333
0.693333 0.501961 0.693333
0.000000 0.000000 0.000000
0 (texture flag)
8 total nodes
0.000000 0.000000 0.000000
0.000000 0.000000 1.000000
1.000000 0.000000 1.000000
1.000000 0.000000 0.000000
0.000000 -1.000000 0.000000
0.000000 -1.000000 1.000000
1.000000 -1.000000 1.000000
1.000000 -1.000000 0.000000
6 faces
0 attribnum
4 vertices
0.000000 0.000000 0.000000
0
1
2
3
0 attribnum
4 vertices
0.000000 0.000000 0.000000
0
3
7
4
0 attribnum
4 vertices
0.000000 0.000000 0.000000
2
6
7
3
0 attribnum
4 vertices
0.000000 0.000000 0.000000
0
4
5
1
0 attribnum
4 vertices
0.000000 0.000000 0.000000
4
7
6
5
0 attribnum
```

```
4 vertices
0.000000 0.000000 0.000000
1
5
6
2
1 sites
0 joints
SITE
base
Site 1 matrix
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000
```

7.3 Alternate .tips format

An alternate format was developed to handle a last-minute change required by the TTES project, namely that the Performer **.tips** format loader run under Performer 1.1 rather than Performer 1.2. The key difference between the two versions with respect to the loader is that Performer 1.1 lacks the pfuBuilder object of 1.2, which performs automatic generation of efficiently-meshed triangle strip GeoSets (geometry objects) and their associated GeoStates (attribute objects) from arbitrary polygon and attribute information.

Our solution was to use the 1.2 pfuBuilder routines to generate GeoStates and meshed geometry which were then written into a slightly modified **.tips** file intended to be read in a Performer 1.1 environment. The only change, therefore, is in the encoding of geometric information:

```
[GEOMETRY] ::=
            [number of attributes] [COMMENT]
            [ATTRIBUTE] *repeated [number of attributes] times*
            [GEOSET] *repeated one or more times*
            endgeosets

[GEOSET] ::=
            geoset
            [attribute index number] [COMMENT]
            [number of triangle strips] [COMMENT]
            [length of strip] *repeated [number of triangle strips] times*
            [indexing mode]
            [VERTEX1] or [VERTEX2] *repeated once for each vertex*
            [color binding]
            [COLOR1] or [COLOR2] *depending upon [color binding]*
            [normal binding]
            [NORMAL1] or [NORMAL2] *depending upon [normal binding]*
            [texture binding]
            [TEXCOORD] *only if [texture binding] = PFGS_PER_VERTEX*

[VERTEX1] ::=
            [vertex index number]
            [vertex 3-vector] *used if [indexing mode] = indexed*
```

[VERTEX2] ::=
    [vertex 3-vector] *used if [indexing mode] = not indexed*

[COLOR1] ::=
    [color 4-vector] *repeated once for each vertex*

[COLOR2] ::=
    [color 4-vector]

[NORMAL1] ::=
    [surface normal 3-vector] *repeated once for each vertex*

[NORMAL2] ::=
    [surface normal 3-vector]

[TEXCOORD] ::=
    [texture coordinate 2-vector] *repeated once for each vertex*

For obvious reasons, this encoding is very similar in structure to that used within a Performer pfGeoSet. [COLOR1] and [NORMAL1] are used when "PFGS_PER_VERTEX" attribute binding is in effect (indicating a value for each vertex) and [COLOR2] and [NORMAL2] for "PFGS_OVERALL" attribute binding (indicating a single shared value for all the vertices).

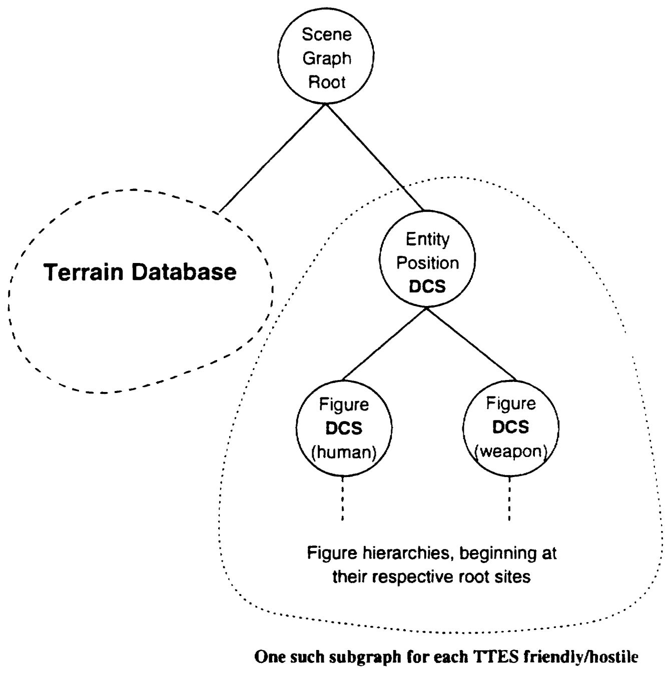
7.4 Runtime database structure

Performer 1.1 and 1.2 are both limited to a maximum scene graph depth[5] of 32. For this reason we use just one pfDCS node for each joint in the Peabody hierarchy. Since a Peabody joint consists of three transformations (site 1, joint angle, and site 2 inverse), two matrix multiplications are required to compute each new DCS value. This single-DCS approach was chosen because the alternative, breaking up the figure hierarchy into several smaller pieces, would entail unnecessary complexity and additional bookkeeping. The matrix multiplications must be done at some point anyway; nothing is lost by making them part of the update process. The two site transforms do not change during simulation, so they are stored by the Performer application and at each frame the *Jack* process need only transfer the updated joint transforms over the socket or shared memory connection, thus minimizing communication bandwidth. Even with only one DCS per joint, the *Jack* 5.8 human approaches the Performer-imposed limit. To minimize its depth we root the human through the waist at all times.

TTES entities[6] are attached to the top level of the Performer scene graph by the sequence of nodes shown in figure 13. The node labeled "Entity Position DCS" allows the human and gun subgraphs to be manipulated as a single unit. More specifically, we update the value of this transformation at each frame to reflect the terrain ground height under the TTES entity. Our geometry being defined such that the coordinate origin is located between the human's feet, this is a simple matter of casting a vertical ray into the scene and using the intersection point directly to determine the appropriate translation matrix. The "Figure DCS" node for a particular figure corresponds to the transformation obtained by multiplying the global position transform of the figure's root site by the inverse of the root site transform. This information is also part of the update packet sent by *Jack* on each frame, mapping figure movements in the *Jack* environment into the Performer environment. Figure 14 explains the structure of the scene subgraph representing a Peabody segment.

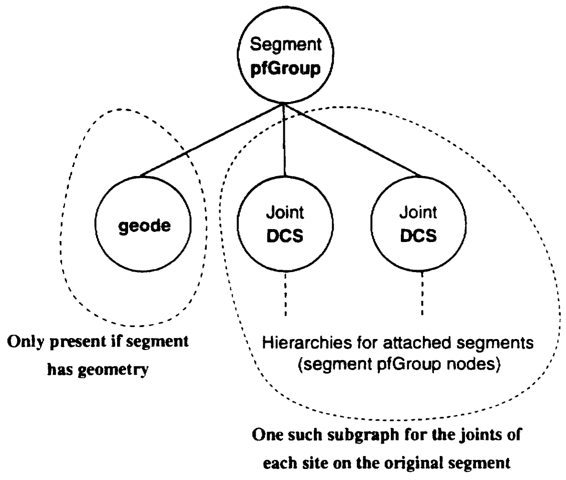
[5]Here we define **depth** to be the maximum number of pfSCS and pfDCS nodes on any path from the scene graph root to one of its leaves.

[6]An entity in our case consists of two figures: a camouflaged human and a rifle.



Scene Graph Root
Terrain Database
Entity Position DCS
Figure DCS (human)
Figure DCS (weapon)
Figure hierarchies, beginning at their respective root sites
One such subgraph for each TTES friendly/hostile

Figure 13: A TTES Entity in the Performer scene graph



Segment pfGroup
geode
Joint DCS
Joint DCS
Only present if segment has geometry
Hierarchies for attached segments (segment pfGroup nodes)
One such subgraph for the joints of each site on the original segment

Figure 14: The subgraph corresponding to a Peabody segment

Finally, the "Joint DCS" nodes are set according to the three composed transformations mentioned earlier. Changing the values stored at these DCS nodes is the main component of the update process, and hence the Performer application also maintains an indexed array of pointers, one to each of the "Joint DCS" nodes in the graph, for fast access.