PLANNING TECHNIQUES FOR AGENT BASED 3D ANIMATIONS

Balasubramanian Kandaswamy, B.E.

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

December 2005

APPROVED:

Paul Tarau, Major Professor
Rada Mihalcea, Committee Member
Elizabeth Figa, Committee Member
Krishna Kavi, Chair of the Department of Computer Science and Engineering
Oscar Garcia, Dean of the College of Engineering
Sandra L. Terrell, Dean of the Robert B. Toulouse School of Graduate Studies

Kandaswamy, Balasubramanian. *Planning techniques for agent based 3D animations*. Master of Science (Computer Science), December 2005, 70 pp., 3 tables, 27 figures, references, 31 titles.

The design of autonomous agents capable of performing a given goal in a 3D domain continues to be a challenge for computer animated story generation systems. We present a novel prototype which consists of a 3D engine and a planner for a simple virtual world. We incorporate the 2D planner into the 3D engine to provide 3D animations. Based on the plan, the 3D world is created and the objects are positioned. Then the plan is linearized into simpler actions for object animation and rendered via the 3D engine. We use JINNI3D as the engine and WARPLAN-C as the planner for the above-mentioned prototype.

The user can interact with the system using a simple natural language interface. The interface consists of a shallow parser, which is capable of identifying a set of predefined basic commands. The command given by the user is considered as the goal for the planner. The resulting plan is created and rendered in 3D. The overall system is comparable to a character based interactive story generation system except that it is limited to the predefined 3D environment.

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all those people who supported me during the course of my Masters program. Firstly, I would like to thank my mom for supporting me emotionally throughout my academic career.

This thesis work would not have been possible without the continuous support and guidance of my advisor Dr. Paul Tarau. I thank him from my heart for steering me in the apt route towards success. I take this opportunity to express my gratitude to Dr. Elizabeth Figa of the SLIS department. I couldn't have completed this literature without her guidance. I thank Dr. Rada Mihalcea and other members of the Natural language Processing Research Group for giving me an opportunity to exhibit my work to them and discuss about it, prior to the thesis defense.

My thanks also go to my friend Sudeep for helping me organize the literature. I also thank my friend Kalyan Voddi for his design contributions. The 3D models that he built served as a valuable resource for my research. I would also like to thank my friend Vivek for helping me investigate and gain knowledge about 3D animations.

Last but not least, I thank all my friends who were there for me when I needed them.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

Interactive 3D graphics is becoming ubiquitous. Today's computers are distributed with some sort of 3D graphics accelerator. We utilize this opportunity to approach 3D graphics research not as a question of rendering speed or improving performance, rather utilizing it in the form of interactive art and entertainment. Story Generation and interactive fiction fall under this category. Our primary objective is to build a prototype for interactive 3D graphics that can act as a common back-end for both interactive fiction and interactive 3D narrative content.

1.1 Motivations

Following sections identify the motivations of our research.

1.2 Interactive 3D Graphics

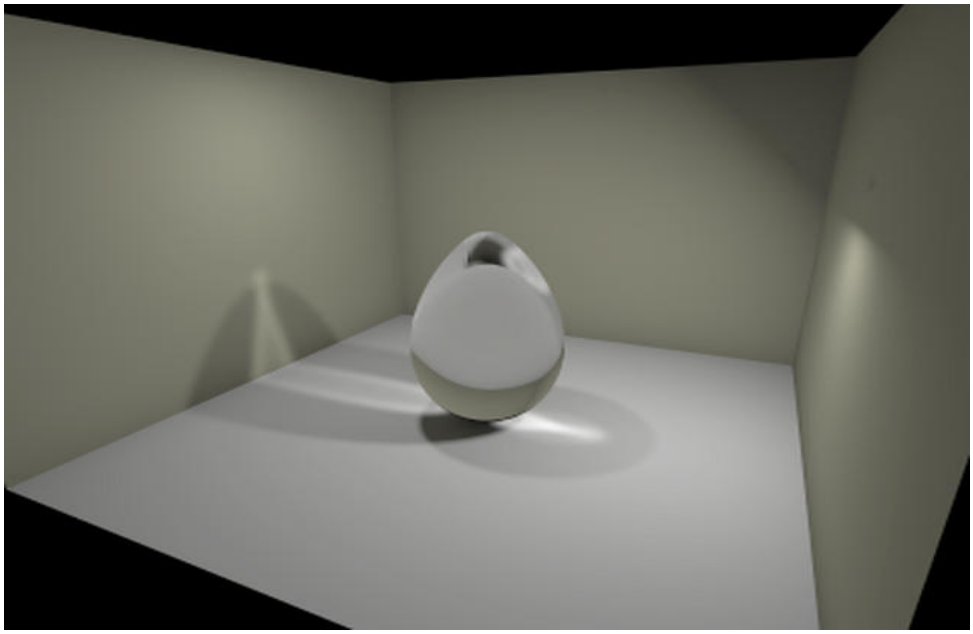

Figure 1-1 Interactive 3D graphics.

One of the most exciting feauture of computer graphics is interactivity. This gives us the ability to affect, influence and immerse in the environment rather than to be simply an observer. Decades back, interactive 3D graphics computers were found only at high-end centers such as research laboratories and corporate engineering centers. It was never in reach for the general masses to have the experience of interacting with an artificial 3D environment. But today's technology allows everyone to have the opportunity to experience interactive 3D.

The advent of invasive superior interactive 3D graphics will allow people to change their way of interaction with the computers. In spite of this advancement, there are many obstacles that still subsist, which prevent the general masses from being comfortable using a computer. The gradual removal of these obstacles will lead to the creation of innovative applications based on the latest types of interactions that are achievable.

1.2.1 Interactive Fiction

Interactive fiction is another terminology used for what once used to be called as "text adventures"[29]. It falls under the category of computer games. The name is as such because of interaction with a fictive world by means of text based interactive agents (not necessarily humanoids). It is a way of problem solving in the 3D world by interacting via the above mentioned agent. Our work discusses a new approach which could be innovatively used in the creation of interactive fiction. We discuss about ways to use

planners for problem solving and navigation. The user and the planner both work in collaboration in the 3D fictitious world for completing the desired task.

1.2.2 Narrative Intelligence

Story telling is a popular performing art all over the world. Every part of the world has developed its own style and tradition for story telling. There are various styles of delivering a story, some combining even musical compositions between the narrations. Narrative in the form of oral, written, or visual stories is portrayed as one of the fascinating means of entertainment in our social and leisure lives. Even adults carry on encircling themselves with stories, furnishing their worlds not just with the facts but with senses. The prevalence of narrative in our lives is partly due to what is called narrative intelligence [1] [2], which refers to the ability of human, and or in collaboration with a computer to orchestrate experiences into narrative [1]. The customary approach to incorporate storytelling into a computer application is by scripting a story at design time. An alternative approach would be to generate stories dynamically by using intelligent planners. Young [9] paints a clear picture of the various advantages of using a planning approach as a model of narrative.

- Plans are comprised of partially ordered steps. If steps are character actions, then a plan is a good model of a story fabula[9] – a chronological enumeration of events that occur in the story world. (from [9])
- Planning algorithms [9] construct plans based on causal dependencies between steps. This assures that all events are part of a causal chain, resulting in a coherent story structure. (from [9])

An adept story generation system is one which is capable of adapting stories according to the user's preference by interacting with the user in ways that were once thought impossible.

1.3 Problem Statement

Designing autonomous agents which are capable of performing a given goal in a 3D domain continues to be a challenge for computer animated story generation and Interactive Fiction systems. This brings up the idea of having a common intelligent back-end planning system, which can resolve various challenges for agent based 3D animations such as navigation, reasoning and many more.

1.4 The Big Picture

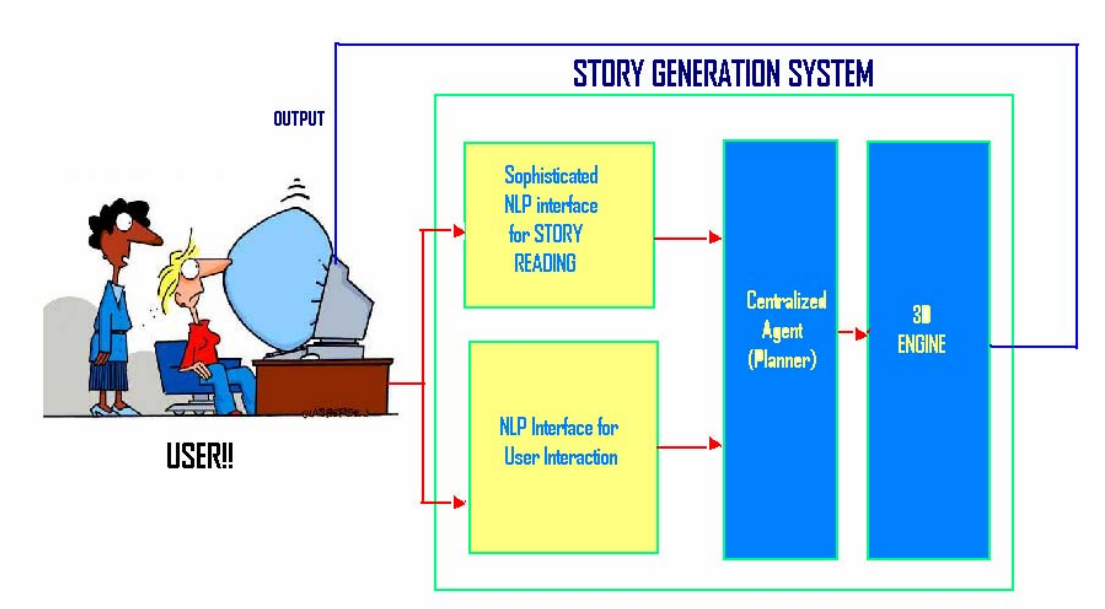

Figure 1-2 Common architecture of a story generation system.

1.4.1 A story generation system

Constructing superordinate Story Generators is an interdisciplinary research area. '*Narratologists'* [30] (The people who deal with the science of Narration) are summoned to disambiguate and systematize basic concepts and theoretical models of narration, where on the other hand computer scientists and AI researchers try to transform these models into viable system architectures and processes. A story generation system [29] [3] is any computer application that creates a written, spoken, or visual presentation of a story as a sequence of actions performed by single or multiple characters.

1.4.2 The Idea of Using a Planner

The efforts for betterment of interactive 3D graphics have given rise to a number of interactive applications, including computer games, training simulations, and intelligent tutoring software. These kinds of applications require an agent to perform coordinated sequences of actions structured as an unfolding story or narrative [1] [4]. One approach used to address the coordination of the actions within these story-based systems is the use of a centralized planning system [9], in which a single planner defines the actions of the agents as a plan and where the plan is rendered visually for the audience.

1.4.3 The Two Approaches and the One we decided to Follow

The technical approach to automated story generation has implications for character believability and story coherence [10]. There are two approaches followed which has edge over the other in achieving character believability and story coherence. Each has

its own set of advantages and disadvantages. The two types of approaches as defined by Young [9] are

- A strong autonomy approach[9]
- A strong story approach.[9]

The strong story approach advocates centralized control by using a single authoring agent [9], which is responsible for making the decisions for the generation of the story. This approach is favorable because the centralized control allows the authoring agent to approach the story from a global point of view, choosing the necessary actions in a pertinent way. For these reasons, we decided to follow the strong story approach and include the planner to act as the centralized agent and play the part of the producer for the story generation, where the user plays the parts of the storywriter and the director.

1.5 The Prototype

The central thesis of the work is to build a prototype to demonstrate a potential way of implementing the intelligent and rendering back-end of the story generation system, with focus on bridging the centralized agent (the planner) to the 3D engine. The various building blocks of the prototype are explained briefly in various chapters of the literature. The prototype is called as PLANNER3D.

Planner3D is an interactive 3D system built completely using JINNI3D [31]. JINNI3D as an extension of JINNI employs JAVA3D [22] as the 3D engine. Planner3D uses WARPLAN-C [26] as its intelligent backend. Planner3D interacts with the user by

means of a simple Natural Language interface. It also employs speech synthesis along with the 3D visual output.

The Planner3D holds responsibility for the creation of the 3D world. It identifies the set of objects that needed to be placed in the world based on the plan and positions them appropriately. Planner3D maps the sequence of actions generated by the planner to its corresponding set of animations with the help of the Knowledge Base and renders them in the 3D world via JINNI3D.

1.6 Literature Overview

Chapter 2 Related Work discusses in detail the functioning of similar and more advanced tools such as Alice3D [12] [14] authoring system from stage 3-development group at Carnegie Mellon University. It also gives an overview about the commercial tool, (Engineering Animation, Inc) EAI's Sense8 WorldUp [12]. The implementation, techniques and salient features of these tools are briefed in detail for the purpose of comparison.

Chapter 3 gives an overview of JINNI3D API, an extension of JINNI (JAVA INference engine and Network Interactor) [21] [31]. JINNI is a lightweight, multithreaded, logic programming language, intended to be used as a flexible scripting tool for gluing together Knowledge processing components and JAVA Objects. This chapter provides a short tutorial about creation of agents and animations using JINNI3D. The second

half of the chapter is committed to elucidate about the embodied agents and enlists the methods for importing and animating them via JINNI3D.

Chapter 4 includes descriptions of the planner WARPLAN-C [26] and explains in brief the implementation of the above-mentioned prototype along with the speech synthesizer. It quotes and explains in a succinct way the prominent parts of the code with examples. More emphasis is given to this chapter as this form the center of the research. It clarifies in detail the implementation of the architecture and the results.

Chapter 5 provides a succinct summary of the research and concludes the results.

Chapter 6 proposes various ways to improvise the prototype. It elucidates about a list of possible additions that can be made to achieve that.

CHAPTER 2

RELATED WORK

This chapter of the literature gives us an overview of the existing software implementations, which has a structure similar to that of our prototype PLANNER3D. This provides us a means for evaluating our architecture through comparison. We borrow for comparison the two of famous interactive 3D graphics authoring tool

- Alice 3D [12] - Authoring system, from the Stage3 Research Group at Carnegie Mellon University
- WorldUp- A commercial virtual reality and visual simulation development tool from Sense8 organization.

Technical specifications of the above software are borrowed for the purpose of comparison. It is to be noted that Stage3 research group of Carnegie Mellon University and Sense8 organization hold all rights for Alice3D and WorldUp respectively. The operations of Alice3D and WorldUp are apprized in this chapter. Their similarities and differences are examined in chapter 4, where we assume that the reader knows about the functioning of PLANNER3D system.

2.1 Alice 3D Introduction

Alice 3D [14], [12] is an authoring tool for interactive 3D graphics. Alice is a 3D graphics-programming environment designed for novices with no 3D graphics or programming experience. The language used for scripting in Alice 3D is object oriented as well as interpreted. This allows the user to update the current state either by interactively evaluating program code fragments, or by manipulating GUI tools.

Virtual environments [32] present a new medium for both the user and the programmer. The best way to accelerate a new medium is to provide tools that allow people without highly technical backgrounds to create 3D animations. Alice is developed to support this goal. Alice is also a rapid prototyping environment [12] for virtual reality. Using Alice 3D we can generate environments such as the one shown below.

Figure 0-1 A sample environment created with Alice 3D. [32]

The name "Alice" honors Lewis Carroll's heroine, who explored a rapidly changing, dynamic environment [32].

2.2 Authoring a 3D Environment Using Alice 3D

Alice 3D follows an authoring structure similar to commercial tools like WorldUp[15]. It consists of two consecutive phases:

- Creating an opening scene
- Scripting

2.2.1 Creating an Opening Scene

Alice 3D has an exhaustive library of precise 3D models. The user can create the Opening scene using the objects in these libraries. The user does this by clicking the add object button (figure 2-2, A). Alice also has the ability to import objects built using several popular 3D modeling software.

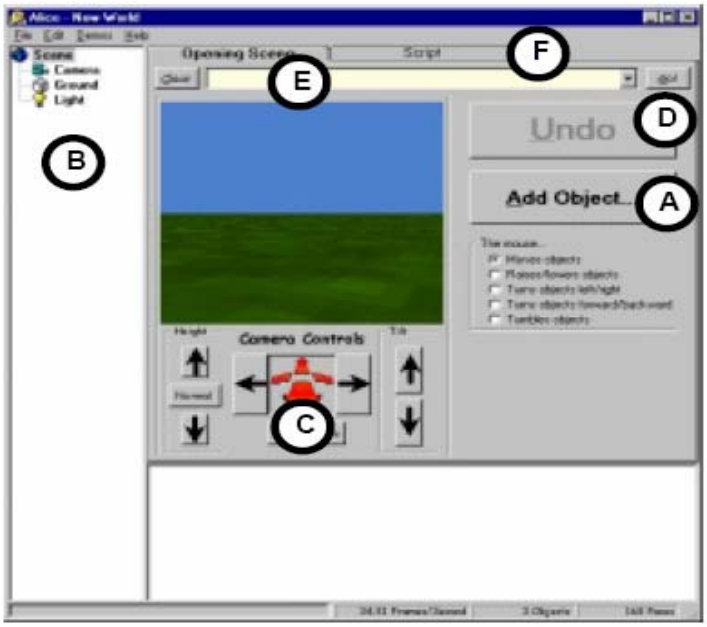

Figure 0-2 Alice 3D interface. (from [12])

Each and every object added to the scene are organized into a tree like structure [17][18] (figure2-2, B), The GUI also contains buttons for the navigation of the camera (figure2-2, C) .

2.2.1.1 The Alice Command Box

As mentioned in section 2.1 the language used in Alice 3D is interpreted. This ability of Alice 3D can be exploited through the Alice Command Box (figure2-2, E)[12][14]. The commands entered in the command box are interpreted and the corresponding

animation is rendered in 3D. For example an object can be moved to a precise distance by using the following syntax

<object name>.move (<direction>, <distance as numerical value>)

The interpretation of the command entered can be intiated by the user either by pressing the GO button [12] or by hitting the Enter key. The animation is rendered over a period of one second. The programmers can also explicitly specify the duration as desired. In addition Alice also provides an animated infinite undo mechanism.

Alice commands are highly overloaded which is considered to be a good feature as the same command can be called via several patterns depending upon user desirability. For example, Move can be called in all these ways (example adapted from [12]):

```
obj.move(forward, 1)
obj.move(forward, 1, duration=3)
obj.move(forward, 1, speed=4)
obj.move(forward, speed=2)
```

2.2.2 Scripting in Alice 3D

The initial state of the world created using Alice 3D is saved into a file called ‘world file’ [12]. All the objects present in the world are referenced in the script by using this file. The scripts in Alice 3D are to be entered in a text editor which is exposed by pressing the scripting tab (figure2-2, F). Then the script entered via the text editor is run by pressing the ‘run script’ button.

2.2.3 The Implementation

Alice runs on Windows based machines so that its GUI can be properly utilized. Hence it uses Microsoft's Direct 3D Retained Mode (D3DRM) for rendering in 3D, which is shown as the basic system level layer in the overall architecture. This layer handles all details related to rendering in 3D including object database management, etc.

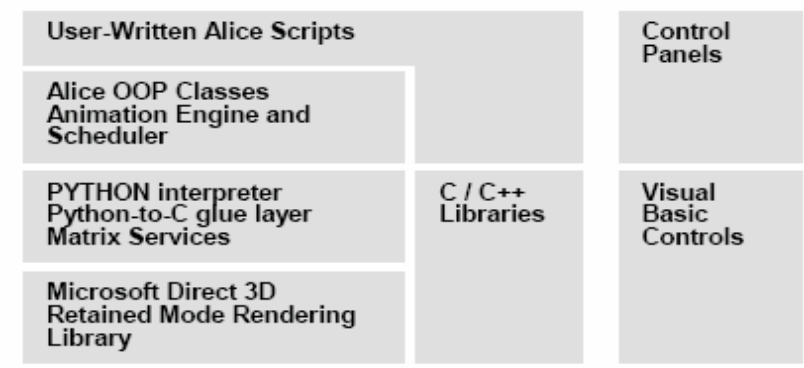

Figure 0-3 Overall structure of Alice3D(from [12])

2.3 Biggest Contribution of Alice

This section describes about the biggest contribution of Alice for the interactive 3D graphics community

2.3.1 The Death of XYZ

"Perhaps Alice's most distinguishing API feature is that it allows people to create behavior for three-dimensional objects without using the traditional mathematical names for the coordinate axes: X, Y and Z. Instead, Alice uses LOGOstyle [13], object-centric direction names: Forward/Back, Left/Right, and Up/Down. This seemingly tiny, cosmetic change is probably Alice's biggest contribution to making a usable API for 3D graphics. By using direction names in lieu of XYZ, the user is relieved of a cognitive mapping step that may occur thousands of times while developing a 3D program." Referred from [12]

2.4 Overview of WorldUp

Parts of this section are referred to [15]. The operation and features of the WorldUp are patented by Sense8 organization.

2.4.1 Introduction

WorldUp[15] is yet another 3D content authoring tool for real-time graphical simulations. The user can efficiently create or import 3D scenes using WorldUp, make them interactive with an easy-to-use GUI which allows drag and drop assembly. Like Alice 3D WorldUp also follows scripting in an object-oriented manner. WorldUp has an incredible feature which allows the 3D graphical objects to behave in a same way as the real world objects. This is achieved by having a proper definition set for every 3D object which describes the behavioral physics associated with that object model. WorldUp allows the user to modify these properties and behaviors concurrently to the animation. One of the important reasons for the commercial success of WorldUp is its ability to support importing 3D models from a wide array of industry standard modelers. The user can change a property of an existing behavior as desired by writing scripts in BasicScript language [15] supported by WorldUp.

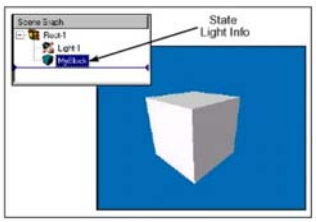

Figure 0-4 Snapshot of a 3D box created using WorldUp

2.4.2 WorldUp Features

“The WorldUp development environment contains many valuable features [15] such as

- A graphical interactive development environment
- Built-in object types
- An object-oriented application framework
- Object hierarchy with dynamic property inheritance.
- The ability to modify and add object properties and behaviors during a simulation
- Support for a wide variety of input/output devices.
- Support for a variety of file formats.
- An interpreted scripting language (BasicScript) syntactically equivalent to Visual Basic
- A script debugger to help resolve errors in your scripts
- Run-time binding of C routines from scripts
- The Scene Graph pane to visualize complex databases
- Data visualization through BasicScript SQL functions.
- Players for distributing your application
- Cross-platform portability (without recompiling)” (from [15])

CHAPTER 3

AGENT BASED ANIMATION WITH JINNI3D

3.1 JINNI3D Introduction

JINNI (JAVA INference engine and Network Interactor)[21] is a light weight, multithreaded, logic programming language, intended to be used as a flexible scripting tool for gluing together knowledge processing components and JAVA Objects. JINNI 3D is an extension of JINNI, which requires the installation of JAVA3D. JAVA3D [22] is a low-level 3D scene-graph based graphics programming API, for the JAVA language. It is a full featured 3D graphics API that can be used to create high performance 3D graphics applications from within the JAVA platform. Now as an extension of JINNI, JINNI 3D enables us to program 3D applications.

3.1.1 The World of JINNI3D

JINNI3D has a JAVA interface consisting of set of classes and methods which provides an abstract for the creation, animation and manipulation of the 3D world. The interface is built over the JAVA 3D API, to provide a hierarchy of classes[22] to enable the programmer to work with the high-level constructs for creating and manipulating 3D geometrical objects. These Geometric objects reside in a virtual universe, which is then rendered.

JINNI 3D provides the programmer with a wide variety of functionality and flexibility to create precise virtual universe with a wide variety of objects of varying sizes from subatomic to astronomical. Despite all these feautures JINNI 3D is straightforward to

use as the details of rendering are handled automatically. The JAVA threads concepts enables the renderer to render objects in parallel. The JAVA 3D renderer is also capable of making automatic optimization for improved performance.

3.1.2 JINNI3D Architecture

JINNI 3D as a component of JINNI is a lightweight, thin client logic programming component , which is based as much as possible on fully portable vendor and version independent JAVA code and requires installation of JAVA3D. It facilitates the user to implement 3D applications using prolog scripting. It works by invoking corresponding JAVA method for the prolog code. It creates Universe and objects as per the Prolog code and returns handles of objects for the user. These handles can be used to invoke scripts generated for complex animations on various 3D objects.

The JINNI3D methods are built in such a way that the user has several levels of abstraction of control over the object. The user has the ability to invoke any method of animation supported by JAVA3D via JINNI3D. JINNI3D also provides exclusive functionalities that provide abstraction for complex, co-operative and multiple animations of objects present in the universe. This set of unique functionalities will be explained in detail later.

3.2 Basic Functionalities of JINNI3D

This section briefly describes the technical steps associated with the creation of virtual universe using JAVA 3D and in process shows how JINNI3D can act as an proficient API for scripting 3D animations.

3.2.1 Building a Scene Graph in JINNI3D via JAVA3D

The preliminary point of a JAVA3D virtual universe is a *scene graph [22]*. A scene graph is created using instances of JAVA 3D classes. The object's geometrical and rendering parameters are defined using this scene graph.

A graph in JAVA3D terminology is defined as a data structure composed of nodes and arcs. A node is an instance of a JAVA3D class which corresponds to a data element. The instance which defines the geometry and appearance of such nodes are called as node component objects. The nodes in a scene graph are interrelated to each other [22].The relationship between various nodes are represented using arcs in the graph. The arcs represent the two kinds of relationships between the JAVA 3D instances.

- *parent-child relationship*.[22] A group node can have any number of children but only one parent. A leaf node can have one parent and no children.
- *reference[22]*. A reference associates a Node Component object with a scene graph Node.

We provide a handy abstraction for the above virtual universe creation of JAVA3D and invoke it in JINNI3D by using a simple predicate

new_world(U).

where U is the variable, which represents the handle for the Virtual Universe. We can then manipulate the Universe using this handle via several other predicates of JINNI3D. These predicates/methods allow us to construct and influence the scene graph by adding Node component objects and the leaf nodes. The following is an example of a scene graph.

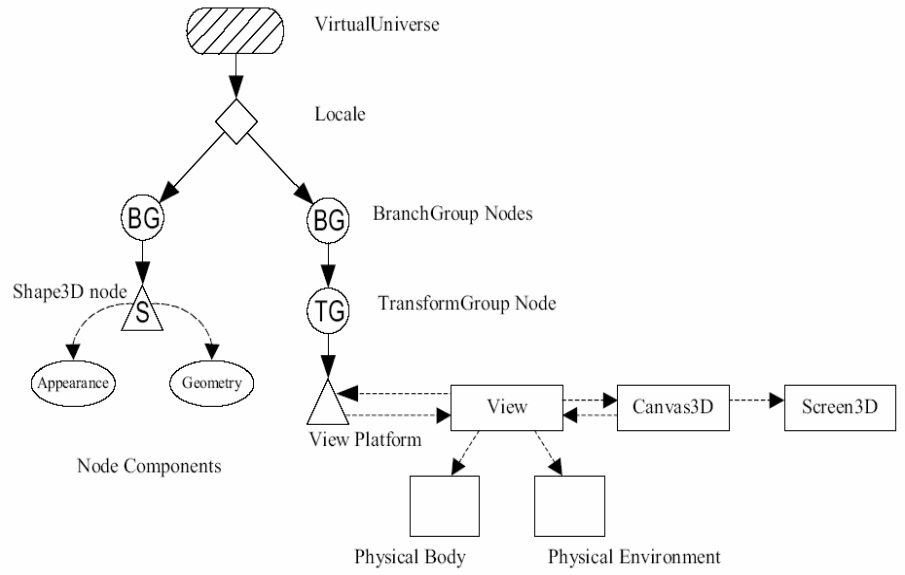

Figure 0-1 Example of a scene graph. (from [22])

3.3 User's Perspective: Comparison between JAVA3D and JINNI3D

This section provides a user's perspective comparison between JAVA3D and JINNI3D as follows. The steps for creating a simple object as given in [22] is compared to the JINNI3D's way of creation of the same object.

3.3.1 Basic Structure of JAVA3D and JINNI3D Programs

The basic outline of JAVA 3D program development consists of seven steps[22] presented below.

1. Create a Canvas3D object
2. Create a VirtualUniverse object
3. Create a Locale object, attaching it to the VirtualUniverse object
4. Construct a view branch graph
    a. Create a View object
    b. Create a ViewPlatform object
    c. Create a PhysicalBody object
    d. Create a PhysicalEnvironment object
    e. Attach ViewPlatform, PhysicalBody, PhysicalEnvironment, and Canvas3D objects
        to View object
5. Construct content branch graph(s)
6. Compile branch graph(s)
7. Insert subgraphs into the Locale

JINNI3D provides us with easier abstractions which allows us to do the same by using just two steps

1. create world
   a. (optional) add objects and position them (default is origin).
2. show world

1.6.1 Sample Program: creation and rendering of a color cube

JAVA3D:

```
public class HelloJAVA3Da extends Applet {
 public HelloJAVA3Da() {
 setLayout(new BorderLayout());
 GraphicsConfiguration config =
 SimpleUniverse.getPreferredConfiguration();
 Canvas3D canvas3D = new Canvas3D(config);
 add("Center", canvas3D);

 BranchGroup scene = createSceneGraph();
 scene.compile();

 // SimpleUniverse is a Convenience Utility class
 SimpleUniverse simpleU = new SimpleUniverse(canvas3D);

 // This moves the ViewPlatform back a bit so the
 // objects in the scene can be viewed.
 simpleU.getViewingPlatform().setNominalViewingTransform();

 simpleU.addBranchGraph(scene);
 } // end of HelloJAVA3Da (constructor)

 public BranchGroup createSceneGraph() {
 // Create the root of the branch graph
 BranchGroup objRoot = new BranchGroup();

 // Create a simple shape leaf node, add it to the scene graph.
 // ColorCube is a Convenience Utility class
 objRoot.addChild(new ColorCube(0.4));

 return objRoot;
 } // end of createSceneGraph method of HelloJAVA3Da
 } // end of class HelloJAVA3Da
```

(sample program from [22]).

JINNI3D:

```
new_world(U),%creates a new universe
color_cube(U,B),%creates a color cube B
show_world(U).%renders the scene
```

As you can see in the above example, JINNI3D supports and provides abstractions for almost all basic functionalities provided by JAVA 3D. Various objects can be created as well as their geometrical and visual attributes can be controlled by using appropriate JINNI3D methods. It is to be noted that JINNI3D is still in development stage and it supports almost but not all features of JAVA3D.

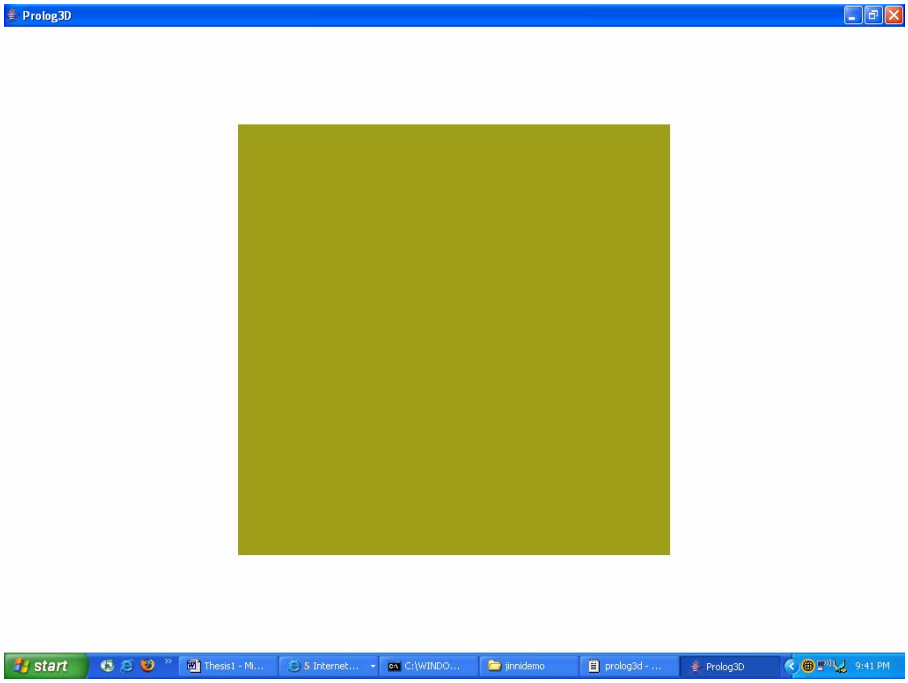

Figure 0-2 the image produced by HelloJAVA3Da.JAVA/HelloJAVA3Da.pl

Further, you can interact with the Universe by adding controls to the created universe by calling the add controls predicate.

```
add_controls(U)
```

The add controls predicate when called, adds controls to the universe allowing the user to interact with the universe using the mouse. Right click is used for the movement of the camera while left click is used for the rotation of the camera.

3.4 JINNI3D Agents

JINNI 3D represents all objects, texts and models in the form of agents. Representing 3D objects as agents, by defining their behavior in 3D world gives us the edge over JAVA3D, when it comes to animation. Most agents fall under the two main category

- Vertex3D agent
- Edge3D agent

Simple geometric objects such as cube, color cube, and sphere are created as Vertex3D agents. Vertex3D agents are created by specifying the vertex coordinates for points, line segments, and/or polygonal surfaces. JINNI3D also makes use of geometric utility classes of JAVA3D for creation of simpler objects. The Edge3D agents allow the user to create free objects by incremental addition of edges.

3.4.1 Creation of JINNI3D agents

JINNI3D agents are created by calling the appropriate predicate, for example a cylinder 'C' can be created and placed in the universe 'U' as follows.

cylinder(U,C)

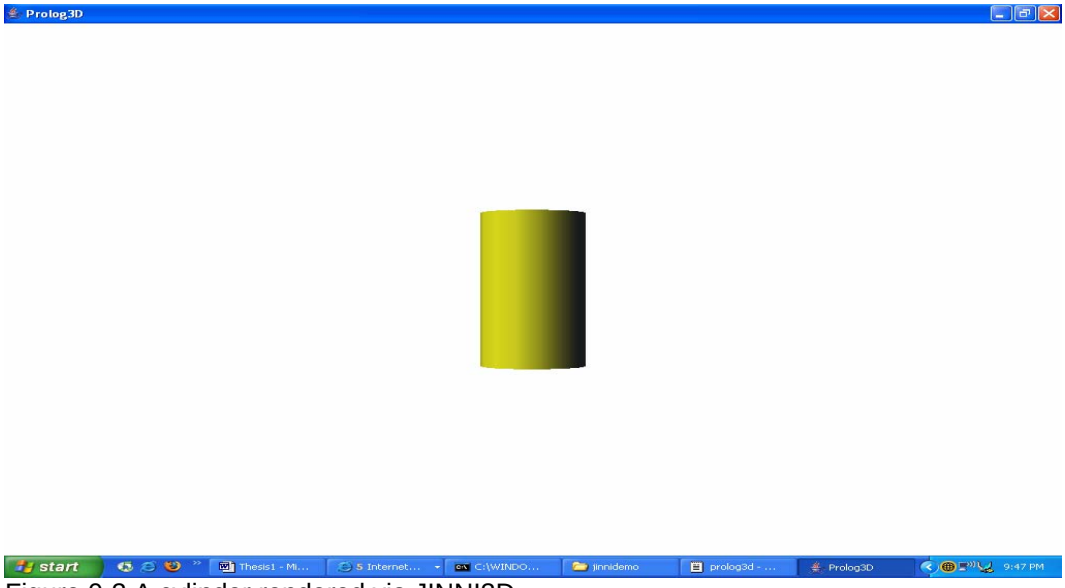

Figure 0-3 A cylinder rendered via JINNI3D

Likewise, JINNI3D allows us to create a wide variety of simple objects and place it in the 3D world. The type of objects varies from box, color cube, sphere, cone, etc. The abstraction of the agent's creation is hidden and is achieved via a call to the JAVA 3D back-end as follows

```
cylinder(U,A):-shaped_agent(U,9,cylinder,A).

shaped_agent(U,ShapeNumber,Data,A):-
invoke_JAVA_method(U,addVertex(ShapeNumber,'$null',Data),A).
```

The JAVA method addVertex handles the creation of the vertex3D agents. The objects are differentiated using the shape number parameter. Each number corresponds to a particular 3D object.

3.4.2 Animating the Agents

Simpler animations in JINNI3D are achieved in ways similar to their creation i.e. by utilizing the JAVA 3D back-end. JINNI 3D also has a valuable collection of predefined predicates, which facilitate the user to impose complex and multiple animations on objects. While the current chapter focuses on simple animations, some of the important tools for complex animations will be briefed in the forthcoming chapters

3.4.2.1 Hide Agent Method

The hide agent method allows us to make an agent invisible in the 3D world; this predicate sends the handle for the agent as the parameter to the corresponding JAVA method, which in turn sends a message to the 3D engine to cease rendering that particular object. The syntax of hide agent method is as follows

hide_agent(<Agent Handle>)

3.4.2.2 Show Agent Method

Show agent method does the exact opposite of H ide Agent method by re-invoking the 3D engine to render the object again and thereby makes it visible. The syntax of show agent method is as follows

show_agent(<Agent Handle>)

3.4.2.3 Rotation

An Agent can be rotated to a desired position by using the rotate_to method. This takes 3 Euler angles as parameters and applies them for rotating the agent with respect to the three axes X, Y and Z.

rotate_to(<Agent handle>,<Rotation w.r.t X-axis>,
<Rotation w.r.t Y-axis>,
<Rotation w.r.t Z-axis>)

We can rotate the agent with respect to any one axis by passing 0.0 as the parameter for the rest of the axis.

(Example) rotate_to(Box,0.0,2.0,0.0) Rotates the agent 'Box' to 2.0 Euler with respect to the Y-axis.

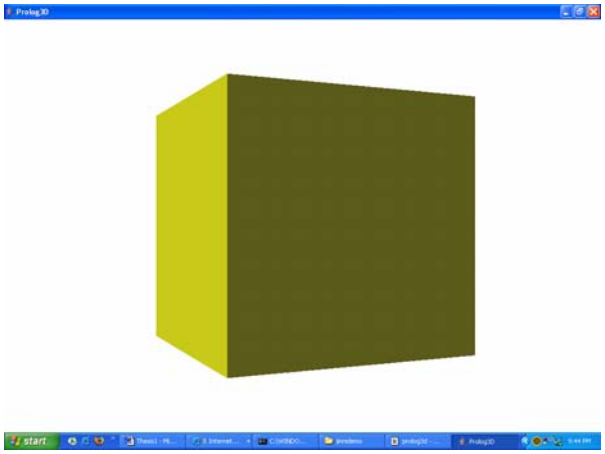

Figure 0-4 A box agent rotated with respect to Y-axis.

3.4.2.4 Positioning

When an agent is created, it is placed at the origin (0,0,0) of the universe. It can be moved from one position to other by using the move_to predicate. The move_to predicate does the combinatorial function of set_x, set_y, and set_z predicates which are used to position the agent with respect to that particular axis. The syntax of move-to predicate is as follows

move_to(<Agent Handle>,<X-axis position>,
<Y-axis position>,
<Z-axis position>)

3.4.2.5 Scaling

An agent can be scaled/resized either uniformly or with respect to the axis via the scale_to predicate. The two methods are differentiated by using the number of arguments passed. It scales uniformly for a single argument or scales with respect to the axis for three arguments.

(uniform) scale_to(<Agent Handle>,<Uniform Scaling factor>)

(Example) scale_to(Cylinder,2.0)

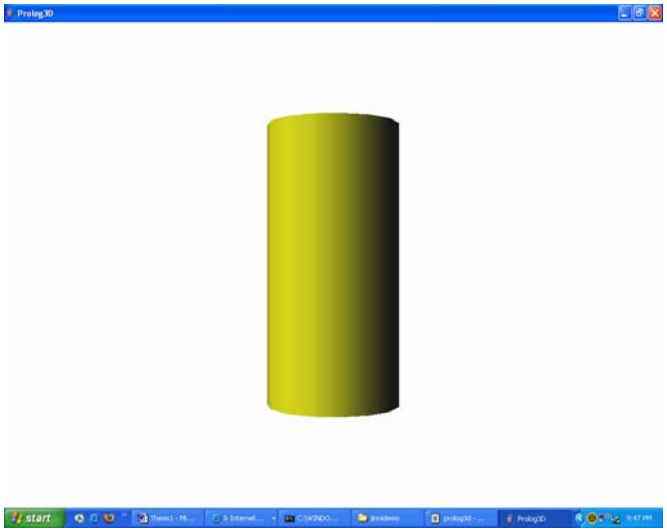

Figure 0-5 A cylinder scaled uniformly to twice its size

(ration based) scale_to(A,S):-scale_to(<Agent Handle>,
<Scaling factor w.r.t X-axis>,
<Scaling factor w.r.t Y-axis>,
<Scaling factor w.r.t Z-axis>),

(example) scale_to(cylinder,1.0,2.0,3.0)

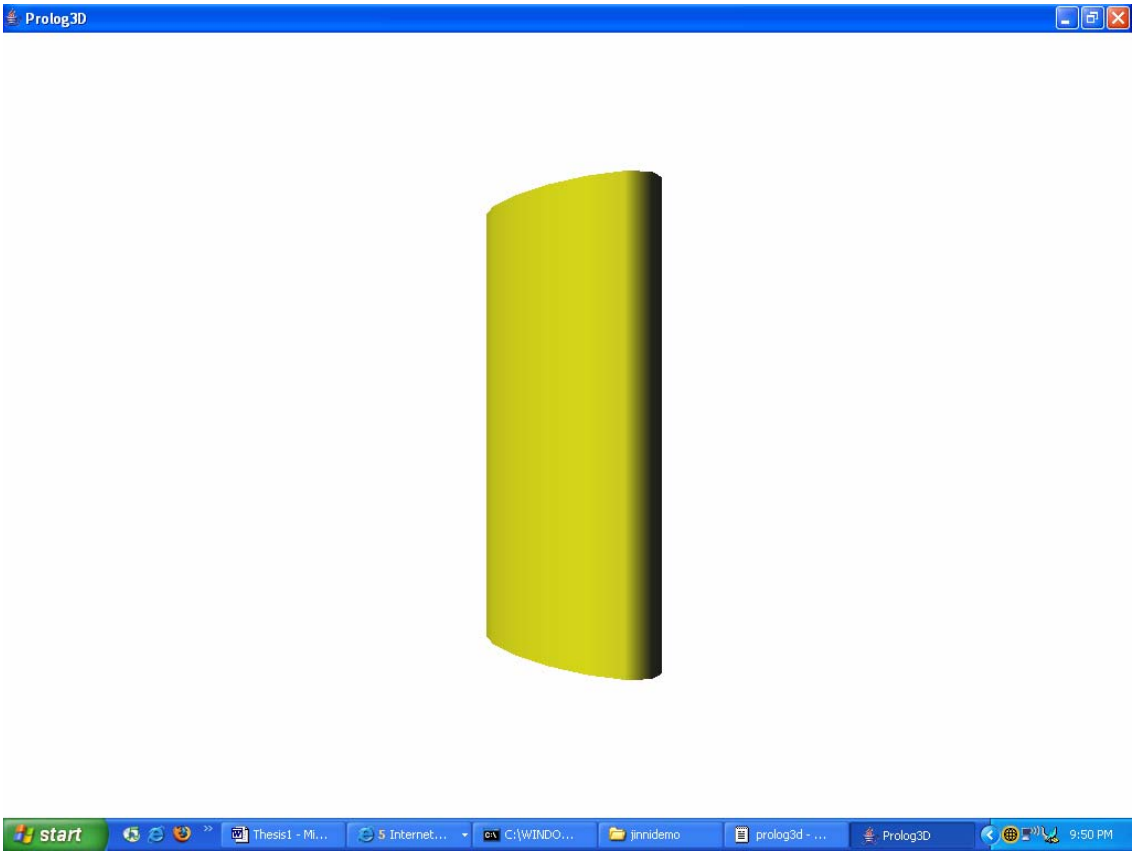

Figure 0-6 A cylinder scaled with respect to each axes

3.4.2.6 Incremental Positioning and Rotation

A JINNI3D agent can be positioned and rotated incrementally with respect to its current position and angle. Predicates like inc_x, inc_y and inc_z, which increments the x,y and z position of the agent respectively. In addition, incremental rotation can be performed using predicates like inc_rot_x, inc_rot_y and inc_rot_z rotating the agent to the prescribed Euler with respect to its current angle.

3.5 Embodied Agents in JINNI3D

As the 3D rendering capabilities continues to grow, there will be an increasing need to represent human beings in virtual environments. Achieving that goal will require the creation of libraries of interchangeable humanoids, as well as tools that make it easy to animate them in various ways. We use a standard way of representing humanoids in VRML 2.0.

3.5.1 VRML 2.0

VRML abbreviates for Virtual Reality Modeling Language[24]. It is a Method of displaying three-dimensional images. VRML is a platform-independent language that creates a virtual reality scene. The Humanoids in our World are represented using the VRML 2.0, the latest version of VRML.

These VRML Humanoids[23] can be animated using key framing, inverse kinematics, performance animation systems, and other techniques. A VRML Humanoid file[23] 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. The file also contains references to all the Joint and Segment nodes. Additional nodes can optionally be included in the file.

3.5.2 Modeling the Humanoid

The Humanoid is modeled[23] in a standing position, facing in the Z direction with Y up and X to the humanoid's left. The origin (0, 0, 0) is located at ground level, between the humanoid's feet.

The foot is placed flat on the ground, spaced apart about the same distance as the width of the hips. The bottom is at Y=0. Initially the arms are straight and parallel to the sides of the body with the palms of the hands facing inwards towards the thighs.

The Humanoid is modeled with actual human size ranges in mind [23]. All dimensions are in meters. A typical human is roughly 1.75 meters tall.

Front View Side View

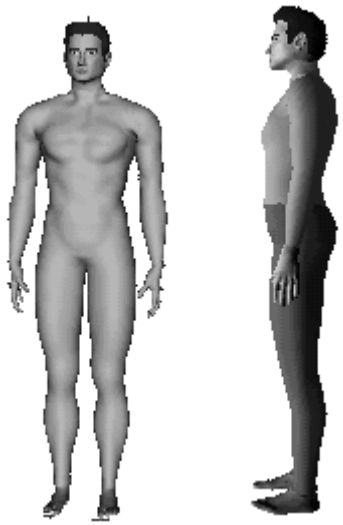

Figure 0-7 Default position of the Humanoid [23]

In this position, all the joint angles are zero. In other words, all the rotation fields, the translation fields [23] and the scale factors are left at their default values. Sending the default values for translation, rotation and scaling to all the Joints in the body must return the body to the neutral position.

3.6 Importing VRML into JINNI3D

The Humanoids Created using VRML 2.0[23] needs to be imported into JAVA3D. The implementation for conversion is integrated along with the basic package of JINNI3D. It has a wide variety of methods, which allows the user to convert 3D models built with external tools to be converted into '.j3f' format recognized by the JAVA 3D engine.

Prior to the conversion, the created models should be placed in a local memory location accessible to JINNI3D. Then the VRML 2.0 models, which have the extension '.wrl', can be easily converted using the following predicates/methods provided by JINNI3D.

convert_vrml(<VRML file name>).

The above method converts the VRML files into J3F files, which is a JAVA3D recognizable format and stores the resulting '.j3f files' and ports them into the folder called 'models', which contains an archive of so called J3F agents.

These J3F agents are imported into JINNI3D via the following syntax

jf2agent(<universe>,<J3F File name>,<Agent handle>)

This command imports the jf2agent into JINNI3D and renders it in the scene. These jf2agents can be used just like any other JINNI3D agent and can be subjected to all sorts of animation. In addition, properly built VRML based agents can support complex animation based on their assembly, which enables the user to animate the parts with respect to the agent.

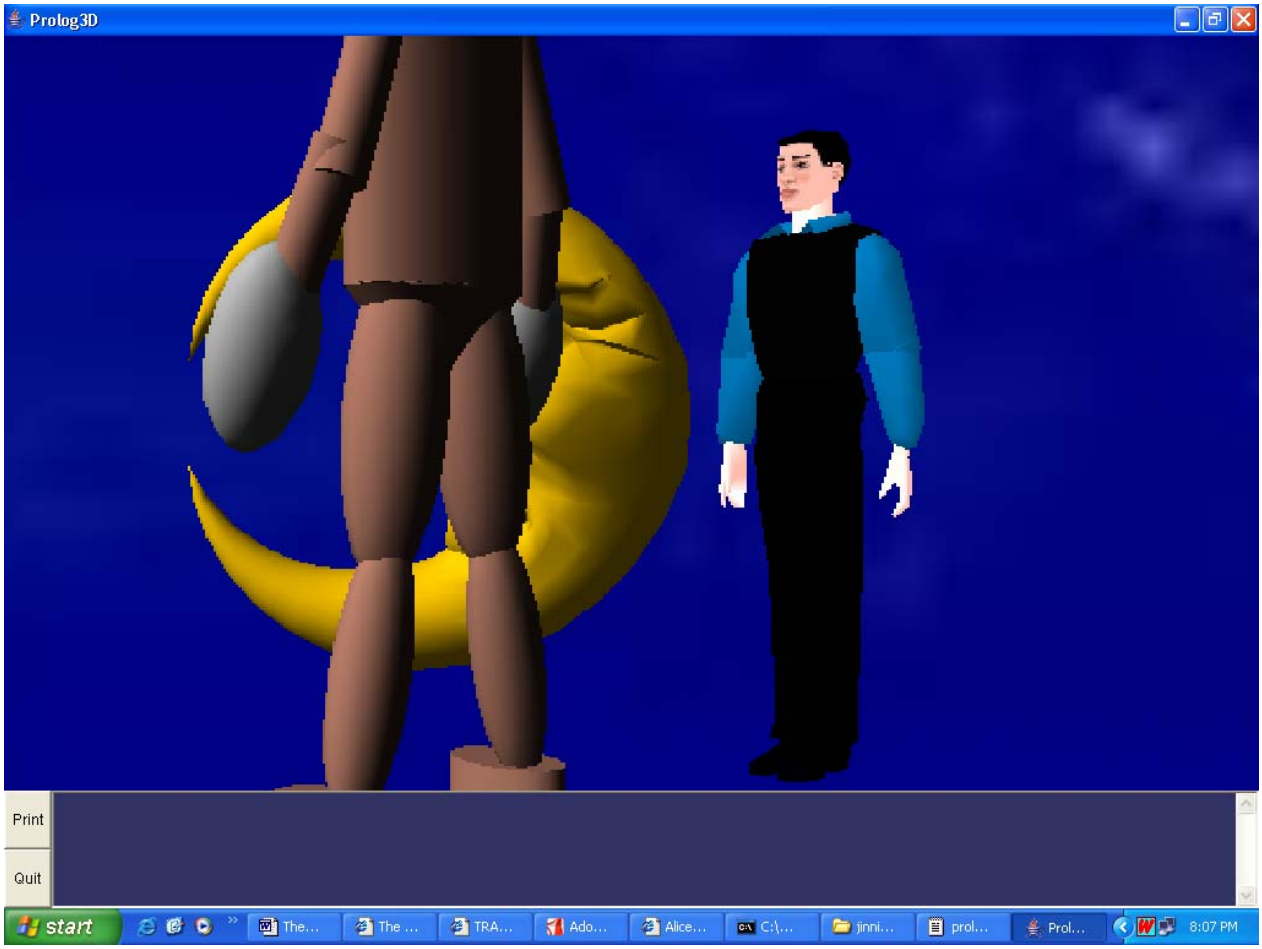

Figure 0-8 Snapshot of imported agents in JINNI3D

3.7 Animating Embodied Agents in JINNI3D

The methods used to animate VRML agents using JINNI3D are listed and elaborated in this section.

3.7.1 VRML Agents and Joints

VRML agents are modeled with joints, which segregates the entire model into segments. These segments can be animated independently or with respect to the other segments. For example, the ‘man’ agent is comprised of the following segments

HEEL_L	→ Left Heel	HEEL_R	→ Right Heel
SH_L	→ Left Shoulder	SH_R	→ Right Shoulder
HAND_L	→ Left Hand	HAND_R	→ Right Hand
THIGH_L	→ Left Thigh	THIGH_R	→ Right Thigh
HAND_L	→ Left Hand	HAND_R	→ Right Hand
ELBOW_L	→ Left Elbow	ELBOW_R	→ Right Elbow
SHANK_L	→ Left Shank	SHANK_R	→ Right Shank
FOOT_L	→ Left Foot	FOOT_R	→ Right Foot
CHEST	→ Chest	HEAD	→ Head
PELVIS	→ Pelvis	STOMACH	→ Stomach
NECK	→ Neck		

Each of the above segments can be controlled/animated using JINNI3D.

3.7.2 Set Focus method

Set Focus method of JINNI3D allows setting the focus of animation on a particular part instead of the entire model. Any animation, positioning or rotation applied to the model affects it only through the part, which is currently focused on. This can be explained briefly with examples

3.7.2.1 Example1: Head Says Yes

This method sets the focus on the HEAD and makes the model to imitate the action of saying yes by turning the head up and down and then finally bringing back to its normal position.

```
head_says_yes(A):-
  body_set_focus(A,'HEAD'),
  turn(up,A),
  turn(down,A),
  turn(down,A),
  turn(up,A).
  body_reset_focus(A).
```

3.7.2.2 Example2: Join Hands

This method joins the hands of the agent and brings it to a posture, which is somewhat similar to '*vanakam*', a south Indian way of greeting. This is achieved by setting the focus on the Shoulders and Elbows and applying rotations appropriately as follows.

```
join_hands(A):-
  join_hand(A,'SH_L','ELBOW_L'),
  ndo(3,tilt(left,A)), %loops 3 times
  join_hand(A,'SH_R','ELBOW_R'),
  ndo(3,tilt(right,A)),
  body_reset_focus(A).

join_hand(A,Sh,El):-
  body_set_focus(A,Sh),
  ndo(3,turn(up,A)), %loops 3 times
  body_set_focus(A,El),
  ndo(3,turn(up,A)). %loops 3 times
```

It is to be noted that after animating the agent, the focus should be reset using the body_reset_focus predicate. This is done to avoid confusion that may occur about the then focus of the agent before scripting a new animation.

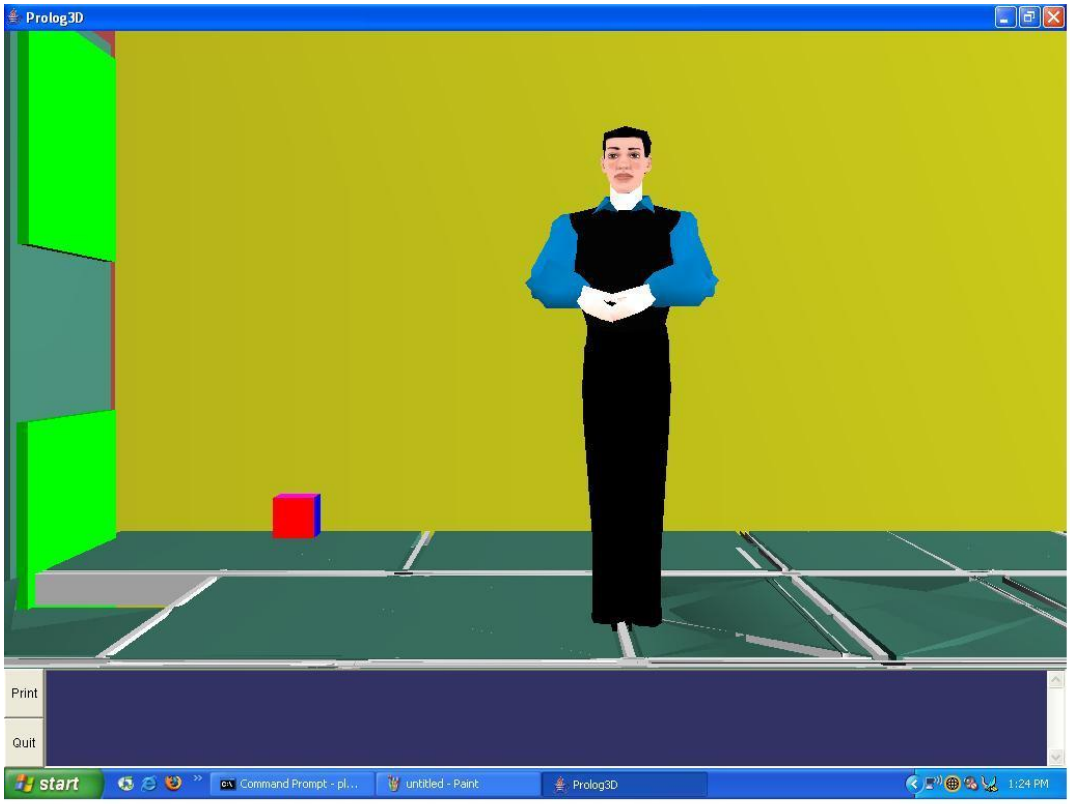

Figure 0-9 A snapshot of agent saying yes and joining hands

CHAPTER 4

PLANNING ANIMATIONS WITH WARPLAN-C PLANNER

4.1 WARPLAN-C Planner

WARPLAN-C [26] planner of D.H.D Warren employs an uncomplicated approach [25] for acquiring conditional linear plans. The method works as follows.

A plan is a sequence of actions that has to be performed to achieve a given goal. The planner has the knowledge, about the world and the set of actions that can be performed on it. The actions which can have more than one possible outcome (Usually two) are termed as conditional actions. All the other actions are termed as unconditional. During the first pass, the planner presumes that all of actions as unconditional ones with a single outcome P. In the resultant plan, the actions which are conditional are tagged. The planner is re-invoked to plan for the other possible outcome of those conditional action (dangling 'else' branch) [25]. This is illustrated below.

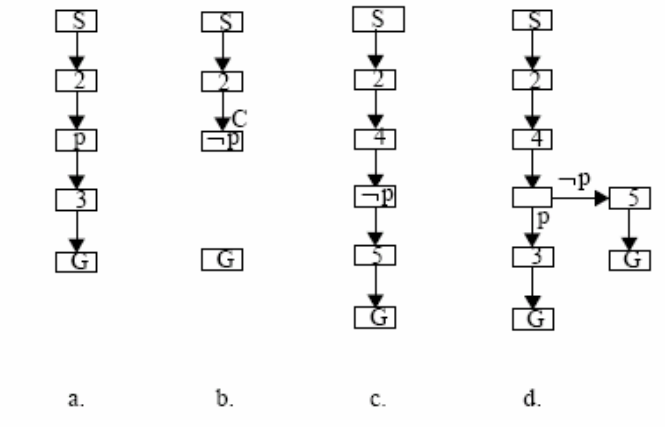

Figure 0-1 Operation of WARPLAN-C. (from [25])

4.1.2 Operation of WARPLAN-C

In the above sample the conditional action is shown a block P. During the first pass the planner creates the plan assuming that P has only one possible outcome. The resultant plan is shown in the figure 4-1.a. The other possible outcome of this action is ~P shown in figure 4-1.b. The planner is invoked to generate a new plan for the ~P part as shown in figure 4-1.c and the result is combined with the actual plan to form the final conditional plan as shown in figure 4-1.d.

4.2 The World of WARPLAN-C

The WARPLAN-C[26] planner defines and operates in a world consisting of a set of rooms, boxes, light switches and the robot. The world and its terrain are described by means of simple prolog facts and rules.

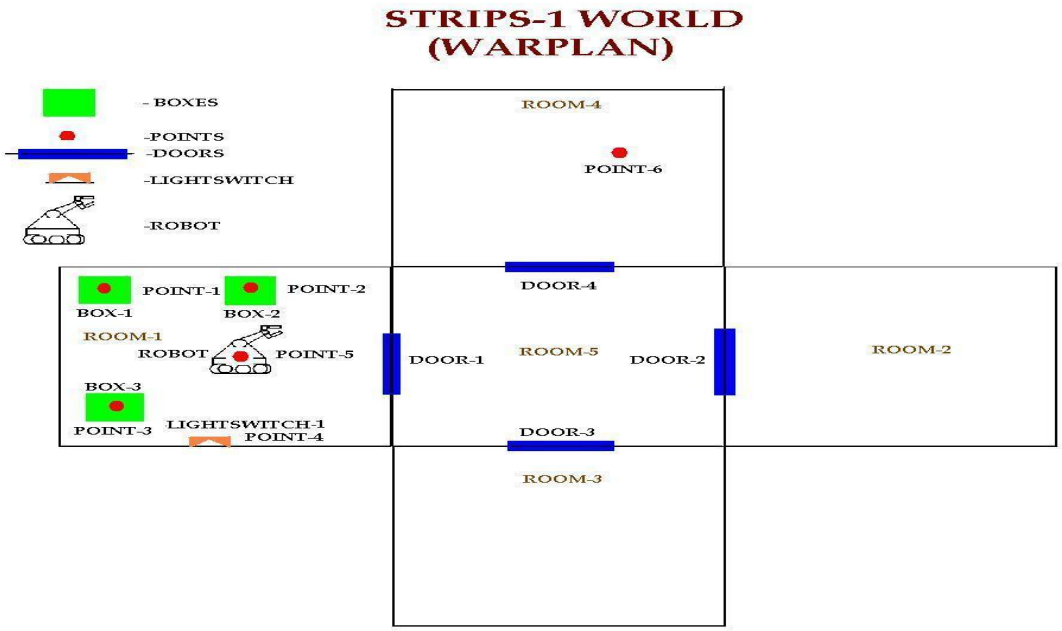

Figure 0-2 Blueprint of the STRIPS-1 world.

The basic blueprint of the World as defined by the planner is given above, along with the initial/default position of the objects. Each object in the world has a set of constraints associated with it. These constraints are represented as simple prolog facts. The planner considers these facts before the generation of the plan. The planner follows a closed world assumption.

4.2.1 Basic Architecture of a Planner

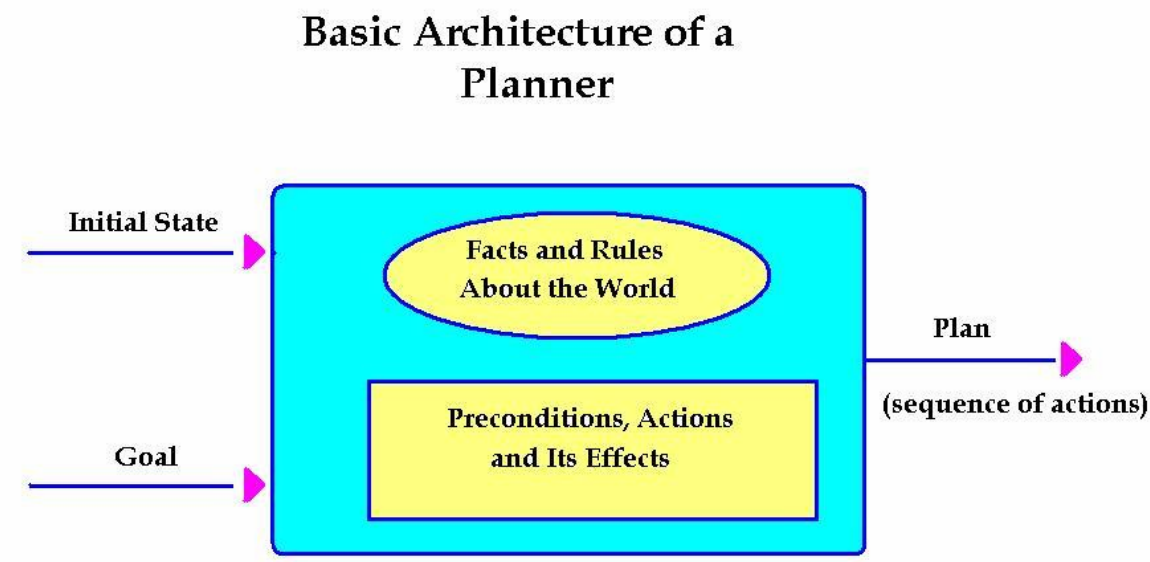

Figure 0-3 Basic Architecture of a Planner

The planner should be provided with two inputs, the initial state and the goal

4.2.2 Sample Initial State and Explanation

We provide the initial state to the planner in form of prolog facts defined through the predicate ‘given()’. This predicate is used to describe the various elements of the world and their respective locations. This predicate is also used to provide the planner with

the initial status of elements before the generation of the plan. A sample set of initial states are given below

```
given( start, at(box(N), point(N))) :-range(N,1,3).
      given( start, at(robot,point(5))).
      given( start, inroom(box(N),room(1))) :- range(N,1,3).
      given( start, inroom(robot,room(1))).
      given( start, onfloor).
      given( start, status(lightswitch(1),off)).
```

The above sample defines the state of the world before the generation of the plan. This state is called as the initial state or the ‘start’ state. According to the above start state:

1. The world consists of three boxes box(1),box(2) and box(3), which are placed at points point(1),point(2) and point(3) respectively.
2. Line two of the code states that the initial position of the robot is at point(5).
3. Lines three and four of the code states that all the boxes and the robot is at room(1) during the start state.
4. According to line 5 and line 6, the robot is on the floor and the light switch is off respectively during the start state.

4.2.3 Representing the Architecture

The architecture of the world as defined by figure 1.0 is represented using the predicate connects1(), as follows.

```
connects1(door(N),room(N),room(5)) :- range(N,1,4).
```

Explanation:

The above code states that room 1, room 2, room 3 and room 4 are connected to room 5 through the doors door 1, door 2, door 3, and door 4 respectively. The world can be expanded by adding more connects1() predicates to the code.

For example room 6 can be added as follows

connects1(door(5),room(6),room(4))

Limitations:

In spite of the ability to expand the world, the planner3D does not allow the user to expand, in order to keep the prototype fair and simple. The methods proposed to accomplish it are explained in the last chapter of the literature.

4.2.4 Universal Facts

The universal facts about the various objects of the world which are set to be always true can be represented using the ‘always’ predicate as follows. These are the rules that (should) always hold.

always(connects(D,R1,R2)) :- connects1(D,R1,R2).

Explanation:

‘connects1’ states that Room R1 is connected to Room R2 through the door D, which implies that door D connects Room R1 and R2.

always(connects(D,R2,R1)) :- connects1(D,R1,R2).

Explanation:

This defines the obvious rule of mutual connectivity; it states that if Room R1 is connected to Room R2 thru door D, then Room R2 is connected to Room R1 through the same door D.

always(inroom(D,R1)) :- always(connects(D,_,R1)).

Explanation:

This states that if Door D is in Room R1, it provides a means of connectivity between R1 and some other room.

```
always( pushable(box(_))).
```

Explanation:

All boxes belong to the set ‘pushable’. This set consists of the list of objects that can be transported or moved.

4.2.5 Enforcing Restrictions

The always() predicate can also be used to enforce certain restrictions to the user as desired by the programmer. The programmer can state his desirability about the world which cannot be manipulated by the user. The facts represented through the always() predicate cannot be altered using the given() predicate. Few examples for enforcing restrictions are given below.

```
always( inroom(lightswitch(1),room(1))).
always( at(lightswitch(1),point(4))).
```

Explanation:

1. Lightswitch1 is always located in room 1.
2. Lightswitch1 is always located at point 4.

4.2.6 Actions, Preconditions and Effects

The planner needs to have a set of actions that can be performed by the agent and the preconditions that should hold for the current state of the world which allows the agent

to perform that particular action. These actions and the corresponding preconditions are provided by means of the ‘can()’ predicate as follows.

can (action(), precondition1 * precondition2 *
precondition 3*……..)

Example:

can(goto2(X,R), inroom(X,R) * inroom(robot,R) * onfloor).

The above example states that for the robot to perform the action goto2(X,R) i.e. to go near an object X in room R, both the object and the robot should be in the same room R. The robot should be on the floor. Every action performed by the agent causes a state transition in the virtual world. These transitions can be termed as effects associated with that particular action. Each action is associated with its corresponding effect by means of add(), del() and moved() predicates.

A state with respect to WARPLAN-C can be defined as a set of predicates which describes about the current status of various elements present in the world. The set can be manipulated by the addition and deletion of the these predicates, which is done by using add() and del(). In addition the moved() predicate allows us to keep in track of the mobile/transportable objects such as the robot and the boxes. For example the effects of the goto2(X,R) would be

add(nextto(robot,X),goto2(X,_)).
moved(robot, goto2(_,_)).

The above code states that the robot has moved from its initial position and is right next to the object X. The del() is not used here as there is no need for deletion of elements from the set.

4.2.7 The Goal and the Plan

The goal can be considered to be a set of predicates which represents a full or partial state. The Goal to the planner is entered as follows:

```
plans( status1 * status2 * status3…., start).
```

The planner is activated by calling the plan predicate. The plan predicate prints out the plan as a sequence of actions that should be performed in order to achieve the goal. The output plan for achieving the goal (status(lightswitch(1),on)) would be:

```
start';'
goto2(box(1),room(1)';'
pushto(box(1),lightswitch(1),room(1))';'
climbon(box(1))';'
turnon(lightswitch(1))'.'
```

4.3 Incorporating WARPLAN-C with PLANNER3D

This section describes briefly about the construction of the prototype PLANNER3D.

4.3.1 Overall Architecture of Planner3D

Planner3D is an interactive 3D system built completely using JINNI3D. JINNI3D as an extension of JINNI employs JAVA3D as the 3D engine. Planner3D uses WARPLAN-C as its intelligent backend. Planner3D interacts with the user by means of a simple Natural Language interface and employs speech synthesis along with the 3D visual output.

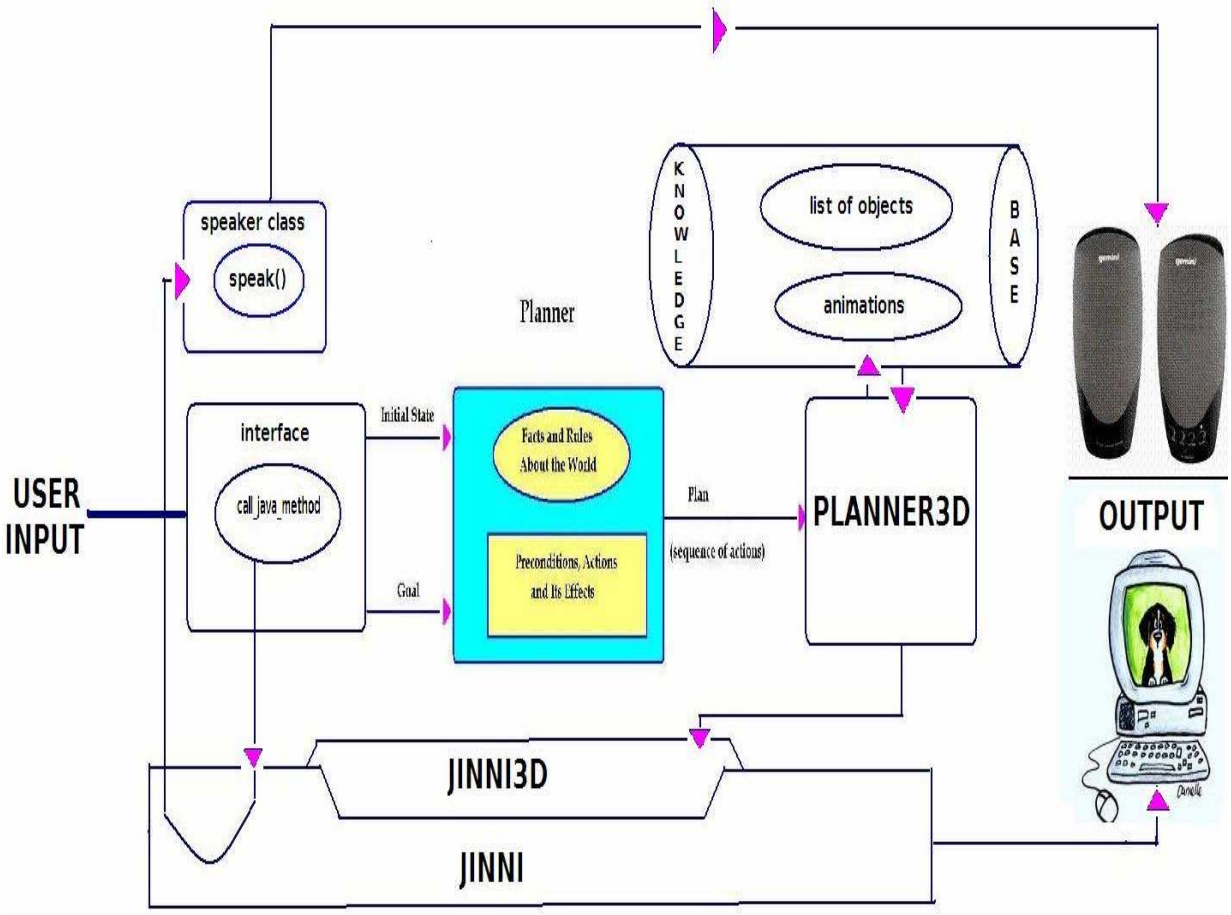

Figure 0-4 The overall architecture of Planner3D

4.3.2 Knowledge Base

The system consists of a knowledge base. The knowledge base has an exclusive collection of models. Based on the generated plan, the planner 3D decides and selects the set of objects to be placed in the 3D world. The knowledge base also consists of the list of animations for all possible actions generated by the planner.

4.3.3 Planner

The Planner (WARPLAN-C) gets the goal and the initial state as inputs via the user interface, and generates the sequence of actions that has to be performed in order to

achieve the goal and passes it over to the Planner3D system.

4.3.4 Interface

The interface consists of a simple Natural language interface. It employs a shallow parser, which matches the input to the goal. It also assists WARPLAN-C through the assertion of the current status of various objects present in the world and provides that in form of the initial state. It also handles the speech synthesis part of the system. The operation of the interface is elucidated briefly later.

4.3.5 PLANNER3D

The Planner3D holds responsibility for the creation of the 3D world. It then identifies the objects that needed to be placed on the world depending upon the plan and positions them appropriately. Planner3D associates the sequence of actions generated by the planner with animations with the help of the Knowledge Base and renders them in the 3D world via JINNI3D. The operation of the system can be explained better by using examples.

4.4 Creation of PLANNER3D World

This section describes about the world in which PLANNER3D operates and also about the various components of the world.

4.4.1 The World of Planner3D

The creation and rendering of the 3D world is somewhat independent to the operation of

Planner3D. The world consists of rooms and doors and these parts of the world are static i.e. cannot be moved or transported to a different place. The default architecture of the world as defined by WARPLAN-C consists of five rooms. For the prototype, we assume the world to be consisting of just three rooms namely room1, room4 and room5 due to the constraints of visibility in 3D. It is to be noted that the tool used for 3D rendering is JINNI3D and it is in development stage and once its development is completed, the above problem can be solved by employing the technique of view point animation.

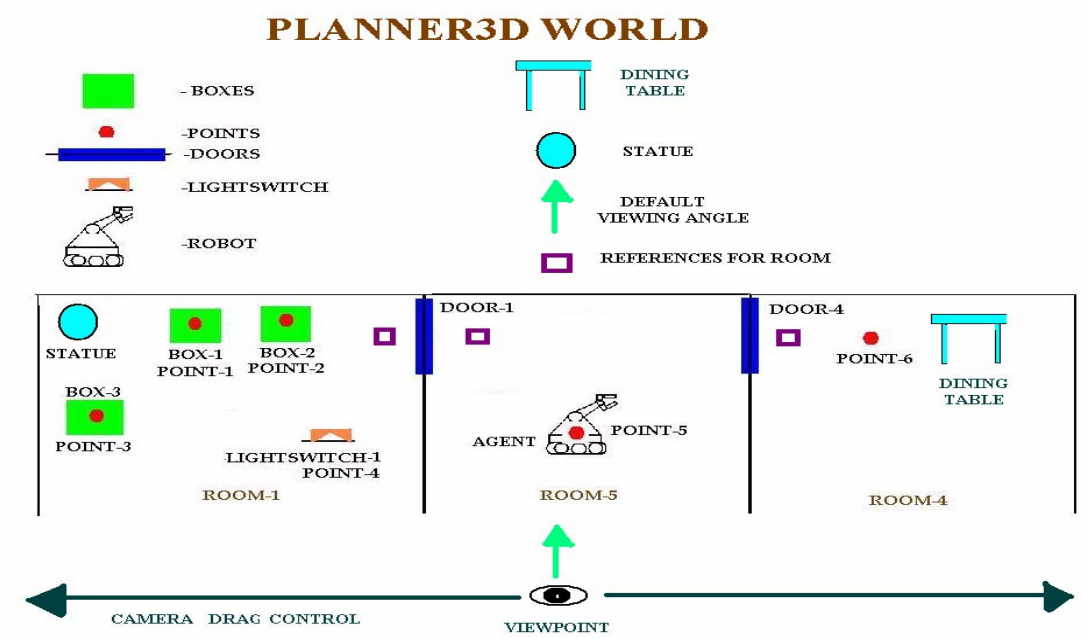

Figure 0-5 Architecture of Planner3D world.

4.4.2 Planner3D Objects

The objects in the world of Planner3D can be categorized into four different types

i. Immutable Objects

ii. State Based Objects

iii. Movable Objects

iv. Locators

4.4.2.1 Immutable Objects

The objects in the World which can neither be moved nor interacted are categorized as immutable objects. The walls of the rooms and decorative materials such as a statue or a poster on the wall can be classified as immutable objects and Planner3D doesn't have any sort of direct control over them. Any variations to these objects can only be caused by direct manipulations to the code. The user version of Planner3D does not allow the user to tamper the code.

4.4.2.2 State Based Objects

The objects in the world which allows limited or no interaction and restricts the interaction to be within a Boolean or a certain number of finite states are categorized as State Based Objects. Objects such as lights, light switches and doors are examples of state based objects. For example, the user can interact with the light switch by turning it on or off and can open or close the door but, can not transport them to a different location.

4.4.2.3 Movable Objects

All objects that allow the user to have a relatively high level of interaction are categorized as Movable objects. These objects belong to the set 'pushable()'. In our prototype, boxes are the only objects which belong to this set and allow this sort of interaction. Theoretically the agent can also be considered to belong to this set.

4.4.2.4 Locators

WARPLAN-C identifies the location of various elements using points. These points should be considered as invisible references to locations. These points can neither be moved nor interacted with. Hence, these points belong to the set of immutable objects, which act as locators. Any number of points can be created in the 3D world. Each point is associated with a distinct x,y,z position in 3D world. Then, any object that is placed in that particular point can be referenced using it. As these points are present only for the comfort of the planner, they are not rendered in 3D world. A sphere is used to represent a point in 3D world and is usually hidden using the hide_agent() method. Likewise, the rooms in the 3D world are associated with the color cubes which act as the reference for the rooms.

4.5 Objects and Animations

This section exhibits the way in which object association and animations can be done using PLANNER3D.

4.5.1 Object Association and Mapping using the Knowledge Base

WARPLAN-C generates the plan and sends the plan to Planner3D. The plan is parsed and for each object in the plan, Planner3D matches it with its corresponding 3D image in the Knowledge Base and places it in the 3D world. This method is employed and experimented successfully. But the current prototype doesn't exploit much of this feature of planner3D in order to achieve a command based 3D interactive system which works incrementally. The following code quoted from Planner3D shows how an object in

the world of WARPLAN-C planner is associated with its corresponding 3D image.

```
add_all_objects(World,
[
        % Name in Planner = JINNI3DHandle
        box(1)=Cube1,
        box(2)=Cube2,
        box(3)=Cube3,
        robot=Man,%Man
        lightswitch(1)=Pyram,
        door(1)=Door1,
        door(4)=Door4,
        point(6)=Point6,
        room(1)=Room1,
        room(4)=Room4,
        room(5)=Room5
         ]):-
        color_cube(World,Cube1),
        set_x(Cube1,-0.5),
        set_z(Cube1,-4.0),
    ……..
```

4.5.1.1 Add_all_objects method

The list of all Movable, State Based and Reference objects are to be passed as parameters to this method. The Add_all_objects method associates each object in that list to a local variable, these local variables are later associated to their corresponding 3D element. Later, by using the local variables, the method positions each object to its default or start position. For example in the above code, the method associates box(1) to the local variable Cube1 and later matches Cube1 to a 3D color cube and positions the color cube in the Planner3D world. It is obvious that Immutable objects require neither associations nor mappings as they are totally independent to the operation of the planner and the agent and doesn't allow any sort of interaction.

4.5.2 Set of Animations in the Knowledge Base with respect to WARPLAN-C

The system requires a complete set of animations which directly corresponds to the set of all possible actions enlisted by the planner. The two sets mentioned above are considered to have a one to one mapping. The animations are grouped and listed using the perform_one predicate which has the following syntax:

```
perform_one(<action>),<Universe>,<list of objects>):-!,
                                    <Script for animation>).
```

The operation perform_one can be better explained with an example as below:

```
perform_one(goto2(box(N),room(K)),_World,Obs):-!,
        once(member(box(N)=BoxHandle,Obs)), %Box
        once(member(robot=RobotHandle,Obs)),%Man
        write(['I',will,move,near,to,BoxHandle,in,room(K)]),nl,
        move_slowly(RobotHandle,BoxHandle), %Moving Man towards Box
        println(done).
```

Explanation:

The perform_one predicate defines the set of operations that has to be performed by Planner3D in the 3D world. For example, the above predicate lists the script for animation which corresponds to the action goto2(box(N),room(K). The action states that the agent should move from its current position and reach the position of the box(N) which is in room(K). For achieving this, we get Local Handles for the plan elements namely the robot (Humanoid) and the Box (Color cube) and employ the move_slowly() which in turn is an exclusive inbuilt Planner3D predicate which moves the robot towards the position of the box(N) with respect to their 3D locations.

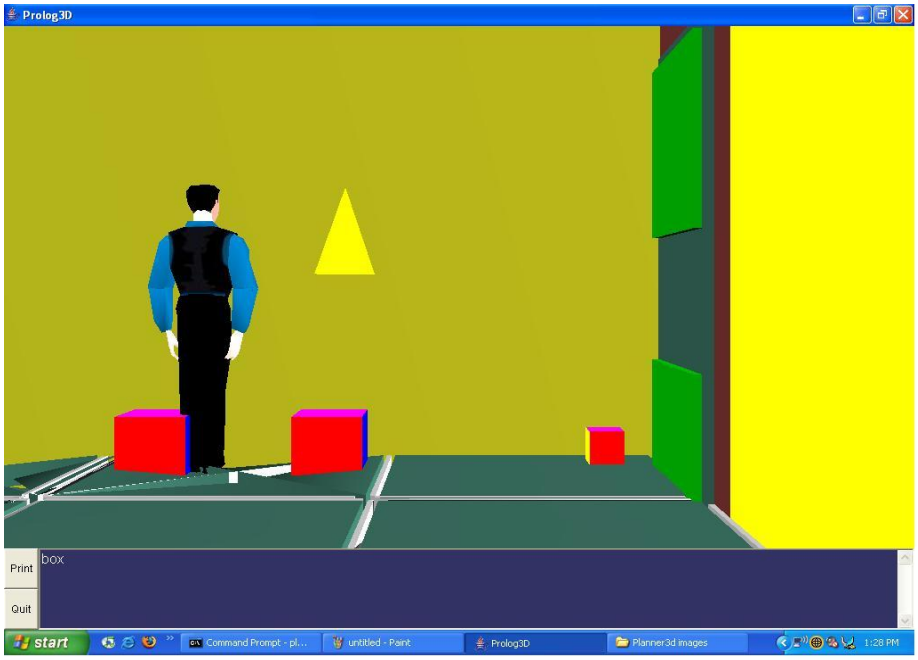

Figure 0-6 Position of agent after goto2(box(1),room(1)).

The script inside the perform_one predicate should not use any direct reference to Movable, State based and Reference objects in order to avoid confusion and achieve independency. All objects should be referenced via the 'Obs' variable which contains the list of all interactable objects present in the 3D world. Likewise, script for animation should be present for all possible actions with respect to the planner.

4.5.3 Planner3D Tools

Several tools have been provided in Planner3D to assist with the animation of objects, some of which are inbuilt predicates present in JINNI3D, while the rest are exclusive to planner3D. Some of the important tools are set_default_dir, go_default, move_slowly, move_from_to, push_from_to and many more.

4.5.3.1 Set_default_dir()

WARPLAN-C does not have the sense of direction, the agent is facing. This tool aids in turning the agents to their default direction before the animation or wherever necessary. The set_default_dir() has the following syntax

set_default_dir(<Current Directional value>,<List of Objects>)

4.5.3.2 go_default method

This method animates the movement of agent from its current position to its default position and can be used whenever necessary. The go_default has the following syntax

go_default(<agent Handle>,<current room number>,<list of objects>)

4.5.3.3 move_slowly method

This method is comparatively a complex method which makes use of several other tools to achieve the task of locomotion of the agent. In order to move the agent from point A to point B, it gets the corresponding (x1,z1) and (x2,z2) of A and B respectively. Then, it sets the agent to its default direction by using set_default_dir() method. It makes a call to the turn_test(), which turns the agent towards the direction of the vector V connecting A and B. Then, move_slowly makes a call to move_from_to method which measures the magnitude of the vector V. move_from_to also calculates the appropriate value for incrementing x,z position of the agent so as to achieve a seamless animation. Then the x,z position of the agent is slowly incremented from point A towards point B.

The syntax of move slowly is as follows

move_slowly(<Agent Handle>,<from position>,<to position>)

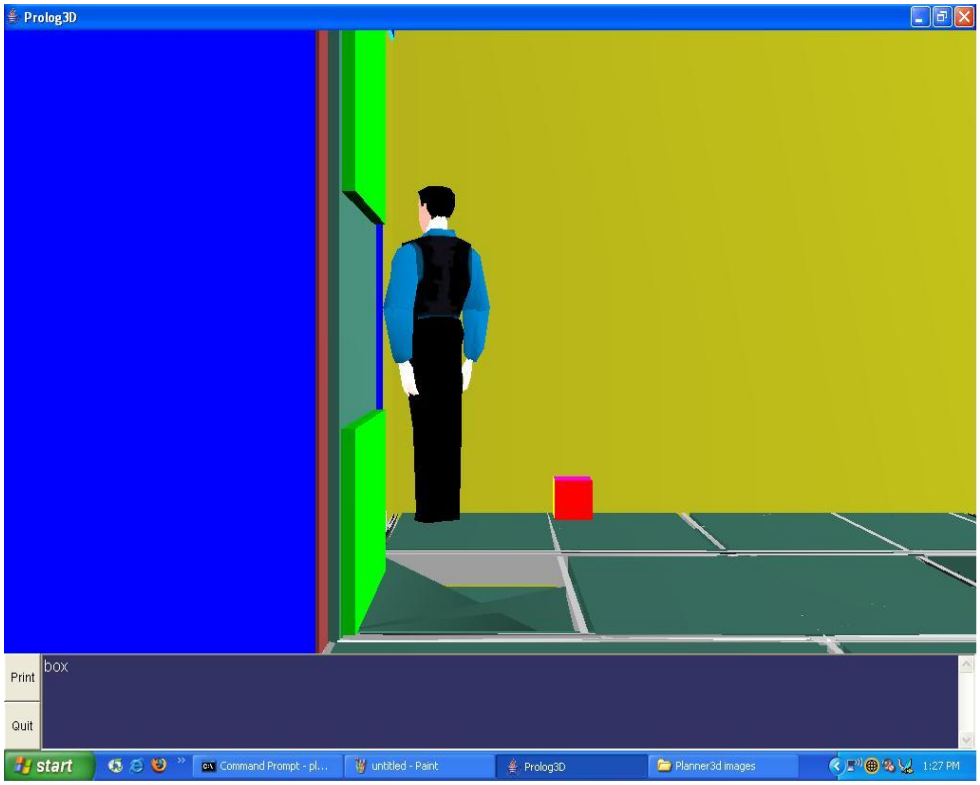

Figure 0-7 The snapshot of the agent moving towards the door

4.5.3.4 push_slowly method

push_slowly() works similar to the move_slowly() predicate. In addition, it takes into consideration, the object which is pushed and increments the position of the object with respect to the agent.

The syntax of push slowly is

push_slowly(<Agent Handle>,<pushed object handle>, <from position>,
<to position>)

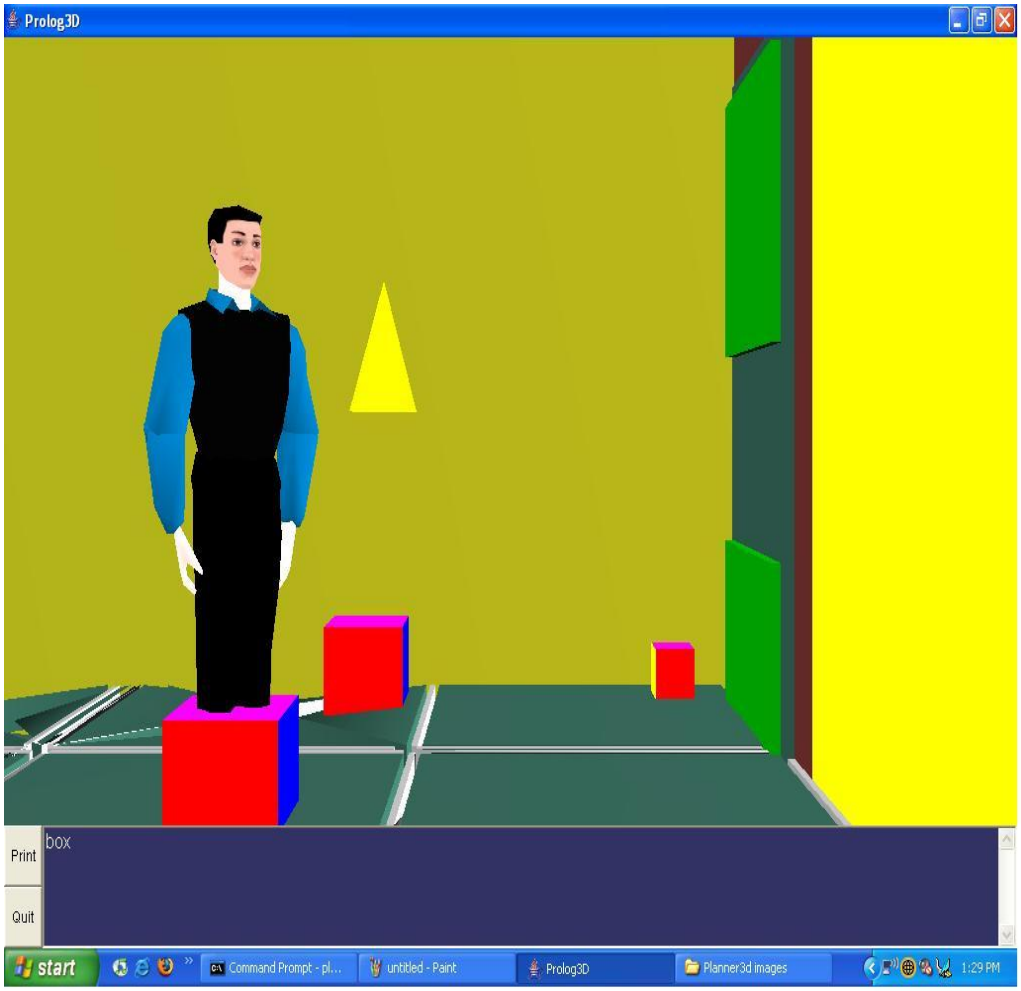

Figure 0-8 A snapshot of the agent pushing box(1) towards lightswitch(1)

4.6 Rendering the Plan in Form of 3D Animations

In this section, we show how the plan generated by WARPLAN-C is rendered as 3D animations.

4.6.1 Conversion of Plan into a List

WARPLAN-C develops a linear plan but does not employ a proper way of storing the created plan, rather it just prints them out in sequence. The Plan needs to be packaged into a Planner3D recognizable format. This is done by converting the plan in form of a

list using the plan2list() predicate. The plan2list predicate adds each action generated in sequence by WARPLAN-C to a list. The created list consists of the entire plan but in reverse, hence it is inverted again before passing it to Planner3D for the generation of 3D animation.

4.6.2 Execution of the Plan

As we now have the entire sequence of action in the form of a list, the job becomes simple from here. The interface waits for an input from the user, with the help of the shallow parser of the interface; the input is linearized into a set of goal elements and fed to WARPLAN-C. The plan generated by WARPLAN-C is acquired in the form of a list and passed to Planner3D. Planner3D maps every action in the list to its corresponding perform_one() in the Knowledge Base and executes the corresponding sequence of animation actions defined by that particular perform_one. Thus, the resultant plan is rendered in form of agent animation in the 3D world.

4.7 The Interface

We briefly describe the components of the interface and their operations in this section of the literature.

4.7.1 Shallow Parser

The interface consists of a shallow parser. The interface prompts for user input. The shallow parser associates the user input, which is in the form of a string, to a numeric value. The numeric value is then mapped with appropriate goal set and a call for the

planner is made. Then the acquired plan is rendered by a call to the perform_all predicate, which initiates Planner3D.

4.7.2 Keeping Track of States

WARPLAN-C is a full order linear planner, which requires the state of all objects to be clearly defined before the generation of the plan. We achieve incremental planning using a full-order planner by keeping track of the states and manipulating the initial state based on that. The interface holds responsibility and makes calls to predicates like get_room(),etc. These predicates acquire the states of every object with respect to the 3D world and provides that state as the initial state to the planner.

4.7.3 Linear and Query Commands

Planner3D is also capable of handling simple linear commands. There is no need to use the planner for such commands. They can be handled independently without using WARPLAN-C. Interface needs to keep track of such commands and map it to appropriate animations. The system also allows the user to query about the status of the agent. Those commands are handled by the interface directly without using neither Planner3D nor WARPLAN-C. The interface has the capability to directly query the knowledge Base and get the results.

(examples) (Query Command) Where are you?

(linear command) turn left

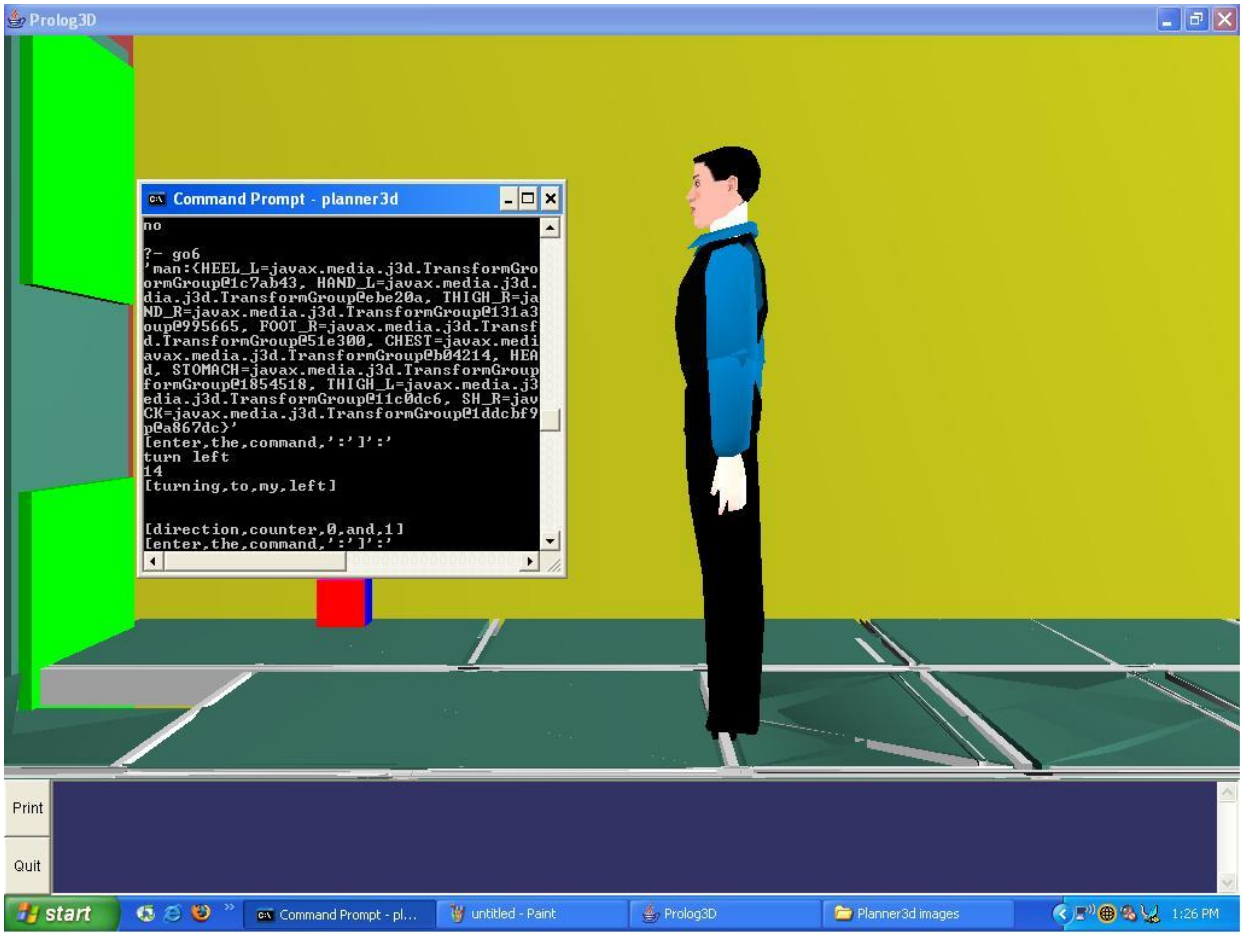

Figure 0-9 Snapshot of the agent for the command turn left.

4.7.3.1 Sample Output

The initial state of the agent is:

< the agent is on the floor and in room1>

[Enter The Command]:

where are you?

[I am in Room 5]

[Enter The Command]:

switch on lightswitch

The first command does not require a planner. So the interface queries through the get_room() and returns the result. For the second command, the planner is initiated and the plan is generated as follows:

```
[ start;
  goto2(door(1),room(5));
  gothru(door(1),room(5),room(1));
  goto2(box(1),room(1));
  pushto(box(1),lightswitch(1),room(1));
  climbon(box(1)),
  turnon(lightswitch(1)).
  ]
```

Figure 0-10 The snapshot of the world after the completion of the task

Now, the state of the world is changed and the planner keeps track of the agent. This can be proven by querying the agent about its current state

[Enter The Command]:

where are you?

[I am in room 1]

For the next command, the current state of the agent is given as the initial state to the planner. So, now the planner can start planning from that particular state by assuming it as the initial state.

4.7.4 Making Use of Points

This is one of the crucial part of the research work, which enables the system to be used for story generation. The planner WARPLAN-C encourages the user to place any number of reference objects called Locators (points) in the virtual world. We exploit this feature of the planner by placing imported models into the Planner3D world and associating them with one such locator. As far as WARPLAN-C is concerned, it is unaware of the model present at the location. Mapping the model to the corresponding locator is concluded by the interface.

(For example) [Enter The Command]:

go near dining table

The Interface has to map dining table to point 6 and send the following goal to the planner

plan_from(at(robot,point(6)),Plan)

Likewise, we can create and place any number of objects in the world and can reference them with the use of points. In the planner3D prototype, we have shown only the sample of the dining table, which can be inflated to numerous objects as necessary.

The resultant Plan would be

start;
goto2(door(4),room(5));
gothru(door(4),room(5),room(4));
goto1(point(6),room(4)).

It is assumed that robot is in its default position room(5).

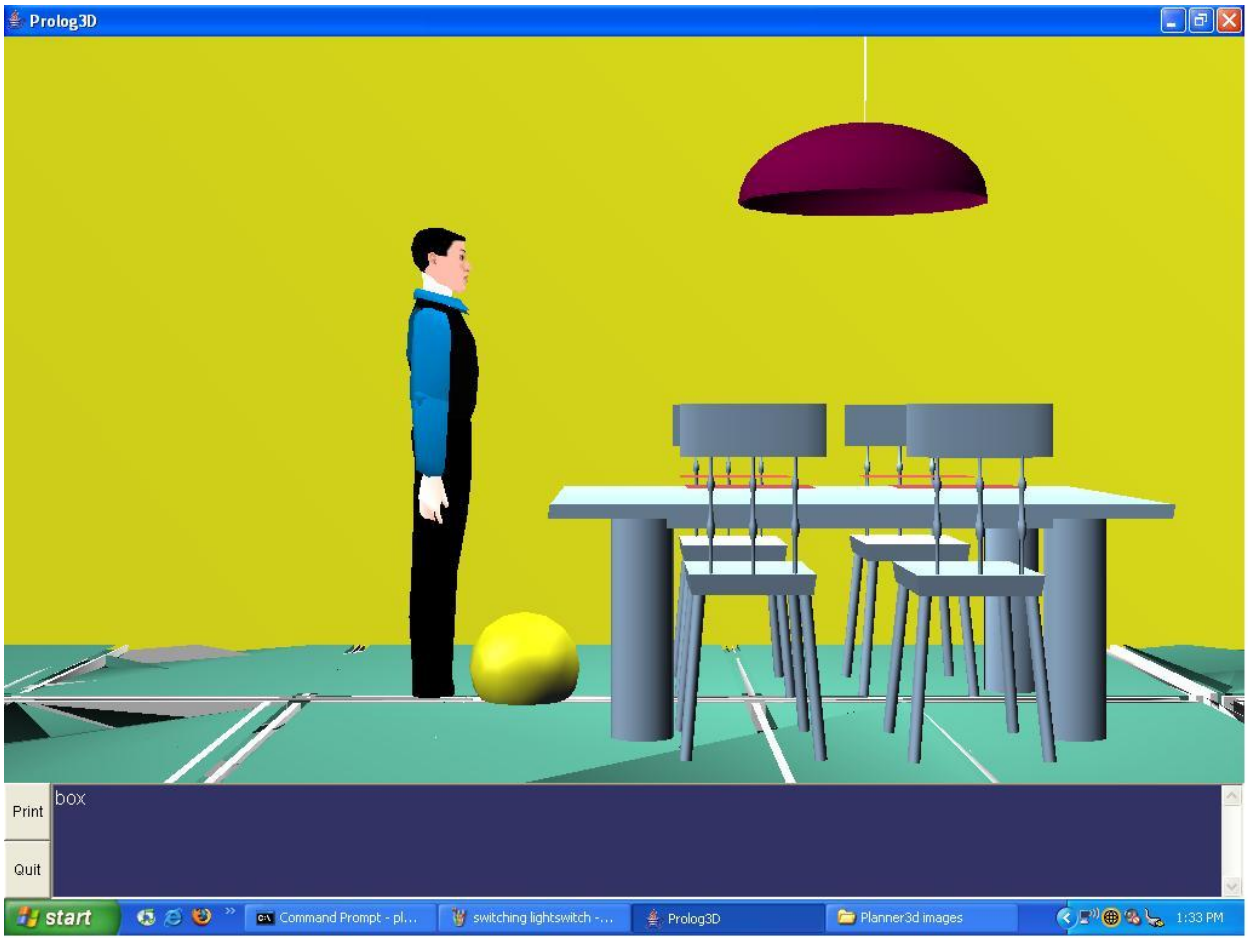

Figure 0-11 Snapshot of the agent near dining table (point6) is given below

4.8 Speech Synthesis

As a supplement to the above system, we present a speech synthesizer built using JAVA speech API [28] and the FreeTTS[29] speech engine. The synthesizer [29] is

provided in form of a JAVA class called speaker. Both the API and the FreeTTS engine are open source and distributed freely.

Table 4.1

Class Summary	
Speaker	The *Speaker* Class provides primary access to speech synthesis capabilities. Through JAVA Speech's Synthesizer

4.8.1 Public Class Speaker

The Speaker Class provides primary access to speech synthesis capabilities. The primary functions provided by the Speaker Class are the ability to speak text, Control the speech properties such as pitch, etc and control an output queue of objects to be spoken. The static methods of the class can be called directly without instantiation. The class can also be instantiated in four different ways by using any of the three provided Constructors or simply by creating a normal instance.

4.8.2 Voice Output

The basic function of the Speaker class is to apply a text to speech algorithm for the provided string arguments and as a result produce voice output. Plain text is spoken using the static speak method. The Speak method calls the speech engine and places the text to be spoken in the synthesizer queue. A synthesizer [29] is monolingual (it speaks a single language), so the text should contain only the single language of the synthesizer. The only available language for now is English. Each object provided to the synthesizer is spoken independently. Sentences, phrases and other structures should not span multiple calls to the speak methods.

4.8.3 Speech Synthesis in Planner3D

We exploit the ability of calling JAVA methods and classes using JINNI. It allows us to call static methods of JAVA classes without instantiation, through the following syntax

call_java_class_method('<public class name>',

<method with parameters>,<result>).

We utilize this and call the speak() method of the Speaker class and pass the text to be spoken in order to attain speech output .

call_java_class_method('Speaker',speak('Welcome'),_).

The above call to the speaker class initializes the speech engine and prompts the speech-bot kevin16 to pronounce the string 'welcome'. It is to be noted that not all methods of speaker class are static, some methods strictly require instantiations due to security reasons. The method summary of the Speaker Class is as follows.

Table4.2

Method Summary	
void	getSynth() gets the Synthesizer ready to speak by allocating it and resuming the speech engine.
void	printProperties() Prints the Current settings of synthesizer properties such as voice, pitch range, pitch, speaking rate, volume and gender.
float	setSpeakRate(float SpeakRate) Sets the current speaking rate to the given value and returns it.
float	incSpeakRate (float incRate) Increments the current speaking rate by the given value and returns the current value.
float	decSpeakRate (float decRate) Decrements the current speaking rate by the given value and returns the current value.

Method Summary	
float	setPitchRange(float PitchRange) Sets the current PitchRange to the given value and returns the current value.
float	incPitchRange (float incRate) Increments the current Pitch Range by the given value and returns the current value.
float	decPitchRange (float decRate) Decrements the current Pitch Range by the given value and returns the current value.
float	setPitch(float Pitch) Sets the current Pitch to the given value and returns it.
float	incPitch (float incRate) Increments the current Pitch by the given value and returns the current value.
float	decPitch (float decRate) Decrements the current Pitch by the given value and returns the current value.
float	setVolume(float Volume) Sets the current Volume to the given value and returns it.
float	incVolume (float incRate) Increments the current Volume by the given value and returns the current value.
float	decVolume (float decRate) Decrements the current Volume by the given value and returns the current value.
void	ListVoices() Lists all available voices of the synthesizer.
void	setVoice(String S) checks for availability and sets the current voice to the given value.
void	speak(String S) puts the given string S in the engine queue and thus makes it speak.
String	setGender (String) sets the current gender by setting the property values to appropriate levels.
String	getGender (float decRate) returns the current gender value.
void	Deallocate() Deallocates the Resource.
void	finalize ( ) Deallocates the resource during object destruction.

Table4.3

Constructor Summary	
Speaker()	Default Constructor
Speaker(String S)	Constructor for setting voice and allocating synthesizer and resuming the engine, so everything is set to speak.
Speaker(String S,String Gender)	Constructor for setting voice, gender and allocating synthesizer and resuming the engine, so everything is set to speak.
Speaker(String S, Float SpeakingRate, Float pitchRange, Float Pitch, Float Volume)	Constructor for setting voice, Speaking Rate, Pitch Range, Pitch, Volume and allocating synthesizer and resuming the engine, so everything is set to speak.

CHAPTER 5

CONCLUSION

The ability to generate narrative is of importance to computer systems that wish to use story effectively for entertainment, training, or education. The ability to computationally generate stories can result in computer systems that interact with humans in a more natural way. To date, story generation systems have used autonomous agent technologies and single authoring agent approaches. One other approach to generate narrative is to use planning.

This thesis presents and investigates several ideas for creation of an Interactive 3D graphics system comprising of a planner which acts as the intelligent back-end for a centralized agent-based story generation system.

We propose an idea for an interactive plan-based narrative virtual environment. The proposed scheme is then formalized into architecture. The essential components of the architecture encompasses of a natural language interface, a planner and a 3D engine. The task of integrating the planner and the 3D engine was identified as the focus of the research. To complete the task, various models of implementations were inspected and the results were scrutinized.

A functional prototype is built based on the results of the above research. The operation of the prototype demonstrates the accomplishment of the architecture proposed for creating the back-end for interactive fiction and story generation systems.

CHAPTER 6

FUTURE WORKS

In this chapter we describe about the various additions and modifications that can be made to the prototype to improve its performance and to make it more user friendly.

6.1 Rest of the Blocks

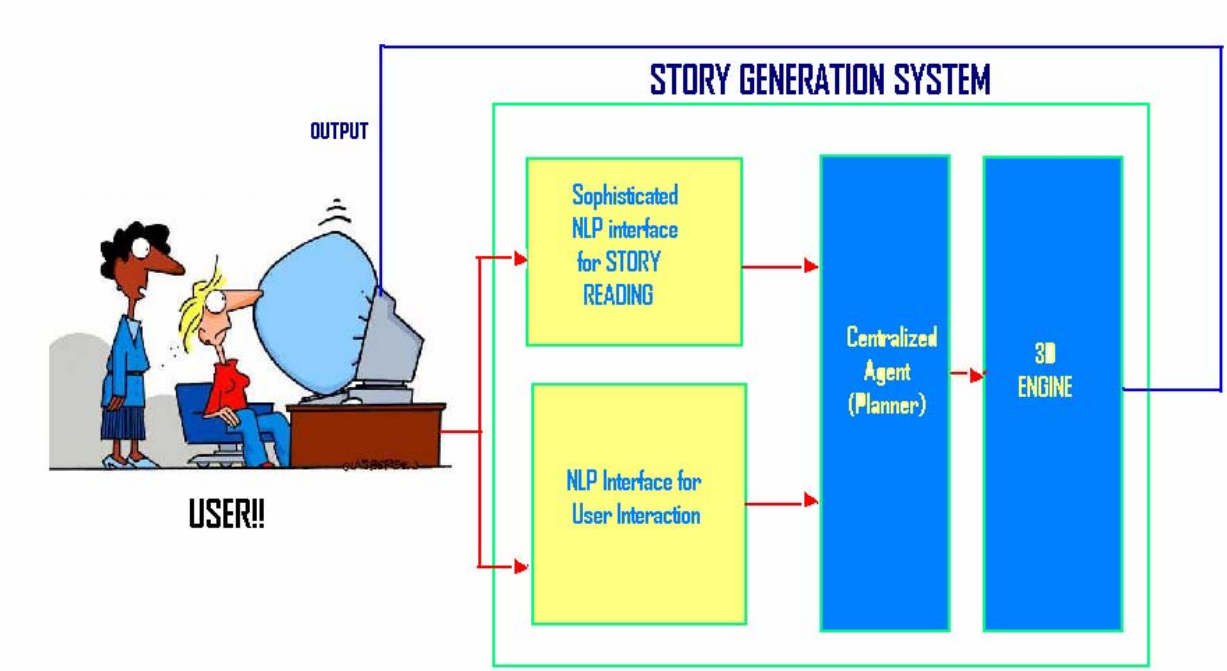

Figure 0-1 Story Generation System (picture from chapter1)

The prototype for back-end intelligence for a story generation system is complete. The performance can be enhanced, by linking the prototype to a more advanced Natural Language Interface. The task of achieving this is highly prioritized and acknowledged as the most essential addition to the above system.

6.2 Make the Shallow Parser Deep

We use a shallow parser in place of a sophisticated natural language system which is

mandatory for story generation. The shallow parser does very basic string comparison operation and has very less fault tolerance. This part of the prototype could be strengthened in the future by the use of a proper Natural Language interface which is capable of doing more than just string comparison.

6.3 Strengthening the 3D API

The 3D engine of the prototype is controlled via JINNI3D API. This API already has within itself incredible features and innovative tools to support agent animation. JINNI3D is in the process of improvisation to support animations that are more complex. The applicability of JINNI3D can be amplified by the addition of archive containing a vast collection of 3D models with defined behaviors.

6.3.1 Collision Detection

The JINNI3D API can be improved by the addition of a basic collision detection [31] algorithm for checking for intersection between two given solids. This can be taken to the next level by making the algorithm more complex by incorporating the behaviors of objects in physical world like in WorldUp which will allow us to calculate trajectories, impact times and impact points, etc.

6.3.2 View Point Animation

View point animation techniques allows us to animate the viewpoint of the 3D universe which is static in the current version of the prototype. The viewpoint can be made to set focus on a particular mobile agent and the agent’s motion can be tracked by animating

the viewpoint with respect to the mobile agent. This removes the visibility constraint described in chapter 4 and allows us to expand the world infinitely as desired.

6.4 Multi Agent Planning Approach

The prototype can be further developed to support multi agent planning by use of a full order or a partial order multi agent planner, which would give groundbreaking results for story generation. The current planner can also be improvised by extending the knowledge set of the planner by addition of complex actions.

6.5 Let Them Speak

The Speech API (Speaker class) can be extended into a fully capable speech engine, which has a wide variety of voices and sounds human rather than the aberrant computer generated voice. This would supplement the multi-agent environment and allows the user to perceive the communication of two characters in the virtual world as voice outputs, which in a natural way increases character believability.

BIBLIOGRAPHY

[1] 1999. In M. Mateas and P. Sengers (Eds.), Working notes of the Narrative Intelligence Symposium , AAAI Fall Symposium Series. Menlo Park: Calif.: AAAI Press.

[2] Cullingford, R. SAM. In Inside Computer Understanding: Five Programs Plus Miniatures, Ed. Roger Schank and Christopher Riesbeck. Hillsdale, New Jersey: Lawrence Erlbaum Associates. 1981

[3] Wilensky, R. PAM. In Inside Computer Understanding: Five Programs Plus Miniatures, Ed. Roger Schank and Christopher Riesbeck. Hillsdale, New Jersey: Lawrence Erlbaum Associates. 1981.

[4] Meehan, J. The metanovel: writing stories by computer. Ann Arbor: University Microfilms International, 1977.

[5] Carbonell, J. Subjective understanding: Computer models of belief systems. Ph.D. Thesis, Computer Science Department, Yale University. 1979

[6] Dyer, M. In Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension. Cambridge, MA: MIT Press. 1983.

[7] Kolodner, J. Retrieval and organizational strategies in conceptual memory: a computer model. Hillsdale, New Jersey: Lawrence Erlbaum Associates. 1984.

[8] Schank, R. and Reisbeck, C. (eds). Inside Compute Understanding: Five Programs Plus Miniatures, Hillsdale, New Jersey: Lawrence Erlbaum Associates. 1981.

[9] Mark O. Riedl and R. Michael Young, An Objective Character Believability Evaluation Procedure for Multi-Agent Story Generation Systems International conference on intelligent virtual agents 2005

[10] Mueller, E. Daydreaming in humans and machines: a computer model of the stream of thought. Norwood, New Jersey: Ablex. 1990.

[11] Turner, S. MINSTREL: a computer model of creativity and storytelling. Ph.D. Thesis, Computer Science Department, University of California, Los Angeles. Technical Report CSD-920057. 1992.

[12] Matthew J. Conway, Alice: Easy-to-Learn 3D Scripting for Novices (PDF), 1997,

[13] Seymour Papert, MindStorms: Children, Computers, and Powerful Ideas, Basic Books,New York, 1980.

[14] Alice: Lessons Learned from Building a 3D System for Novices (PDF), Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, Kevin Christiansen, Rob Deline, Jim Durbin, Rich Gossweiler, Shuichi Kogi, Chris Long, Beth Mallory, Steve Miale, Kristen Monkaitis, James Patten, Jeffrey Pierce, Joe Schochet, David Staak, Brian Stearns, Richard Stoakley, Chris Sturgill, John Viega, Jeff White, George Williams, and Randy Pausch, CHI 2000

[15] Sense8 Corporation: www.sense8.com

[16] Randy Pausch, Jon Snoddy, Robert Taylor, Scott Watson, Eric Haseltine, Disney's Aladdin: First Steps Toward Storytelling in Virtual Reality, ACM SIGGRAPH 96 Conference Proceedings, August 1996.

[17] James H. Clarke, Hierarchical GeometricModels for Visible Surface Algorithms, Communications of the ACM, 19(10), October 1976, pp. 547-554.

[18] Andries vanDam, et. al. PHIGS+ Functional Description Revision 3.0, Computer Graphics 22, 3, (July 1988), 124-218.

[19] John Lasseter, Principles of Traditional Animation Applied to 3D Computer Animation, SIGGRAPH 87 Conference Proceedings, pp. 35-44.

[20] G.G. Robertson, S.K. Card, J.D. Mackinlay. The Cognitive Coprocessor Architecture For Interactive User Interfaces. In ACM Symposium on User Interface Software and Technology (Nov. 13-15, Williamsburg, VA), ACM/SIGGRAPH/SIGCHI, 1989, pp. 10-18.

[21] Inference and Computational Mobility with JINNI , Dr.Tarau,Paul,UNT.

[22] JAVA3D API Tutorials by Sun Microsystems JAVA 3D Engineering Team September 200(http://JAVA.sun.com/developer/onlineTraining/JAVA3D/)

[23] Humanoid Animation Working Group (H-Anim) (http://h-anim.org/Specifications/H-Anim1.0/)

[24] Don Brutzman, The Virtual Reality Modeling Language and JAVA Code UW/Br, Naval Postgraduate School Monterey California 93943-5000 USA brutzman@nps.navy.mil Communications of the ACM, vol. 41 no. 6, June 1998, pp. 57-64.

[25] Mark A. Peot and David E. Smith, Conditional Nonlinear Planning, Appears in Proceedings of the First International Conference on Artificial Intelligence Planning Systems, College Park, Maryland, 1992.

[26] Warren, D. H. D., Generating Conditional Plans and Programs, in Proceedings of the Summer Conference on AI and Simulation of Behavior, Edinburgh,1976.

[27] JAVA Speech API Programmers Guide (Sun Microsystems) (http://JAVA.sun.com/products/JAVA-media/speech/forDevelopers/jsapi-guide.pdf)

[28] FreeTTS 1.2 - A speech synthesizer written entirely in the JAVA™ programming language(http://freetts.sourceforge.net/docs/index.php)

[29] Referred from www.wikipedia.org

[30] Birte Lönneker, Jan Christoph Meister, Pablo Gervás , Federico Peinado , Michael Mateas Story Generators: Models and Approaches for the Generation of Literary Artefacts.

[31] from www.binnetcorp.com and www.cs.unt.edu/~tarau (Dr.Tarau's website)

[32] A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality(appeared in the May 1995 issue of IEEE Computer Graphics and Applications) User Interface Group,Computer Science Department,University of Virginia

Bibliography Notes:

- Chapter1: Figure 1.1 is via web from a free distribution 3D gallery.
- Chapter2: Figure2.1 Sample images of Alice 3D is from alice3D community.
- Chapter2: Figure2.2 and Figure2.3 is adapted from [12],[14]
- Chapter2: Figure2.4 is adapted from [15]
- Chapter2: Most part of chapter 2 describes about Alice3D (patented 3D authoring tool). So technical part of the literature is reproduced as such from [12].
- Chapter2: Likewise, for the commercial software WorldUp
- Chapter3: Few parts of section 3.2 (*such as the steps for creation of a simple universe and a sample program*)have been recreated from [22] to compare with JINNI3D. Figure 3.1 is adapted from [22].
- Chapter4: Figure 4.1 is adapted from [25].