NOTE TO USERS

Page(s) not included in the original manuscript and are unavailable from the author or university. The manuscript was scanned as received.

This reproduction is the best copy available.

UMI[®]

Script Language for Avatar Animation in a 3D Virtual Environment

By Xiaoli Yang

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of**

**Master of Applied Science
in
Electrical Engineering**

**Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering**

**Carleton University
Ottawa, Ontario
September 5, 2002**

**© copyright
2002, Xiaoli Yang**

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79721-X

Canada

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

**Script Language for Avatar Animation
in a 3D Virtual Environment**

Submitted by Xiaoli Yang
in partial fulfillment of the requirements for
the degree of Master of Applied Science

Co-Supervisor

Co-Supervisor

Chair, Department of Computer Engineering

Carleton University

September 5, 2002

Abstract

Virtual environments can be used in a variety of application areas such as engineering, science, education and entertainment. An avatar, which represents a virtual human in the virtual world, plays an important part in any virtual environment.

This thesis develops an effective way to control an avatar defined in VRML according to the international ISO/IEC Humanoid Standard proposed by the Humanoid Animation Working Group. The advantage of using a standard representation is that the same control system can be used for a large variety of avatars. The control system is built in three levels of abstraction. The lower level controls the rotation of the avatar's joints. Every joint has its own rotation restriction, defined for anatomical reasons. The second level implements basic behaviors (such as walk, run, jump, etc.) that coordinate different joint rotations at different times. The basic behaviors can be either independent or interactive, and can be combined in sequence and in parallel. Finally, the last level is represented by a script language described by a context-free grammar, which is translated into a set of basic behaviors. The entire control system was implemented in Java.

Acknowledgement

My sincere gratitude goes to my academic supervisor Dr. Dorina C. Petriu, who helped and guided me towards my academic as well as my personal success. Also, the thesis could not have been done without the help from my co-supervisor Dr. Thomas Eric Whalen, who provided me with a substantial amount of help and suggestions throughout my work. A special gratitude also goes to fellow students from the DISCOVER laboratory, University of Ottawa and the RADS laboratory, Carleton University, for many helpful discussions.

I would like to give my sincere thanks to my husband and parents for their support throughout my graduate studies. I am also grateful to my friends who helped me either with suggestions or with positive affirmations.

The financial support from the National Capital Institute of Telecommunications (NCIT) given for the Distributed Virtual Environments with Training Applications (DIVERTONS) project is also gratefully acknowledged.

Acronyms

3D — three-dimensional

2D — two-dimensional

A.I — Artificial Intelligence

VRML — Virtual Reality Modeling Language

H-Anim — Humanoid Animation Working Group

API — application programmer interface

EAI — External Authoring Interface

JVM — Java Virtual Machine

ACE — Agent Common Environment

AWT — Abstract Window Toolkit

Table of Contents

ABSTRACT	I
ACKNOWLEDGEMENT	II
ACRONYMS	III
TABLE OF CONTENTS	IV
LIST OF FIGURES	VIII
LIST OF TABLES.....	IX
CHAPTER 1 INTRODUCTION.....	1
1.1 MOTIVATIONS	1
1.2 RELATED WORK ON AVATAR ANIMATION	2
1.2.1 VRML Animation	3
1.2.2 Animation with Neural networks	3
1.2.3 Real-Time Avatars	4
1.2.4 ACE-Real Time Simulation of Virtual Human Agents	5
1.2.5 Improv System	6
1.3 THESIS CONTRIBUTIONS.....	6
1.4 THESIS CONTENTS.....	8
CHAPTER 2 TECHNICAL BACKGROUND	9
2.1 VIRTUAL ENVIRONMENT	9
2.1.1 Three Dimensional World.....	9
2.1.2 Java 3D.....	10
2.1.3 Virtual Reality Modeling Language -- VRML	11
2.1.4 Humanoid – Avatar Structure	14

2.2	EXTERNAL AUTHORIZING INTERFACE – EAI	16
2.2.1	Animation in VRML	17
2.2.2	Control through EAI	17
2.2.3	How EAI works.....	19
2.3	JAVA PROGRAMMING LANGUAGE	22
CHAPTER 3	VIRTUAL ENVIRONMENT AND AVATAR	26
3.1	VIRTUAL ENVIRONMENT AND AVATAR DEFINITION	26
3.1.1	Virtual Environment Definition	26
3.1.2	Avatar	29
3.2	VRML - JAVA INTERFACE	32
3.2.1	Sacroiliac Events	33
3.2.2	Left Thigh Color Events.....	34
3.2.3	The View Events	36
3.2.4	Sacroiliac Rotation Method Definition	37
3.2.5	View Bound Method Definition.....	38
3.2.6	VRML-Java Interface Class Diagram	39
3.3	GRAPHIC USER INTERFACE	40
3.4	AVATAR ANIMATION DESIGN	42
3.4.1	Design Description.....	42
3.4.2	Class Diagram for Avatar Animation Control	45
CHAPTER 4	AVATAR ANIMATION: JOINTS CONTROL AND BASIC BEHAVIORS	47
4.1	JOINTS CONTROL.....	47
4.2	BASIC BEHAVIORS	49
4.2.1	Independent Behavior	49

4.2.1.1 Walk Class Diagram.....	50
4.2.1.2 Walk Sequence Diagram.....	51
4.2.2 Interactive Behavior.....	55
4.2.2.1 Forward Kinematics and Inverse Kinematics	55
4.2.2.2 Inverse Kinematics Algorithm	55
4.2.2.3 Interactive Behavior Implementation.....	59
4.3 COMBINED BEHAVIORS	62
4.3.1 Sequential Composition	62
4.3.2 Parallel Composition.....	64
4.3.3 Class Diagram of Combined Behavior – Open Door.....	66
CHAPTER 5 AVATAR ANIMATION: SCRIPT LANGUAGE.....	69
5.1 GRAMMAR.....	69
5.2 SCRIPT LANGUAGE INTERPRETER.....	73
5.3 EXAMPLE	76
CHAPTER 6 CONCLUSIONS AND FUTURE WORK.....	79
6.1 CONCLUSIONS	79
6.2 FUTURE WORK.....	80
REFERENCES	82
APPENDIX A THE PROTO DEFINITION OF THE HUMANOID STANDARD	89
APPENDIX B JOINT ROTATION CLASS EXAMPLES.....	92
Joint Skullbase Rotation Definition Class.....	92
Joint Sacroiliac Rotation Definition Class.....	92
APPENDIX C WALK BEHAVIOR EXAMPLE.....	93
Class For Pre-defined Walk Data.....	93

Class For Walk Definition.....	95
Class For Walk Thread Definition.....	98
APPENDIX D SCRIPT LANGUAGE INTERPRETER.....	101
APPENDIX E SCRIPT LANGUAGE SENTENCES.....	104

List of Figures

Figure 2.1	Skeleton description for H-Anim.....	15
Figure 2.2	Communications between Java and VRML.....	19
Figure 3.1	The entry view of the virtual house.....	27
Figure 3.2	The bird's eye view of the virtual house.....	27
Figure 3.3	A front view of the virtual room.....	29
Figure 3.4	Nancy.....	30
Figure 3.5	Left hand joint definition of Nancy.....	32
Figure 3.6	ControlJoint class diagram.....	39
Figure 3.7	The interface for user to control avatar.....	40
Figure 3.8	Class diagram of for avatar animation control	46
Figure 4.1	Class diagram of "sacroiliac" thread	48
Figure 4.2	Class diagram of "walk" behavior.....	50
Figure 4.3	UML sequence diagram illustrating "walk" behavior	52
Figure 4.4	UML sequence diagram illustrating "open door" behavior.....	60
Figure 4.5	UML sequence diagram illustrating "go to door" behavior.....	63
Figure 4.6	UML sequence diagram illustrating "walk" and "wave" behavior.....	65
Figure 4.7	Class diagram of "open door" thread.....	68
Figure 5.1	A parse tree for an input.....	72
Figure 5.2	Nancy is walking.....	77
Figure 5.3	Nancy is turning on the light.....	78

List of Tables

Table 3.1	Joint and segment names of Nancy.	31
Table 4.1	Basic movements and parameters.....	54
Table 4.2	Interactive behaviors and parameters.....	62
Table 5.1	Grammar for the script language: Set of rules R.....	71
Table 5.2	Semantic Table	75

CHAPTER 1 INTRODUCTION

1.1 Motivations

A three-dimensional (3D) virtual world provides a new way to describe and explain the objects in the real world. It supports interesting, live and accurate images or models used in many application fields. A 3D virtual environment with avatars, the computer model of human beings, has many applications in education, industry and military areas. For example, a virtual environment with avatars can be used for education applications, where an avatar can give lectures instead of a real professor. This saves a lot of effort and time for real human beings. In today's manufacturing industry, the human element should be considered thoroughly when designing assembly lines or maintenance processes. Simulations done with virtual environments and avatars help to achieve shorter design time, lower development cost, and enhanced safety. Virtual environments and avatars can also be used in military applications for simulating military actions that are very dangerous to real human beings.

In this context, the problem of how to control avatars is more and more important. An avatar is not only a part of the virtual environment, but is also the controller that interacts with other objects in that environment. Avatars can be used:

1. As substitutes for “the real thing” in ergonomic evaluations of computer-based designs for vehicles, work areas, machine tools, assembly lines, etc.
2. For embedding real-time representations of ourselves or other live participants into the virtual environment.

The research relevant to the problem of simulating human motion includes several fields: computer graphics, robotics, biomechanics, and psychology Artificial Intelligence (A.I). In the thesis, we focus on computer graphics. Nowadays, many computer applications begin to change from two-dimensional (2D) to 3D virtual environments. After all, the real world is a 3D world. In [23,30] it is claimed that it will be easier for people to understand and use some software applications if a 3D environment is provided with it. This can improve the user’s interactivity and save a lot of time for the users. Nowadays, people go shopping, share information, and get entertainment through the Internet [37]. Virtual environments and animated avatars can be put on the Internet to make all these applications look more realistic and be easier to use. In the near future, avatars can also be used as agents between computer and user.

1.2 Related Work on Avatar Animation

There are many approaches to avatar control to simulate real human beings in the computer graphics research area. The following is a short review of related relevant research.

1.2.1 VRML Animation

An example presented in [32, 15] uses the Virtual Reality Modeling Language (VRML) [54, 55] as the visualization and integration technology for manufacturing system simulation. VRML translators are used to produce reusable VRML components. Avatars defined in VRML simulate the real human beings in the manufacturing system. The example shows how the assembly line works and how the workers cooperate together to fulfill a task. In this way, some simple and repeatable procedures can be simulated well. VRML suits this scenario because the manufacturing system doesn't change a lot. However, VRML itself is not a programming language, so the user can only create simple animations with it. Also VRML does not give the user many options to interact with the virtual environment.

1.2.2 Animation with Neural networks

One of the major problems in the field of virtual human animation is to make the avatar move like a real human being. Specific behaviors of a certain human need to be simulated sometimes. Neural network models can be used to implement such specific behaviors, as shown in [33, 31, 3, 14, 18]. The approach is based on the acquisition of real human kinematics data, collected by means of an automatic motion analyzer. Starting from the kinematics database of a given subject, a neural network model can be established and trained to generate the movements of the virtual human. This method is useful when a specific person's behavior needs to be reproduced.

1.2.3 Real-Time Avatars

Another example introduces real-time agents represented as avatars [4]. It simulates people from different cultures with different personalities. Various aspects of real-time virtual humans are considered, such as appearance and motion, interactive control, autonomous actions, gesture, attention [1, 5, 16, 21, 40, 7]. This richness and diversity are reflected in avatars.

The appearance of the avatar can be developed as a polygonal model with rigid segments and joint motions [16]. The model is prototyped into the JackMOO system [35, 28, 2, 9], which can be used to build avatars in many other applications. A “smooth body” is developed using free-form deformation techniques to aid in the portrayal of visually appealing virtual humans [34].

For the avatar movements, a time series of joint angles is stored so that specific motions can be re-played under real-time constraints [20]. The major advantages of pre-stored motions are fast speed of execution and safer algorithmic security (by minimizing computation). However, the principal disadvantages of pre-stored motion are the lack of generality (since every joint must be controlled explicitly) and anthropometrical extensibility (since changing joint-to-joint distances will change the computed locations of end effectors such as feet, making external constraints and contacts impossible to maintain) [4].

This real-time avatar system was already developed with existing 3D technologies to provide training scenarios across the Internet in a form as close as possible to the

appearance and interaction expected of live situations with human participants. This approach is very useful when real-time avatars are needed in some diversified virtual environments. According to [23], the system is more oriented for human factors applications rather than social and behavior animations. It is not suitable for the applications that need many behavior animations and interactive communications.

1.2.4 ACE-Real Time Simulation of Virtual Human Agents

ACE (Agent Common Environment) developed by the EPFL computer Graphics Lab is a platform for virtual human agent simulation that is able to coherently manage a shared virtual environment [22, 23, 6, 11, 12, 10]. This system can simulate scenarios in the virtual environment [22]. It provides built-in commands for perception and acting. It also defines an external collection of behavioral plug-ins. One kind of plug-in can define agent-object interactivity by using a feature modeling approach; another controls the reasoning and behavior of each agent through a scripted module.

This system already contains many behavioral plug-ins. The user can use these behaviors to create a story for the virtual environment. However, the user must extend some control functions if he wants to use this system for a specific task. For example, the system has a built-in walking motor but does not have a navigation control, as navigation requirements can change depending on many issues: real time interactivity, human-like navigation and exploration, optimized path planning, etc [22]. Therefore, a user who wishes to use the built-in walking motor has to construct its own navigation control.

1.2.5 Improv System

The Improv System from New York University is mainly controlled by behavioral scripts [31, 29, 30, 8]. The behavioral scripts are designed such that they can be translated easily from a given storyboard. In this system, noise techniques are used to change image rendering to character movement. Characters created in this manner can smoothly blend and layer animations. This type of smooth blending is often referred to as procedural animation. These characters can also use noise-influenced models for decision making at many levels: ranging from low-level animation triggering (e.g., eye blinking), to mid-level behaviors (e.g., approach/avoid), to high-level attitudes that develop over time. These characters aren't attempting to be intelligent in their behavior, but rather to use carefully crafted statistical models to engage their audience of users.

The system is useful for predefined scenarios in the virtual environment. External functions need to be added if it is used in the real-time controlled virtual environment and the characters need to make their decisions often.

1.3 Thesis Contributions

The main contributions of the thesis are divided into two categories: contributions to knowledge and practical contributions.

A. Contributions to knowledge

1. Propose and implement a three-level hierarchical control system for the animation of avatars that are represented according to the International ISO/IEC

Humanoid Standard proposed by the Humanoid Animation Group [49]. The lower level controls the rotation of the avatar's joints. The second level defines basic behaviors (such as walk, run, jump). The basic behaviors can be combined in sequence or in parallel. The last level of abstraction is represented by a script language, which can be used to describe stories that will be enacted by the avatars.

2. Define a context-free grammar for the script language and implement an interpreter for it. The interpreter translates the stories entered by users into a set of commands that represents basic behavior, and triggers their execution by the avatar.

B. Practical contributions

1. Build an independent and flexible virtual environment and avatar in VRML. The virtual environment includes a virtual house and a virtual room. The avatar, named Nancy, is implemented according to the Humanoid Standard.
2. Implement a three-level hierarchical control system for the avatar in Java. The control system can access the VRML objects representing the virtual environment and the avatar through a Java-VRML interface, which was also developed in the thesis. The control system uses Java threads for performing concurrent tasks, and implements a 2D inverse kinematics algorithm to let the avatar reach different objects from the virtual environment.

Publications arising from the thesis: the following paper was presented at *VIMS2002* and another paper is in preparation:

D.C. Petriu, X.L. Yang, and T.E. Whalen, "Behavior-Based Script Language for Anthropomorphic Avatar Animation in Virtual Environments," Proc. of *VIMS2002, IEEE Intl. Symposium on Virtual and Intelligent Measurement Systems*, Anchorage, AK, USA, May 2002.

1.4 Thesis Contents

Chapter 1 describes the motivation, related work and the contribution of the thesis.

Chapter 2 presents the relevant background: VRML, Humanoid Standard, interface between Java and VRML, and Java.

In chapter 3, the creation of the virtual environment and the avatar is described. The technique of how to build the concrete interface between Java control and VRML objects is also discussed.

Chapter 4 presents how to control the basic joint movements and how to compose the avatar behaviors based on joint movements.

Chapter 5 focuses on the script language. This chapter describes the context-free grammar for the script language and presents the implementations of the interpreter.

In chapter 6 are given the conclusions of the thesis and directions for future work.

Chapter 2 Technical Background

This chapter presents an overview of the techniques related to the thesis, such as VRML, Java3D, standard VRML Humanoid, EAI (External Authoring Interface) and Java programming language.

2.1 Virtual Environment

2.1.1 Three Dimensional World

A 3D world is more attractive than a traditional 2D world in many aspects. 3D virtual worlds can be used in a variety of application areas such as engineering, science, education, entertainment, multimedia presentations, web pages, and shared virtual worlds. For example, in chemistry, a 3D virtual model is used to illustrate the molecules structures. In architecture, a 3D virtual model can be used to visualize the design of houses. The virtual model gives the user a more direct and clear view of the real object because the user can view the object from many angles and different positions. Many models can be now distributed on the web and viewed by anybody who can access the Internet [3]. Classical web browsers such as Netscape Navigator or Internet Explorer can be used now to explore 3D worlds as easily as to view 2D HTML pages.

There are many ways for building and describing 3D shapes and interactive worlds such as VRML [54] [43], Java3D [55], OpenGL [48], Direct3D [46], QuickDraw 3D [52], and SGL [53]. We will briefly describe the first two of them.

2.1.2 Java 3D

Java3D is a high-level application programmer interface (API) not tied to VRML, or any 3D file formats. It is mainly developed by Sun. The Java 3D API is a set of classes for writing three-dimensional graphics applications and applets, and is a part of JavaMedia suite of APIs [51]. It gives developers high level constructs for creating and manipulating 3D geometry and for constructing the structures used in rendering the geometry [51].

As a programming language, Java has the remarkable character of platform independence (“write once, run anywhere”) due to the Java Virtual Machine. Java 3D inherits this advantage. Java3D also integrates well with the Internet because applications and applets written using the Java 3D API have access to the entire set of Java classes.

Java 3D’s low-level graphics constructs synthesize the best ideas found in low-level APIs such as Direct3D [46], OpenGL[48], QuickDraw3D[52], and XGL [57]. Similarly, its higher-level constructs synthesize the best ideas found in several scene graph-based systems.

Java3D is a program-centric way to build 3D worlds. Neither the application nor the content is standardized. Both are variable. Java3D is good when the operations and the content are complex and application-specific. Java3D is a set of standardized classes for use by Java Programmers. It is roughly a Java-based superset of VRML.

When this thesis was started, a choice had to be made between using Java 3D or VRML + Java. We decided to use the later because we wanted to apply the Humanoid-standard,

which was defined in VRML. Another reason is the fact that the definition of a virtual environment in VRML is easier than in Java 3D.

2.1.3 Virtual Reality Modeling Language -- VRML

VRML is a well-known language for describing interactive 3D objects and worlds. The design of VRML makes it easy to be used on the Internet, intranets, and local client systems [27, 55]. VRML is also intended to be a universal interchange format for integrated 3D graphics and multimedia. VRML files are standard text files that can be interpreted by browsers.

VRML has been designed to be platform independent like HTML. While HTML files describe a 2D page containing various text constructs, VRML files describe a 3D space, or “world”. It can represent static and animated dynamic 3D and multimedia objects with hyperlinks to other media such as text, sounds, movies, and images. VRML browsers, as well as authoring tools for the creation of VRML files, are widely available for many different platforms [55]. The user can explore the virtual world, zooming in and out, moving around and interacting with the virtual environment by use of VRML browser. This allows complex 3D graphics to be transmitted across networks without very high bandwidth.

For the early versions, VRML was a simple 3D file format. From version 2.0 on, VRML becomes a scene description language which is human readable.

VRML has been designed to fulfill the following requirements [26, 55]:

- Authorability: Enable the development of computer programs capable of creating, editing, and maintaining VRML files, as well as automatic translation programs for converting other commonly used 3D file formats into VRML files.
- Composability: Provide the ability to use and combine dynamic 3D objects within a VRML world and thus allow re-usability.
- Extensibility: Provide the ability to add new object types not explicitly defined in VRML.
- Capable of implementation on a wide range of systems.
- Performance: Emphasize scalable, interactive performance on a wide variety of computing platforms.
- Scalability: Enable arbitrarily large dynamic 3D worlds.

Besides the above requirements, the VRML language has the following capabilities [55]:

- Built-in geometric primitives including face sets and solids
- Lighting, material, texture, movie control
- Spatial sound
- Absolute time for animations
- The concept of avatar to allow collision detection
- Hyperlinking, viewpoints and navigation methods

- The ability to extend the language through prototyping
- Event handling to detect when geometry is interacted with in some way
- Routing of events to allow one object to affect another
- Scripting in various languages which allows logic in the world.

With the event capability, VRML has a system of events that permits it to establish a kind of communication between any VRML objects. With the extension ability, VRML can also include a piece of program inside to establish a connection between an external program and the VRML application. It allows new dynamic 3D objects to be defined explicitly. The application communities can develop interoperable extensions to the base standard. There are mappings between VRML objects and commonly used 3D API features [25, 45].

VRML is a content-centric approach to build 3D worlds [27]. The application (browser) is standardized; only the content is variable. VRML is useful when the operations on the content are simple and well known (walk around and draw). VRML is primarily for content-developers.

An advantage of VRML is that any web server can store VRML files. It is easy to use, display and publish. In this thesis, the virtual world and avatar both were built in VRML because of the above advantages.

However, VRML has some disadvantages when it comes to the control of the avatars and virtual environments. These are due to the fact that VRML is not a programming

language per se, so more complex animations algorithm cannot be easily expressed. For this reason, we chose to implement the avatar controller in Java.

2.1.4 Humanoid – Avatar Structure

As the 3D virtual world continues to grow, there is an increasing need to represent human beings in virtual environments. The human being called avatar in virtual environment can be immersed into a computer generated multi-sensory environment, which may include sight and feel. Furthermore, the user is able to interact with this environment through an avatar in various ways derived from the real world [49].

Humanoid Animation Working Group (H-Anim) [49] created the standard humanoid-avatars in the virtual environment. This standard is an International Standard ISO/IEC. It has libraries of interchangeable humanoids, as well as authoring tools that make it easy to create new humanoids and animate them in various ways [49]. The standard humanoid has the following characters: compatibility, flexibility and simplicity. H-Anim humanoids can be animated using keyframing, inverse kinematics, performance animation systems and other techniques.

The human body consists of a number of segments (such as the forearm, hand and foot) which are connected with each other by joints (such as the elbow, wrist and ankle).

Figure 2.1 is the skeleton description for H-Anim 1.1 [49]. It describes the standard joints, segments and sites definitions. This figure is just for the left side of human being. The concrete description is in the figure.

Figure. 2.1 Skeleton description for H-Anim [49].

A H-Anim file contains a set of joint nodes that are arranged to form a hierarchy. Each joint node can contain other joint nodes, and may also contain a segment node which

describes the body part associated with that joint. Each segment can also have a number of site nodes, which define locations relative to the segment. Sites can be used for attaching clothing and jewelry, and can be used as end-effectors for inverse kinematics applications [49].

Some applications for animating a humanoid obtain access to the joints and alter the joint angles. The applications may also need to retrieve information such as joint limits and segment masses. A mesh of polygons typically defines each segment of the body. In some applications, locations of the vertices in that mesh may need to be altered.

The H-Anim file also contains a single Humanoid node that stores references to all joints, segments, site nodes, and serves as a "wrapper" for the humanoid. In addition, the node provides a top-level transform for positioning the humanoid in its environment.

2.2 External Authoring Interface – EAI

The fundamental goal of the External Authoring Interface (EAI) working group is to produce an external interface for VRML. It defines the way to communicate with the VRML world. VRML enables users to control shapes and some sample animations and interactions. For a better control of objects in VRML, the external control can be added and connected with VRML objects. The following part explains how VRML implements animation and how to extend the control for objects in VRML from external controls.

2.2.1 Animation in VRML

In the thesis, the virtual environment and avatar are both built in VRML. VRML can be displayed on any machine using plug-ins like BlaxxunVRML [43], Cosmo Player [45], Corona [44]) or stand-alone applications like VRWave [56].

The animations or interactions are fulfilled by sending/receiving events to/from objects in VRML. Each kind of object (known as node: Group, Material, Texture) has a set of events which can be received and/or sent [3]. For instance, a rotation node can generate a rotation_changed event which specifies an update of the rotation of the object and receive a set_rotation event which updates the position of the object. VRML world builders can then define which outgoing event field will be sent to which incoming event field in order to make nodes communicate by using a kind of command – called route command [3] in VRML. But route are only possible between events of the same type: it is illegal to send a translation_changed eventOut to a set_rotation eventIn. ExposedField is a combination of both eventIn and eventOut. It is capable of sending and receiving an event [3].

2.2.2 Control through EAI

The route command can be defined in a static way. It can also be defined dynamically by using a Script node. It is possible to add a route command or to delete an existing command at runtime. For example, this is useful when the user wants to attach an object to a moving character by routing the position of the body to the position of the object. When the actor releases the object, the script can delete the route command and the

object will stay in place. A Script node gives the possibility of performing intelligent changes in the VRML world. By including the capabilities of a programming language, updates of the VRML world can be from a complex computation instead of a direct route of basic information from one node to another. The Script node can contain several types of scripting languages or programming languages [3]:

- Integrated language: “vrmlscript”, “ECMAScript” which use the JavaScript Authoring Interface (JSAI).
- Compiled language: Java, which is the most powerful solution, by using EAI.

Figure 2.2 shows the event communications what VRML permits.

Figure 2.2 Communications between Java and VRML

2.2.3 How EAI works

Java is chosen to control VRML objects in the thesis. EAI is needed to connect Java and VRML. EAI is not a part of the VRML 2.0/97 specification. EAI allows a Java Applet to communicate with a VRML browser on the same web page. It is designed to allow an external environment to access nodes in a VRML scene using the existing VRML event model.

Conceptually, EAI allows four types of access to the VRML scene:

1. Accesses the functionality of the Browser Script Interface.
2. Sends events to eventIns of nodes inside the scene.
3. Reads the last value sent from eventOuts of nodes inside the scene.
4. Gets notified when events are sent from eventOuts of nodes inside the scene.

The following part explains how EAI works with both VRML and Java. It also gives examples of how to get the browser and how to communicate with events of VRML objects.

The external environment communicates with a VRML world by obtaining an identifier for a browser object [44]. This allows the external environment to uniquely identify a particular VRML scene in environments where multiple scenes are available. This object can be created, in which case the protocol is responsible for providing all information needed to introduce the object into the environment [25]. An instance of an existing scene is obtained by instantiating a Browser class. This class is used when the application is embedded in another environment (such as a HTML browser). The following code is the specification of the external interface for a VRML applet browser. Line 2 gets browser from the environment. Line 3 gets node by it's name from the VRML file [25].

```
1 public class Browser implements IBrowser {  
2     public Browser(Applet pApplet);  
3     public Node getNode(String name)  
4         throws InvalidNodeException;  
5 }
```

The following code shows the methods of the node in VRML. Line 2 gets a handle to an eventIn of this node. Line 4 is to get a handle to an eventOut of this node [25].

```

1 public class Node {
2     public EventIn getEventIn(String name)
3         throws InvalidEventInException;
4     public EventOut getEventOut(String name)
5         throws InvalidEventOutException;
6 }

```

The following code gives an example of how to send events to eventIns of nodes inside the scene. It is the example of eventIn of SFRotation. Line 2 is the method that sends the event to the eventIn of the node[25].

```

1 public class EventInSFRotation extends EventIn {
2     public void setValue(float[] value)
3         throws IllegalArgumentException;
4 }

```

The following code shows how to read the last value sent from eventOuts of nodes inside the scene. It is the example of eventOut of SFRotation. Line 2 is the method that reads the value from the eventOut of the node [25].

```

1 public class EventOutSFRotation extends EventOut {
2     public float[] getValue();
3 }

```

Here in the thesis, CosmoPlayer [45] is used to display the VRML file and is in fact the EAI between VRML and Java. CosmoPlayer is the VRML browser from Cosmo Software. We chose it because it can implement all of the required functions and renders scenes well. Cosmo Player runs in a web browser as a plug-in. It lets users navigate and manipulate 3D worlds and objects created in the VRML. It supports Java class files allowing the user to connect a Java Applet with the VRML files.

2.3 Java Programming Language

Java is an object-oriented language. Originally, it was designed mainly for Internet programming, but it is also used as a general-purpose programming language. A Java program consists of a number of class and interface definitions. A class definition is a "blue print" to construct objects. A class definition specifies the state of an object by defining the state variables, and what is the behavior of the objects, i.e., how the objects are to be manipulated by defining methods [50].

The Java programming language is robust and versatile. Developers can do the following tasks [50]:

- Write software on one platform and run it on another.
- Create programs to run within a web browser.
- Develop server-side applications for online forums, stores, polls, processing HTML forms, and more.
- Write applications for cell phones, two-way pagers, and other consumer devices.

The Java platform is based on the power of networks and the idea that the same software should run on many different kinds of computers, consumer gadgets, and other devices. Since its initial commercial release in 1995, Java technology has achieved more popularity and usage because of its true portability. The Java platform allows users to run the same Java application on a lot of different kinds of computers. Java works differently from the other languages. The Java source code (with an extension ".java") is "compiled"

into the "machine language" of the "Java Virtual Machine" (JVM) which is a kind of hypothetical computer. JVM can be implemented by an interpreter that translates general Java platform instructions into tailored commands that make the devices do their work. The machine language of JVM is called the byte-code. The byte-code program is stored in a file with extension "class". When the program is executed, another program is used to interpret the byte-code instructions into machine language. This technique enables the Java's remarkable character "write once, run anywhere"[50].

Java technology allows programmers and users to do things with Web pages that were not possible before. With Java technology, the Internet and private networks becomes a computing environment. For example, users can securely access their personal information and applications when they're far away from the office by using any computer connected to the Internet. Sooner or later, they'll even be able to access tailored applications from a cell phone based on the Java platform, or even use smart cards as a passkey to a variety of devices from cash machines to ski lifts.

An applet is a program written in the Java programming language that can be included in an HTML page, much in the same way an image is included. When a Java technology-enabled browser is used to view a page that contains an applet, the applet's code is transferred to the client system and executed by the browser's JVM [50].

The Java Applet class extends the "Abstract Window Toolkit " (AWT) Panel class, which extends the AWT Container class, which extends the AWT Component class. From Component, an applet inherits the ability to draw components and handle events. From Container, an applet inherits the ability to include other components and to have a layout

manager to control the size and position of those components. From the Applet class itself, an applet inherits several capabilities, including the ability to respond to major milestones, such as loading and unloading.

The applets are included in HTML pages using the `<APPLET>` tag. When a browser user visits a web page that contains an applet, the process works as follows [50]:

1. The browser finds the class file for the applet's Applet subclass. The location of the class file (which contains Java bytecodes) is specified with the `CODE` and `CODEBASES` attributes of the `<APPLET>` tag.
2. The browser brings the bytecodes over the network to the user's computer.
3. The browser creates an instance of the Applet subclass. When we refer to an applet, we're generally referring to this instance.
4. The browser calls the applet's `init` method. This method performs any one-time initialization that is required.
5. The browser calls the applet's `start` method. This method often starts a thread to perform the applet's duties.

An applet's Applet subclass is its main controlling class, but applets can use other classes as well. These classes can be either local to the browser, provided as part of the Java environment, or custom classes. When the applet tries to use a class for the first time, the browser tries to find the class on the host running the browser. If the browser can't find the class there, it looks for the class in the same place that the applet's Applet subclass

came from. When the browser finds the class, it loads its bytecodes (over the network, if necessary) and continues executing the applet.

Loading executable code over the network is a classic security risk. For Java applets, some of this risk is reduced because the Java language is designed to be safe -- for example, it doesn't allow pointers to random memory. In addition, Java-compatible browsers improve security by imposing restrictions. These restrictions include disallowing applets from loading code written in any non-Java language, and disallowing applets from reading or writing files on the browser's host.

In the thesis, the Java controller is implemented as a Java applet.

Chapter 3 Virtual Environment and Avatar

In this chapter the definition of the virtual environment and the avatar in VRML is presented. In order to achieve a more flexible and better control of the avatar, Java is used to design and execute all the control tasks of the avatar. To connect the Java control with the avatar and its environment, the interface between them is built and explained in detail.

3.1 Virtual Environment and Avatar Definition

3.1.1 Virtual Environment Definition

The 3D virtual environment is the place for avatars to move, and interact with other objects within the environment. In the thesis, three axes are defined for the 3D virtual environment in Cartesian coordinates as X — pointing left, Y — pointing upward, and Z — pointing inside. The virtual environment built in the thesis includes a virtual house and a virtual room.

Figure 3.1 and Figure 3.2 show the virtual house from two different perspectives. The virtual house includes a back yard with trees, a front yard with a paved path and a house. The only active object defined is the door, which can be open and closed. The other objects are for decoration only.

Figure 3.1 The entry view of the virtual house

Figure 3.2 The bird's eye view of the virtual house

The virtual environment is built with VRML. The basic VRML files contain:

- The file header
- Comments - notes for author himself

- Nodes – nuggets of scene information
- Fields & values – node attributes
- Routes – animation wiring
- Prototypes – custom nodes

VRML uses a node to describe a virtual object. The following code is an example of a simple node – the roof on the right side of the house:

```

1 Transform {
2     Translation -21.185 7.60802 22.68
3     rotation 0 0 -1 0.885
4     children[
5         Shape {
6             appearance Appearance {
7                 texture ImageTexture {
8                     url "brick1.jpg"
9                 }
10            }
11            geometry Box{
12                size 0.18 5.75 8.3
13            }
14        }
15    ]
16 }
```

First of all, the Translation in the Transform defines the position of the roof on line 2. The three numbers define the X, Y and Z position in the virtual space. If the object needs to be rotated for a specific angle, the rotation will be defined. The four numbers are X, Y, Z factors for the axis of rotation and the overall angle on line 3. The roof's appearance will use the brick image file as the texture on line 8. Line 11 indicates this object's shape is a

box shape. The size of this roof is defined by the size parameter on line 12. The three numbers are the sizes for X, Y and Z-axis.

The whole virtual environment in the thesis is built with the objects defined as shown above. After the avatar opens the door of the virtual house, the user will see the virtual room shown in Figure 3.3. In the virtual room, there are a table with a computer, a telephone, two speakers and a table lamp, a file container besides the table, a virtual chair. The avatar can turn the computer and table lamp on/off.

Figure 3.3 A front view of the virtual room

3.1.2 Avatar

There are many example avatars defined and available through the Internet based on the standard humanoid – H-Anim humanoid body representation. In the thesis, the avatar named Nancy was chosen from H-Anim examples. Nancy was created by the 3Name3D

Company [47]. Nancy has standard joints and segments according to H-Anim's definition.

Nancy shown in Figure 3.4, wears red top and blue pants.

Figure 3.4 Nancy [49].

Before a standard avatar is defined in VRML, the Joint, Segment and Humanoid prototypes (called Proto in VRML) are defined first. The definition of a new avatar is just an implementation of Proto. The user can choose to implement the definitions of some or all the joints and segments for different avatars. The Proto definition and the implementations of the joints and segments are given in Appendix A.

The difference among avatars is that the segment shapes and the positions of the joints are different. This difference doesn't influence the avatar control in the thesis because it only controls the rotation of different standard joints. The control is not related to the

segment shape or the joint positions. Nancy is chosen in the thesis because the most representative joints are defined. The controller can control any other avatars defined with the H-Anim standard in VRML if it can control Nancy. When Nancy is chosen for the thesis, not all of the joints are defined according to the standard. 3Name3D Company only defined the imperative joints of the human body for Nancy. For example, the original definition contained the palm segment but did not define the finger joints. However, the finger joints were needed in the thesis to open the door or turn on the computer in our virtual environment, so that Nancy's original definition was extended here with a more detailed hand.

Nancy's new definition has 45 joints and 15 body segments altogether. Table 3.1 lists all of them (l_ means the left part, r_ means the right part).

Table 3.1 Joint and segment names of Nancy

Joints			segments
Body joints	Left hand joints	Right hand joints	
humanoidRoot	l_thumbbase	r_thumbbase	pelvis
sacroiliac	l_thumbext	r_thumbext	l_thigh
l_hip	l_indexbase	r_indexbase	l_calf
l_knee	l_indexmid	r_indexmid	l_hindfoot
l_ankle	l_indexext	r_indexext	r_thigh
r_hip	l_middlebase	r_middlebase	r_calf
r_knee	l_middlemid	r_middlemid	r_hindfoot
r_ankle	l_middleext	r_middleext	c7
vl1	l_ringbase	r_ringbase	l_upperarm
l_shoulder	l_ringmid	r_ringmid	l_forearm
l_elbow	l_ringext	r_ringext	l_hand
l_wrist	l_pinkybase	r_pinkybase	r_upperarm
r_shoulder	l_pinkymid	r_pinkymid	r_forearm
r_elbow	l_pinkyext	r_pinkyext	r_hand
r_wrist			c4
vc4			
skullbase			

Figure 3.5 depicts the left hand joints definition. It gives the joints name and position on the left hand.

Figure 3.5 Left hand joint definition of Nancy [49].

3.2 VRML - Java Interface

As mentioned before, the avatar and the virtual environment are defined in VRML. Because VRML is not a programming language and cannot create complex control, the controller is defined with Java to manage Nancy's movements and interactions with the virtual objects in the environment. The VRML-Java interface is the bridge to connect the two parts.

Every node in VRML has its own unique name. The name is the identification of the node. The interface defines `eventIn` and `eventOut` for each field of each VRML node that Java wants to control. The Java controller can trigger `eventIn` of the field to change its values in VRML through the interface, and can also read the values of the field through

eventOut. There are many different fields in VRML. For example, the Boolean field SFBool contains a single Boolean value; the color field SFCOLOR specifies one RGB (red-green-blue) color triple, MFCOLOR specifies zero or more RGB triples; the rotation field SFRotation specifies one arbitrary rotation, and the MFRotation specifies zero or more arbitrary rotations [55, 25].

3.2.1 Sacroiliac Events

To control the joint rotations of the avatar, both eventIn and eventOut for rotation should be defined for all of the joints. For example we want to control the rotation of sacroiliac joint of Nancy. In VRML, the sacroiliac joint is defined by the following code:

```

...
1 children [
2     DEF hanim_sacroiliac Joint {
3         name "sacroiliac"
4         rotation 0 0 1 0
5         center 0 1.01 -0.0204
6         children [DEF hanim_pelvis_s Segment {
7             name "pelvis"
8             children Shape
...

```

The sacroiliac joint is inherited from hanim joint. The node name of sacroiliac in VRML is hanim_sacroiliac on line 2. This joint has three exposedFields: name, rotation and center on line 3, 4 and 5. These three exposedField are of types of SFString, SFRotation and SFVec3f respectively.

Both eventIn and eventOut of sacroiliac rotation exposedField should be defined for control the sacroiliac. The code is listed below:

```

1 EventOutSFRotation sacroiliacOrientOut ;
2 EventInSFRotation sacroiliacOrientIn ;
3 node = browser.getNode ("hanim_sacroiliac" ) ;
4 try {
5   sacroiliacOrientIn = (EventInSFRotation)node . getEventIn ( "set_rotation" ) ;
6   sacroiliacOrientOut = (EventOutSFRotation)node . getEventOut
7     ("rotation_changed" ) ;
8   }
9 catch ( InvalidEventInException e ) {}
10 catch ( InvalidEventOutException e ) {}

```

Line 1 and 2 are the definitions of SFRotation eventIn and eventOut: sacroiliacOrientOut, sacroiliacOrientIn. Line 3 enables the browser to fetch the sacroiliac joint node. In VRML, this node's name is hanim_sacroiliac. On line 5 and 6, both the sacroiliacOrientIn and sacroiliacOrientOut are connected with the rotation of sacroiliac joint node through the library classes of API.

3.2.2 Left Thigh Color Events

The following part explains how to define the events for color of the segment lthigh. In VRML, the definition of lthigh is as follows:

```

...
1 children [ DEF hanim_l_thigh_s Segment {
2   name "l_thigh"
3   children Shape {
4     appearance Appearance {
5       material DEF hanim_l_thigh Material {
6         ambientIntensity 0.25
7         diffuseColor 0.1 0.3 0.3
8       }
9     }
10    geometry IndexedFaceSet {
11      coord Coordinate {...}

```



```

12         }
13     }
...

```

For the `hanim_l_thigh_s` segment in VRML, there are several exposedFields according to the code. In order to change the segment color we need to find the node first, then exposedField for the color. The name for the node is `hanim_l_thigh` on line 5. The exposedField is `diffuseColor` defined on line 7.

To access the exposedField `diffuseColor`, both `EventIn` and `EventOut` of this exposedField need to be defined. The following code illustrates how to define them :

```

1  EventInSFCOLOR lthighDiffuseColorIn;
2  EventOutSFCOLOR lthighDiffuseColorOut;
3  colorNode = browser.getNode ( "hanim_l_thigh" ) ;
4  try {
5
ThighDiffuseColorIn=EventInSFCOLOR)colorNode.getEventIn("set_diffuseColor");
6
lthighDiffuseColorOut=EventOutSFCOLOR)colorNode.getEventOut("diffuseColor_changed
");
7      }
8  catch ( InvalidEventInException e ) {}
9  catch ( InvalidEventOutException e ) {}

```

Line 1 and 2 define SFCOLOR eventIn and eventOut: `lthighDiffuseColorIn` and `lthighDiffuseColorOut`. Then the segment node is obtained by the browser on line 3. From the VRML code above, this node's name is `hanim_l_thigh`. The `lthighDiffuseColorIn` and `lthighDiffuseColorOut` are connected with the `diffuseColor` of `hanim_l_thigh` node on line 5 and 6 through the library classes of API.

3.2.3 The View Events

Besides the control of the avatar, the virtual environment itself also needs to be controlled. When the avatar enters into the virtual room from the virtual yard, the sky color, ground color, light, view and so on all need to be changed. All the eventIn and eventOut of these should be defined. For example, the code to define the viewpoint in the house is as follows:

```

1 DEF view_house Viewpoint {
2     fieldOfView 1.6499
3     jump TRUE
4     orientation 0 0.7555966 0.001 3.00795
5     position -17.132 0.9516 9.802
6     description "viewpoint_house"
7 }

```

The node view_house has several exposedFields such as fieldOfView on line 2; jump on line 3; orientation on line 4 and position on line 5. This node still has eventIn of set_bind and eventOut of isBound. These eventIn and eventOut are not shown in the code because they don't have values like exposedField does.

The following Java code defines the events of node view_house in order to change the viewing point for the virtual environment.

```

1 EventInSFBool bindIn_h;
2 EventOutSFBool bindOut_h;
3 EventOutSFVec3f viewOut_h;
4 EventInSFVec3f viewIn_h;
5 node = browser.getNode ("view_house" );
6 try {
7     viewIn_h = (EventInSFVec3f)node . getEventIn ( "set_position" );
8     viewOut_h = (EventOutSFVec3f)node . getEventOut ("position_changed" );

```

```

9  bindIn_h = (EventInSFBool)node . getEventIn ( "set_bind" ) ;
10 bindOut_h = (EventOutSFBool)node . getEventOut ( "isBound" ) ;
11  }
12 catch ( InvalidEventInException e ) {}
13 catch ( InvalidEventOutException e ) {}

```

The control is similar to the joint rotation or segment color change above. From line 1 to 4, the eventIn and eventOut of both SFBool and SFVec3f are defined for the virtual environment viewing: bindIn_h, bindOut_h, viewOut_h and viewIn_h. Then the house node is obtained by the browser on line 5. In VRML, this node's name is view_house. From line 7 to line 10, viewIn_h and viewOut_h are connected to the position of the view; These both events are eventIn and eventOut for the exposedField position in VRML code. bindIn_h and bindOut_h are connected to the bind of the view through the library classes of API.

To set the view of the virtual environment is a little different from rotating a joint or changing the segment's color. In VRML, several views can be defined simultaneously. When the user wants to change from one view to another, he needs to change the view's position exposed field and generate an eventIn named set_bind.

All of the active objects such as the virtual computer, the virtual table light are defined by the EventIn and EventOut of different fields in the virtual environment.

3.2.4 Sacroiliac Rotation Method Definition

After the definition of all of the EventIn and EventOut, the method to change the field will be defined. The following code is an example of the method definition to control the rotation of sacroiliac:

```

1 public void sacroiliacRotate(float angle[ ]){
2     if ( node != null )
3         sacroiliacOrientIn.setValue ( angle ) ;
4 }
5 public float[ ] getsacroiliacRotate () {
6     float[ ] rotAngle = new float[4];
7     rotAngle = sacroiliacOrientOut.getValue() ;
8     return rotAngle ;
9 }

```

Line 2 and 3 indicate the eventIn of sacroiliac joint – sacroiliacOrientIn can send the float array angle which is the parameter of the method to VRML. VRML change the joint sacroiliac to rotate according to the value of angle. Line 6 and 7 shows that Java can get the current rotation angle of sacroiliac in VRML through sacroiliacOrientOut.

3.2.5 View Bound Method Definition

The following example is the methods definition of setBind_h of the view. Line 2 and 3 indicate the eventIn of the view – bindIn_h can send the Boolean value t to VRML. VRML can adjust whether to bind the view or not according to that value passed to it. Java can know if the view is bound or not in VRML by the eventOut of the view -- bindOut_h on line 7 and 8.

```

1 public void setBind_h(boolean t){
2     if ( node != null )
3         bindIn_h.setValue(t) ;
4 }
5 public boolean getBind_h() {
6     boolean bind;
7     bind= bindOut_h.getValue() ;
8     return bind ;
9 }

```

3.2.6 VRML-Java Interface Class Diagram

The name of interface between VRML and Java in the thesis is ControlJoint. It defines all of the EventIn and EventOut. Figure 3.6 is the class diagram of ControlJoint. Only the class name and a subset of variable definitions are shown, without giving the method definitions. There are corresponding methods to change the VRML values of every variable defined in the class.

Figure 3.6 ControlJoint class diagram

3.3 Graphic User Interface

A Java GUI (graphic user interface) was created in order that the user can send commands to the avatar. Figure 3.7 shows the GUI for the user to control the avatar.

There are three parts of this GUI. Part one is for controlling different joint rotations and segment colors, part two is for basic behaviors, and part three is the text editor for entering script language statements.

Figure 3.7 The interface for the user to control the avatar

For the part 1, the upper section of part 1 is for the user to control different joint rotations. The user needs to choose the joint first. He can input the joint name in the joint textfield or choose the joint from the joint list. After choosing the joint, the user can input the rotation angle for X, Y and Z-axis or use scrollbars to control them. There are three

textfields for the user to input the angles of rotations around X, Y and Z-axis. The rotation limit can be different for every joint, but the maximum is 360 degrees and the minimum is -360 degrees. The limit is set for every joint. If the angle that the user inputs is out of the range, the joint won't rotate. After the user inputs the angle, he can press the Enter or Rotate command button to execute the rotation. Another choice to control the joint rotation is by using the scrollbars for the three axes.

The lower section of part 1 is used to control the segment colors. It's usage is similar to the joint control. The user can choose the segment from the list or input the segment name by himself. He can input the color range into the textfields or control the three control bars for three colors (red, green blue). The color range is from 0.0 to 1.0 (float number) for all three colors. User can press Enter or Set color command to change the color after entering the color.

Part 2 is the basic movement (behavior) control panel. The user can choose the movement name from the list or input it by himself. After he presses the Act command, the movement can begin. The user can press the Stop command if he wants the active movement to be stopped. There are some parameters for the user to set for each movement. The first parameter named times will set the numbers of repetitions for the movement. For example, if the user inputs 3 for walk, Nancy will only walk three steps. The second parameter indicates the speed of the movement. For example, if the user inputs 1.2 for run, each step will last 1.2 seconds. The third one is the angle for turning or some movements that need an angle. For example, if the user inputs 60 for turn left, then Nancy will turn 60 degrees left.

There are two functions the user can use on the right panel. One is avatar's position; another one is the user's viewpoint. The avatar's default position is set to 0, 0, 0 for X, Y and Z axis. The user can adjust the avatar's or his own position according to his desire.

The part 3 in the GUI is the highest level of avatar's control — the script language part. The user can input his own little story into the textArea and press Execute to execute it. Nancy will act according to the content of the story and the rules defined by the script language, described in chapters 4 and 5.

3.4 Avatar Animation Design

3.4.1 Design Description

The avatar Nancy is composed of joints and segments. We choose to control the joint rotations in order to control the avatar movement. The avatar can rotate several different joints to form a basic behavior or combine different movements together to complete a complex behavior.

In the thesis, there are three levels to control the avatar's animations : the joint control, the basic behavior control and the script language. Every higher level of control is built upon the lower level. This makes it easier to extend each level and decrease the dependency among the different levels. When we want to build the controller for another avatar, we just need to define the first level because this is the basic level, which accesses the specific avatar's joints in VRML. The other levels of controls can be reused without any changes. It makes the control system more configurable and extensible.

The lowest level controls all of the joints of the avatar like sacroiliac, elbow, wrist or hip and so on. Every joint can rotate in three directions, X, Y and Z. However, every specific joint has it's own constraints on each direction due to the human anatomy. For example, the rotation of the hip is restricted to 90 degrees maximum on the Y direction.

The control code for every joint rotation is in a separate Java class. This makes it easier to add more joint control classes without affecting existing code at the higher level of control. If we want to add more joints to the avatar and to control them, then we need to build more joint rotation classes. The second level of control can use these new classes for the new behaviors without affecting the behaviors that were set up before. In appendix B, some examples of joint rotation classes are given.

The role of the second control level is to combine different joints rotations together at different times to form basic behavior such as walking or running. There are two kinds of basic behaviors, independent and interactive.

Independent behaviors are performed by the avatar alone, without direct interaction with environment objects. Examples of independent behavior are walking or waving a hand. In order to separate the responsibilities related to a behavior, we decided to use three classes working together for each independent behavior. For example, the walking behavior is realized by:

1. WalkData class which stores the data for different joint rotations.
2. Walk class which defines a different joint rotation groups at different time.

3. WalkThread class which controls the timing and coordinates the different rotation groups.

Sample code for these classes is given in appendix C. In the thesis, the data for three behaviors (“walk”, “run” and “jump”) are taken from the VRML Nancy example in [47].

Interactive behaviors are those where the avatar interacts with objects from the virtual environment. For example, “turn on/off the computer” is an interactive behavior realized by two classes. The class IKEngine calculates the data for the avatar’s arm to reach the computer’s button. TurnComputerThread controls the avatar to go to the computer and turn it on/off.

Separating different behaviors into different classes makes it easier for the behaviors to be used. Different behaviors don’t affect each other. New classes can be built if we want to add new behaviors. Adding new behaviors doesn’t affect the behaviors already defined. The new behaviors can be invoked easily from the third level of control.

By combining the basic behaviors together in sequence or in parallel, the user can get a composed behavior. For example, by combining going to the door and turning the doorknob, we can fulfill the behavior of opening the door.

The third level of control is an interpreter for an English-like script language. The user needs to input commands according to the script language’s syntax (grammar). There is a text editor to input the commands. The script language interpreter will translate them into a sequence of commands that will be executed by the avatar. A more detailed explanation is given in Chapter 5.

3.4.2 Class Diagram for Avatar Animation Control

Figure 3.8 is the class diagram of the avatar's animation control system. It includes three main parts: joint control represented by Joint Thread class, behavior movement represented by Behavior Thread class and script language interpreter. Not all joint or behavior classes are given in this class diagram. The Controller class coordinates different classes. This diagram also shows how the three different levels of control for the avatar relate to each other, how they get the corresponding data and how they relate with the objects in VRML.

Figure 3.8 Class diagram of for avatar animation control

Chapter 4 Avatar Animation: Joints

Control and Basic Behaviors

As described before, the avatar control contains three levels of abstraction. The basic level is joint-level control, the second level is basic behaviors/skills, and the third level of control is the script language. Each higher level is built up on the lower level. This hierarchical approach was inspired from the field of robotics [19, 41]. Robotics systems usually have several levels of control, ranging from joint-oriented (the lowest level) to task-oriented (the highest level). This chapter will describe the first and second level of avatar control designed and implemented in the thesis.

4.1 Joints Control

A joint is the basic unit for avatar movement. Our avatar Nancy has 45 joints altogether. The first level to control the avatar is to define different joint rotations. Every joint can rotate along three axis (X, Y and Z). There are pre-set constraints on the rotation degree for each joint.

The joint control level defines the basic control for the joints and segments of the avatar. Through the JAVA interface, the Java controller is connected with the joints and segments of the avatar defined in VRML. A Java thread controls the rotation of each joint. This makes it easier to build second level operations, which need some joints to

rotate or move concurrently in order to realize a complete behavior. The Java threads are a convenient mechanism to implement concurrent movements.

Figure 4.1 shows the class diagram for the control of the sacroiliac joint, that shows how the sacroiliac thread connects with other classes.

Figure 4.1 Class diagram of “sacroiliac” thread

Although not directly related to the control of the avatar’s movement, the user can also change the color of the 15 segments of the avatar at this level. The color change of segment can show where the segment is on the avatar and which joints it connects. There is no restriction for the colors of the segments, which can be any combination of Red, Green and Blue. The user can adjust the three basic colors to change the color of the whole segment in the range from “white” to “black”.

4.2 Basic Behaviors

The joint rotations can be combined together to implement basic behaviors. As mentioned before, there are two kinds of basic behaviors, independent and interactive. For the later, the avatar interacts with objects from the virtual environment. In such cases, the avatar control uses the algorithm of inverse kinematics [13, 17], well known from robotics, in order to reach the objects from the environment the avatar is interacting with.

Java threads are used to achieve concurrent movements. The “sleep” time of a thread is used to control the timing of a movement cycle (such as a walking step). Threads are used in two ways:

1. Single-action thread which controls a basic behavior such as walking, running or waving a hand.
2. Compound-action thread which controls different behaviors performed concurrently, such as “walk and wave a hand” at the same time.

4.2.1 Independent Behavior

An independent behavior is the combination of different joint movements according to a given schedule. An independent behavior gets the data from different data classes for controlling the joint rotations at different times.

4.2.1.1 Walk Class Diagram

Here is an example for a basic independent behavior “walk”. Fig 4.2 represents the class diagram for controlling the avatar’s “walk”.

Figure 4.2 Class diagram of “walk” behavior

4.2.1.2 Walk Sequence Diagram

There are 21 walk stages in the definition of “walk”. A control thread controls all of the walk stages. Figure 4.3 shows a simplified UML sequence diagram to illustrate the “walk” behavior; it just shows two walk stages repeated for five steps.

The user inputs or chooses the “walk” command through the user interface. There are two parameters associated with “walk”: “step” and “cycle”. The parameter “step” is used to define how many walk steps the avatar should perform. The parameter “cycle” sets the time cycle for one walk step. The default values for these two parameters are 3 steps and 1 second, respectively. Suppose the user set “step” to 5 and “cycle” to 1. The controller will initiate a “walk” action thread after receiving the command and the parameters, which will control different walk stages for each step. Each walk stage controls different joints to act concurrently.

The following is the brief description of how all of the walk stages work together. The first walk stage initiates three different control threads concurrently. They are the threads for the left arm, the head and the right leg. All of these three threads can send control events to the ControlJoint class. The ControlJoint class can send all of the events to the VRML at the same time. VRML changes the position or orientation of the left arm, the head and the right leg. After the “walk” action thread creates the first walk stage, it will sleep for 42 milliseconds before it creates the next walk stage. The second walk stage will be initiated after this 42 milliseconds sleep time. The second walk stage initiates other

Figure 4.3 UML sequence diagram illustrating "walk" behavior

NOTE TO USERS

Page(s) not included in the original manuscript and are unavailable from the author or university. The manuscript was scanned as received.

This reproduction is the best copy available.

UMI[®]

two different control threads concurrently. They are the threads for the right arm and the left leg. All of the threads for the right arm and the left leg can send control events to the ControlJoint class. The ControlJoint class will send all of the events to VRML. VRML will change the position or the orientation of the right arm and the left leg. After “walk” action thread creates the second walk stage, it will sleep for 83 milliseconds before next walk stage. The “walk” action will repeat this control until all of the walk stages are finished. Then it will subtract the count for step by one and repeat all of the walk stages again until the count for steps is zero.

The user can set the following parameters for the independent behaviors defined in the thesis:

Table 4.1 Basic movements and parameters

behavior	parameters		
	Cycle (duration time to finish one step)	Times (steps to execute)	Angle (amplitude to turn)
Walk	yes	yes	no
Run	yes	yes	no
Jump	yes	yes	no
Go to door	yes	no	no
Go to yardDoor	yes	no	no
Go to table	yes	no	no
Turn left	no	no	yes
Turn right	no	no	yes
Wave left hand	yes	yes	yes
Wave right hand	yes	yes	yes
Kick left leg	yes	yes	yes
Kick right leg	yes	yes	yes
Turn head left	yes	yes	yes
Turn head right	yes	yes	yes

4.2.2 Interactive Behavior

For the interactive behaviors, the avatar needs to find some objects in the virtual environment and interact with them. Since the relative position of the avatar with respect to different objects changes dynamically, the controller has to obtain the control data at run time. The object in the Virtual Environment and the avatar itself work together to decide how to implement the behavior. For example “open the door”, the avatar could be in different places and stand at different angles when receiving the command. In order to get the avatar to touch the doorknob, the “inverse kinematics” algorithm is used in the thesis.

4.2.2.1 Forward Kinematics and Inverse Kinematics

Forward kinematics involves a transformation from joint angles to 3D positions [13, 17, 24, 38, 42, 39] as it relates to computer graphics. Given some articulated figure, we can describe the figure by relating each joint angle to the limb it is attached to. Given the angles in question, we can straightforwardly calculate the end points of the limbs using coordinate transforms. But if a desired end point or position for an articulated figure is given, how to calculate the angles? This will be the problem of Inverse kinematics.

4.2.2.2 Inverse Kinematics Algorithm

Forward kinematics is simple because a set of joint angles specifies exactly one position. Inverse kinematics, however, is more difficult: most real systems are underconstrained, so for a given goal position, there could be an infinite numbers of solutions (i.e. many

different joint configurations could lead to the same endpoint). The field of robotics has developed many inverse kinematics systems, which have closed-form solutions due to their constraints. The inverse kinematics problem for computer animation is much harder because it must work for arbitrary figures, like human arms or legs.

Several techniques for solving inverse kinematics problems exist. An iterative approach utilizing the Jacobian matrix was chosen in the thesis [39]. In general, the Jacobian matrix for a set $y = f(x)$ of n equations in n variables $x_1, x_2, \dots, x_n$ is defined as follows [39]. The set of equations can be written explicitly as

$$y \equiv \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix}$$

or more explicitly as

$$\begin{cases} y_1 = f_1(x_1, \dots, x_n) \\ \vdots \\ y_n = f_n(x_1, \dots, x_n) \end{cases}$$

The Jacobian matrix, sometimes simply called "the Jacobian" [36] is defined by

$$J(x_1, \dots, x_n) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \dots & \frac{\partial y_n}{\partial x_n} \end{bmatrix}$$

The determinant of J is the Jacobian determinant, and is denoted by:

$$J = \left| \frac{\partial(y_1, \dots, y_n)}{\partial(x_1, \dots, x_n)} \right|$$

Taking the differential

$$dy = y_x dx$$

shows that J is the determinant of the matrix y_x , and therefore gives the ratios of n -dimensional volumes (contents) in y and x ,

$$dy_1 \cdots dy_n = \left| \frac{\partial(y_1, \dots, y_n)}{\partial(x_1, \dots, x_n)} \right| dx_1 \cdots dx_n$$

The Jacobian Matrix is used to calculate the angles for the linked joints by the iterative approach of inverse kinematics. Suppose that we have N joints. For a sufficiently small movement, the change in angle θ and change in position P are linearly related by the Jacobian matrix. We denoted by $\theta = (\theta_1, \theta_2 \dots \theta_N)$ the angle vector, and by $P = (P_1, P_2 \dots P_N)$ the position vector for every joint.

$$dP = J(\theta)d\theta \tag{1}$$

where the Jacobian matrix J is the defined as follows:

$$J = [dP/d\theta_1, dP/d\theta_2 \cdots dP/d\theta_N]$$

The column vectors $dP/d\theta_i$ are the partial derivatives of the end point with respect to the i^{th} joint angle θ_i .

For getting the joint angles, we can rewrite (1) as

$$d\theta = J^{-1}dP$$

This is only valid when J is nonsingular. With articulated figures, however, J is quite often irregularly shaped, and has no inverse. A Moore-Penrose pseudoinverse is a good approximation, and will work for matrices of any shape. The pseudoinverse of a matrix J is the unique matrix denoted J^+ that has the following properties [13, 17]:

$$JJ^+J = J$$

$$J^+JJ^+ = J^+$$

$$(J^+J)^T = J^+J$$

$$(JJ^+)^T = JJ^+$$

Boullion and Odell [13] present an iterative method for calculating the pseudoinverse of arbitrary matrices. Thus, the heart of the inverse kinematics iterative solution is the equation.

$$d\theta = J^+dP$$

Choose dP to be a small vector in the direction from the end point to the goal, calculate the pseudoinverse of the Jacobian, and then calculate the change in θ . Increase θ by $d\theta$ and repeat the process. This technique will eventually converge on the goal point.

The Jacobian method, with the nice property of convergence, is not necessarily optimal. Other methods exist which minimize the energy spent (mainly used in robotics), as well as methods which minimize the length of the path traveled. For this thesis, the Jacobian works nicely because it looks natural when the avatar moves an arm to reach a certain object.

Korein and Badler [24] present an optimized approach to inverse kinematics. The technique is hierarchical. Instead of calculating the change in all the angles at once, the base segment is moved so that the goal point is in the "reach space" of the rest of the segments. Then, the base segment is fixed, and the next segment is moved so that the goal point is in the reach space of the rest of the segments. The process is repeated until the final segment moves to the goal point. In the thesis, this optimized approach is used to calculate the angles for the joints and to control the joints when moving toward the goal.

4.2.2.3 Interactive Behavior Implementation

An example of how to implement an interactive behavior is given in this section. Assume that the user enters the command "open the door" through the user interface. Figure 4.4 is the sequence diagram for opening the door.

Figure 4.4 UML sequence diagram illustrating “open door” behavior

The controller will send appropriate events to the ControlJoint class to ask for the positions of the doorknob and of the avatar's right arm (which are contained in the VRML part of the program). The ControlJoint class will send events to VRML to get these positions and will send them back to the Controller. The Inverse Kinematics approach is used by the controller to calculate the angles of joints on the right arm according to the doorknob position and the end point (hand) position. After getting the angles for the right arm joints, the Controller will create a "Touch doorknob" thread. This thread initializes the threads for the right shoulder, the right elbow and the right wrist at the same time and sends the angles obtained from the Controller to the joint threads as parameters. These threads send their corresponding events to the ControlJoint class. The ControlJoint class will control the rotation of the right shoulder, the right elbow and the right wrist. After the "Touch doorknob" thread creates all of the arm joint threads, will sleep for 20 milliseconds to wait for the rotations and translations in the virtual environment to be performed. Then this thread sends appropriate events to ControlJoint to open the door. ControlJoint informs the correspondent VRML objects to implement the action. The "Touch doorknob" thread will sleep for another 20 milliseconds to wait for the door to open. Then it will initialize the "walk" thread for the avatar to enter the virtual room. After the avatar walks into the room, the scene will be changed from the virtual house to the virtual room. The "Touch doorknob" thread sends events to ControlJoint to change the scene.

The only parameter for this behavior that the user can choose is the time cycle, which decides how fast the avatar walks to the object (i.e., door) and interacts with that object. Table 4.2 lists interactive behaviors and their parameters.

Table 4.2 Interactive behaviors and parameters

behavior	parameters		
	Cycle (duration time to finish one step)	Times (steps to execute)	Angle (amplitude to turn)
Open the door	yes	no	no
Turn on/off computer	yes	no	no
Turn on/off light	yes	no	no

4.3 Combined Behaviors

In order to imitate better a real human being, an avatar is required to have the ability to combine different simple behaviors. In the thesis, we have considered only two types of behaviors composition: sequential and in parallel.

4.3.1 Sequential Composition

This is a very simple composition, as it executes two basic behaviors in sequence: one behavior can start only after the other one is finished. For example, “go to the door” is a sequential composition, as it has two basic behaviors in sequence. Figure 4.5 is the sequence diagram for this scenario.

The user inputs the “go to the door” command through the user interface. After the Controller receives this command, it will send events to the ControlJoint class to get the door’s position and the avatar’s position. The ControlJoint class will contact the corresponding VRML objects (here is the door and the avatar) and get the position values. Then the ControlJoint class sends these position values to the Controller. In the thesis, the trajectory for the avatar is always a straight line, as we have not implemented

Figure 4.5 UML sequence diagram illustrating “go to door” behavior

an obstacle avoidance algorithm. The Controller needs to calculate the angle between the avatar and the door, as well as how many steps are needed to walk from the avatar’s current position to the door position. The steps number is related to the length of one step. In this thesis, the parameter is not a variable that the user can change from the user interface, but it can be an option in the future. After the calculation is done, the Controller

will create a “go to the door” thread. Which in turn will create a “turn left” or “turn right” thread according to the angles it gets from the Controller. The “turn left” or “turn right” thread will send the events to the ControlJoint class. The ControlJoint class will make the avatar turn into the right direction with a specific angle that the controller has calculated. The Controller will sleep for 10 milliseconds in order to wait for the avatar to implement the commands. The Controller creates a “walk” thread after the turn is done. The “walk” thread will send events to the ControlJoint class with the numbers of steps it gets from the Controller. The ControlJoint class will control the avatar to complete the behavior.

4.3.2 Parallel Composition

A parallel composition is a combination of the basic behaviors executed concurrently. The basic behaviors are classified into different groups depending on whether they can be combined or not. The basic skills in the same group compete with each other because they use the same joints and they cannot be executed at the same time. The behaviors in different groups can be combined together. For example, the user can ask the avatar to wave its left hand while it is walking. But the avatar cannot walk and run at the same time. In order to combine different behaviors, the group of each behavior is indicated by a flag. Besides the group flag, body part flags are also set to control different body parts involved in the movements (the head, the left arm, the right arm, the left leg and the right leg). For example, the avatar can turn the head toward to an object while walking. This means that the head is controlled by the command “turn head” whereas the rest of the body is controlled by the command “walk”. These two behaviors can be combined together smoothly.

The sequence diagram of figure 4.6 illustrates the parallel composition of “walk” and “wave left hand”.

Figure 4.6 UML sequence diagram illustrating “walk” and “wave” behaviors

In this sequence diagram: the avatar waves the left hand three times after he begins walking. The “walk” behavior keeps on after the hand waving is finished.

The user inputs first “walk” for five steps. After the avatar walks for some time, the user inputs “wave left hand” three times. After receiving “wave left hand”, the Controller checks and decides that these two behaviors are in two different groups and can be combined concurrently. Then the Controller sets “left arm” flag for walking to be false. This command will stop the left arm’s movement for walking, but continue the movements of the rest for the avatar. The Controller will create a “wave left hand” thread after it informs the walk action thread to stop the left arm movements. The “Wave left hand” thread controls the left arm to wave three times. After finishing the waving, the corresponding thread will send an event to the Controller to inform it that the action is done. The Controller informs then the “walk” action thread to resume the left arm’s movement for walking. The Walk action thread will send new events to ControlJoint class, which controls the avatar in the virtual environment to keep on walking until the number of steps is completed.

4.3.3 Class Diagram of Combined Behavior – Open Door

Figure 4.7 is the class diagram for a combined behavior namely “open door”. It is a sequential combination of independent behaviors (“go to door” and “side walk”) and interactive behavior (“open door”).

The behaviors are all extended from SetParam class. The OpenDoor class controls several other classes. This behavior controls Turn and Walk classes to go to the door first.

If the avatar is too far away from the door, the SideWalk class can control the avatar to adjust its position. Then the IKEngine calculates the data for the arm to reach the doorknob and open the door. After the door is open, OpenDoor class can control the ChangeScene and ChangeViewpoint classes.

Figure 4.7 Class diagram of "open door" thread

Chapter 5 Avatar Animation: Script Language

The highest level of avatar control uses a script language by which the user can input a little story or a sequence of commands in batch, and the avatar can act the story afterwards. The script language is a restricted English-like language described by a context-free grammar. The language has its own interpreter to recognize and execute the commands. This chapter explains the script language and its interpreter.

5.1 Grammar

Our intention was to design an English-like script language that would allow the user to express easily a short story or a set of commands to be enacted by the avatar. However, it is well known that unconstrained natural language are difficult to parse, so we decided to settle for a script language defined by a context-free grammar. This type of grammar is largely used for the definition of today's programming languages. Efficient parsers can be built for context-free grammars.

The script language defined in the thesis is constrained both by:

1. The size of its vocabulary (terminal symbols) that depends on the set of basic behaviors implemented in the second level of control.
2. The sentence syntax defined by the grammar.

According to [19], a context-free grammar G is a 4-tuple (V, Σ, R, S) , where:

V is a finite set of nonterminal symbols.

Σ is a finite set of terminal symbols.

R is a finite set of rules that define each nonterminal as a string of symbols (either terminal or nonterminal)

$S \in V$ is the set of start symbols.

For the script language grammar defined in the thesis, the definitions of V , Σ and S are the following. The set of rules R is given in Table 5.1.

$V = \{ \langle \text{story} \rangle, \langle \text{sentence} \rangle, \langle \text{composed sentence} \rangle, \langle \text{simple sentence} \rangle, \langle \text{separator} \rangle, \langle \text{subject} \rangle, \langle \text{predicate} \rangle, \langle \text{direct object} \rangle, \langle \text{noun} \rangle, \langle \text{indirect object} \rangle \}$

$\Sigma = \{ \langle \text{Nancy} \rangle, \langle \text{she} \rangle, \langle \text{stand} \rangle, \langle \text{wait} \rangle, \langle \text{run} \rangle, \langle \text{jump} \rangle, \langle \text{open} \rangle, \langle \text{wave} \rangle, \langle \text{kick} \rangle, \langle \text{walk} \rangle, \langle \text{go} \rangle, \langle \text{turn} \rangle, \langle \text{left} \rangle, \langle \text{right} \rangle, \langle \text{door} \rangle, \langle \text{arm} \rangle, \langle \text{leg} \rangle, \langle \text{head} \rangle, \langle \text{to} \rangle, \langle \text{on} \rangle, \langle \text{yardDoor} \rangle, \langle \text{table} \rangle, \langle \text{window} \rangle, \langle \text{tree} \rangle, \langle \text{chair} \rangle, \langle \text{light} \rangle, \langle \text{computer} \rangle, \langle \text{fast} \rangle, \langle \text{faster} \rangle, \langle \text{slow} \rangle, \langle \text{slower} \rangle \}$

$S = \langle \text{story} \rangle$

How can a string of terminal symbols be generated from the rules? We can begin with the start symbol and choose a rule to apply. If the resulting string contains nonterminals, we

Table 5.1 Grammar for the script language: Set of rules R

Left-hand side		Right-hand side
<story>	::=	<sentence> <sentence><story>
<sentence>	::=	<simple sentence> "." <composed sentence> "."
<composed sentence>	::=	<simple sentence> <separator> <composed sentence> <simple sentence>
<separator>	::=	"," <conjunction>
<conjunction>	::=	"and"
<simple sentence>	::=	[<subject>] <predicate>
<predicate>	::=	<verb> [<adverb>] * [<direct object>] [<indirect object>]
<direct object>	::=	[article] [<adjective>] <noun>
<noun>	::=	[article] <noun>
<indirect object>	::=	[article] [<adjective>] * <noun> <preposition> [article] [<adjective>] * <noun>
<subject>	::=	"Nancy" "she"
<verb>	::=	"stand" "wait" "run" "jump" "open" "wave" "kick" "walk" "go" "turn"
<adjective>	::=	"left" "right"
<noun>	::=	"door" "arm" "leg" "head" "yardDoor" "table" "window" "tree" "chair" "light" "computer"
<adverb>	::=	"fast" "faster" "slow" "slower"
<article>	::=	"a" "the"
<preposition>	::=	"to" "on"

choose rules to apply to each of those nonterminals. Each time we apply a rule to a nonterminal, we replace that nonterminal in the current string by the right-hand side of the rule applied. We continue making replacements according to the rules until, the current string contains no nonterminals finally. When the current string then consists of terminal symbols only, it is finished [19].

The following convention was adopted for the composition of basic behaviors, each one represented by a simple sentence:

1. Sequential composition: behaviors separated by "." (i.e. separate sentences)
2. Parallel composition: behaviors separated by "," or "and" (i.e. simple sentences in a composed sentence)

This approach aims for maximum parallelism between the behaviors expressed in the same composed sentence.

For example, the parse tree for the following sentence is given in Figure 5.1:

"Nancy turns to the yardDoor. She walks fast, waves the left hand. "

Figure 5.1 A parse tree for an input

5.2 Script Language Interpreter

The role of the interpreter is to parse the input string in order to identify the sentences corresponding to different basic behaviors, then to invoke the appropriate methods and threads implemented in the second control level for executing them.

A typical top-down parser without backtracking was built for the script language grammar by using recursive methods for all the rules containing nonterminals in the right-hand side given in Table 5.1. If such a rule has multiple alternatives, we look ahead in the input string until we can decide which alternative is the right one for that case. Thus, backtracking is avoided.

The rules that contain only terminals in the right-hand side (such as those for <noun>, <verb>, etc) have been implemented with the help of a linked table shown in Table 5.2. The reasons for this design choice are multiple:

1. Accelerate the search among multiple alternatives.
2. Make the parser easier to extend when new terminals are added to the grammar.
3. Simplify the selections of the right sentence structure (some verbs can be followed by a direct object, others by an indirect object, etc).
4. Point to the method or thread that executes the respective actions.

When designing the script language, we decided to make it less like a programming language and more like English. Therefore, no quantitative parameters for different

behaviors are expressed in the script language (such as exact duration of a step, number of steps or angles for turning, etc). In order to allow the user to control to some degree these parameters, we took the following approach:

1. If no adverb is given in a sentence, use pre-defined default values. For example "Nancy walks" will be translated into a walk command that corresponds to a "normal" walk (The actual parameter values are: number of steps = 3; step duration = 1s).
2. If the user wants to change some of the movement characteristics, he/she can use adverbs such as "fast" or "slow" to increase or decrease the speed of the movement to some pre-set values. For example, "Nancy walks fast" will be translated into a rapid walk (the step duration parameter is now 0.9s).
3. Another possibility is to use comparative adverbs, such as "faster", to change the speed by a pre-defined increment. By repeating the command "walk faster" several times, the avatar walking speed can be raised up to the level desired by the user.

This approach of using adverbs to qualify the characteristics of the movements instead of giving quantitative values has the advantage of being simpler for a human user. However, it has the disadvantage of being less accurate than using numerical parameters. More experiments are necessary to judge the effectiveness of this approach, and maybe to refine it by introducing more pre-defined levels (such as "very fast", "fast", "slow", "very slow") for a better accuracy.

The Java feature used to implement the semantic table 5.2 is a Hashtable. The advantages of using such a table are the following:

1. Disallows sentences that are syntactically correct but don't make sense.
2. Hashtable accelerates key-based searches.
3. It is easy to add new key-and-value pairs to a Hashtable, so that the semantic table can be easily extended.

Table 5.2 Semantic Table

Sentence						execution	
subject	verb	direct object		indirect object			
		adj.	noun.	prep	noun		
Nancy She	stand(s)					Stand thread	
	wait(s)					Wait thread	
	run(s)					Run thread	
	jump(s)					Jump thread	
	open(s)		door			Open door thread	
	wave(s)	left	arm			Wave left arm thread	
		right				Wave right arm thread	
	kick(s)	left	leg			Kick left leg thread	
		right				Kick right leg thread	
	walk(s)					Walk thread	
	go(es)			to	door	Go to door thread	
					yardDoor	Go to yard door thread	
					table	Go to table thread	
	turn(s)			to	door	Turn to door thread	
					yardDoor	Turn to yardDoor thread	
					window	Turn to window thread	
					tree	Turn to tree thread	
					chair	Turn to chair thread	
				on	light	Turn on light thread	
					computer	Turn on computer thread	
			head	to	door	Turn head to door thread	
					yardDoor	Turn head to yardDoor thread	
					window	Turn head to window thread	
					light	Turn head to light thread	
			left				Turn left thread
			right				Turn right thread

One of the design goals was to make the interpreter easy to modify when new avatar behaviors will be defined in the second level of control, or new objects will be introduced in the virtual environment. This means that the set Σ of terminals will be extended with new verbs, nouns, etc. If the new behaviors can be expressed with the same sentence syntax as defined by the grammar G , then the code of the interpreter does not need to be changed, only the linked table. If, however, the syntax of the script language needs to be extended with new sentence structures, then the code of the interpreter will have to be modified, as well.

The sample sentences of the script language generated by grammar G that conform to the semantic table 5.2 are listed in Appendix E.

5.3 Example

This section gives an example of how to use the script language for avatar animation.

The user can input the following story in the appropriate field at the bottom of the user interface:

“Nancy walks. She turns to the tree. She runs. Nancy goes faster to the yardDoor, turns the head to the door and waves the right arm. She turns right. Walks longer. She goes to the door, turns the head to the window and waves the left arm. Nancy opens the door. Turns the head to the light and turns on the light.”

After pressing the “execute” command button, the whole text commands will be executed.

Figure 5.2 is the screenshot that Nancy is executing the first sentence. “*Nancy walks*”, and Figure 5.3 is the screenshot that Nancy is executing the command “*turns on the light*”.

Figure 5.2 Nancy is walking

Figure 5.3 Nancy is turning on the light

Chapter 6 Conclusions and Future Work

6.1 Conclusions

In this thesis, we described the design and implementations of a 3-level control system for avatar animation in a virtual environment. Here is a summary of our work.

1. A virtual environment is created with VRML, which contains many useful features for such a task. VRML files are standard text files that can be interpreted by browsers. The virtual environment built with VRML can be uploaded to the Internet, which increases its availability.
2. The Humanoid Standard is used to create the avatar Nancy. The advantage to use this standard is that the control for Nancy can be easily applied to any other avatars built with the same standard.
3. Java is used as the developing language for the controller. As a programming language aimed for the Internet, Java cooperated with VRML seamlessly. Due to the advantage of its pure object-oriented nature, Java increased the efficiency of the development, and decreased the difficulties of implementing concurrency in the avatar's behavior.
4. A three-level control system was developed for the avatar animation. The lower level controls the joint rotations; the middle level controls the basic behaviors and their composition, and the higher level implements and executes a script

language. This hierarchical approach, inspired from the field of robotics, has the following advantages:

- decoupling between different control levels
- flexibility and extensibility
- user friendliness, as the lower levels are more or less hidden from the user.

The actual design and implementation has a number of limitations that can be addressed in future work, and are discussed in the following sections.

6.2 Future Work

There are several directions for future work that were identified during the thesis research.

First of all, more functions and behaviors can be added to the avatar. More artificial intelligence can be used to enable the avatar to decide in impromptu situations without the interference of human beings. Also, intelligence can be added to other objects in the virtual environment. When the avatar wants to interact with such objects, it can interact with them according to their intelligent properties.

Another big extension is to introduce more avatars into the virtual environment. Collaborative interactions among avatars can be developed, and collision avoidance can be introduced. Speech and text can be also added to enrich the whole activities in the virtual environment.

Different behaviors can be integrated into models or plug-ins that can be used in other software applications, such as computer games, educational applications, weather forecast or news report.

Neural network can be introduced in the second level of control to learn how to imitate a given person's behavior. This will allow to give different "personalities" to the avatars.

All these extensions will have an impact on the script language for avatar animation. The language will become more complex, and maybe closer to English than it is now.

References

1. F.Azuola, N. Badler, P.-H. Ho, I. Kakadiaris, D. Metaxas, and B. Ting. "Building anthropometry-based virtual human models," *In proc. IMAGE VII Conf.*, 1994.
2. F.Azuola, N. I. Badler, P.H. Ho, I. Kakadiaris, D. Metaxas and B. Ting. "Building anthropometry-based virtual human models," *Proc. IMAGE VII Conf.*, Tucson, AZ, June, 1994.
3. C. Babski, D. Thalmann, "Define Virtual Humans On The Web," *Software Focus*, Wiley, Vol. 1, No1, pages 6-14.
4. N. Badler, C. Phillips, and B. Webber. "Simulating Humans," *Computer Graphics Animation and Control*. Oxford University Press, New York, NY, 1993.
5. N. Badler, M. Hollick and J. Granieri, "Real-time control of a virtual human using minimal sensors," *Presence*, 2(1) pages 82-86, 1993.
6. N. Badler, "Animation 2000++", *IEEE Computer Graphics and Applications*, January/February 2000, pages 28-29.
7. J. Bates, "The role of emotion in believable agents," *Comm. Of the ACM*, 37(7): pages 122-125, 1994.

8. J. Bates, A. Loyall, W. Reilly, "Integrating Reactivity, Goals and Emotions in a Board Agent," *Proceedings of the 14th Annual Conference of the Cognitive Science Society*, Indiana, July 1992.
9. R. Bindiganavale, W. Schuler, J. M. Allbeck, N. I. Badler, A. K. Joshi, and M. Palmer, "Dynamically Altering Agent Behaviors Using Natural Language Instructions," *Autonomous Agents 2000*, pages 293-300.
10. C. Bordeaux, R. Boulic, and D. Thalmann, "An Efficient and Flexible Perception Pipeline for Autonomous Agents", *Proceedings of Eurographics '99*, Milano, Italy, pages 23-30.
11. R. Boulic, N. Magnenat-Thalmann, and D. Thalmann, "A Global Human Walking Model with Real Time Kinematic Personification", *The Visual Computer*, 6, pages 344-358, 1990.
12. N. Badler, M. Hollick and J. Granieri, "Real-time control of a virtual human using minimal sensors," *Presence*, 2(1): pages 82-86, 1993.
13. T. Boullion and P. Odell, "Generalized Inverse Matrices". New York. : Wiley-Interscience, 1971.
14. M. Cavazza , R. Earnshaw, N. Magnenat-Thalmann, and D. Thalmann, "Motion control of virtual humans," *IEEE Computer Graphics Application* vol. 18, pages 24-31, 1998.

15. Cheney, Ichnowski, Forsyth, "Efficient Dynamics Modeling for VRML and Java," in *Proceedings of VRML 98* Monterey CA Feb 1998.
16. D. Chi, N. Badler, J. Clarke, and B. Webber, "Casualty modeling for real-time medical training," *Presence*, 5(4): pages 359-366, 1995.
17. R. Cline, "Elements of the Theory of Generalized Inverses for Matrices". Newton, Mass.: EDC/UMAP, 1979.
18. H. Cruse and U. Steinkuhler, "Solution of the direct and inverse kinematics problems by a common algorithm based on the mean of multiple computations," *Biol. Cybern.* Vol. 69, pages 345-351, 1993.
19. K.S.Fu, R.C.Gonzales, C.S.G.Lee, "Robotics: Control, Sensing, Vision, and Intelligence", McGraw Hill, 1987.
20. J. Granieri, J. Crabtree and N. Badler, "Off-line production and real-time playback of human figure motion for 3D virtual environments", In *VARIS Conf. IEEE Press*, 1995.
21. J. K. Hodgins, "Three-Dimensional Human Running," *Proceedings of the IEEE Conference on Robotics and Automation*. 1996.
22. M. Kallmann and D. Thalmann, "A Behavioral Interface to Simulate Agent-Object Interactions in Real-Time" *Proceedings of Computer Animation 99, IEEE Computer Society Press*, 1999, Geneva, pages 138-146.

23. M. Kallmann, J.-S. Monzani, A. Caicedo and D. Thalmann, "ACE: A Platform for the Real Time Simulation of Virtual Human Agents", *EGCAS'2000 - 11th Eurographics Workshop on Animation and Simulation*, 2000, Interlaken, Switzerland, August, 2000.
24. J. Korein, and N. Badler, "Techniques for generating the goal directed motion of articulated structures." *IEEE Computer Graphics and Applications*, 2(9), pages 71-81.
25. C. Marrin, Proposal for a VRML 2.0 Informative Annex, External Authoring Interface Reference,

"http://www.vrml.org/WorkingGroups/vrml-eai/ExternalInterface.html#2.2".
26. D. R. Nadeau, Publishing 3D Content with VRML,

"http://www.sdsc.edu/~nadeau/Talks/NASA_EOSDIS/whatis2.htm".
27. D. R. Nadeau, Visualization in VRML,

"http://www.sdsc.edu/~nadeau/Talks/WorldMovers/".
28. T. Noma, L.W. Zhao and N. I. Badler, "Design of a Virtual Human Presenter," *IEEE Computer Graphics and Applications*. 0272-1716/00 2000 IEEE.
29. K. Perlin, A. Goldberg "Improv: A System for scripting interactive Actors in Virtual Worlds," *Computer Graphics*, Vol. 29 No. 3., 1996.

30. K. Perlin, "Real Time Responsive Animation with Personality," *IEEE Transactions on visualization and computer graphics*. Vol. I. No. 1. March 1995.
31. K. Perlin, "An image Synthesizer. Computer Graphics," *SIGGRAPH '85 Proceedings*, 19(3): pages 287-293, 1985.
32. S. Ressler, A. Godil, Q. Wang, G. Seidman, "A VRML Integration Methodology for Manufacturing Applications," *Proceedings of VRML99 Fourth Symposium on the Virtual Reality Modeling Language*, Paderborn Germany, Feb. 1999.
33. C. Rigotti, P. Cerveri, G. Andreoni, A. Pedotti and G. Ferrigno "Modeling and Driving a Reduced Human Mannequin through Motion Captured Data: A Neural Network Approach". *IEEE Transactions on systems. Man and Cybernetics-part A: systems and humans*. Vol. 31, No 3, May 2001.
34. T. Sederberg and S. Parry. "Free-form deformation of solid geometric models," *ACM Computer Graphics*, 20(4): pages 151-160, 1986.
35. J. Shi, T. J. Smith, J. Granieri, and N. Badler. 1999. "Smart avatars in JackMOO". In *Proceedings of IEEE Virtual Reality*, 156--163. Los Alamitos: IEEE Computer Society.
36. Simon, P. Carl and L. Blume (1994) "Mathematics for Economists". W.W.Norton and Co., New York, NY.

37. Xiaojun Shen, Saeid Nourian, Isabelle Hertanto, Nicolas D. Georganas: vCOM: virtual commerce in a collaborative 3D world. *ACM Multimedia 2001*: 605-606
38. A. Watt, and M. Watt "Advanced Animation and Rendering Techniques: Theory and Practice". New York: *ACM Press*, 1992.
39. E. Weisstein's world of Mathematics: Algebra, Linear Algebra, Determinants, Jacobian "<http://mathworld.wolfram.com/Jacobian.html>".
40. W. L. Wooten, J. K. Hodgins, "Animation of Human Diving," *Computer Graphics Forum* 15(1) pages 3-13. 1996.
41. T. Yoshikawa, "Manipulatability and redundancy of robotic mechanism," in *Proc. IEEE Conf. Robot. Automat. Silver Spring, Md*, pages 1004-1009.
42. J. Zhao and N. Badler, "Inverse kinematics positioning using nonlinear programming for highly articulated figures." *ACM Transactions on Graphics*, 13(4), pages 313-336.
43. Blaxxun VRML browser, "<http://www.blaxxun.com>".
44. Corona VRML browser, "<http://www.parallelgraphics.com>".
45. Cosmo Player support, "<http://www.cai.com/cosmo/>".
46. Direct 3D/DirectX Graphics Game Programming, "[Http://www.shaderx.com/direct3d.net](http://www.shaderx.com/direct3d.net)".
47. h-anim examples,

- [“http://www.ballreich.net/vrml/h-anim/h-anim-examples.html”](http://www.ballreich.net/vrml/h-anim/h-anim-examples.html).
48. High Performance 2D/3D Graphics, [“http://www.opengl.org”](http://www.opengl.org).
 49. Humanoid Animation Working Group, [“http://www.h-anim.org”](http://www.h-anim.org).
 50. Java 2 Platform, Standard Edition: [“http://java.sun.com/j2se/”](http://java.sun.com/j2se/).
 51. Java 3D API, [“http://java.sun.com/products/java-media/3D”](http://java.sun.com/products/java-media/3D).
 52. Mac’s quickdraw3D [“http://www.apple.com/quicktime/”](http://www.apple.com/quicktime/).
 53. Sydney Gamers League, Network Gaming on the North Shore of Sydney,
[“http://www.sgl.org.au”](http://www.sgl.org.au).
 54. VRML Introduction:

[“http://www.web3d.org/Specifications/VRML97/part1/introduction.html”](http://www.web3d.org/Specifications/VRML97/part1/introduction.html).
 55. The VRML Specification for Field and Event Reference,
[“http://www.vrml.org/VRML2.0/FINAL/spec/part1/fieldsRef.html”](http://www.vrml.org/VRML2.0/FINAL/spec/part1/fieldsRef.html).
 56. VRWaveVRML browser, [“http://www.iicm.edu/vrwave”](http://www.iicm.edu/vrwave).
 57. XGL File Format Specification, [“http://www.xglspec.org/”](http://www.xglspec.org/).

Appendix A The Proto definition of the Humanoid Standard

```

PROTO Joint [
  exposedField SFVec3f   center      0 0 0
  exposedField MFNode    children    []
  exposedField MFFloat   llimit      []
  exposedField SFRotation limitOrientation 0 0 1 0
  exposedField SFString  name        ""
  exposedField SFRotation rotation    0 0 1 0
  exposedField SFVec3f   scale        1 1 1
  exposedField SFRotation scaleOrientation 0 0 1 0
  exposedField MFFloat   stiffness    [0 0 0]
  exposedField SFVec3f   translation  0 0 0
  exposedField MFFloat   ulimit       []
]
{
  Transform {
    center      IS center
    children    IS children
    rotation    IS rotation
    scale       IS scale
    scaleOrientation IS scaleOrientation
    translation  IS translation
  }
}
PROTO Segment [
  field      SFVec3f   bboxCenter    0 0 0
  field      SFVec3f   bboxSize      -1 -1 -1
  exposedField SFVec3f   centerOfMass 0 0 0
  exposedField MFNode    children    []
  exposedField SFNode    coord      NULL
  exposedField MFNode    displacers  []
  exposedField SFFloat   mass        0
  exposedField MFFloat   momentsOfInertia [0 0 0 0 0 0 0 0 0]
  exposedField SFString  name        ""
  eventIn    MFNode    addChildren
  eventIn    MFNode    removeChildren
]

```

```

{
  Group {
    addChildren      IS addChildren
    bboxCenter      IS bboxCenter
    bboxSize        IS bboxSize
    children         IS children
    removeChildren  IS removeChildren
  }
}

PROTO Humanoid [
  field      SFVec3f  bboxCenter      0 0 0
  field      SFVec3f  bboxSize        -1 -1 -1
  exposedField SFVec3f  center         0 0 0
  exposedField MFNode  humanoidBody   []
  exposedField MFString info          []
  exposedField MFNode  joints         []
  exposedField SFString name          ""
  exposedField SFRotation rotation    0 0 1 0
  exposedField SFVec3f  scale          1 1 1
  exposedField SFRotation scaleOrientation 0 0 1 0
  exposedField MFNode  segments       []
  exposedField MFNode  sites          []
  exposedField SFVec3f  translation    0 0 0
  exposedField SFString version        "1.1"
  exposedField MFNode  viewpoints     []
]

{
  Transform {
    bboxCenter      IS bboxCenter
    bboxSize        IS bboxSize
    center          IS center
    rotation        IS rotation
    scale           IS scale
    scaleOrientation IS scaleOrientation
    translation     IS translation
    children [
      Group {
        children IS viewpoints
      }
      Group {
        children IS humanoidBody
      }
    ]
  }
}

```



```

    }
  ]
}
}
DEF hanim_All Transform{
  translation  -20.52 -0.792 12.0
  rotation      0 1 0 0
  scale         1 1 1
  children[
    DEF Humanoid Humanoid{
      humanoidBody [
        DEF hanim_HumanoidRoot Joint {
          name      "HumanoidRoot"
          translation 0 0 0
          rotation   0 1 0 0
          center     -0.00405 0.855 -0.000113
          children [
            DEF hanim_sacroiliac Joint {
              name      "sacroiliac"
              rotation   0 0 1 0
              center     0 1.01 -0.0204
              children [ DEF hanim_pelvis_s Segment {
                name      "pelvis"
                children Shape {
                  appearance Appearance {
                    material DEF hanim_pelvis Material {
                      ambientIntensity 0.25
                      diffuseColor 0.1 0.3 0.3
                    }
                  }
                }
              }
            }
          ]
        }
      ]
      geometry IndexedFaceSet {
        coord Coordinate {
          point[.....]
          creaseAngle 1.14
        }
      }
    }
  ]
}
DEF hanim_l_hip Joint {
  :
  :

```

Appendix B Joint rotation class examples

Joint Skullbase Rotation Definition Class

```
import java.util.*;
class SkullbaseT extends Thread
{
    ControlJoint cj;
    float t[] = new float[4];
    public SkullbaseT(ControlJoint a, float f[])
    {
        cj = a;
        t = f;
    }
    public void run()
    {
        cj.skullbaseRotate(t);
    }
}
```

Joint Sacroiliac Rotation Definition Class

```
import java.util.*;
class SacroiliacT extends Thread
{
    ControlJoint cj;
    float t[] = new float[4];
    public SacroiliacT(ControlJoint a, float f[])
    {
        cj = a;
        t = f;
    }
    public void run()
    {
        cj.sacroiliacRotate(t);
    }
}
```

Appendix C Walk Behavior Example

Class For Pre-defined Walk Data

```
public class WalkData
{
    float[ ][ ] r_ankle = new float[10][4];
    float[ ][ ] r_knee = new float[9][4];
    :
    :
    float[ ][ ] humanRoot = new float[18][4];

    public WalkData()
    {
        /**
         *data for r_ankle
         */
        r_ankle[0][0] = 0f;
        r_ankle[0][1] = 0f;
        r_ankle[0][2] = 1f;
        r_ankle[0][3] = 0f;

        r_ankle[1][0] = -1f;
        r_ankle[1][1] = 0f;
        r_ankle[1][2] = 0f;
        r_ankle[1][3] = 0.3533f;
        :
        :

        /**
         * data for l_shoulder
         */
        l_shoulder[0][0] = 1f;
        l_shoulder[0][1] = 0f;
        l_shoulder[0][2] = 0f;
        l_shoulder[0][3] = 0.1189f;
        :
        :
```

```

/**
 * data for vl1
 */
vl1[0][0] = 0f;
vl1[0][1] = 1f;
vl1[0][2] = 0f;
vl1[0][3] = 0.0826f;

vl1[1][0] = -0.01972f;
vl1[1][1] = -0.5974f;
vl1[1][2] = 0.8017f;
vl1[1][3] = 0.08231f;

vl1[2][0] = 0.009296f;
vl1[2][1] = -0.9648f;
vl1[2][2] = 0.2627f;
vl1[2][3] = 0.1734f;

:
:
/**
 * data for whole body translation
 */
humanRoot[0][0] = 0f;
humanRoot[0][1] = -0.00928f;
humanRoot[0][2] = 0f;

humanRoot[1][0] = 0f;
humanRoot[1][1] = -0.003858f;
humanRoot[1][2] = 0f;

humanRoot[2][0] = 0f;
humanRoot[2][1] = -0.008847f;
humanRoot[2][2] = 0f;

:
:
}
}

```

Class For Walk Definition

```

import java.util.*;

class Walk
{
    ControlJoint wcj;
    WalkData walkData;
    boolean t;
    public Walk(ControlJoint a)
    {
        wcj = a;
        walkData = new WalkData();
        t = Position.houseChosen;
    }
    /**
     * from here, they are inner methods that define the walk animation. For each stage, *
     * it has five parts to make this movement.
     */
    public void walk1HeadApp()
    {
        new SkullbaseT(wcj, walkData.skullbase[0]).start();
    }
    public void walk1LArmApp()
    {
        new LwristT(wcj, walkData.l_wrist[0]).start();
        new RelbowT(wcj, walkData.l_elbow[0]).start();
        new LshoulderT(wcj, walkData.l_shoulder[0]).start();
    }
    public void walk1RArmApp()
    {
        new RwristT(wcj, walkData.r_wrist[0]).start();
        new RelbowT(wcj, walkData.r_elbow[0]).start();
        new RshoulderT(wcj, walkData.r_shoulder[0]).start();
    }
    public void walk1LLegApp()
    {
        new LankleT(wcj, walkData.l_ankle[0]).start();
        new LkneeT(wcj, walkData.l_knee[0]).start();
        new LhipT(wcj, walkData.l_hip[0]).start();
    }
}

```

```

public void walk1RLegApp()
{
    new RankleT(wcj, walkData.l_ankle[0]).start();
    new RkneeT(wcj, walkData.r_knee[0]).start();
    new RhipT(wcj, walkData.r_hip[0]).start();
}
public void walk1App()
{
    if(t)
    {
        Position.position_h[0] = walkData.humanRoot[0][0]+Position.position_h[0];
        Position.position_h[1] = walkData.humanRoot[0][1];
        Position.position_h[2] = walkData.humanRoot[0][2]+Position.position_h[2];
        new RootT(wcj, Position.position_h).start();
    }
    else
    {
        Position.position_r[0] = walkData.humanRoot[0][0]+Position.position_r[0];
        Position.position_r[1] = walkData.humanRoot[0][1];
        Position.position_r[2] = walkData.humanRoot[0][2]+Position.position_r[2];
        new RootT(wcj, Position.position_r).start();
    }
    new SacroiliacT(wcj, walkData.sacroiliac[0]).start();
    new V11T(wcj, walkData.v11[0]).start();
}

/**
 * second method of joint rotation definition for walk
 */
:
:
/**
 * The twenty one method of joint rotation definition for walk
 */
public void walk21HeadApp()
{
    new SkullbaseT(wcj, walkData.skullbase[8]).start();
}
public void walk21LArmApp()
{
    new LwristT(wcj, walkData.l_wrist[2]).start();
}

```

```

    new LelbowT(wcj, walkData.l_elbow[2]).start();
    new LshoulderT(wcj, walkData.l_shoulder[2]).start();
}
public void walk21RArmApp()
{
    new RwristT(wcj, walkData.r_wrist[2]).start();
    new RelbowT(wcj, walkData.r_elbow[2]).start();
    new RshoulderT(wcj, walkData.r_shoulder[2]).start();
}
public void walk21LLegApp()
{
    new LankleT(wcj, walkData.l_ankle[5]).start();
    new LkneeT(wcj, walkData.l_knee[6]).start();
    new LhipT(wcj, walkData.l_hip[6]).start();
}
public void walk21RLegApp()
{
    new RankleT(wcj, walkData.r_ankle[9]).start();
    new RkneeT(wcj, walkData.r_knee[8]).start();
    new RhipT(wcj, walkData.r_hip[8]).start();
}
public void walk21App()
{
    if(t)
    {
        Position.position_h[0] = walkData.humanRoot[17][0]+Position.position_h[0];
        Position.position_h[1] = walkData.humanRoot[17][1];
        Position.position_h[2] = walkData.humanRoot[17][2]+Position.position_h[2];
        new RootT(wcj, Position.position_h).start();
    }
    else
    {
        Position.position_r[0] = walkData.humanRoot[17][0]+Position.position_r[0];
        Position.position_r[1] = walkData.humanRoot[17][1];
        Position.position_r[2] = walkData.humanRoot[17][2]+Position.position_r[2];
        new RootT(wcj, Position.position_r).start();
    }
}
/**
 * The end of walk automation definition.
 */
}

```

Class For Walk Thread Definition

```

/**
 * "walk thread" definition. Control all of the "walk" stages together. For every "walk"
 * stage, it defines the joints angles at one time.
 */
import java.util.*;

class WalkThread extends SetParam
{
    private ControlJoint wtcj;
    private Walk walkB;
    private float cycle;
    private int times;

    public WalkThread(ControlJoint a)
    {
        wtcj = a;
    }

    public SetParam set(Hashtable h)
    {
        if(h.get(Position.cycle) != null)
            cycle = ((Float)h.get(Position.cycle)).floatValue();
        if(h.get(Position.times) != null)
            times = ((Integer)h.get(Position.times)).intValue();
        else
            times = 3;
        if(h.get(Position.longer) != null)
            times = 6;
        walkB = new Walk(wtcj);
        Position.walkTimes = times;
        Position.walkHeads = times;
        Position.walkLeftArms = times;
        Position.walkRightArms = times;
        Position.walkLeftLegs = times;
        Position.walkRightLegs = times;
        if(h.get(Position.fast) != null)
            cycle = cycle - 0.1f;
        if(h.get(Position.slow) != null)
            cycle = cycle + 0.1f;
        if(h.get(Position.faster) != null)

```



```

{
    cycle = Position.walkduration - Position.incre;
    Position.walkduration = cycle;
}
if(h.get(Position.slower) != null)
{
    cycle = Position.walkduration + Position.incre;
    Position.walkduration = cycle;
}
return this;
}

public void run()
{
    if(Position.bodyFlag)
    {
        while (Position.walkTimes>0)
        {
            /**
             * continue to turn head to the object while avatar walks
             */
            Position.bodyFlag = false;
            if(Position.walkHeads>0)
                walkB.walk1HeadApp();
            if(Position.walkLeftArms>0)
                walkB.walk1LArmApp();
            if(Position.walkRightArms>0)
                walkB.walk1RArmApp();
            if(Position.walkLeftLegs>0)
                walkB.walk1LLegApp();
            if(Position.walkRightLegs>0)
                walkB.walk1RLegApp();
            walkB.walk1App();
            stopW((new Float(42*cycle)).intValue());
            walkB.walk2App();
            :
            :
            if(Position.walkHeads>0)
                walkB.walk21HeadApp();
            if(Position.walkLeftArms>0)
                walkB.walk21LArmApp();

```

```

if(Position.walkRightArms>0)
    walkB.walk21RArmApp();
if(Position.walkLeftLegs>0)
    walkB.walk21LLegApp();
if(Position.walkRightLegs>0)
    walkB.walk21RLegApp();
walkB.walk21App();
stopW((new Float(83*cycle)).intValue());
/**
 * when the user presses stop or make walkTimes equals zero, It will set
 * walkTimes zero. But this cycle can still be finished, then set all
 * of numbers zero, stop the behavior and stand at last.
 */
if(Position.walkTimes != 0)
{
    Position.walkTimes--;
    :
    :
}
if(Position.walkTimes == 0)
{
    Position.walkHeads = 0;
    :
    :
    Hashtable hts = new Hashtable();
    new Stand(wtcj).set(hts).start();
}
/**
 * changing viewpoint of user according to the movement of avatar.
 */
new ChangeViewpoint(wtcj).start();
}
}
Position.bodyFlag = true;
Position.sentover = true;
}
public void stopW(int d)
{
    :
    :
}

```

Appendix D Script Language Interpreter

```
import java.util.*;

class Script extends Thread
{
    private ControlPanel scp;
    private ControlJoint scj;
    private String separation;
    private String parallelSep1;
    private String parallelSep2;
    private String[ ] sentence;

    private Hashtable persontable = new Hashtable();
    :
    :
    private Hashtable turntable = new Hashtable();

    public Script(ControlPanel a, ControlJoint b)
    {
        scp = a;
        scj = b;
        separation = ".";
        parallelSep1 = ",";
        parallelSep2 = "and";

        /**
         * add keys and values to the tables.
         */
        persontable.put(Position.nancy, movetable);
        :
        :
        onttable.put(Position.computer, new TurnComputerThread(scj));
    }

    public void run()
    {
        /**
         * separate all of the sentences.
```

```

    */
    String text = scp.getUserList().getText();
    StringTokenizer stoken = new StringTokenizer(text, separation);
    int length = stoken.countTokens();
    sentence = new String[length];
    int i = 0;
    while(stoken.hasMoreTokens())
    {
        sentence[i] = stoken.nextToken();
        i++;
    }

    /*
    * for each sentence, separate it with "," "and" if it has concurrent movements.
    */
    for (int j = 0; j < length; j++)
    {
        :
        :
        String sen = sentence[j];
        StringTokenizer st1 = new StringTokenizer(sen, parallelSep1);
        Vector vec = new Vector();
        while(st1.hasMoreTokens())
        {
            String str1 = st1.nextToken();
            int p = str1.indexOf(parallelSep2);
            if (p == -1)
                vec.add(str1);
            :
            :
            beginExe(length1, singles);
        }

    public void beginExe(int s, String[] b)
    {
        int length2 = s;
        :
        :
        /*
        * To parse the whole sentence to corresponding movement.
        */

```

```

int count = 0;
String keystr = sn[count];
Hashtable ht = new Hashtable();
Hashtable ht1 = new Hashtable();
/**
 * to judge which table it can begin.
 */
if(persontable.containsKey(keystr))
    ht = persontable;
else
    if(movetable.containsKey(keystr))
        ht = movetable;
    else
        error("Please input person or movement name");

while(ht.get(keystr) instanceof Hashtable)
{
    ht1 = (Hashtable)ht.get(keystr);
    ht = ht1;
    count++;
    if(count<length3)
        keystr = sn[count];
    if(keystr.equals("the"))
        :
        :
}
}
public void stopS(int d)
{
    try{
        this.sleep(d);
    }
    catch(InterruptedException c){}
}
}

```

Appendix E Script Language Sentences

Nancy|she stands.

Nancy|sh| waits.

Nancy|she runs *fast|faster|slow|slower*.

Nancy|she walks *fast|faster|slow|slower*.

Nancy|she jumps *fast|faster|slow|slower|longer*.

Nancy|she opens *the* door.

Nancy|she turns to *the* door.

Nancy|she turns to *the* yarddoor.

Nancy|she turns to *the* window.

Nancy|she turns to *the* tree.

Nancy|she turns to *the* chair.

Nancy|she turns *the* head to *the* door.

Nancy|she turns *the* head to *the* yarddoor.

Nancy|she turns *the* head to *the* window.

Nancy|she turns *the* head to *the* light.

Nancy|she goes to *the* door *fast|faster|slow|slower*.

Nancy|she goes to *the* yarddoor *fast|faster|slow|slower*.

Nancy|she goes to *the* table *fast|faster|slow|slower*.

Nancy|she turns on *the* computer.

Nancy|she turns on *the* light.

Nancy|she waves *the* left arm *fast|faster|slow|slower*.

Nancy|she waves *the* right arm *fast|faster|slow|slower*.

Nancy|she kicks *the* left leg *fast|faster|slow|slower*.

Nancy|she kicks *the* right leg *fast|faster|slow|slower*.

Optional elements are represented in italic font. The sentences can be written either in lowercase or uppercase.