AUTONOMOUS AGENTS FOR REAL-TIME ANIMATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

James J. Kuffner, Jr.
December 1999

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jean-Claude Latombe
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Leonidas J. Guibas

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Christoph Bregler

Approved for the University Committee on Graduate Studies:

Abstract

Advances in computing hardware, software, and network technology have enabled a new class of interactive applications involving 3D animated characters to become increasingly feasible. Many such applications require algorithms that allow both autonomous and user-controlled animated human figures to move naturally and realistically in response to task-level commands.

This thesis presents a research framework aimed at facilitating the high-level control of animated characters in real-time virtual environments. The framework design is inspired by research in motion planning, control, and sensing for autonomous mobile robots. In particular, the problem of synthesizing motions for animated characters is approached from the standpoint of modelling and controlling a "virtual robot".

Two important classes of task-level motion control are investigated in detail. First, a technique for quickly synthesizing from navigation goals the collision-free motions for animated human figures in dynamic virtual environments is presented. The method combines a fast 2D path planner, a path-following controller, and cyclic motion capture data to generate the underlying animation. The rendering hardware is used to simulate the visual perception of a character, providing a feedback loop to the overall navigation strategy. Second, a method for automatically generating collision-free human arm motions to complete high-level object grasping and manipulation tasks is formulated. Given a target position and orientation in the workspace, a goal configuration for the arm is computed using an inverse kinematics algorithm that attempts to select a collision-free, natural posture. If successful, a randomized path planner is invoked to search the configuration space (C-space) of the arm, modeled as a kinematic chain with seven degrees of freedom (DOF), for a collision-free path connecting the arm initial configuration to the goal configuration.

Results from experiments using these techniques in an interactive application with high-level scripting capabilities are presented and evaluated. Finally, how this research fits into the larger context of automatic motion synthesis for animated agents is discussed.

Acknowledgments

First, I would like to thank my advisor Professor Jean-Claude Latombe for his helpful guidance and support over the past six years. Through him, I have been introduced to a wealth of new ideas ranging from motion planning to mountaineering. I also gratefully acknowledge Professor Leo Guibas and Professor Christoph Bregler for serving on my reading committee, and for their mentorship during my time at Stanford.

While conducting this research, I had the opportunity to work with and learn from many people. I am grateful to Steve LaValle for allowing me to collaborate with him, and for offering his thoughtful advice and friendship. I thank Brian Mirtich of MERL, whom I enjoyed working with to write a software library based on his thesis research. I thank the team of Yotto Koga, Koichi Kondo, and Jonathan Norton, who made creating the ENDGAME computer animated short so much fun.

The students, staff, and visitors of the Stanford Computer Science Robotics Laboratory have enriched my educational experience immeasurably. In particular, I thank David Hsu, Lydia Kavraki, Craig Becker, Tsai-Yen Li, David Lin, Hector Gonzalez, Mark Ruzon, Diego Ruspini, KC Chang, Scott Cohen, Mehran Sahami, Stephen Sorkin, Joel Brown, Serkan Apaydin, Jing Xiao, Frank Hickman, Jutta McCormick, and everyone in the motion-algo group for their friendship and for the many interesting and insightful discussions.

I thank the Department of Computer Science for providing the facilities for conducting this research. I am very grateful to Professor Mark Levoy, Professor Pat Hanrahan, and the Stanford Computer Graphics Laboratory for allowing me to utilize their rendering hardware and video equipment.

Financial support for this research was provided in part by a National Science

Foundation Graduate Fellowship in Engineering, and MURI grant DAAH04-96-1-007, of which I am very grateful.

I thank Professor Hirochika Inoue and Professor Masayuki Inaba of the University of Tokyo for allowing me to join their robotics research laboratory in Japan upon completion of this dissertation.

I am indebted to the many friends who have helped me during my time at Stanford. In particular, I am grateful for the friendship of my former roommates, Apostolos Lerios, Daisuke Terasawa, and Iacovos Koumas, who survived living with me. Thanks also to Michael Svihura, Salli Tazuke, and Phil and Mariko Yang who have been such supportive friends. I owe a special debt of gratitude to Bill Behrman, who generously donated his time and energy to help me print and file the final copy of this dissertation.

I thank my family: Maureen, Andy, Teresa, and Joseph, for filling my childhood with wonderful memories. Special thanks to my parents, James and Kathleen Kuffner, to whom I dedicate this thesis, for their unconditional love and for teaching me the value of education.

Finally, I thank my wife Haruko, for her patience and encouragement throughout.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Challenge of Model Complexity

Computer graphics involves the creation of images of things both real and imagined. *Computer animation* entails the generation and display of a series of such images. In either case, mathematical models are used to represent the *geometry*, *appearance*, and *motion* of the objects being rendered. Recent improvements in computer hardware and software have enabled mathematical models of increasing complexity to be used. Indeed, synthetic images of some objects have been created which bear remarkable similarity to actual photographs of real objects. As computing technology continues to improve, we can expect computer graphics and animation of increasing realism in the future. Ultimately, the dream of *virtual reality* may be fully realized, in which artificial scenes can be created that are indistinguishable from real life.

The fundamental challenges in computer graphics and animation lie primarily in having to deal with complex geometric, kinematic, and physical models. Researchers have developed 3D scanning, procedural modeling, and image-based rendering techniques in order to automate the modeling of the geometry and appearance of complex objects. For the purposes of animation, algorithms are needed to automate the generation of motions for such objects. Moreover, even objects whose geometry and appearance are very simple, may be difficult to animate due to the number of moving parts/joints which must be dealt with.

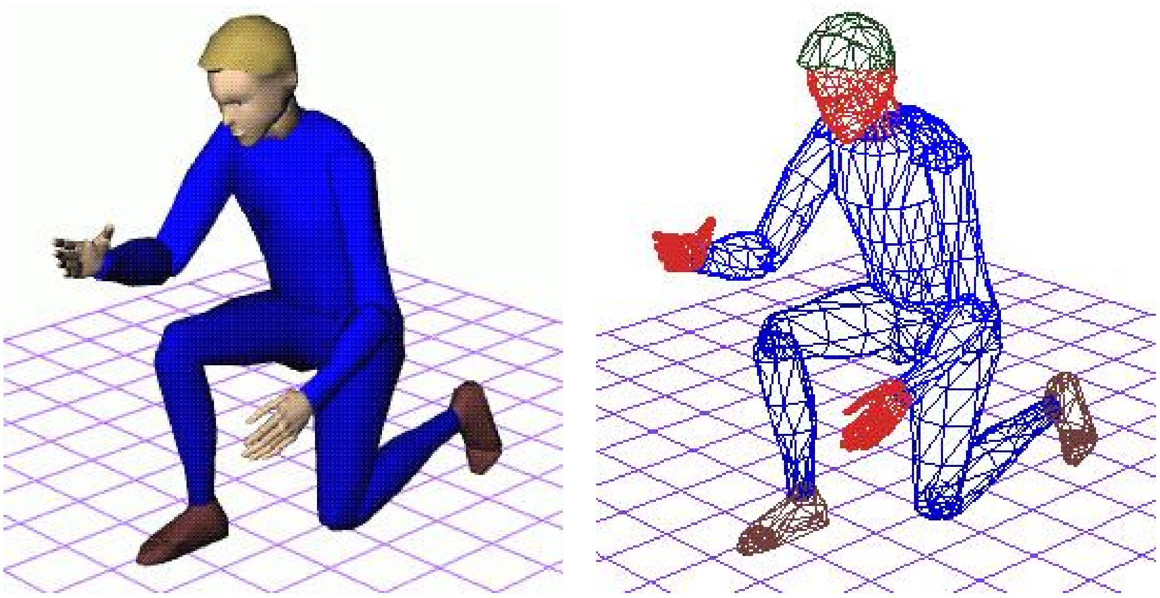
Figure 1.1: An animated human character composed of 52 moving parts with a total of 104 degrees of freedom

1.2 Task-Level Character Animation

This thesis concerns the automatic generation of motion for animated characters. Character animation poses particular challenges to motion generation due to the often complex, multi-jointed structure of the models (see Figure 1.1). In addition, the motion of each joint may be subject to multiple kinematic or dynamic constraints in order to produce body poses and motions that appear natural. Fundamentally, our goal is to create software that will enable a character to move naturally given a *task-level command* such as “walk over to the table and pick up the book”. The underlying software must automatically generate the motion necessary to animate the character performing the given task.

We have concentrated on the animation of human-like character models, though the basic ideas contained in this thesis are applicable to other types of characters. We have focused our experimental efforts on human-like models for the following reasons:

1. Human characters (or characters with a human-like structure) are commonly portrayed in animated films or video games.

2. Human character models involve a complex, multi-jointed kinematic structure that is challenging, such that synthesizing human motion can be considered a benchmark for motion generation algorithms.

3. Real humans are very good at evaluating human motion and detecting subtle flaws, thus making it easier to compare the output of various motion generation strategies.

Researchers in the field have recently coined several new terms that refer specifically to animated human characters. Some of the more popular terminology includes "virtual humans", "virtual actors", "digital actors", "synthetic actors", "animated agents", and "avatars". Many of these terms have found their way into the mainstream media. Although these terms are often used synonymously, their origins can be traced back to disparate applications involving animated human characters that are very different in terms of scope and purpose. These differences stem primarily from the required characteristics of the animated characters, two of which are discussed in the next section. In order to avoid confusion due to pre-conceived notions, we will simply use the term *animated characters* as it applies in the broad sense.

1.3 Autonomy and Interactivity

Beyond geometry and kinematics, animated characters possess two fundamental characteristics which are primarily determined by the application domain in which they are being used:

The degree of *autonomy*: This generally refers to how much intervention the character requires from a user. At one extreme, are completely fine-grained, *user-controlled* characters that demand a human operator specify all joint motions by hand. At the other extreme, are *completely autonomous* characters whose motions are controlled entirely by a software program requiring no user intervention whatsoever.

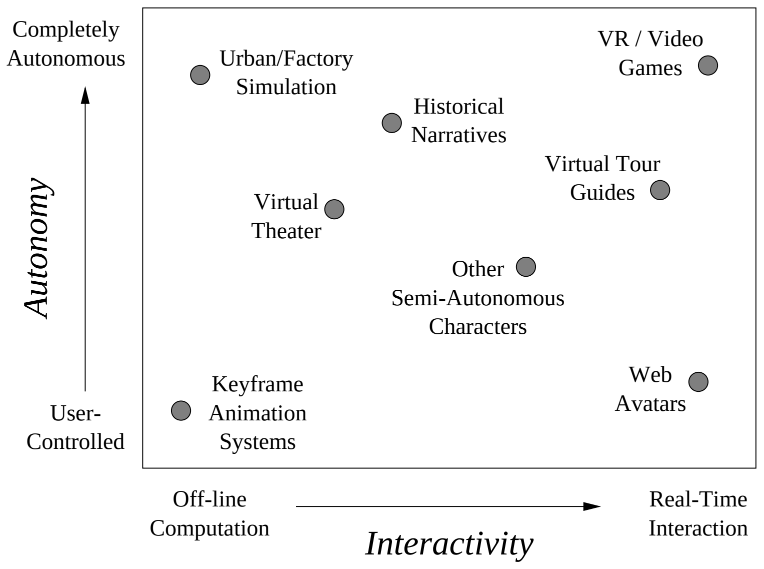

Figure 1.2: The characteristics of animated characters vary according to the application domain.

The degree of *interactivity*: This has to do with the time constraints of the application in relation to the user. At one extreme are completely *non-interactive* (off-line) software applications, such as simulations that have no strict execution time requirements. At the other extreme are highly-interactive applications that have strict execution time constraints. A common requirement is that of maintaining display update rates at 30 frames per second in order to allow *real-time interaction* with a user.

These two characteristics are orthogonal, and form a space of potential uses for character motion generation software that covers a wide range of applications. These include virtual reality, video games, web avatars, digital actors for desktop movie studios, and real-time virtual human simulations for urban, industrial, or military purposes. Some of these are plotted on the diagram in Figure 1.2.

As researchers, we are intrigued by the most challenging problems, namely fully-autonomous human character motion generation for applications that provide real-time interaction (such as virtual reality or video games). Clearly, any motion generation algorithm that is able to satisfy these criteria can also be used in applications with less demanding constraints. For example, generated motion for a character in an interactive application could be simply stored and used later to animate the same character off-line. However, generated motion that is acceptable for interactive video games may *not* be of an acceptable quality for off-line animation production (in terms of smoothness, naturalness, precision, or general artistic impression). Thus, tradeoffs exist between the quality, flexibility, and the time needed to compute motions for various applications.

1.4 Motivation and Previous Work

The primary motivation for task-level control in off-line animation systems stems from the time and labor required to specify motion trajectories for complex multi-jointed characters, such as human figures. Traditional keyframe animation techniques are extremely labor-intensive and require the artistic skill of highly-trained animators. For real-time interactive animation, keyframing “on-the-fly” is not an option. Rather, animation for both autonomous characters and characters under high-level control by the user must be generated via software.

The body of research relating to algorithms for generating animation is young but vast[TT90, BPW92, TDT96]. We only highlight here a few of the key ideas and briefly explain how they fit into the context of human figure animation. Like many things in computer graphics, some of these ideas were originally developed in other fields, such as robotics, computer vision, and artificial intelligence (AI), and have been slowly making their way into the field of computer animation.

The obvious first option to motion generation is to simply “play back” previously stored animation clips (“canned” motions). The animations may have be key-framed by hand, or obtained by *motion capture*. Motion capture techniques offer a fairly simple means of obtaining realistic-looking motion. The key idea is to record the motion of a real human performing the desired task via magnetic sensors, optical tracking, or other means[Mai96, BRRP97]. The motion is stored and used later to animate a character. With only minor editing, motion capture data can provide striking realism with little overhead. However, motion capture data alone (as well as all clip motions) are inflexible in the sense that the motion may often be only valid for a limited set of situations. Such motions must often be redesigned if the relative locations of other objects, characters, or starting conditions change even slightly.

Motion warping, blending, or signal processing algorithms[WP95, BW95, UAT95] attempt to address this problem by offering some added flexibility through interpolating or extrapolating joint trajectories in the time domain, frequency domain, or both. In practice however, they can usually only be applied to a fairly limited set of situations involving minor changes to the environment or starting conditions. Significant changes typically lead to unrealistic or invalid motions.

Alternatively, flexibility can be gained by adopting kinematic models that use exact, analytic equations to quickly generate motions in a parameterized fashion. Forward and inverse kinematic models have been designed for synthesizing walking motions for human figures[KB82, GM85, BC89, BTMT90]. Some kinematic models are based on the underlying physics of walking[AG85, OK94, KB96]. While most of these techniques assume level terrain, or a straight-line trajectory, some can handle curved trajectories and limited variations in the environment terrain[CH99]. Noise functions have also been proposed as a means of adding lifelike behavior to individual motions, or blends between motions[Per95, PG96]. These methods generate motions that exhibit varying degrees of realism.

Dynamic simulation and physically-based modeling techniques nicely handle the problems of physical validity and applicability to arbitrary situations. Given initial positions, velocities, forces, and dynamic properties, an object's motion is simulated according to natural physical laws[AG85, Bar89, MZ90, Mir96]. Dynamic simulation techniques are well-suited for generating *non-intentional motions* (such as animating falling objects, clothing, hair, smoke, wind, water, and other natural effects). While simulations are fairly straightforward to initialize and compute, specifying the desired outcome of the simulation is typically not possible. Thus, we are faced with a classic tradeoff between *physical realism* and *control.* If the animator wants full control (i.e. keyframing all motions), they will often sacrifice physical realism. Alternatively, if an animator desires physical realism, they sacrifice control. For animating *intentional motions* (such as walking, jumping, or lifting), the problem is reduced to designing a good controller that produces the necessary forces and torques to solve a particular task. Given the forces and torques, the simulation generates the resulting motions. Due to the complex dynamics of a multi-jointed system, designing a good controller is an extremely difficult task. Nonetheless, effective controllers have been successfully designed (by hand) for human running, diving, and gymnastic maneuvers[RH91, BH95, Hod96, HW98]. The advantage to designing a controller is that it is generally reusable for a given character and behavior. However, new controllers must be designed each time a new behavior or character morphology is desired. There has been some work on adapting existing controllers to other character morphologies[HP97]. Still other research focuses on automatically generating controllers via dynamic programming[dPFV90], genetic algorithms[NM93, Sim94], control abstraction[GT95], or through training a neural network[dPF93, GTH98]. These techniques have not yet been applied to complex human models.

Spacetime constraints provide a more general mathematical framework for addressing the problem of control [IC88, WM88, LGC94, GRBC96]. Complex animations with multiple kinematic[Gle98], and dynamic[PW99] constraints have been generated. Constraint equations imposed by the initial and final conditions, obstacle boundaries, and other desired properties of the motion are solved numerically. Related constraint-based and optimization techniques include [ZB94] and [BB98]. Other constraint-based techniques involving physically-based models are described in [LM99] and [AN99]. The potential downside to these methods is that the large number of constraints imposed by complex obstacle-cluttered environments can sometimes result in poor performance.

Various forms of motion planning have also been used to animate human figures. Motion planning techniques are designed to calculate collision-free trajectories in the presence of obstacles, and handle arbitrary environments. Motion planning algorithms have been successfully applied to the automatic animation of goal-directed navigation [LRDG90, HC98, BT98, Kuf98], object grasping and manipulation tasks [KKKL94a, BT97, Ban98], and body posture interpolation [BGW$^+$94, JBN94]. These algorithms operate by searching the system configuration space ($\mathcal{C}$-space) for a collision-free path connecting a start configuration to a goal configuration. Though reasonable performance can be achieved for low degree of freedom problems (low dimensional $\mathcal{C}$-spaces), motion planning algorithms typically run slowly when faced with many degrees of freedom. Much of this thesis focuses on presenting techniques aimed at making motion planning practical for the real-time interactive animation of goal-directed navigation (Chapter 3) and object manipulation tasks (Chapter 5).

Nearly all of the motion generation techniques described in the previous paragraphs can be unified under a common framework by viewing the motion generation problem as a problem involving two fundamental tools: *model-building* and *search*. More specifically, solving a motion generation problem almost always involves *constructing a suitable model* and *searching an appropriate space* of possibilities.

Parallel to the research in solving motion generation problems for specific tasks, there has also been work on creating suitable architectures for designing autonomous animated characters in virtual worlds. Related research in robotics has been focused on designing control architectures for autonomous agents that operate in the physical world[Bro85, Ark92]. For animation, the ultimate goal is the realization of fully-autonomous, interactive, artificial human-like characters. Reactive behaviors applied to simple simulated creatures appeared in the graphics literature with Reynolds' BOIDS model[Rey87]. Tu and Terzopoulos implemented a strikingly realistic simulation of autonomous artificial fishes, complete with integrated simple behaviors, physically-based motion generation, and simulated perception[TT94]. This work was extended to include high-level scripting and cognitive models[Fun98]. Researchers at EPFL (Switzerland) have been working on realizing similar goals for human characters. Noser, *et al.* proposed a navigation system for animated characters using ideas for synthetic vision originally developed by Renault, *et al.*[RTT90]. Their implementation also included memory and learning models based on dynamic octrees[NRTT95]. These ideas were later expanded to include virtual tactility and audition[NT95, TNH96]. Researchers at the University of Pennsylvania have pioneered efforts at human simulation. Various algorithms for controlling their Jack human character model have been developed[JBN94, GBR$^+$95], incorporating body dynamics[KMB95, LM99], and high-level scripting[BWB$^+$95, Bad97]. Other systems include Perlin and Goldberg's Improv software for interactive agents[PG96, Per95], the ALIVE project at MIT[BG95, MTBP95, Blu96], Johnson's WavesWorld, the Oz project at CMU[BLR94], and the work of Strassman[Str91, Str94], and Ridsdale, *et al.*[RHC86]. Despite these achievements, creating autonomous animated human-like characters that respond intelligently to task-level commands remains an elusive goal, particularly in real-time applications.

1.5 The "Virtual Robot" Approach

Much research effort in robotics has been focused on designing control architectures for autonomous robots that operate in the real world[Lat91, Bro85, Ark92]. Researchers at U. Penn and EPFL have designed robotics-like architectures for animating human figures in virtual worlds [BWB^{+}95, TDT96]. This thesis also approaches the problem of automatically generating motion for interactive character animation from a robotics perspective. Specifically, the problem of controlling an autonomous animated character in a virtual environment is viewed as that of controlling a *virtual robot* complete with virtual sensors. A virtual robot is essentially an *autonomous software agent* that senses, plans, and acts in a virtual world. User commands or high-level scripts provide task commands, while the software automatically generates the underlying motion to complete the task. This thesis focuses on developing practical methods for implementing a virtual robot approach to synthesizing motions for high-level navigation and manipulation tasks for animated human figures.

1.6 Overview of the Thesis

Chapter 2 describes a framework for automatically generating animation. Chapter 3 deals with the design of an efficient motion generation strategy for *goal-directed navigation.* Chapter 4 describes a method to efficiently simulate the visual perception and visual memory of a character, and explains how to combine this with the planning method of Chapter 3 to obtain a *perception-based navigation strategy.* Chapter 5 describes a new technique for single-query path planning used to quickly synthesize *object manipulation* motions. Chapter 6 describes how the algorithms from the previous chapters can be integrated using high-level scripting to create more interesting and complex behaviors. Finally, Chapter 7 summarizes the key concepts in this thesis along with a concluding discussion.

Chapter 2

Character Animation Framework

2.1 Introduction

One of the fundamental problems concerning the creation of fully-autonomous animated characters is the design of an overall motion-control software framework. Our approach is to view the problem from a robotics perspective, drawing upon the tools and techniques used in the design and control of actual robots. Such tools include algorithms from computational geometry, artificial intelligence, computer vision, and dynamic control systems. The animated character is viewed as an *autonomous agent* in a virtual environment (i.e. a “virtual robot”)

2.2 Autonomous Agents

Background

The concept of an *autonomous agent* has received a fair amount of attention in the recent artificial intelligence literature. Indeed, autonomous agents have been touted as representing “a new way of analysing, designing, and implementing complex software systems”[JSW98]. Ironically, even though several international conferences are held annually on the topic, the term *agent* lacks a universally accepted definition (for a summary discussion, see [FG97]). The word *autonomy* is similarly difficult to define

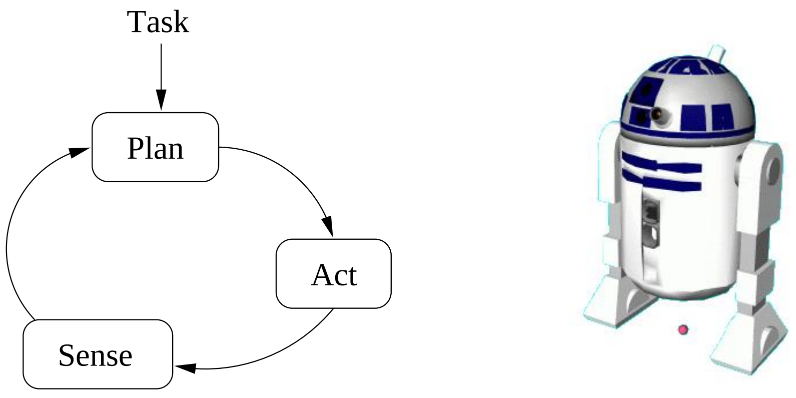

Figure 2.1: The control loop of a robot (physical agent).

precisely. In the context of animated characters, we will consider *autonomous* to mean not requiring the continuous intervention of a user, and adopt the following general definition as a starting point:

> "An *autonomous agent* is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."
>
> Franklin & Graesser, 1996
> *"Is it an Agent, or just a program?"*
> Proc. of 3rd. Int. Workshop on Agent Theories

Physical Agents

A physical agent (what we commonly conceive of as a "robot") is essentially a collection of hardware and software that operates in the physical world. It has control inputs (motors) that apply forces and torques on its joints, which in turn cause the robot to move itself or other objects in it environment. The robot also comes equipped with various kinds of sensors that provide feedback on its motion, along with information about its surroundings. Periodically, the robot gathers input from its sensors and formulates a plan of action based upon its current goals and internal state. The plan of action is converted into control inputs, and the cycle repeats.

Figure 2.1 shows the control loop of a generalized physical agent. In reality, the underlying architecture may be much more complex, with potentially several servo or high-level planning control loop modules executing in parallel. However, the overall operation is essentially the same. Given a program of tasks, the robot utilizes the cycle of sensing, planning, and acting, in an effort to accomplish its goals. If the robot has been well-designed and programmed, it will correctly perform its given tasks.

Animated Agents

An *animated agent* is a software entity that exists and moves within a graphical virtual world. We consider the problem of motion generation for an animated agent as essentially equivalent to that of *designing and controlling a virtual robot.* Instead of operating in the physical world, an animated agent operates in a simulated virtual world, and employs a “virtual control loop”. The virtual control loop operates as follows:

Control Inputs: The animated agent has a general set of *control inputs* which define its motion. These may be values for joint variables that are specified explicitly, or a set of forces and torques that are given as input to a physically-based simulation of the character’s body.

Virtual Sensors: The animated agent also has a set of “virtual sensors” from which it obtains information about the virtual environment.

Planning: Based on information provided by the virtual sensors and the agent’s current tasks and internal state, appropriate values for its control inputs are computed.

The animated agent exists entirely as a software program that runs in a simulated virtual environment. This environment is periodically rendered on a graphic display to produce a series of images. The viewer of the display (the user) interprets the sequence of images as apparent motion of the objects and characters within the virtual environment.

In some cases, the user is capable of directly interacting with the simulation through input devices. This distinguishes the situation from the passive act of simply viewing a television or movie screen, where the information flow is unidirectional (from the screen to the user). The animated agent can respond to the interactions of the user, as it goes about attempting to fulfill its agenda of tasks.

The advantages that a virtual robot has over a physical robot are numerous. While a physical robot must contend with the problems of uncertainty and errors in sensing and control, a virtual robot enjoys "perfect" control and sensing. As an aside, this means that it should be *easier* to design an animated agent that behaves intelligently, than a physical agent that does so. In fact, creating an intelligent autonomous animated agent can be viewed as a necessary step along the way towards creating an intelligent autonomous physical robot. If we cannot create a robot that behaves intelligently in simulation, how can we expect to create one in the real world? Thus, in some sense, this research goes beyond just computer animation, but also extends into the realm of artificial intelligence.

2.3 Motion Synthesis

Animation Variables

Let us now assume that we are faced with the problem of generating motion for an animated character given a task command. With so many possible ways for a character to move, how can a software program select from among the near infinite choices available to complete a given task? The difficulty of this problem generally increases with the complexity of the character. Human-like characters have a large number of animation variables (*avars*), or degrees of freedom (DOF) that must be specified. The variables that control the positions of the parts of the human body model are sometimes referred to as the *joint variables.* These variables typically control approximately 15 to 60 movable parts arranged hierarchically, with anywhere between 40 to 110 degrees of freedom (e.g. the model shown in Figure 2.2). This total does not include the detailed degrees of freedom in the face, which can number in the

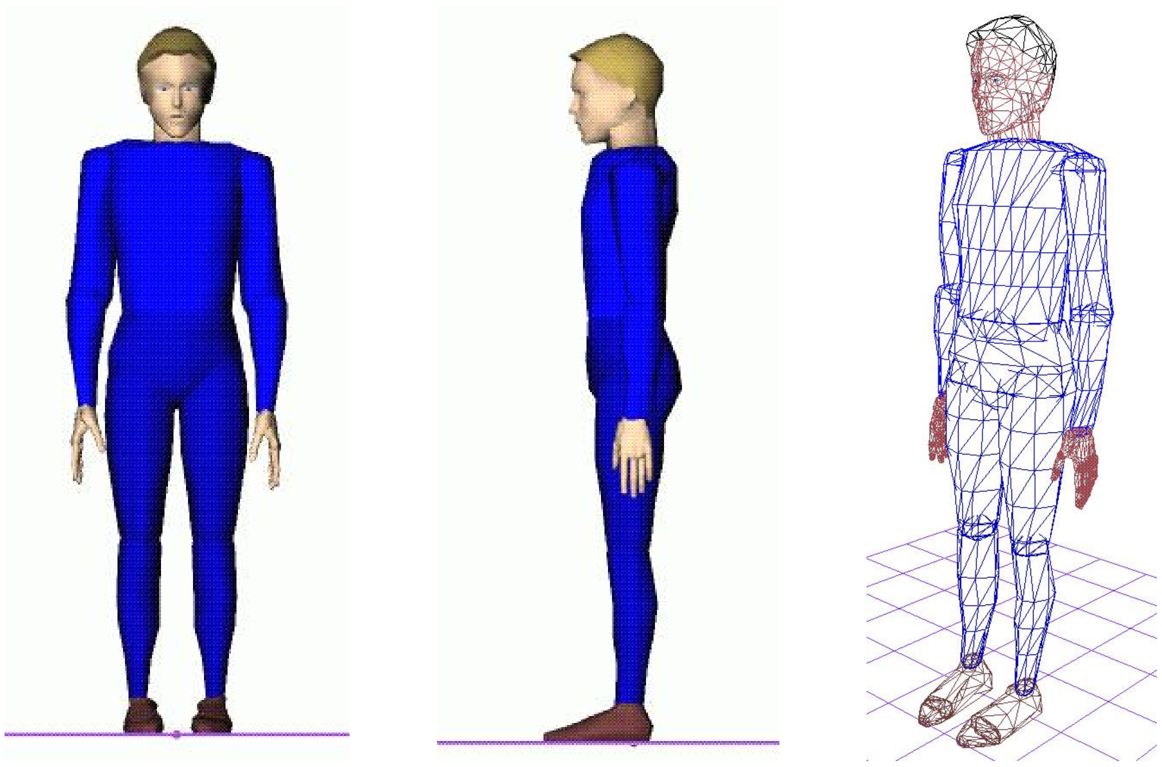

Figure 2.2: Multiple views of a human character model composed of 52 moving parts with a total of 104 degrees of freedom

hundreds depending upon the complexity of the face model.

Mathematically, the animation variables for a character together form a parameter space of possible *configurations* or *poses.* From a robotics standpoint, this space can be considered the *joint space*, or the *configuration space* ($\mathcal{C}$-space) of the character[LP83]. Any motion for the character will trace out a curve (i.e. a *path*) in this multi-dimensional space as illustrated in Figure 2.3. The curve is normally a continuous function of time (i.e. a *trajectory*). In addition, the topology of the space may be multiply connected so as to reflect the fact that certain variables "wrap around" in value (e.g. global orientation). Conceptually, the fundamental goal of any motion synthesis strategy is *to generate a trajectory in the configuration space that accomplishes the desired task.* In addition to this basic requirement of accomplishing the task, other important criteria may include generating smooth motion, physically-valid motion, or simply motion that is perceived by a user as generally aesthetically-pleasing. This

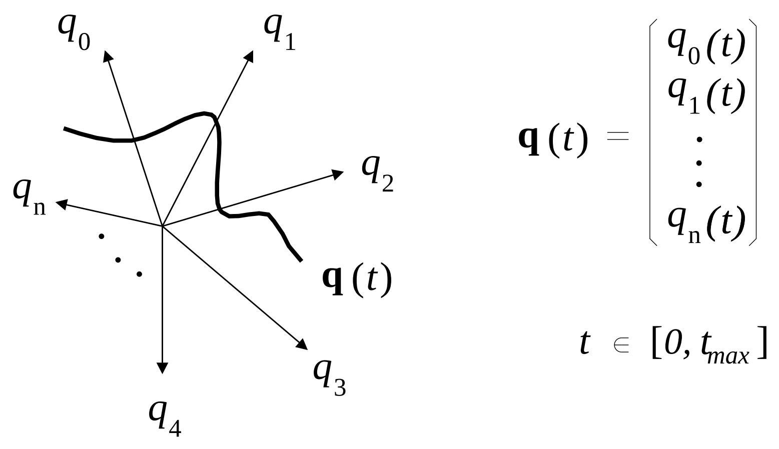
q_0
q_1
q_2
q_n
q_3
q_4
$\mathbf{q}(t)$
$\mathbf{q}(t) = \begin{pmatrix} q_0(t) \\ q_1(t) \\ \vdots \\ q_n(t) \end{pmatrix}$
$t \in [0, t_{max}]$

Figure 2.3: Motions for a character trace out a time-parameterized curve in the multi-dimensional joint space.

last criterion is perhaps the most important for computer animation, yet also the most difficult to satisfy, since "aesthetics" cannot be readily defined mathematically or objectively.

Classes of Motion

The motions that we may wish to generate for animated characters tend to fall into two categories:

Primary Motions: These include gross body movements (e.g. walking, jumping, reaching), as well as facial animation (e.g. speech, expressions). Primary motions are sometimes referred to as *active* or *intentional* motions, since they often involve joints that are under a character's direct control. Thus, they are driven by a character's willful (intentional) acts.

Secondary Motions: These include the animation of hair, skin, clothing, or motion related to environmental effects (e.g. smoke from a character's cigarette). Secondary motions are sometimes referred to as *passive* or *non-intentional* motions, since they involve animation variables that are typically not under a character's direct control. Instead, the motions are an indirect result of the character's actions (e.g. the wrinkling of a character's shirt in response to a reaching motion).

The approaches taken to solving motion generation problems among these two classes of motion will often differ. Some motion synthesis strategies are more suited to primary motions (e.g. motion planning), while other are more readily useful for creating secondary motions (e.g. physically-based modeling).

Facial animation is a highly-specialized topic with its own unique challenges, and is beyond the scope of this thesis. However, as any skilled animator will testify, facial animation is a vitally important aspect to creating believable action and drama. Thus, any serious automatic motion generation software should support the animation of the face.

Within the remaining chapters of this thesis, we will focus our attention on the automatic generation of gross body motions, specifically that of goal-directed navigation (walking and running), and object grasping and manipulation.

2.4 High-Level Tasks

Specification of Task Commands

We will consider the generation of motion for human-like animated characters that can respond to task-level commands. such as "walk to the kitchen", "open the refrigerator", "take out the milk", "pour a glass", "sit down at the table", "take a drink", "wave hello to Mr. Smith", or "move the teapot next to the Luxo lamp in front of the ray-traced sphere". Task commands may be given to a character in a number of ways:

User-specified: Task commands generated explicitly by a user through keystrokes, mouse clicks, voice commands, or other direct means.

Software scripts: High-level behavior programs specified via combinations of typical programming language constructs, such as IF-THEN-ELSE, WHILE-DO, REPEAT, etc.

Behavior simulations: Software that attempts to model the physical, mental, or emotional state of the character, and generates task-level commands based on its internal goals, needs, desires, or preferences. This approach may be considered a special case of a software script approach that employs a detailed model of the character's internal functions or thought processes.

Given a task command (or a set of high-level goals) provided by one of the above sources, the software must automatically generate the motion for the character to complete the task. From here on, we shall refer to this problem as the *motion synthesis problem*, and the corresponding software as the *motion synthesis software*.

Desired Motion

From any given task command, the input and the output are essentially of the same form. The input given is:

1. A geometric description of the environment.

2. A kinematic and geometric description of the character (possibly including joint limits, or physical attributes such as mass, moments of inertia, etc., for each link).

3. The *initial state* of the character (the initial values for each joint parameter of the character, and any auxiliary animation variables).

4. The *goal state* of the character (either directly specified or computed from the task command).

The desired output will be a specification for each of the character's joint variables expressed as a continuous function of time. The simultaneous playback of all of the joint trajectories will cause the character to move so as to accomplish the given task.

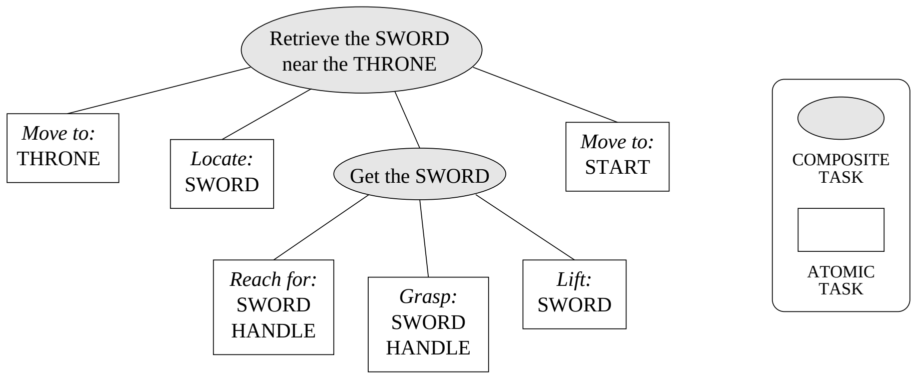

Figure 2.4: A composite task broken down into a linear sequence of atomic tasks

Composite Tasks and Atomic Tasks

The ability to synthesize motion for a character depends upon restricting the domain of task commands allowable. Therefore, some task commands can be considered as basic *atomic tasks* such as “move to a location” or “grasp an object”. Atomic tasks such as these have final goal states which are generally less ambiguous, or may have default parameters specified to allow a fast and fairly straightforward computation of the goal state.

From these atomic task primitives, one can build up more complicated *composite* task commands. For example “retrieve the sword near the throne” may be decomposed into the following sequence of sub-tasks: “move to the throne”, “locate the sword”, “get the sword”, and “move to the starting location” (see Figure 2.4). The sub-task “get the sword” may be further subdivided into the tasks “reach towards the sword handle”, “grasp the sword handle”, and “lift the sword”. Each sub-task will generally involve an initial and a goal state which is less ambiguous, and therefore simpler to compute. The goal state of one sub-task will become the initial state of the subsequent sub-task. Structuring task commands based on a hierarchy of sub-tasks allows one to perform a recursive decomposition until a linear chain of atomic task commands is obtained. To complete the composite task, the character needs only to complete all atomic sub-tasks in the chain.

2.5 A Virtual Robot Architecture

Overview

If every atomic task was very narrowly and explicitly defined, one could imagine simply maintaining a vast library of pre-computed, captured, or hand-animated motions to accomplish every possible task. The reality is that, in general, tasks cannot be so narrowly defined, lest the set of possible tasks become infinite. Instead, atomic tasks are typically specified at a high-level, and apply to a general class of situations, such as "move to a location" or "grasp an object". Thus, they are specific in one sense (the required end result), but also very vague (how the result is achieved).

At present, no single motion synthesis strategy can adequately handle the variety of tasks that can arise. Instead, a flexible underlying software framework is needed to bring together different tools and techniques. As a foundation, we propose an animated agent framework based on a virtual control loop as outlined in Section 2.2. At regular intervals, the agent computes a motion strategy based on the current task, its internal state, and its perceived model of the environment. The motion is generated using a collection of fundamental software modules and data libraries depending upon the task.

Some possible software components include a large library of "canned motions" (clip motions), a simulated sensing module, a physically-based simulation module, and a library of motion planning and geometric algorithms. Other potential component modules include numerical optimization libraries, inverse kinematics routines, and collections of biomechanical or neurophysiological data appropriate to specific classes of character morphology (e.g. humans, dogs, birds). As an example, Chapter 3 describes a goal-directed navigation strategy that combines cyclic motion capture data, a path planning module, and a path following controller. Chapter 4 extends this technique to incorporate a simulated sensing module. Chapter 5 describes a motion strategy for object grasping and manipulation that utilizes an inverse kinematics module and a randomized path planner and optimizer. What is important is that different software modules are accessible for use by different motion synthesis strategies depending upon the task.

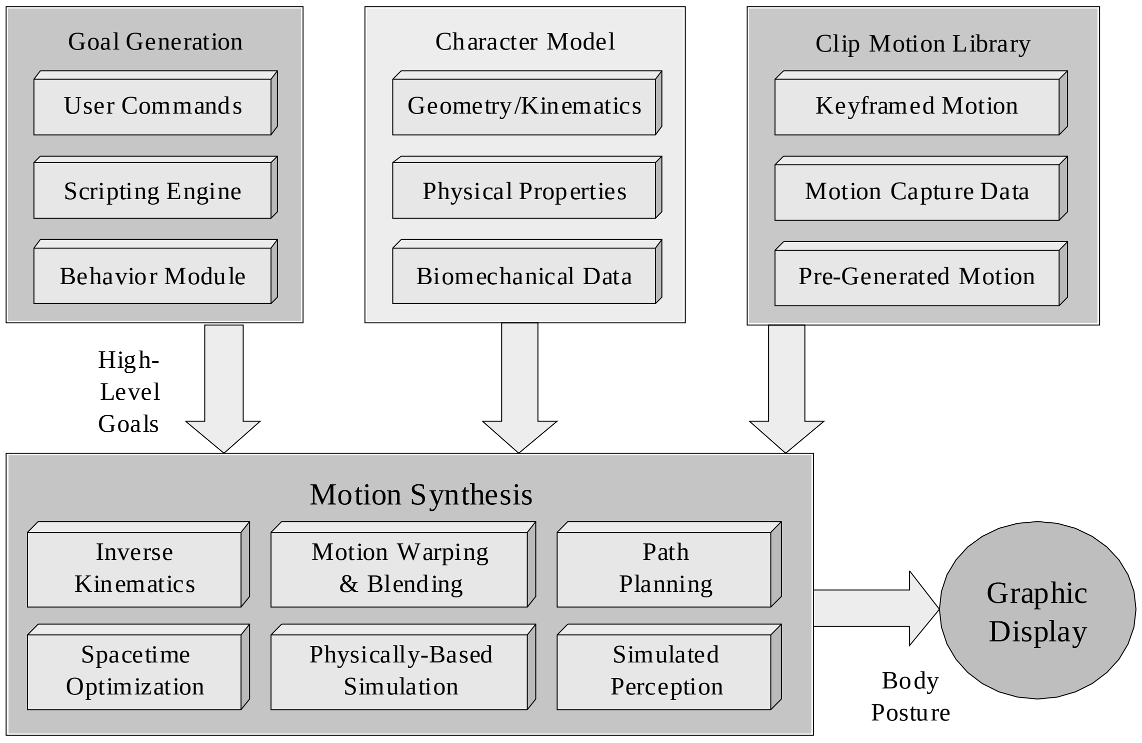

Figure 2.5: Resources available for motion synthesis.

A block diagram of the agent's available software resources is depicted in Figure 2.5. A set of high-level goals are generated and passed to the motion synthesis software. Utilizing some or all of the fundamental software components and data libraries, the resulting motion for the character is computed and passed to the graphic display device for rendering.

Mathematical Formulation

To make things more precise, we now give a more formal formulation of the motion synthesis problem for animated characters. The notation adopted here is loosely based on the conventions used in [Lat91], which are often used in the robotics and motion planning literature.

1. A character or *agent* is called $\mathcal{A}$. If there are several characters, they are called $\mathcal{A}_i$ $(i = 1, 2, \ldots)$.

2. The 3D environment in which the characters move is denoted by $\mathcal{W}$ (commonly called the *workspace* in robotics), and is modeled as the Euclidean space $\Re^3$ ($\Re$ is the set of real numbers).

3. Each character $\mathcal{A}$ is a collection of p links $\mathcal{L}_j$ ($j = 1, \ldots, p$) organized in a kinematic hierarchy with cartesian frames $\mathcal{F}_j$ attached to each link.

4. A *configuration* or *pose* of a character is denoted by the set $\mathcal{P} = \{T_1, T_2, \ldots, T_p\}$, of p relative transformations for each of the links $\mathcal{L}_j$ as defined by the frame $\mathcal{F}_j$ relative to its parent link's frame. The *base* or *root* link transformation T_1 is defined relative to some world cartesian frame $\mathcal{F}_{world}$.

5. Let n denote the number of generalized coordinates or *degrees of freedom* (DOFs) of $\mathcal{A}$. Note that n is in general *not equal* to p. For example, a simplified human arm may consist of three links (upper arm, forearm, hand) and three joints (shoulder, elbow, wrist) but have seven DOFs ($p = 3$, $n = 7$). Here, the shoulder and the wrist are typically modeled as having three rotational DOFs each, and the elbow as having one rotational DOF, yielding a total of seven DOFs.

6. A *configuration* of a character is denoted by $\mathbf{q} \in \mathcal{C}$, a vector of n real numbers specifying values for each of the generalized coordinates.

7. Let $\mathcal{C}$ be the *configuration space* or $\mathcal{C}$-space of the character $\mathcal{A}$. $\mathcal{C}$ is a space of dimension n.

8. Let $\textsc{Forward}(\mathbf{q})$ be a *forward kinematics* function mapping values of $\mathbf{q}$ to a particular pose $\mathcal{P}$. $\textsc{Forward}(\mathbf{q})$ can be used to compute the global transformation G_j of a given link frame $\mathcal{F}_j$ relative to the world frame $\mathcal{F}_{world}$.

9. Let $\textsc{Inverse}(\mathcal{P})$ be a set of *inverse kinematics* (IK) algorithms which maps a given global transformation G_j for a link frame $\mathcal{F}_j$ to a set $\mathcal{Q}$ of values for $\mathbf{q}$. Each configuration $\mathbf{q} \in \mathcal{Q}$, represents a valid inverse kinematic solution ($\textsc{Forward}(\mathbf{q})$ positions the link $\mathcal{L}_j$, such that the frame $\mathcal{F}_j$ has a global transformation of G_j relative to $\mathcal{F}_{world}$). Note that the set $\mathcal{Q}$ may possibly be infinite, or the empty set (no valid solutions exist).

10. The obstacles in the environment $\mathcal{W}$ are denoted by $\mathcal{B}_k$ $(k = 1, 2, \ldots)$.
11. We define the *$\mathcal{C}$-obstacle region* $\mathcal{C}B \subset \mathcal{C}$ as the set of all configurations $\mathbf{q} \in \mathcal{C}$ where one or more of the links of $\mathcal{A}$ intersect (are in collision) with another link of $\mathcal{A}$, any of the obstacles $\mathcal{B}_k$, or any of the links of another character $\mathcal{A}_i$ with $\mathcal{A}_i \neq \mathcal{A}$. We also regard configurations $\mathbf{q} \in \mathcal{C}$ where one or more *joint limits* are violated as part of the $\mathcal{C}$-obstacle region $\mathcal{C}B$.
12. The open subset $\mathcal{C} \setminus \mathcal{C}B$ is denoted by $\mathcal{C}_{free}$ and its closure by $cl(\mathcal{C}_{free})$, and it represents the *space of collision-free configurations* in $\mathcal{C}$ of the character $\mathcal{A}$.
13. Let $\tau : \mathcal{I} \mapsto \mathcal{C}$ where $\mathcal{I}$ is an interval $[t_0, t_1]$, denote a motion trajectory or *path* for a character $\mathcal{A}$ expressed as a function of time. $\tau(t) = \mathbf{q}_t$ represents the configuration $\mathbf{q}$ of $\mathcal{A}$ at time t, where $t \in \mathcal{I}$.
14. A trajectory τ is said to be *collision-free* if $\tau(t) \in \mathcal{C}_{free}$ for all $t \in \mathcal{I}$.

In response to an atomic task command for a character, we wish to compute a motion trajectory τ that satisfies certain criteria. At a minimum, the resulting motion of $\mathcal{A}$ should connect the initial configuration $\mathbf{q}_{init}$ and the goal configuration $\mathbf{q}_{goal}$. Though not strictly required, it is also helpful if the motion trajectory is *collision-free* (the character's geometry does not pass through objects in the environment). In other words, τ should be computed such that:

1. $\tau(t_0) = \mathbf{q}_{init}$ for the start time t_0.
2. $\tau(t_1) = \mathbf{q}_{goal}$ for some final time t_1.
3. $\forall t \in [t_0, t_1]$, $\tau(t) \in \mathcal{C}_{free}$.

The forward kinematics function FORWARD($\mathbf{q}$) can be used to map intermediate configurations defined by τ to the corresponding character poses used during graphic rendering and display. A software module that computes τ for specific task commands is called a *motion synthesis strategy.*

There are several criteria which are important for the purpose of evaluating any proposed motion synthesis strategy.

1. *Computational efficiency*: Given the available computing resources, how much time is required to compute a motion? Is this strategy practical for the desired application?

2. *Reliability and consistency*: Do the algorithms always terminate? Do they find valid solution trajectories whenever they exist? Do similar task queries yield similar results?

3. *Quality of the motion*: Is the generated motion physically valid? Does it appear natural? Does it reflect the personality or unique aspects of the character? Overall, is it aesthetically pleasing?

These criteria (especially naturalness and aesthetics) are more difficult to define mathematically, and typically must be evaluated by viewing the output motions resulting from repeated trials.

In designing a motion synthesis strategy, a variety of general-purpose software tools (*software modules*) are available for use. Different modules will be more effective and useful for certain classes of task commands. In addition, typically one or more modules can be used in combination in order to solve specific motion synthesis problems.

The Roles of Specific Software Modules

This section describes a few of the fundamental software components identified in Figure 2.5, and briefly indicates how they might be utilized for synthesizing motion. This is by no means an exhaustive list, but adequate for creating interesting animations (see Chapter 6). Again, we believe that no single component can provide a general solution to the motion synthesis problem, but rather each technique in combination with one or more of the others can provide a viable approach to generating animation for a given set of tasks.

Clip Motion Libraries

Clip motions or "canned motions" are short animations that have been either key-framed, pre-generated, or obtained via motion capture systems. Such motions can be stored and played back very efficiently. Clip motions obtained via motion capture systems are recorded directly from a live subject, and then applied to a character for the purposes of animation. Captured motions are often preferred due to their high level of visual realism. This is especially true of motions which contain high-frequency components in the data (e.g. rapid gestures), or motions in which a character's particular body mannerisms (style of motion) must be reproduced.

Large libraries of clip motions can potentially become a powerful resource for animation. Eventually, animating a character in a given situation could ultimately involve selecting a pre-recorded motion from a vast *motion dictionary* indexed by task or motion characteristics. For example, there may be hundreds of walking motions stored, from among which a character might utilize a "medium-fast walk with a slight limp in the left leg".

As mentioned in Chapter 1, the primary drawback to clip motions is that any given data set can usually only be used in a very specific set of situations. For example, consider a captured motion of a human character opening a refrigerator and taking out a carton of milk. The motion will only appear perfect if the virtual model of the character, the refrigerator, the carton, and their relative positions match the actual objects used when the motion was captured. What happens if the carton is placed on the lower shelf instead of the upper shelf? What if the refrigerator model is made larger, or the location of the handle on the refrigerator door changes? What happens if a bottle of orange juice is placed in front of the milk carton? Motion warping, interpolation, or spacetime constraint techniques exist that adapt clip motions to a broader class of situations while enforcing kinematic and/or dynamic constraints[WP95, Gle98, BB98, PW99]. However, larger deviations can often cause such adapted motions to lose their visual realism, and hence their aesthetic quality. Finding new techniques to adapt individual motions and connect sequences of clip motions is an active area of research.

Captured motions are most useful for *efficiently animating activities that would* *otherwise be very difficult to model in software* (e.g. sitting in a chair, dancing, or animating human speech and facial expressions). The human eye is very keen at detecting subtle flaws in human motion animation. Thus, most software simulations have so far been unable to match the pure visual realism obtained by simply using captured motions. For this reason clip motions will continue to play an important role in real-time animation systems.

Motion Planning

Motion planning algorithms were initially developed in the context of robotic systems. Such algorithms generate motion given a high-level goal and a geometric description of the objects involved[Lat91]. In the context of computer animation, motion planning can be used to effectively compute primary (intentional) motions. Examples include computing collision-free motions to accomplish high-level navigation or object manipulation tasks[KKKL94a, Ban98, Kuf98], or connecting different body configurations[CB92, BMT97]. Motion planning is particularly suited to such tasks, since there is a near infinite number of possible goal locations and obstacle arrangements in the environment. This combinatorial explosion of possibilities currently prohibits the direct use of pre-recorded motion sequences. Instead, flexible and efficient planning algorithms can be designed to compute collision-free motions for specific categories of tasks commands that apply to a broader set of situations (see Chapter 3 and Chapter 5).

However, motion planning is less useful for tasks in which few obstacles to motion exist and/or aesthetics are of paramount importance (such as facial animation). As an example, consider the task of animating a human figure sitting down on a chair. This task results in a highly-specialized class of motions involving a direct interaction of the entire body with an environment object (in this case, a chair). Grasping the armrests during the sitting motion and adjusting the body posture afterwards are subtleties of motion that are important for naturalness, and are unlikely to be obtained by directly searching the body configuration space. In this case, using motions derived from captured data can potentially yield much better results, when compared with attempts to construct complicated kinematic or physically-based models, or apply motion planning techniques.

The primary challenge when using motion planning to generate animation is to design efficient algorithms that achieve visual realism. Aesthetics of motion are of little concern for robots, but are vitally important for animated characters. The computed motion must look natural and realistic. It may be possible to encode aesthetics as search criteria to use during planning, or to perform post-processing on the planned motion. For example, the naturalness and realism of a planned motion could arise from an underlying physically-based model that guides the search. Alternatively, search criteria might be ultimately derived from clip motion libraries that represent a particular "style" of motion.

Physically-Based Simulation

All motions in the physical world are driven by the laws of physics. Motions in virtual worlds typically aspire to give the appearance that they are also driven by the laws of physics. Graphical models simulate the visual appearance of objects, while physical models simulate their behavior in the physical world.

Animation generated using physically-based models (dynamic simulation) has the potential advantage of exhibiting a very high level of realism. Given a set of initial conditions and force/torque models, the motions of objects and their interactions can be calculated precisely. However, since the underlying motion is dictated by physics, it is difficult to control the simulation at a task-level. Spacetime constraint optimization techniques can alleviate some of these difficulties[IC88, WM88, LGC94], but at a computational cost that is largely prohibitive for interactive animation systems.

Physically-based techniques are very well-suited for generating secondary motions. Examples include the animation of rigid objects[AG85, Bar89, Mir96], hair, skin, and clothing[TF88, VCNT95, BW98], or environmental effects (wind, water, smoke, fire)[SOH99]. Animating special effects whose motion derives primarily from the force of gravity (e.g. a human character falling from a bicycle or slipping on a banana peel) are ideal situations where physically-based simulations can be applied.

However, it is more difficult to apply such techniques to the animation of primary (intentional) motions, using a physically-based model of a character. For example, consider tasks in which the dynamics of the underlying physical system (e.g. the human body) plays a significant role in the overall quality of the motion (e.g walking, dancing, running, jumping, kicking, etc). Fundamentally, the key difficulty lies in computing the required controls necessary to achieve a particular task. Some complex controllers have been carefully hand-designed for animating human walking, running, and athletic maneuvers[MZ90, RH91, Hod96, HW98]. Other researchers have proposed automatically synthesizing controllers via dynamic programming[dPFV90], genetic algorithms[NM93, Sim94], control abstraction[GT95], or through training a neural network[dPF93, GTH98].

Deriving the necessary controls to animate a physically-based model may be another area in which both motion planning and libraries of captured motions might be useful. Motion planning problems that involve dynamic constraints (*kinodynamic planning*) can be formulated as a search among the space of available controls to move a physical system from one state to another[DXCR93, BL93, Fer96, LK99, LM99]. Alternatively, one can envision using the "inverse dynamics" of a given physically-based model in order to compute the set of controls necessary to achieve a particular captured motion. Ultimately, libraries of "canned motions" may eventually be replaced by libraries of "canned sets of controls" that can be used in combination with a physically-based model of a character.

Clearly, as the computational resources available to desktop systems grows, increasingly sophisticated physically-based models can be used in a variety of ways in order to generate increasingly realistic animations.

Simulated Sensing

Creating an autonomous animated agent with believable behavior in an interactive virtual environment will ultimately require some kind of simulated sensing (see Chapter 4). This can include one or more of simulated visual, aural, olfactory, or tactile sensing. The basic idea is to more realistically model the flow of information from the virtual environment to the character. The character should act and react according to what it perceives.

Sensory information can be encoded at both a low level and a high level and utilized by high-level decision-making processes of the animated agent. Examples of sensory encodings include “all objects that are currently visible”, “all other characters that are currently nearby”, or “sounds that can be currently heard”. Because animated agents operate in virtual environments, they can potentially avoid many of the problems that physical agents (robots) have when dealing with sensory information (e.g. noisy data, conflicting data, etc). However, in the interests of realism, it may not always be desirable to equip animated agents with perfect sensors. For example, real humans can become confused in rooms with mirrors, or mistake a tiger-skin for a real tiger. In the same way that some computer-generated images of real-life objects are sometimes criticized as being “too perfect”, an animated agent that never makes mistakes might also be criticized as being unrealistic.

The output of the agent’s sensors can be used to incrementally build a perceived model of the world, with which the agent utilizes to plan its motion. This completes the cycle of the virtual control loop. Note that the agent’s perceived model of the virtual world need not coincide with the actual model. This means that agents can sometimes make poor judgments, or be forced to make decisions based on incomplete information (which is what happens to humans in real life).

Chapter 3

Goal-Directed Navigation

3.1 Introduction

This chapter combines several of the software component modules discussed in Chapter 2 to design a motion generation strategy for *goal-directed navigation.* Navigation is an important task that frequently arises in the context of animated characters. Navigation task commands require that a character move from one location to another within a virtual environment filled with obstacles.

Consider the case of an animated human character given the task of moving from a starting location to a goal location in a flat-terrain virtual environment (such as the virtual office in Figure 3.1). An acceptable navigation algorithm should produce a set of natural-looking motions to accomplish the task while avoiding obstacles and other characters in the environment.

Related Work

The problem of goal-directed navigation has received much attention in the robotics literature, as it is fundamental to designing practical mobile robots. In particular, the objective has been the design and implementation of task-level motion planning algorithms for real-world robotic systems[Lat91]. Variations on an assortment of robot navigation techniques have been used for the purposes of animation.

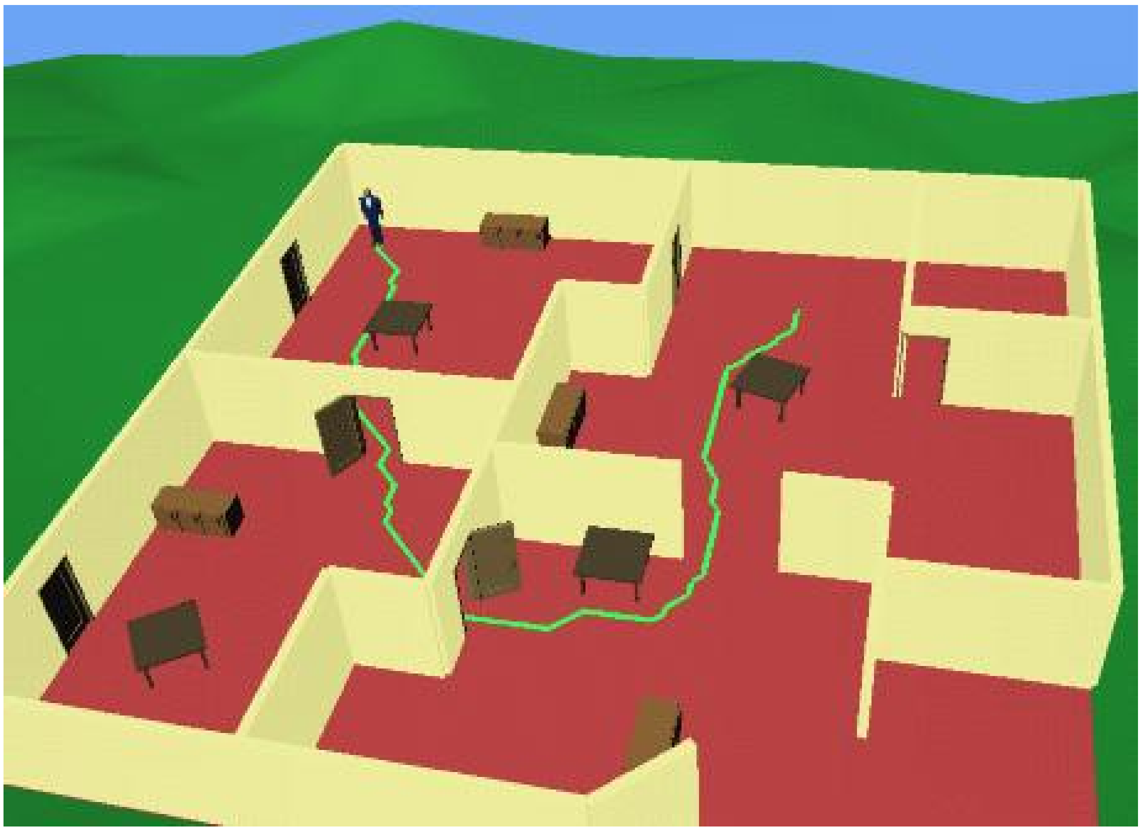

Figure 3.1: A human character navigating in a virtual office.

Local approaches use sensors and reactive behaviors to avoid obstacles in the immediate vicinity[Bro85, Ark98]. Some popular techniques utilize artificial potential fields for obstacle avoidance[Kha86]. Reynolds used a kind of potential field to simulate flocking and schooling behaviors[Rey87], and developed an array of reactive controllers for creating interesting autonomous behaviors[Rey99]. Tu and Terzopoulos used similar reactive behaviors for their artificial fish[TT94]. Reich, *et al* used potential fields for animating navigation for human characters[RBKB94]. Blumberg and Galyean used a reactive controller along with simulated perception for their artificial dog, in which motion energy of the visual field is used as a heuristic for avoiding obstacles[BG95]. Van de Panne proposed a footprint-based motion generation strategy for human figures along a pre-determined global path[vdP97]. Footprints that intersect obstacles are perturbed locally until a collision-free position is found. Noser, *et al.* used potential fields for obstacle avoidance for human characters, along with an integrated simulated visual perception system for exploring unknown environments[NRTT95].

The advantages to potential fields are that they are efficient, simple to compute, and can be used for both known and unknown environments. The primary disadvantage is their tendency to become trapped in local minima induced by the artificial potential field. In general, all local, behavior-based approaches can become trapped in *behavior loops*, resulting in the inability to find a path to the goal even if one exists.

Global approaches to navigation consider global information to form a plan of action, as opposed to simply reacting to local information. Global path planning techniques using a configuration-space approach have been extensively studied in robotics [LP83, Lat91]. For navigation in 2D, the configuration space can often be explicitly represented and complete algorithms are known[SS83, Can88]. Most path planning methods operate by first building a convenient representation of the free space (e.g. a graph), and then searching that representation for a path.

Some of these planning methods have been adopted for the purposes of animation. Lengyel, *et al* used rasterizing computer graphics hardware for fast motion planning[LRDG90]. Bandi and Thalmann implemented a global path planning approach for human navigation using space discretization that can handle environments with stairs and ramps[BT98]. Kuffner developed a similar approach for navigation on a flat-terrain suitable for interactive applications[Kuf98]. Hsu and Cohen used an exact cell-decomposition path planner for human figure navigation on an uneven terrain[HC98].

The advantages to global approaches to navigation is their ability to find paths even in complex environments, and (for some methods) their ability to always return optimal paths to the goal (if one exists). The primary disadvantages are that they typically assume that the complete environment is known in advance, and they generally involve more computation than local approaches.

3.2 Navigation using Planning and Control

Overview

The navigation strategy presented in this chapter is a global approach based on [Kuf98], and is similar in spirit to the work of Bandi[Ban98]. Our strategy will be to divide the computation into two phases: *path planning* and *path following*. The planning phase computes a collision-free path to the goal location, while the path following phase synthesizes the character's motion along the computed path. Navigation goals are specified at the task level, and the locomotion animation is created using cyclic motion capture data driven by a path-following controller that tracks the computed path. Although the planner presented here is limited to finding navigation paths that lie in a plane, the algorithm can potentially be modified to compute paths that include vertical motions of the character (see Section 3.8).

The speed of the navigation planner enables it to be used effectively in interactive environments where obstacles and other characters change position unpredictably. Chapter 4 extends the navigation algorithm to unknown environments by simulating the visual perception of a character. This is used to provide a feedback loop to the overall navigation strategy, as the character reacts in response to obstacles in its visual field. A record of perceived objects and their locations is kept as the character explores an unknown environment, allowing the character to construct motion plans based solely on its own internal model of the virtual world. The internal model is updated as new sensory information arrives, and an updated motion plan is computed if necessary. The resulting animation can be generated at interactive rates and looks fairly realistic. The overall data flow for the navigation strategy is illustrated in Figure 3.2.

Limiting the Degrees of Freedom

The theory and analysis of path planning algorithms is fairly well-developed in the robotics literature, and is not discussed in detail here. For a broad background in motion planning, readers are referred to [Lat91]. For any path planning technique,

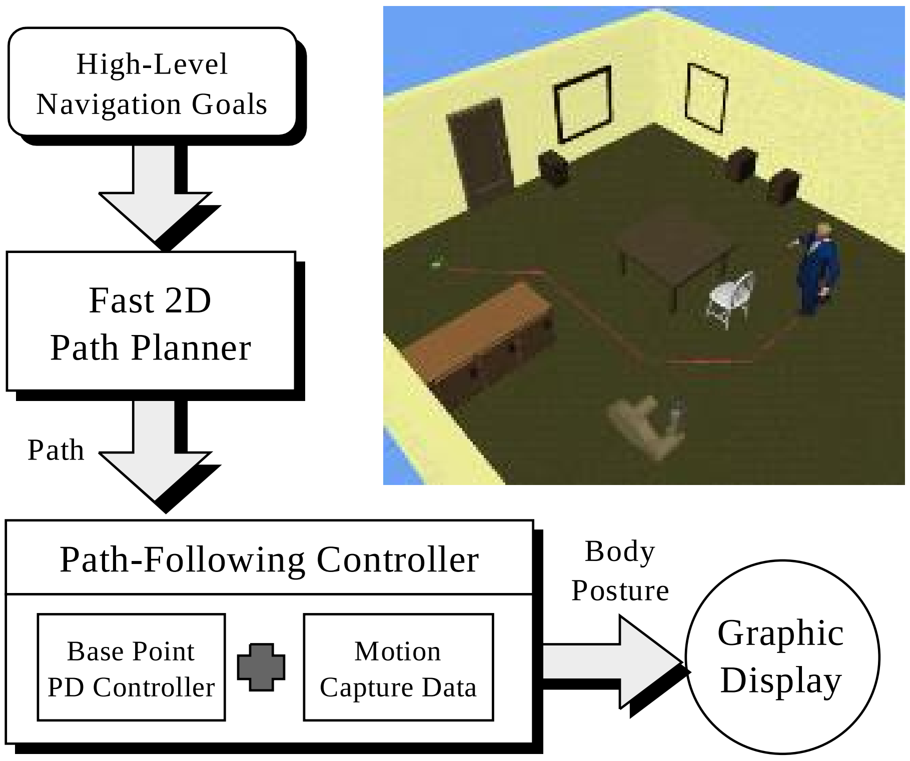

Figure 3.2: Data flow for the navigation strategy

it is important to minimize the number of degrees of freedom (DOFs), since the time complexity of known algorithms grows exponentially with the dimension of the $\mathcal{C}$-space[Rei79]. For human locomotion (e.g. walking, running, crawling), the most important joints that must be animated are the major appendages of the body (i.e. the torso, legs, and arms). The joint variables in the face, toes, and fingers of the hands typically play a minimal role. Figure 3.3 illustrates the major joints of a human figure used for navigation and their hierarchical arrangement. Motion capture data used for animating the gross body motions of human figures almost invariably specifies trajectories for these degrees of freedom.

For fast motion synthesis, we want to only consider planning for the degrees of freedom that affect the overall global motion of the links of a character (i.e. the *root* joint DOFs). For the joint hierarchies we used, each of the 6 degrees of freedom of the *Hip* joint falls into this category. The idea is to compute motion plans for these

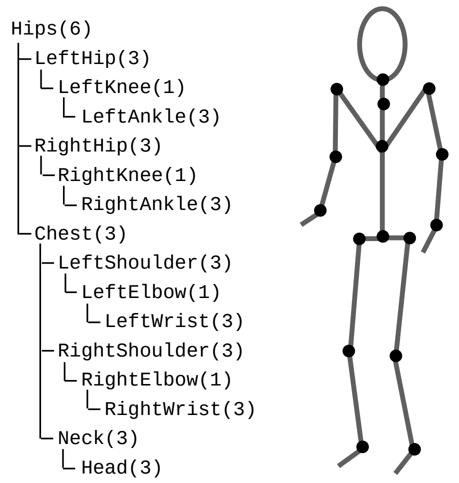

Figure 3.3: The major joints in the kinematic hierarchy used for locomotion are shown, along with the number of DOF for each joint.

DOFs (which we shall refer to as *active DOFs* or *active joints*), while the remaining joints (which we refer to as *passive DOFs* or *passive joints*) are either held fixed, or controlled by a simple algorithm. For our navigation strategy, passive joints are animated by cycling through a clip motion locomotion gait. However, passive joints could also potentially be driven by the active joints with their motion computed using a simple mathematical relation or even a sophisticated physically-based simulation (e.g. the motion of a character's hair in response to the gross motions of the head).

Using a Bounding Volume

Moreover, we can further reduce the dimensionality (and hence the efficiency) of the planning phase by considering only a subset of the active joints when possible. For instance, if we assume that the character navigates on a flat surface, we can omit one of the active translational DOFs (the height of the Hip joint above the surface), as well as two of the active rotational DOFs (by assuming the figure's main axis remains aligned with the normal to the surface). Suppose all of the passive DOFs and the omitted active DOFs are driven by a controller that plays back a simple locomotion gait derived from motion capture data. By approximating the character by a suitable

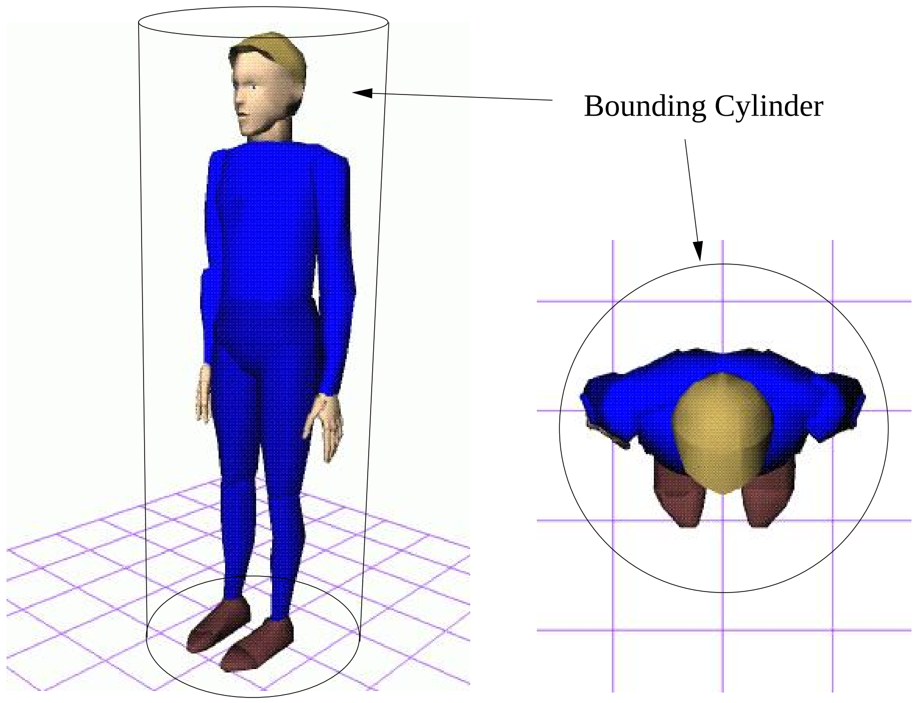

Figure 3.4: The character's geometry is bounded by an appropriate cylinder.

bounding volume (such as a cylinder), that bounds the extremes of the character's motion during a single cycle of the locomotion gait, we have effectively reduced the navigation planning problem to the 3-dimensional problem of planning the motion for an oriented disc moving on a plane. Figure 3.4 shows one of the characters used in our experiments along with its bounding cylinder.

As previously mentioned, for character navigation on flat-terrain, the most important DOFs are the position and orientation (x, y, θ) of the base of the character on the walking surface. We can reduce the dimensionality further still by having the the orientation (forward-facing direction) of the character computed by a controller during the path following phase (see Section 3.6). Thus, we need only to consider the position (x, y) of the base of the character during the planning phase. This means that path planning for flat-terrain navigation can be reduced to planning collision-free paths for a circular disk in 2D.

3.3 Algorithm Outline

There are four basic parts to the navigation algorithm. Each step is briefly described below, and then discussed in detail in subsequent sections:

1. **Initialization:** A general 3D description of the environment and characters suitable for rendering is provided, along with a goal location. Motion capture data for a single cycle of a locomotion gait along a straight line is pre-processed as described in Section 3.4.

2. **Projection:** The planning phase begins by projecting all obstacle geometry within the vertical extents (the height range) of the character to a discretized 2D bitmap grid. The projected obstacles are "grown" by a character's *bounding radius*, so that the grid represents an approximate map of the free space.

3. **Path Search:** The embedded graph defined by the grid from the previous step is searched directly for a collision-free path using any standard dynamic programming technique (e.g. Dijkstra's algorithm, or A* with some appropriate heuristic function). In this case, we use a modified version of Dijkstra's algorithm optimized for searching square grids. If planning is successful, the complete path is sent to the path following phase. Otherwise, the controller is notified that no path exists.

4. **Path Following:** Cyclic motion capture data along with a PD controller on the position and velocity is used to generate the final motion for the character as it tracks the computed path. If no path exists, an appropriate stopping motion or waiting behavior is performed.

The path planning method adopted here is one instance of an *approximate cell decomposition* method[Lat91]. The $\mathcal{C}$-space (in this case, the walking surface) is discretized into a fine regular grid of *cells*. All obstacles are projected onto the grid and "grown" as detailed in Section 3.5. Hence, the grid approximately captures the free regions of the search space at the given grid resolution, and can ultimately be used for fast collision checking.

3.4 Initialization

Environment Representation

The navigation strategy imposes no restrictions on the makeup of the environment obstacles, so long as it is possible to project their geometry to a plane. Most virtual environments currently used in practice contain obstacles comprised of a collection of polygons suitable for rendering. The polygonal models may be arbitrarily complex, with holes, discontinuities, and self-intersections (i.e. a "polygon soup").

The walking surface is assumed to be flat in the implementation described here. It may be possible to remove this restriction by first mapping an uneven surface with obstacles onto a plane prior to projection. However, this has currently not been implemented. The walking surface is also assumed to be finite. It should have maximal extents that are of reasonable size compared to the dimensions of the character. This restriction ensures that the resolution of the grid is not too coarse, while the number of cells is kept to a minimum. We prefer to have fewer cells, since this affects the speed of the path search (see Section 3.5). A useful rule of thumb is to restrict the relative size of a bitmap cell to be no larger than the character's bounding radius.

Since the number of cells grows quadratically with the size of the walking surface (i.e doubling the maximal extents quadruples the number of cells), it helps to have smaller, finite-sized virtual environments. For indoor scenes, these constraints typically pose little difficulty. For environments with large open expanses (e.g. outdoor environments), additional data structures are needed to manage the free space and limit the fine-grained path searching to a local area. Possible solutions to this problem involve a number of design decisions and tradeoffs that typically depend upon the application (see the paragraph on *Large Environments* in Section 3.8).

The Goal Location

The goal location is specified either directly by the user, or derived from a high-level script. It determines a location in the virtual environment relative to the character's starting location, that the character is commanded to navigate towards.

Although the goal location will typically map to a single cell in the grid, it should be noted that the goal location need not occupy a mere single cell. Rather, it could represent a collection of several cells (e.g. the interior cells of a given room if the task command doesn't specify a precise location within the room). Moreover, the collection of cells can be disjoint. For example, if the task command is to exit a building, goal locations can be specified at multiple exit doors and the planner will return a path to the nearest one.

Preparing the Motion Capture Data

As a one-time pre-processing step, the motion capture data to be used as the *navigation gait* is prepared for the path following phase. Here we assume that the original motion capture data is *cyclic* (the motion of the final frame smoothly connects to the first frame), and roughly follows a straight line in the global (world) frame. To begin, the transformation of the root joint (for example, the *Hip* joint) for each data frame is factored into two parts:

1. The transformation of a *base point* (the point of projection of the origin of the root joint to the walking surface) moving in a straight line at a constant speed throughout the duration of the cycle.

2. The relative transformation of the root joint with respect to the moving base point.

Figure 3.5 illustrates the decomposition. The first transformation relates the world frame to the base frame, while the second transformation represents the relative offset of the hip frame to the moving base frame. Factoring the motion data in this way is commonly known as computing the motion with respect to the *treadmill frame of reference*. The name derives from the fact that if the resulting motion is played back while holding the base point fixed, it will appear as if the character is moving on a treadmill.

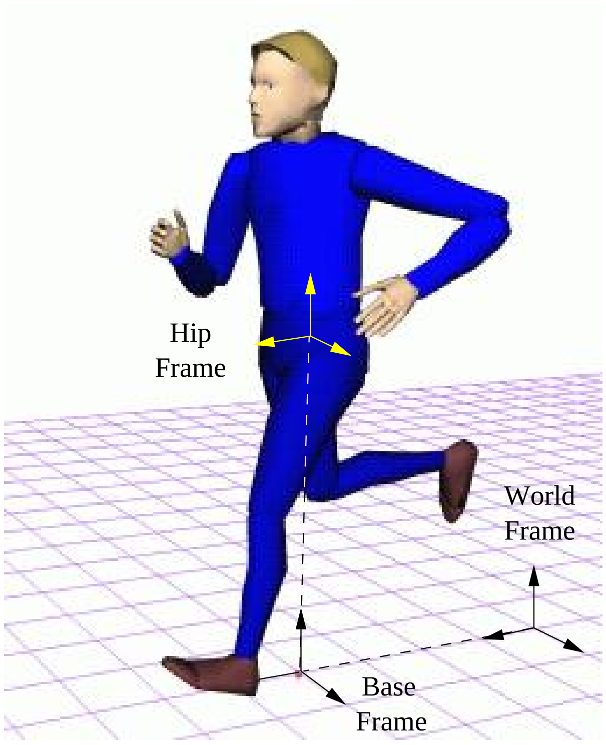

Figure 3.5: Computing the “treadmill” frame of reference.

The primary reason for computing this treadmill frame decomposition is to allow us to apply the motion capture data to an arbitrary curve much more easily. Traditionally, motion capture data comes in a format that specifies the transformation of the root joint relative to some fixed global frame. Instead of trying to control the root joint directly to follow a computed path, we can control the base point, which by construction travels in a straight line and at a constant speed throughout the duration of the motion cycle.

The trajectory of the base point is computed by first projecting the origin of the root joint onto the walking surface for both the first frame and the last frame of the motion cycle. The base point should move in the direction corresponding to the translational difference between these two projected points. The total distance traveled by the base point should be the length of this translational difference. The velocity of the base point is simply this distance divided by the time elapsed during playback of the motion cycle. The base point is the point to which we will map the center of our oriented disc model for path following. The direction of motion of the base point corresponds to the forward-facing direction of the disc model.

The velocity of the base point is the *canonical average velocity* V of the root joint over the duration of the motion. If the base point is made to move along a straight line at this velocity, the motion capture data will appear as it does in its raw form. Since we will be controlling the base point for path following, it will move along curved paths, and at differing velocities. In order to improve the appearance of the motion for velocities other than the average velocity, we can incorporate other motion capture data sets taken at different walking or running speeds. An alternative to this, is to pre-compute a table of interpolation factors for the joint rotations, indexed by velocity. Smaller base point velocities will result in smaller joint rotations. For example, for a basic walk cycle, the interpolation factors can be pre-selected for each base point velocity such that sliding of the character's feet along the walking surface is minimized. This simple interpolation method is used in the implementation described here, and results in fairly reasonable motions for transitioning between a standing position to a full-speed walk, and in coming to a stop at the end of a path. A more sophisticated method might utilize inverse kinematics (IK) or constraint-based techniques[Gle98] to prevent the feet of the character from sliding on the walking surface.

3.5 Planning

The path planning phase consists of projecting the obstacles to a grid to obtain a "map" of the walking surface, and then searching the resulting embedded graph for a free path. Performing these two steps of projection and search, we will refer to as one *planning cycle*. For *static* environments (i.e. non-moving obstacles), the projection step needs only to be performed once. The resulting grid can then be reused to search for a new path if the start or goal locations change. For *dynamic* environments, where the locations of other characters or objects in the environment can change without warning, a complete planning cycle must be performed whenever objects move. This is due to the fact that object motions may invalidate the optimality or the collision-free nature of the current path being followed, and necessitate re-planning.

Obstacle Projection

All obstacle geometry within the character's height range $[z_{min}, z_{max}]$ is projected orthographically onto the grid. The height range limitation assures that only obstacle geometry that potentially impedes the motion of the character is projected. Cells in the grid are marked as either *FREE* or *NOT-FREE*, depending upon whether or not the cell contains any obstacle geometry. The resulting 2D bitmap $\mathcal{B}$ now represents an occupancy grid of all obstacles within the character's height range projected at their current locations onto the walking surface. Note that a binary bitmap such as this does not distinguish between small, low obstacles (that could potentially be stepped on or over), large columns (that must be circumvented), and overhangs (which the character could duck under). However, the overall navigation strategy could potentially be extended to include such kinds of motion (see Section 3.8).

In the robotics literature, a technique of growing the obstacles and shrinking the robot was introduced by Udupa[Udu77] and later refined by Lozano-Perez and Wesley[LPW79]. We adopt a similar approach here. Cells in $\mathcal{B}$ marked as *NOT-FREE* are "grown" by R, the radius of a cylinder that conservatively bounds the character's geometry. This is done by marking all neighboring cells within a circle of radius R from each original *NOT-FREE* cell as *NOT-FREE*. In effect, this operation computes the Minkowski difference of the projected obstacles and the character's bounding cylinder, thus reducing the problem of planning for a circular disc, into planning for a point object. To check whether or not a disc whose center is located at (x, y) intersects any obstacles, we can simply test whether $\mathcal{B}(x, y)$, the cell in $\mathcal{B}$ containing the point (x, y) is marked *FREE*.

For relatively simple environments with few total polygons, software rasterization algorithms are appropriate for computing the 2D projection. For large environments with tens of thousands of polygons, the obstacle projection operation can be performed much faster using rasterizing computer graphics hardware[LRDG90, Ban98]. Here, the rendering pipeline enables us to quickly generate the bitmap needed for fast point collision checking. An orthographic camera is positioned above the scene pointing down along the negative normal direction of the walking surface with the clipping planes set to the vertical extents of the character's geometry. The *near* clipping plane of the camera is set to correspond to the maximum vertical height of the character's geometry, while the *far* clipping plane is set to be slightly above the walking surface. The other dimensions of the orthographic view volume are set to enclose the furthest extents of the walking surface at the current grid resolution as depicted in Figure 3.6. All geometry within the view volume is projected (rendered) into either the back buffer or an offscreen bitmap via the graphics hardware.

Further performance improvements are possible. Since we are only concerned with whether or not any obstacle geometry maps to each pixel location, we can effectively ignore the pixel color and depth value[1]. Thus, we can turn off all lighting effects and depth comparisons, and simply render in wireframe all obstacle geometry in a uniform color against a default background color.

Under most reasonable graphics hardware systems, these simplifications will significantly speed up rendering performance. However, we can do even better. If the graphics hardware supports variable line-widths and point-sizes, we can perform the obstacle growth at no additional computational cost! We simply set the line-width and point-size rendering style to correspond to the projected pixel length of R, the radius of the character's bounding cylinder. In this way, we can efficiently perform both obstacle projection and growth simultaneously.

Path Search

The bitmap $\mathcal{B}$ is essentially an approximate map of the occupied and free regions in the environment. Assume that the goal location $\mathcal{G} = (g_x, g_y)$ and the starting location $\mathcal{S} = (s_x, s_y)$ in the bitmap are both *FREE*. Let us consider $\mathcal{B}$ as an embedded graph of cells, each connected to its neighboring cells. For a bitmap (a square grid) each interior cell has 8 neighboring cells, while border and corner cells have 5 and 3 neighboring cells respectively. Each pixel in the bitmap corresponds to a node in the graph, and edges are placed between pairs of neighboring pixels that are both *FREE*. By searching the graph defined by $\mathcal{B}$, we can conservatively determine whether or not a collision-free path exists from the start location to the goal location at the current

[1]One possible use for the depth value could be to extend the navigation algorithm to handle irregular terrain, or the ability of the character to step over low obstacles (see Section 3.8).

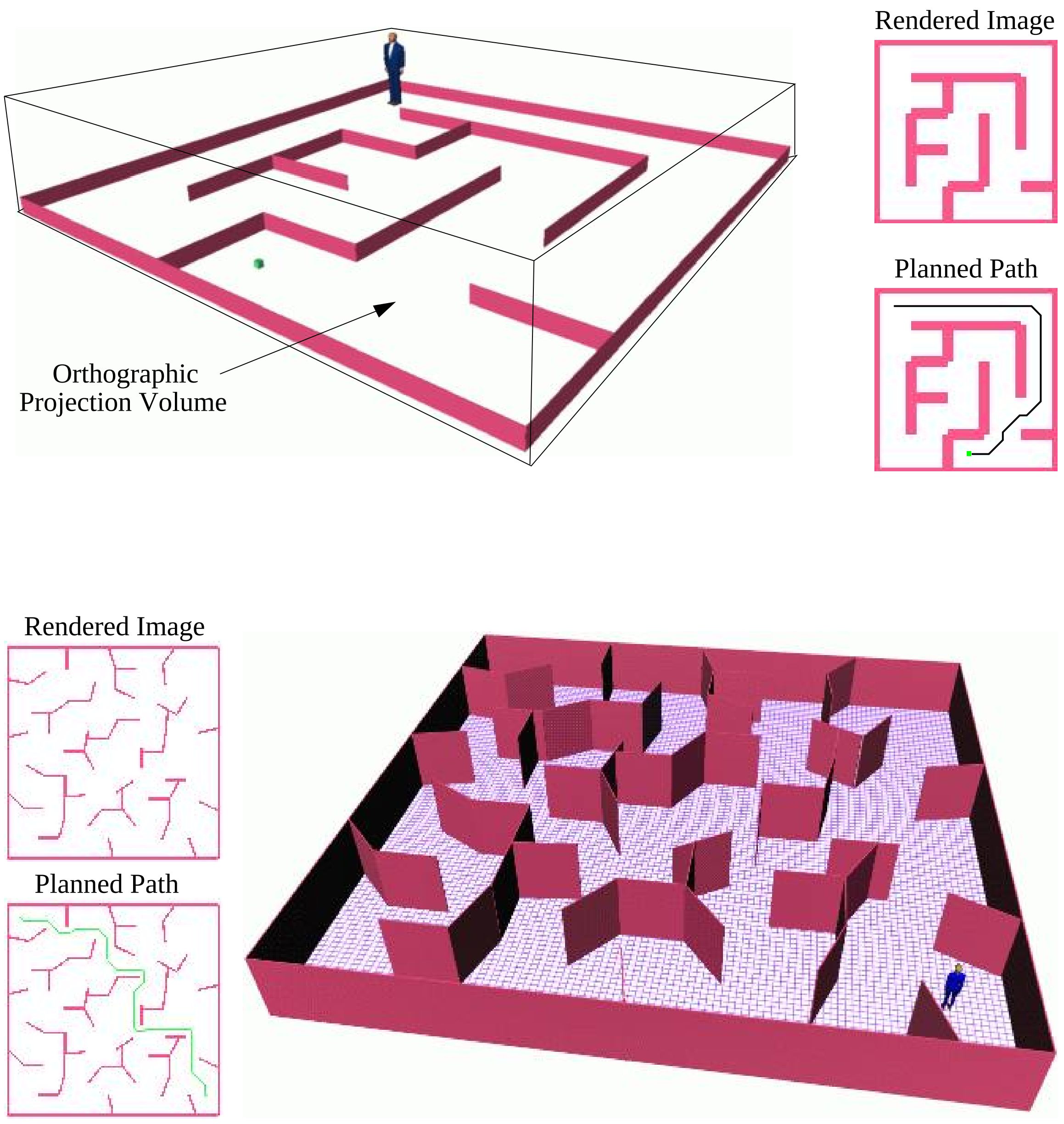

Figure 3.6: The view volumes resulting from an overhead orthographic camera. The near and far clipping planes of the camera are defined according to the vertical extents of the character geometry. All obstacle geometry contained within the view volume is rasterized on a 2D grid for planning. The resulting bitmap is searched directly for a collision-free path connecting the character's starting position to the goal.

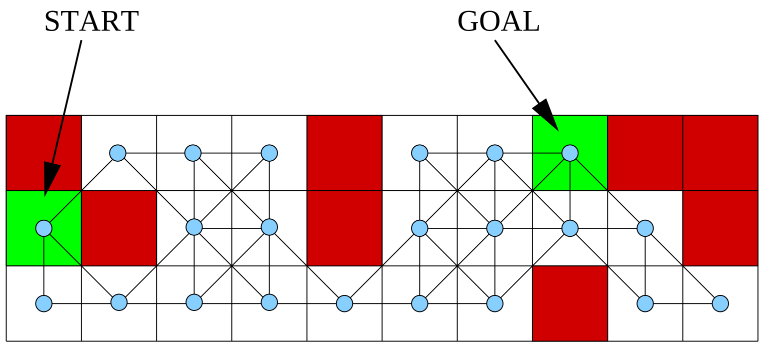

Figure 3.7: Searching an embedded graph defined by a grid

grid resolution. Figure 3.7 shows a small example of an embedded graph defined by a bitmap grid.

If we assign a relative cost to each arc between cells, we can search for a path that connects $\mathcal{S}$ and $\mathcal{G}$ while minimizing our cost function. Here, we use a relative measure of the Euclidean distance between adjacent cells as our cost. Arcs between directly adjacent cells are assigned a relative cost of 1, while diagonally-adjacent cells are assigned a relative cost of $\sqrt{2} \approx 1.4$. Our task has been effectively reduced to that of searching for a path between two nodes in a weighted graph.

Searching a Weighted Graph

Any number of standard dynamic programming techniques may be used to search the graph defined by $\mathcal{B}$. Some possibilities include Dijkstra's algorithm, best-first search (BFS), and A*. For a detailed review of graph searching techniques and analysis, see [KK88, Mit98]. However, most path-finding algorithms are written for arbitrary graphs rather than regular grids. We prefer to use something that can take advantage of the grid nature of the graph.

With this goal in mind, we implemented a modified version of Dijkstra's algorithm that has been optimized for finding paths on square grids. In particular, it reduces the overall total number of expected neighboring node expansions. The same technique can be applied to optimize the A* algorithm, though the size of the grids used in our experiments were not large enough to justify the extra cost of computing a

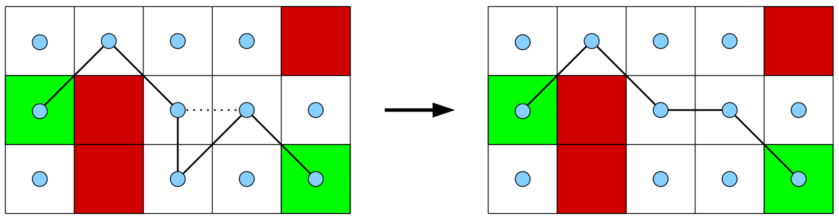
Figure 3.8: Paths which form angles less than 90 degrees can always be shortened.

heuristic function (see Section 3.8). Fixed-point math is used in calculating the distance and cost computations, resulting in a planner that runs almost entirely using fast integer arithmetic. In addition, the implementation makes extensive use of inline functions and uses an array-based indexed priority queue data structure for improved performance.

This search strategy will always return a path containing a list of *FREE* cells between $\mathcal{S}$ and $\mathcal{G}$ if one exists at the given grid resolution. Moreover, since a relative measure of the Euclidean distance is used as the single-step cost between cells during the search, the returned path will be of minimal length (for any 8-neighbor path at the given grid resolution). From the list of free cells connecting $\mathcal{S}$ to $\mathcal{G}$, a final path $\mathcal{P}$ is constructed by linking in sequence the line segments connecting the centers of adjacent cells along the path.

Fast Planning on a Square Grid

In this section, we show how to speed up path searching on a regular 2D grid with edge weights assigned as described earlier. The basic idea is to modify Dijkstra's algorithm so as to reduce the overall total number of expected neighboring node expansions by limiting the number of neighboring cells inserted into the priority queue. We can do so by exploiting the geometry of optimal 8-neighbor paths on a square grid.

The minimum angle an optimal path on a square grid can ever form with itself is 90 degrees. This fact is apparent by noting that any 8-neighbor path that forms an angle smaller than 90 degrees can be made shorter by "cutting corners" (see Figure 3.8). Because of this, we can effectively limit the number of neighboring node expansions by ignoring those cells whose inclusion would produce a path with an angle less than

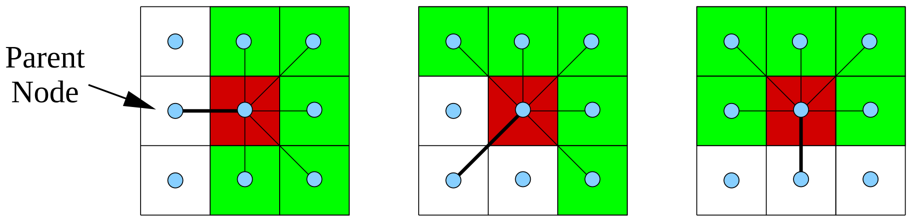

Figure 3.9: Reducing the number of neighbor node expansions.

90 degrees. By keeping track of which direction a given node was expanded from, we can reduce the number of of potential neighbor node expansions from 8 to 5 (see Figure 3.9). For example, suppose we extract a node from the queue that was inserted when its neighbor node to the “west” was expanded (the left portion of Figure 3.9). Obviously, there is no need to consider the node to the west when the current node is expanded. In addition, there is no need to consider the neighbor nodes to the northwest and southwest. This is because the edges connecting them to the current node cannot be part of an optimal path that includes the edge between the current node and its western neighbor.

Properties of Computed Paths

There are several desirable properties of the paths computed by the search phase of the planner:

1. **Optimality:** Paths generated by the planner are optimal according to the graph metric used. An optimal path may not be unique, as there may be several optimal paths with an equivalent cost. The planner is guaranteed to return one of them.

2. **No Crossings:** This property derives naturally from path optimality. Computed paths will never cross themselves, otherwise any path loops could simply be eliminated to yield a lower-cost path.

3. **Bounded Curvature:** The computed paths have a bounded curvature due to the geometry of optimal paths on a square grid (i.e. the minimum angle a path can ever form with itself is 90 degrees).

Complexity Analysis

If planning is successful, the complete path is sent to the path following controller. Otherwise, the controller is notified that no path exists. The planner is *resolution-complete*, meaning that it is guaranteed to find a collision-free path from $\mathcal{S}$ to $\mathcal{G}$ if one exists at the current grid resolution, and otherwise report failure[Lat91].

The running time of the obstacle projection step is proportional to the number and geometric complexity of the obstacles. Searching for a path in the bitmap using Dijkstra's algorithm runs in quadratic time with respect to the number of free cells in the grid. Extracting the path (if one exists) runs in time proportional to the length of the path. Overall, this planning strategy can be implemented very efficiently and robustly even for complex environments. Note that the search phase of the planner may run *faster* for complex, obstacle-cluttered environments, since such environments result in fewer free cells to search.

3.6 Path Following

Simply computing a collision free path in the environment is not enough to produce realistic animation. The navigation system described here uses cyclic motion capture data applied to the joints of the character, plus a simple low-level proportional derivative (PD) controller to follow the computed path. The quality of the final motion arises primarily from the motion capture data, but there are other alternatives that could be used for path following. So-called "footstep"-driven animation systems could be applied to place the feet at points nearby the computed path, along with real-time inverse kinematics (IK) to hold them in place[BC89, BTMT90, vdP97]. As computer processing power increases, physically-based models of the character dynamics along with complex controllers such as the one presented in [Hod96] could also potentially be used to simulate locomotion gaits along the path. For the purposes of these experiments, applying cyclic motion capture data proved to be a fast and simple method of obtaining satisfactory motion.

Although the path planning and following concept generally applies to many types of characters and motions, we will concentrate on generating walking or running motions for a human-like biped. We would like the character's motion to be smooth and continuous, natural-looking, and follow the computed path as closely as possible. Though many kinds of path following techniques could potentially be used, the one described here was chosen for its simplicity and efficiency.

Mathematical Model

Human figure walking or running is essentially a quasi-nonholonomic system, since the typical turning radius is usually subject to some minimum value depending upon the velocity. Of course, a person can turn in place, but generally, this will only happen at the beginning or end of a path following procedure, not in the middle of a path. Humans tend to only walk forward, not backward or sideways (no direction reversals during path following).

With this in mind, the path following phase is modeled as one involving an oriented disc smoothly tracking a geometric path in the plane. The disc center corresponds to the projected point at the base of the character's geometry, and the orientation of the disc corresponds to the character's forward-facing direction as illustrated in Figure 3.10. The linear velocity of the disc is constrained to always lie along the forward-facing direction. This corresponds to the character's ability to walk or run forward. Turning is modeled by considering the disc's rotational velocity about its center.

A discrete time simulation of the following state variables is used:

$\mathbf{p}_t$	position (x_t,y_t) of the disc center
θ_t	orientation (forward-facing direction)
v_t	linear speed along the direction of θ_t
ω_t	angular speed about $\mathbf{p}_t$

The tuple $(\mathbf{p}_t, \theta_t, v_t, \omega_t)$ represents the simulated state of the character at time t. At

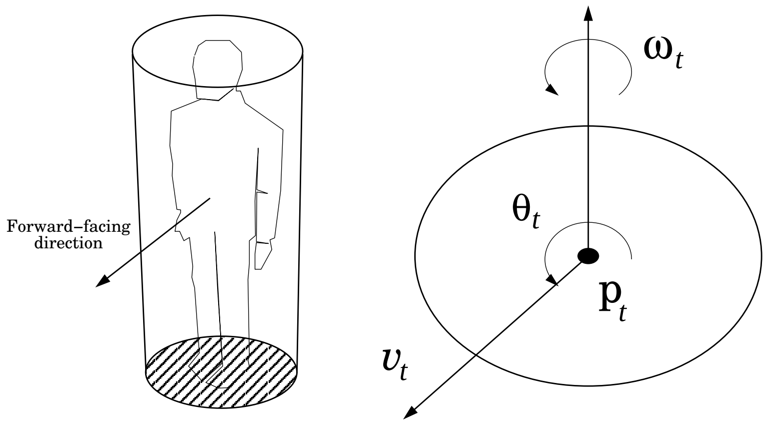

Figure 3.10: The controller model considers the character's motion as that of an oriented disc in the plane. The center of the disc corresponds to the projection of the origin of the root joint of the figure onto the walking surface.

each time step, any combination of the following two controls may be applied:

a_t	linear acceleration along the direction of θ_t
α_t	angular acceleration about $\mathbf{p}_t$

These controls model the four basic fundamental actions for our character: (*speed up, slow down, turn left, turn right*). Speeding up and slowing down are represented by positive and negative values of a_t respectively. Similarly, positive values of α_t correspond to left turns, while negative values correspond to right turns. Section 3.6 explains how these controls are calculated at each time step.

Once the controls a_t and α_t have been specified, the state variables are integrated forward discretely by the time step Δt. In these experiments, simple fixed-step Euler integration was used, but more sophisticated integration methods may be used if desired. For Euler integration, the state propagation equations can be approximated by:

$$\begin{aligned}
x_{t+\Delta t} &= x_t + (v_t \cos\theta_t)\Delta t \\
y_{t+\Delta t} &= y_t + (v_t \sin\theta_t)\Delta t \\
\theta_{t+\Delta t} &= \theta_t + \omega_t \Delta t \\
v_{t+\Delta t} &= v_t + a_t \Delta t \\
\omega_{t+\Delta t} &= \omega_t + \alpha_t \Delta t
\end{aligned}$$

The simulation proceeds in this fashion iteratively. As long as the values of the controls are reasonable relative to the size of the time step Δt, the motion will be smooth and continuous.

Calculating the Controls

In this section, we describe a simple method for computing the two control variables, a_t and α_t for our character during each time step of the simulation. The method is based on proportional derivative (PD) control. Given the current state of the system $(\mathbf{p}_t, \theta_t, v_t, \omega_t)$, a *desired* state $(\hat{\mathbf{p}}_t, \hat{\theta}_t, \hat{v}_t, \hat{\omega}_t)$ is calculated. The controls a_t and α_t are then computed to move the system towards the desired state.

The computation proceeds as follows: Given a path $\mathcal{P}$ computed by the planning phase, a desired position along the path $\hat{\mathbf{p}}_t$ is calculated relative to the current position $\mathbf{p}_t$. The desired position is typically set to be slightly ahead of the point on the path that is closest to $\mathbf{p}_t$, as this tends to smooth out any sharp corners on the path. Next, the desired orientation $\hat{\theta}_t$ is computed so as to face the character towards the desired position $\hat{\mathbf{p}}_t$. The desired angular speed $\hat{\omega}_t$ is set proportional to the difference (error) between the current orientation θ_t and the desired orientation $\hat{\theta}_t$. The desired linear speed $\hat{v}_t$ has three alternatives:

1. If the error in orientation is *small*, $\hat{v}_t$ is simply set to be the canonical average velocity V of the motion capture cycle.

2. If the error in orientation is *large*, the character is facing the wrong direction, so the speed is set to be some small non-zero value to force the character to slow down and turn around.

3. If the character is nearing the end of the path (the goal location), $\hat{v}_t$ is calculated proportional to the difference between the current position and the goal location. This will cause the character to slow down and ultimately stop at the goal. In order to guarantee that the character eventually comes to a stop and doesn't overshoot the goal, it is useful to define a goal region (a small *epsilon*-neighborhood around the goal location). As soon as $\mathbf{p}_t$ comes within a distance ϵ of the goal location, $\hat{v}_t$ is automatically set to zero (forcing the character to halt).

After $\hat{v}_t$ is obtained, the controls a_t and α_t are calculated. The linear acceleration a_t is set proportional to the difference between the current and desired linear speed, while the angular acceleration α_t is set proportional to the difference between the current and desired angular speed. The state of the system is integrated forward by a discrete time step Δt, and the entire process is repeated for the next time step.

All of the calculations involving proportional derivative terms above require the following gains to be specified:

k_p	position gain
k_θ	orientation gain
k_v	linear speed gain
k_ω	angular speed gain

The gains represent how quickly errors (differences between the current and the desired) are resolved. Since a discrete time step is being used, some care must be taken when setting the gains. Gains set too high will cause oscillations (an *under-damped* system), while gains set too low will fail to correct errors (an *overdamped* system). Setting the gains properly will result in a *critically-damped* system, that asymptotically corrects errors without overshoot. For more detailed information and analysis, see any textbook on feedback control, for example [Isi95].

As long as the gains are set to reasonable values relative to the size of the time step Δt, the resulting motion will be smooth and continuous. The controller will

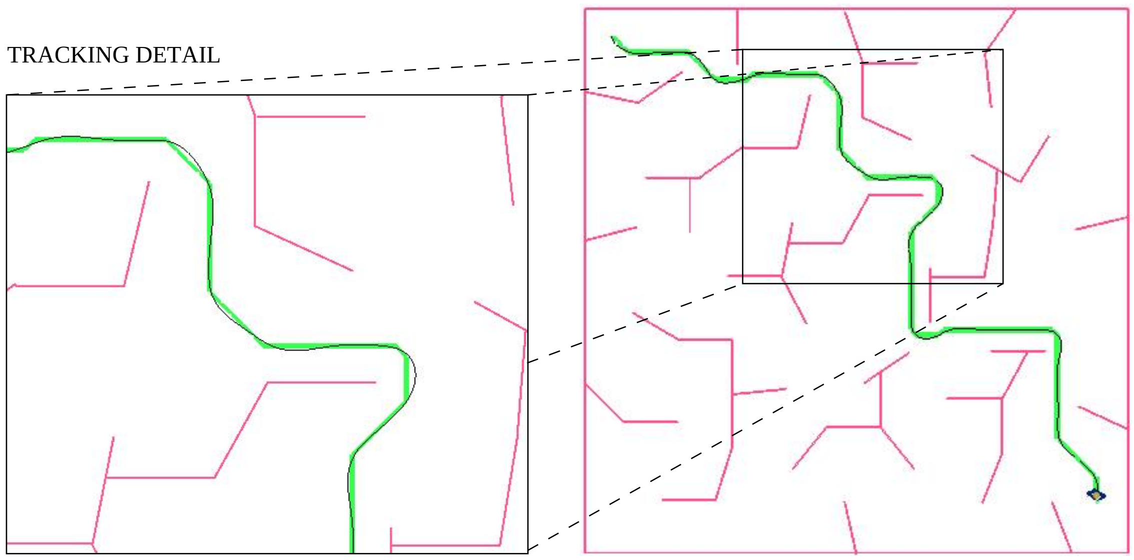

Figure 3.11: Base point tracking performance for path following.

exhibit better tracking performance (less deviation from the planned path) for locomotion gaits with lower canonical speeds (i.e. lower values of V). Since the paths computed by the planner have a bounded curvature, the deviations from the path are relatively small in practice. An example of the tracking performance for following a path through a maze is shown in Figure 3.11.

To summarize, all of the control calculations are listed below. First, we calculate the desired state $(\hat{\mathbf{p}}_t, \hat{\theta}_t, \hat{v}_t, \hat{\omega}_t)$, and then compute the controls needed to move towards the desired state.

$$
\begin{aligned}
\hat{\mathbf{p}}_t &= (\hat{x}_t, \hat{y}_t) \text{ from path } \mathcal{P} \\
\hat{\theta}_t &= \text{atan2}(\hat{y}_t - y_t, \hat{x}_t - x_t) \\
\hat{\omega}_t &= k_\theta(\hat{\theta}_t - \theta_t) \\
\hat{v}_t &= \begin{cases} \text{canonical speed } V & \text{if } |\hat{\theta}_t - \theta_t| \leq \theta_{turn} \\ \text{small speed } \epsilon & \text{if } |\hat{\theta}_t - \theta_t| > \theta_{turn} \\ k_p(|\hat{\mathbf{p}}_t - \mathbf{p}_t|) & \text{if near the goal} \end{cases} \\
a_t &= k_v(\hat{v}_t - v_t) \\
\alpha_t &= k_\omega(\hat{\omega}_t - \omega_t)
\end{aligned}
$$

In the computation of $\hat{\theta}_t$, atan2(y, x) is the standard two-argument arctangent function.

After all controls are calculated, the state is integrated forward discretely by the time step Δt, as described in Section 3.6. The subsequent state becomes the new location and orientation for the base point of the character. To animate the remaining joints, the current velocity v_t is used to index into the motion interpolation table as described in Section 3.4.

3.7 Multiple Characters

Sharing the Initial Bitmap

By incorporating minor modifications to the navigation algorithm, scenes involving multiple characters are possible. The cost of the projection step can be amortized over several characters by caching the initial projected bitmap $\mathcal{B}$ (prior to obstacle growth). Each character expands the cells marked as *NOT-FREE* in $\mathcal{B}$ by its bounding radius R to produce a final bitmap for planning. If all multiple characters have the same bounding radius, then this step needs only to be performed once for each unique value of R. The same path search algorithm is invoked for each character to connect their start and goal cells (see Section 3.13 for a simple example and Chapter 6 for examples of more complex behaviors involving multiple characters).

Coordinating Multiple Individuals

There are several possible strategies for avoiding collisions between characters. One simple idea is have each character "plan around" the others. For example, we can project the geometry of the other characters at their current locations prior to obstacle growth. Since each character may change its location over time, replanning may be necessary periodically in order to avoid the possibility of a collision between two characters. One simple coordination scheme might replan whenever other moving characters cross a character's current path.

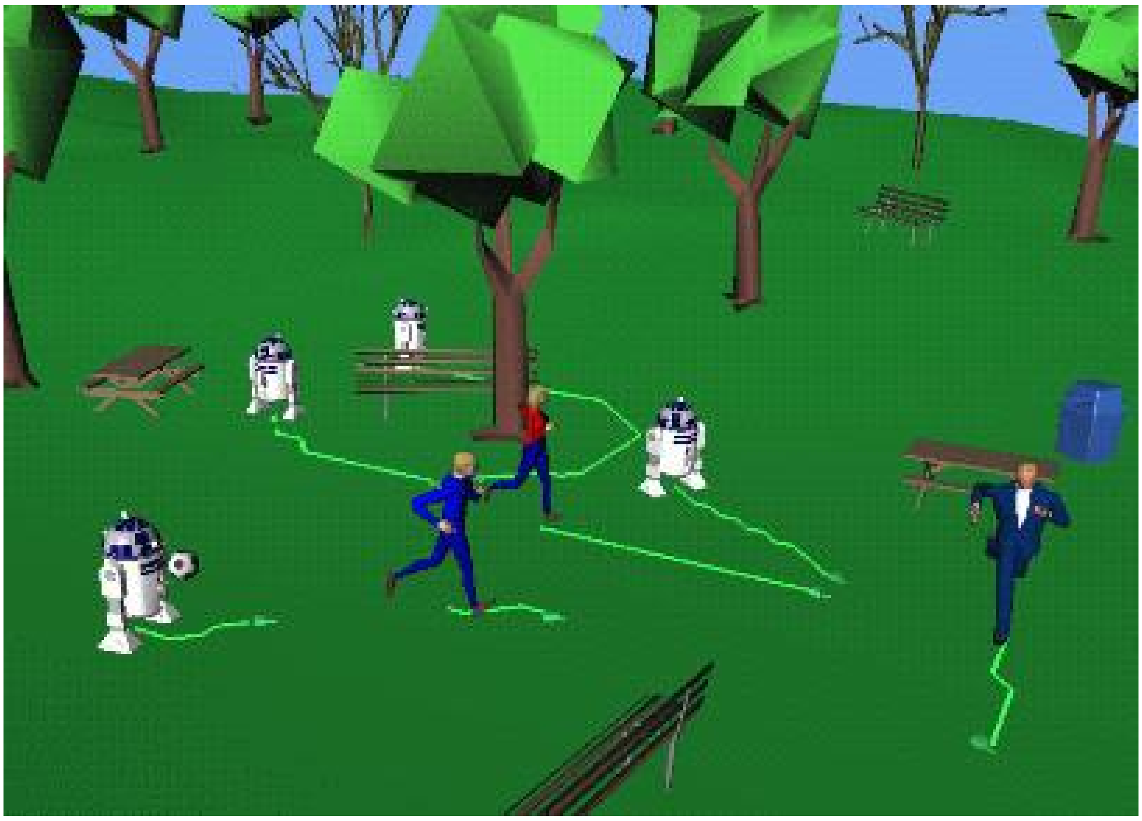

Figure 3.12: Wandering and following behaviors using a velocity-prediction scheme to avoid inter-character collisions.

More sophisticated schemes that account for character velocities are also possible. Instead of planning around the other characters at their *current positions*, each character could plan around the other characters' *predicted future positions*. A character's future position could be calculated by a simple extrapolation of its current actual or estimated velocity. The examples shown in Figure 3.12, Figure 3.13, and those in Figure 6.1 and Figure 6.4 utilize this technique in order to avoid potential inter-character collisions (see Chapter 6).

Customized Controllers

Different characters can also be outfitted with different controllers and motion cycles in order to reflect their unique styles of motion or behavior. Some characters may

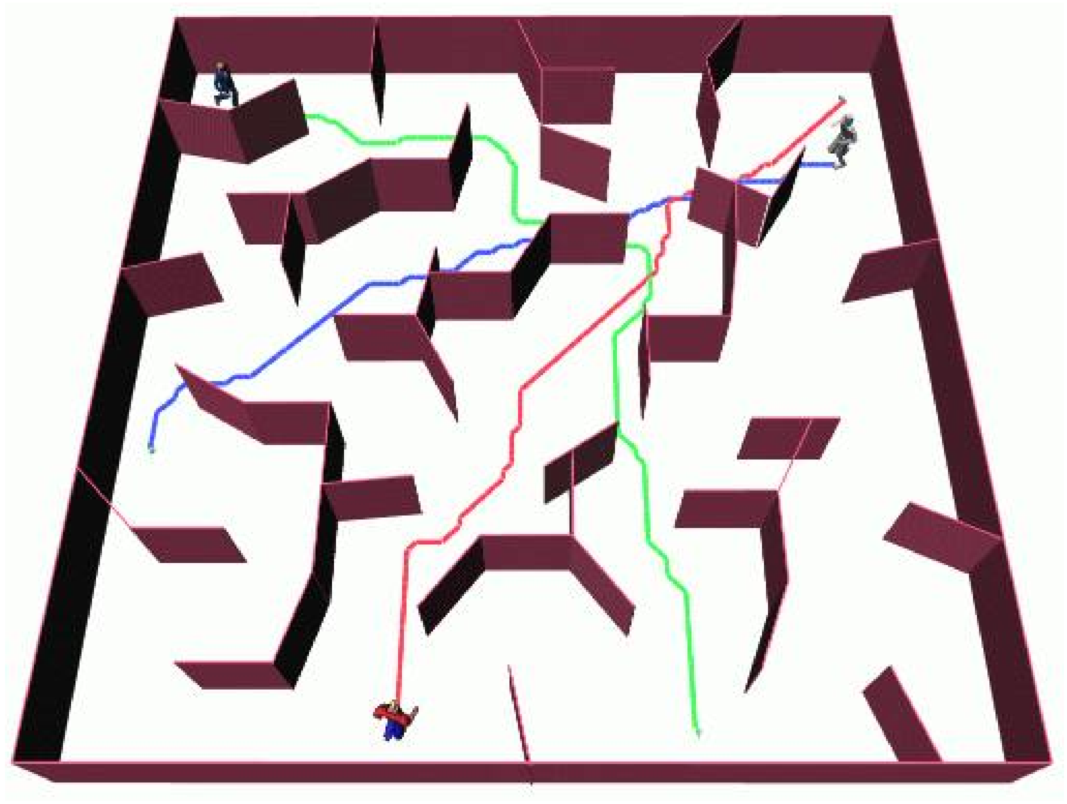

Figure 3.13: Multiple Characters navigating in a maze.

tend to walk slowly, while others frequently jog or run. Alternatively, the speed with which a character follows its current path may depend upon its current emotional state (e.g. sad, frightened, excited).

3.8 Results and Discussion

The algorithms described in this chapter have been implemented on a 200MHz SGI Indigo2 running Irix 6.2 with 128MB of RAM and an SGI EXTREME graphics accelerator card. Interactive performance has been achieved, even for complex scenes with multiple characters (see Figure 3.12 and Figure 3.13). During an active session, the user can click and drag on the goal location or obstacles. The path planner will then calculate an updated, minimal-length, collision-free path (if one exists) in approximately one-tenth of one second on average for a scene with over 10,000 triangle primitives. Running the software on an SGI R10000 with Solid Impact graphics resulted in a speedup by about a factor of 4, making it possible to replan at every frame at rates in excess of 30 frames per second. If a path is found, it is sent directly to the controller, and the character will immediately begin following the new path. Since the goal is allowed to move arbitrarily, it is possible for one character to perform simple tracking or following of another character (see Chapter 6). This can be useful for pursuit games, or having a group of characters follow a tour guide.

The generation of the projection bitmap for path planning was accomplished by rendering all obstacles to the back buffer using standard OpenGL calls. The rendering style optimizations for faster performance as described in Section 3.5 were implemented. The rendered pixel values were subsequently read directly from the framebuffer. The modified Dijkstra's algorithm implementation uses fixed-point math, resulting in a planner that runs almost exclusively using fast integer arithmetic.

For the purposes of path following, the simple PD controller described in Section 3.6 was implemented for a human-like character with 17 joints. Two sets of motion capture data were used in the experiments: a walk cycle and a jog cycle. As expected, the slower canonical speed of the walk cycle exhibits better tracking performance along the path compared with the jogging motion. The values of the gains used for the controller were as follows: (k_p= 1.0, k_θ= 5.0, k_v= 5.0, k_ω= 10.0). These gain values are compatible with standard units of meters, radians, and seconds, which were used throughout the experiments. The value of the time step was $\Delta t = 0.0333$ seconds (1/30 sec).

Figure 3.14 and Figure 3.15 show several snapshots of an interactive session involving a human figure navigating in both a maze and an office environment. The user can dynamically reposition obstacles and the goal location as the character moves. The planner rapidly computes a new path based changes in the environment.

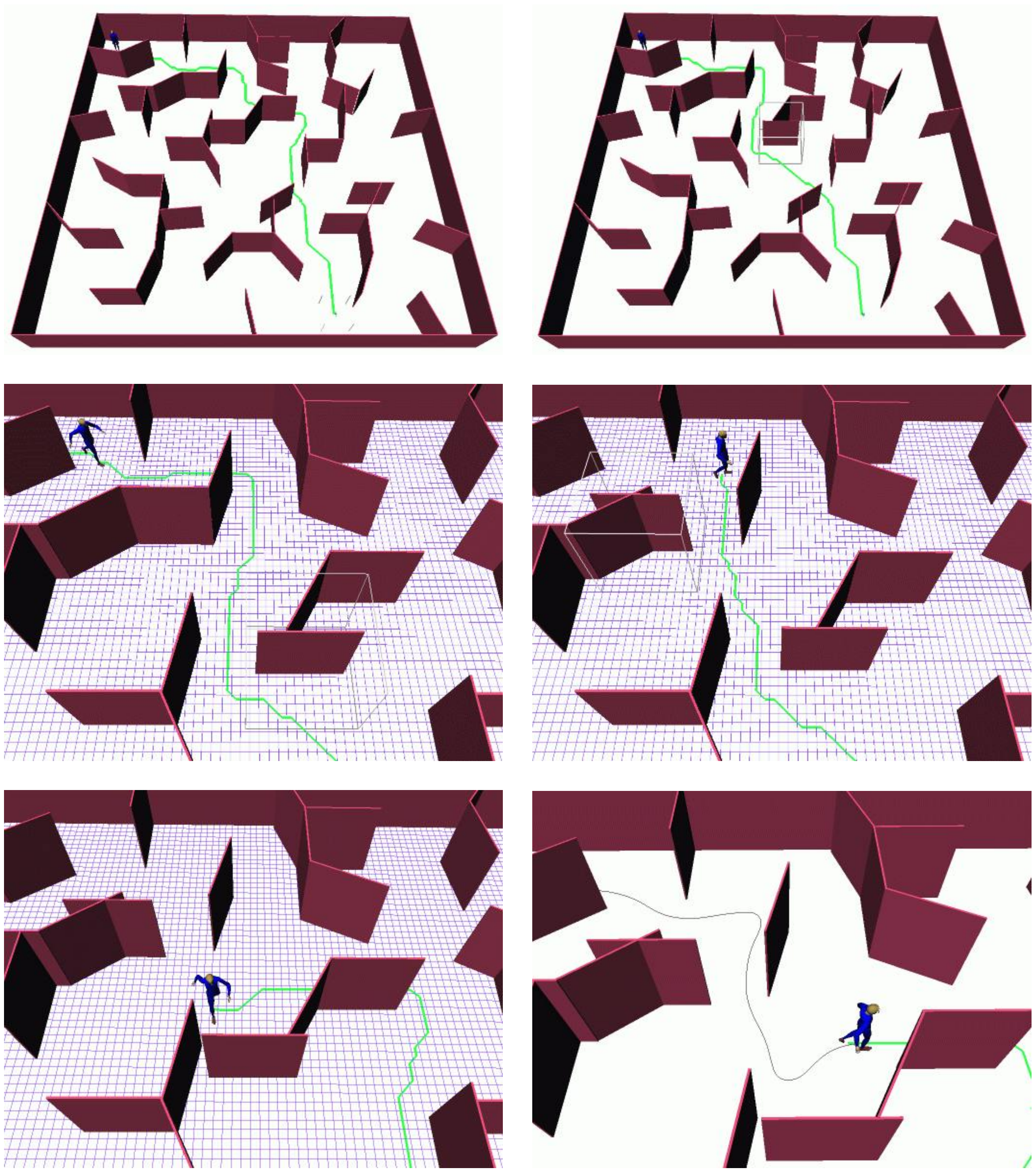

Figure 3.14: A human figure navigating in a maze environment. The path is dynamically recomputed as the goal or obstacle locations change. The detail images illustrate walls being repositioned interactively by the user and the resulting effect on the character's motion. The last image shows the trace of the trajectory taken by the character.

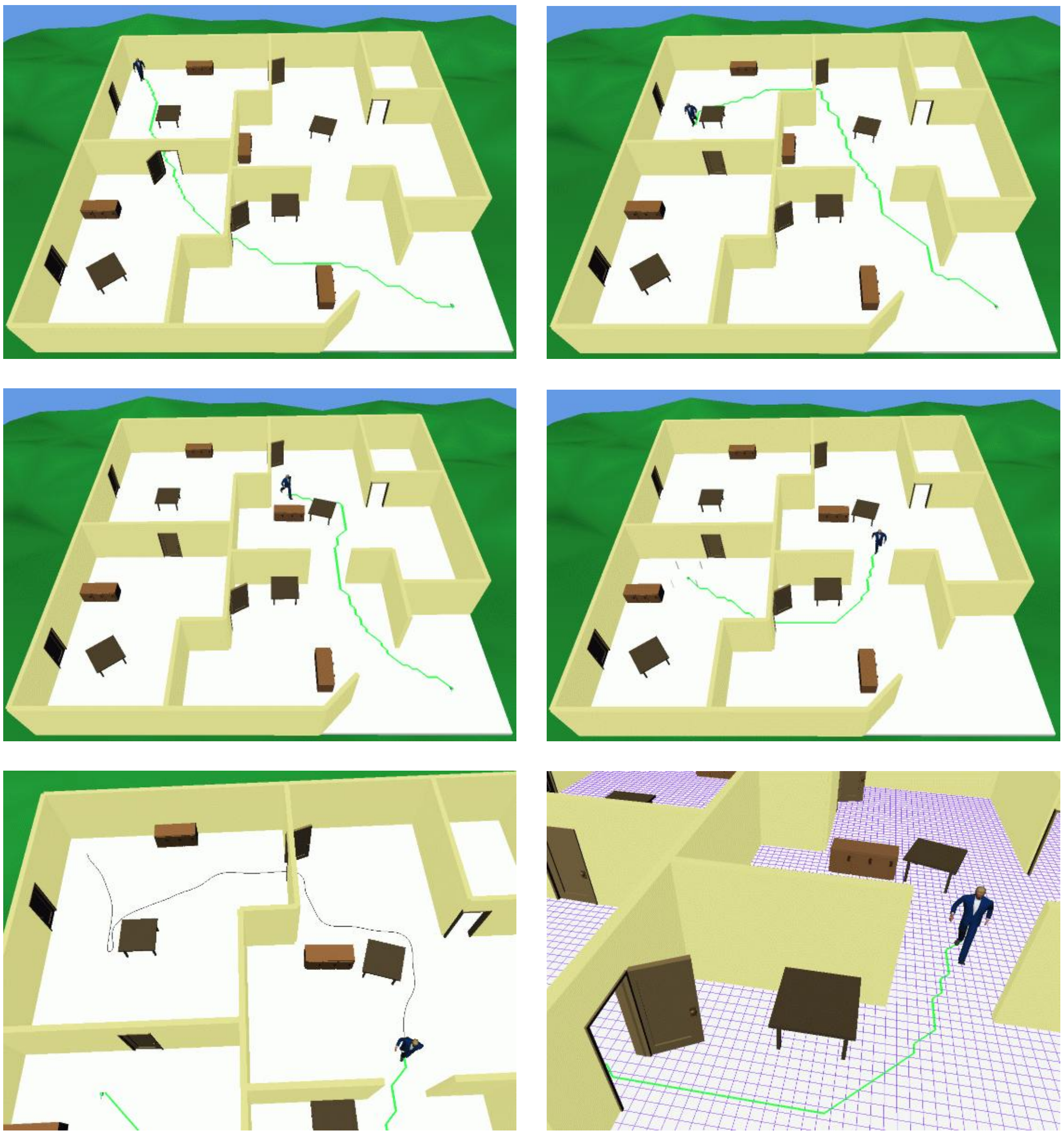
Figure 3.15: A human figure navigating in an office environment. Initially, the character follows a path towards the goal located in the lower-right corner. Later, a door closes forcing the character to select an alternate route. Later still, the character avoids an obstacle placed in its path, and the goal is moved to a new location. The bottom two images show the grid used for planning, and a trace of the trajectory of the character.

Scene (grid size)	Project	Search	Total (msec)
Maze (50 x 50)	9.5	7.2	16.7
Maze (100 x 100)	34.5	37.4	72.0
Maze (150 x 150)	82.9	79.8	163.0
Office (45 x 45)	47.7	13.9	61.7
Office (90 x 90)	62.7	27.4	90.2
Office (135 x 135)	139.2	56.8	196.0

Table 3.1: Average total execution time for planning.

Performance Statistics

The average projection, search, and total *elapsed* execution times during repeated invocations of the planner during an interactive session were tabulated. The timing results are summarized in Table 3.1. All values listed in the table are in units of milliseconds, and were averaged from N = 100 independent trials with varying goal locations and obstacle positions. Different grid resolutions were tested ranging between 45 and 150 cells on a side. The total number of triangle primitives in the Maze scene and the Office scene were 2,780 and 15,320 respectively.

The start and goal locations used in these trials were specifically designed to force a majority of cells in the bitmap to be examined. Simpler path planning queries will produce proportionally faster path search times. As expected, newer and faster machines yield better performance. Although detailed timing statistics were not collected for machines other than the SGI Indigo2, running the software on an SGI R10000 with Solid Impact graphics resulted in a speedup by about a factor of 4.

Discussion

Although useful in its present form, the navigation algorithm could be improved in a number of important ways. Each of the following paragraphs outlines one of several of these issues, along with a discussion of potential methods for improvement.

Uneven Terrain

The planner presented here is limited to finding navigation paths that lie in a plane. Extending the algorithm to handle uneven-terrain is possible, though it might involve redesigning the geometry clipping and projection operations. For static environments, the techniques from [Ban98] or [HC98] are appropriate. Another idea might be to utilize the depth information information that is generated, but is currently being ignored during the projection process. The hardware Z-buffer stores a relative measure of the depth, yielding a simple height field of the environment, which can potentially be used for deciding a navigation strategy.

Large Environments

As mentioned in Section 3.4, the navigation strategy implemented here an its basic form is impractical when dealing with large environments, since the number of cells grows quadratically with the size of the walking surface. For such large open expanses (e.g. outdoor environments), additional data structures are needed to manage the free space and limit the fine-grained path searching to a local area. There is a large body of work on finding optimal and approximately optimal paths in large networks (for a broad overview, see the survey by Mitchell[Mit98]). For example, maximum distance cutoff values can potentially be used to define a local area. Goals outside the local area can then be mapped to the nearest free border cells in the grid. Alternatively, a multi-resolution hierarchical subdivision grid structure can potentially be used to first find a coarse path on the meta-grid, and then successively finer-grained paths in the sub-grids. Other possibilities include the use of a global network of *landmarks* known to be connected by free paths (a similar idea is proposed in [Ban98]).

Path Following

The path following controller as described here is overly-simplistic, and ignores such subtleties of human motion as turning in-place, or side-stepping between narrow openings. Incorporating more motion capture data sets at different velocities, or along curved paths would also likely improve the final appearance of the animation.

In addition, the ability to automatically compute the optimal values for the controller gains based on the simulation constants and the canonical speed of the motion capture data would be a very useful improvement. Knowing these optimal gains might also facilitate the calculation of conservative error-bounds on the performance of the path following controller.

Planner Completeness

Another limitation of the current approach is the approximate nature of the grid, which may cause the planner to fail to find a free path even when one exists. This problem is more pronounced if traversing the environment involves navigating through narrow corridors. Perhaps a multi-resolution strategy or an exact cell-decomposition approach would be appropriate.

Path Searching Alternatives

The path search algorithm need not be the one proposed here. Consider for example, the A* algorithm (since A* can be similarly optimized for grid-based searches). A* operates essentially the same as Dijkstra's algorithm except that a heuristic function that estimates the cost to the goal is added to the cost of a node when inserting it into the priority queue[Nil80]. This causes the algorithm to expand more promising nodes first, potentially saving a significant amount of computation. The size of the grids used in our experiments were not large enough to realize this potential savings given the added cost of computing the heuristic function. However, larger grids will likely benefit greatly from the use of A* along with a reasonable heuristic function. For even larger grids, there exists a linear time algorithm due to Henzinger, Klein, and Rao for computing all shortest paths from a single source in planar graphs[HKR97]. However, due to the overhead involved in running the algorithm, the potential execution time savings may only be realized for very large graphs. Other search strategies of interest cache information for dynamic or unknown environments in order to avoid having to search from scratch each time, such as the D* (Dynamic A*) algorithm[Ste95].

Varied Walking Surfaces

Adjacent cells with different heights or surface properties could be assigned higher costs for traversal (different edge weights in the graph). For example, sections of mud or gravel on the walking surface could be identified during the projection step such that the corresponding cells are assigned higher edge weights (see [Jon97]). Thus, the character will still be able to traverse such regions, but will prefer to go around them if convenient. However, since non-uniform edge weights are used, a standard implementation of Dijkstra's algorithm or A* is needed in order to guarantee the optimality of the computed path.

Additional Obstacle Avoidance Maneuvers

The navigation strategy described in this chapter does not distinguish between small, low obstacles (that could potentially be stepped on or over), overhangs (which the character could potentially duck under), and large columns (that must be circumvented). We would like to incorporate into the planning process the ability to step over low obstacles, or duck under overhangs. One idea might be to classify each projected obstacle based on its vertical extents. Associating specific character poses to be used for collision avoidance with different kinds of obstacles is suggested in [Ban98]. Alternatively, the projected cells of low obstacles could be assigned higher cost edge weights according to their relative height and difficulty to traverse. A similar procedure could be performed for overhanging obstacles. If the final computed path includes such cells, the path following phase could be modified to alter the character's locomotion behavior appropriately. For example, if the controller begins to track parts of the path defined by an overhang, a new set of motion capture data for ducking under an obstacle could be incorporated into the character's motion accordingly.

The same basic idea could perhaps be applied to even more aggressive means of circumventing obstacles, such as utilizing stairs/elevators, climbing, jumping, or crawling. These other modes of navigation (e.g. jumping over barriers such as walls and trenches) could be considered during planning and assigned appropriate relative costs (e.g. energy expended, risk of injury, etc.).

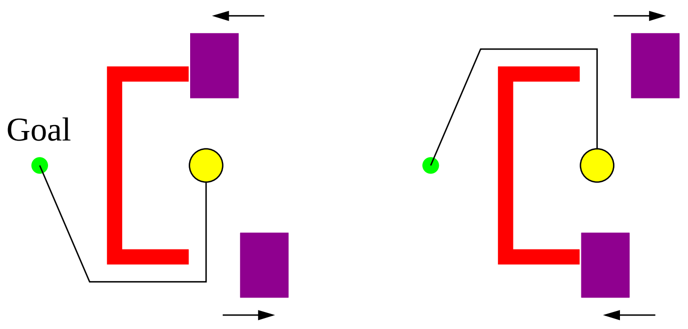

Figure 3.16: Behavior oscillation due to dynamic obstacles. A character attempting to navigate around the wall towards the goal will repeatedly oscillate between following the two paths shown above if the large obstacles at the edges of the wall alternately move back and forth.

Preventing Behavior Loops

The proposed navigation strategy will never cause a character to become trapped in a behavior loop for static environments. Unfortunately, the same guarantee cannot be made for dynamic environments. Although rare, situations can be constructed with dynamic obstacles that will cause a character to oscillate between following two paths initially leading in opposite directions (see Figure 3.16). Incorporating simulated perception often reduces this problem (see Chapter 4), but does not eliminate it entirely. One could potentially keep a record of the character's position and heuristically detect when a behavior loop occurs and alter the navigation strategy accordingly. Similar schemes can also be used if multiple characters become mutually trapped while attempting to navigate around each other.

Obstacle Semantics

Providing the character with simple obstacle semantics can also potentially improve the sophistication of the navigation strategy. Identifying certain objects as "movable" objects might allow a character to dynamically create free space in which to navigate. For example, if a door is unlocked or slightly open, it might be considered traversable free space during planning if the character has the ability to open doors. Another example might incorporate the ability of the character to reposition a chair or other movable obstacle blocking the middle of a hallway. The cells that correspond to the obstacle's location could be assigned relative weights in the graph that account for the time/energy required to move the obstacle out of the way. Some of these issues are discussed further in Chapter 4.

Chapter 4

Sensing the Environment

4.1 Introduction

Sensing for autonomous agents may be defined as *"the process of gathering or receiving data about the environment and the agent itself"*[Che96]. Architectures for autonomous agents usually provide some method of interfacing the agent to the environment through *sensors*. Sensory information can be encoded at both a low level and a high level and utilized by high-level decision-making processes of the agent.

Sensing consists of two fundamental operations: *gathering data* about the environment, and *interpreting the data*. For a physical agent, gathering data involves devices such as cameras, laser rangefinders, and sonars, while interpreting data involves software algorithms (e.g. image segmentation, 3D model reconstruction, object recognition, motion estimation, etc). For an animated agent, sensing is performed via software, without the use of devices. However, the two fundamental operations of gathering and interpreting data remain.

Previous researchers have argued the case for employing some kind of virtual perception for animated characters[RTT90]. This may include one or more of simulated visual, aural, olfactory, or tactile perception. The key idea is to somehow realistically model the flow of information from the environment to the character. Giving each character complete access to all objects in the environment yields unrealistic animation and can be impractical to implement for large environments with many objects.

One way to limit the information a character has access to, is to consider only objects within the immediate vicinity of a character. For example, Reynolds considered all objects and neighboring creatures within a sphere centered around each character in his BOIDS flocking model[Rey87]. However, most characters of interest (including human characters) do not have such omni-directional perception. Rather, sensory information from the environment flows from a primary direction, such as the cone of vision for a human character. Synthetic audition may often be considered to be omni-directional, but for tasks involving navigation and obstacle avoidance, some kind of synthetic vision is needed. In this chapter, we develop an approximate model for synthetic vision that is suitable for interactive animation systems.

4.2 Simulated Visual Perception

Overview

Simulated visual perception, also known as *synthetic vision* or *animat vision*, can provide a powerful means for a character to react based on what it perceives. The key challenge is to devise a scheme for simulated perception that realistically reflects both a character's sensing *capabilities* and *limitations*. In addition, the method must be practical to implement for the target application.

The processes of gathering and interpreting visual data are considered separately. Gathering visual data for an animated agent generally involves determining which object surfaces in the environment are currently visible to a character in order to produce one or more *visual images*. This problem can be viewed as one instance of computing a 3D visibility set (i.e. calculating all visible surfaces from a particular viewpoint given a collection of objects in 3D). Typically, the character has a limited field of view, and possibly a limited range of vision. Thus, computing the visibility set from a character's current location involves intersecting all environment geometry with the character's cone of vision and performing hidden-surface removal to determine which objects are visible. There is an extensive history of visible-surface determination algorithms in the graphics literature, as it is a fundamental operation for most display systems[FKN80]. For a good overview and summary of various methods, see [FDHF90, Tel92].

Once the visual image data is obtained, it is interpreted by the character. Data interpretation can involve a wide range of computations. At one extreme, a character could run a stereo-matching algorithm on a pair of images to obtain a depth map, use image segmentation software and shape-matching algorithms for object recognition, and optical flow for motion estimation. This is what an actual physical agent might do in a real environment. At the other extreme, Z-buffer data can be used directly as a depth map, and the visible object shape and semantic information can be instantly accessed by the character. Clearly, since an animated agent operates in a virtual environment, it has unique advantages over a physical agent. We can exploit some of these advantages in order to meet the performance requirements of interactive animation systems, while approximately modeling the flow of visual information. Naturally, care must be taken to model the visual limitations as well as the capabilities of a character in order to generate realistic behavior.

Approximate Synthetic Vision

There have been several proposals for simulated visual perception. Tu and Terzopoulos implemented a model of synthetic vision for their artificial fishes based on ray-casting[TT94, Tu96]. Blumberg experimented with image-based motion energy techniques for obstacle avoidance for his autonomous virtual dog[Blu96]. Terzopoulos and Rabie proposed using a database of pre-rendered models of objects along with an iterative pattern-matching scheme based on color histograms for object recognition[TR95]. Noser, *et al.* presented a synthetic vision model that uses object false-coloring and dynamic octrees to represent the visual memory of the character[NRTT95].

We are primarily interested in synthetic vision techniques that are practical for real-time systems. Specifically, our goal is to allow an autonomous character endowed with synthetic vision to explore at interactive rates an unknown environment, while maintaining a visual memory or "cognitive map" of what it has perceived. This map may then be used as input to a planning or navigation algorithm. Due to the time constraints inherent to interactive systems, the synthetic vision module must be reasonably fast, and the visual memory model must be simple and efficient to update. In addition, we require the vision and memory modules to handle changing environments, where objects can appear, move, or disappear without warning.

Consider the problem of determining the set of object surfaces currently visible to a character given the environment scene description along with a specification of the character's current viewing frustum. Exact geometric algorithms for 3D visibility are complex, and suffer from poor performance relative to hardware Z-buffers. This is especially true for large scenes with moving obstacles, which typically cannot be preprocessed. Recently, there has been work on designing data structures that can handle dynamic scenes[Com99, CS92], but at present, no software algorithm outperforms a Z-buffer for computing approximate 3D visibility sets. Fortunately, if our goal is to realistically simulate vision for human characters, exact algorithms are not likely needed, since human vision is also not perfect.

We adopt an approach to synthetic vision similar to the one described by Renault [RTT90] and later refined by Noser, *et al.*[NRTT95, NT95]. A preliminary version of the approach is described in [KL99]. The general idea is to render an unlit model of the scene (flat shading) from the character's point of view, using a unique color assigned to each object or object part. The pixel color information is extracted to obtain a list of the currently visible objects. As pointed out by Thalmann, *et al.* in [TNH96], synthetic vision differs from vision computations for real robots, since the data interpretation problem can be made much easier. This allows us to implement a reasonable model of visual information flow that operates fast enough for real-time systems. Furthermore, since the object visibility calculation is fundamentally a rendering operation, all of the techniques that have been developed to speed up the rendering of large, complex scenes can be exploited. This includes scene-graph management and caching, hierarchical level-of-detail (LOD) approximations, and frame-to-frame coherency[Str93, FDHF90].

From the list of currently visible objects computed by the vision module, a set of observations is formed by combining this list with each object's current location.

Finally, this set of observations is added to the character's internal memory model of the environment, and a navigation plan is computed. The navigation planning algorithm uses the fast path planner, motion controller, and cyclic motion capture data, as described in Chapter 3. However, any appropriate real-time navigation planning strategy may be used.

For our system to work, we assume the environment is broken up into a collection of small to medium-sized objects or surface patches, each assigned a unique ID. For example, a single object may be a chair or a door. Large objects such as walls or floors are further subdivided, and each piece is assigned an ID. This assignment could potentially be done automatically using object-level spatial subdivision techniques[Nay92, PY90, FKN80], though some human intervention may ultimately be required. Fortunately, this assignment need only be performed once. Afterwards, a color table is initialized to represent a one-to-one mapping between object IDs and colors.

To check which objects are visible to a particular character, the scene is rendered offscreen from the character's point of view, using flat shading and using the unique color for each object as defined by the object ID (see Figure 4.1). Note that this color is used only when rendering the visibility image offscreen, and does not affect renderings of the object seen by the user, which may be multi-colored and fully-textured. The size of the rendered image need not be very large (usually 200x200 pixels yields sufficient detail).

When rendering is complete, the resulting image pixels are scanned, and a list of visible objects is obtained from the pixel color information. This list may then be combined with other environment state information to encode higher-level aspects of a character's perception. Possible examples include encoding semantic information about certain objects, velocities of objects, and relationships between objects.

4.3 Internal Representation and Memory

Each character maintains an internal model of the world as it explores a virtual environment. Noser, *et al.* used an occupancy grid model (e.g. an octree) to represent the

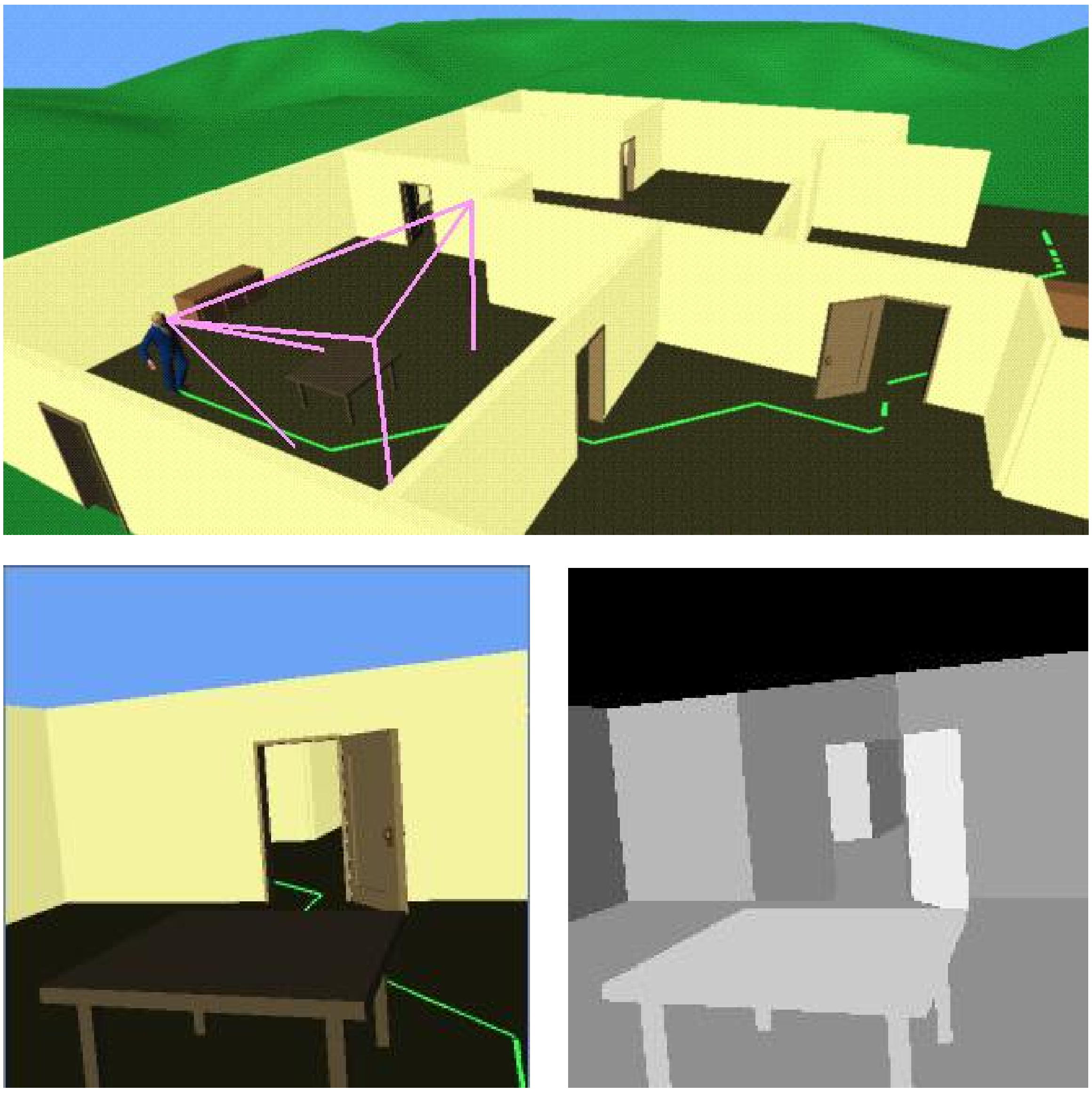

Figure 4.1: The top image shows a character navigating in a virtual office. An outline of a representative portion of the character's viewing frustum is shown. The bottom two images show the scene rendered from the character's point of view. The image on the right is the true-color, lighted model, while the image on the left is the unlit, color-coded visibility image.

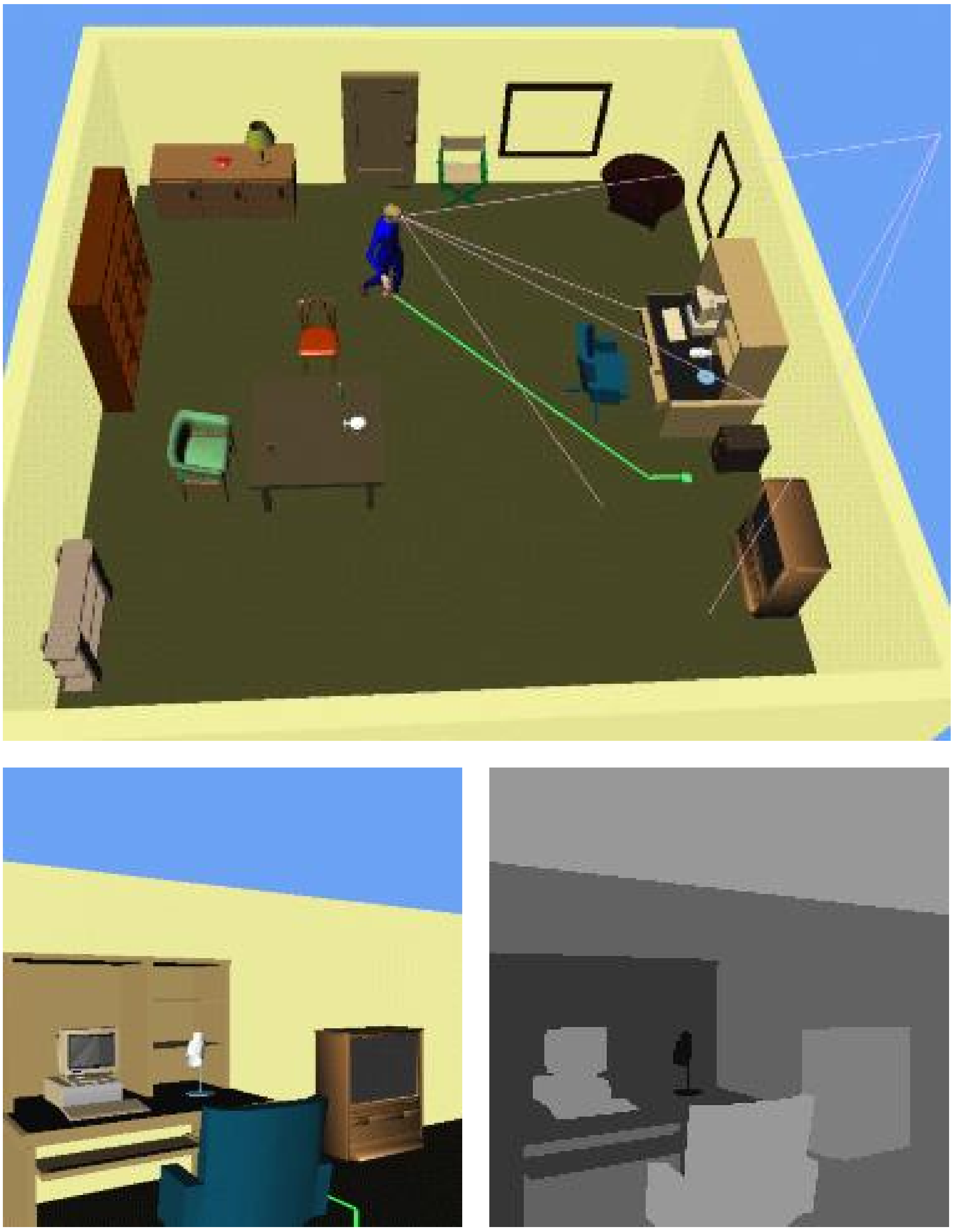

Figure 4.2: Another example view in a virtual office scene.

visual memory of each character[NRTT95]. We instead rely upon the object geometry stored in the environment along with a list (or possibly an array) of object IDs and their most-recently observed states. This provides a compact and fast representation of each character's internal world model that is scalable to large environments with many characters.

When a character observes the environment, the vision module returns the set of object IDs representing objects that are currently visible. Each object ID is combined with other state information, such as the corresponding object's current 3D transformation. The state information of each of the visible objects is obtained directly from the environment, and the character updates its internal model of the world with each object's recently observed state. Previously unobserved objects are added to the character's list of known objects, and the rest of the visible objects are simply updated with their current state. Objects that are not currently visible but have been observed in the past, retain their most recent previously-observed states. This process maintains a kind of spatial memory for each character.

4.4 Perception-Based Navigation

Using such a sensing and memory model, a character can be made to explore in real-time an unknown environment, and incrementally build its own internal model of the world. We now describe the algorithm in more detail. First, we will describe the basic sense-plan-control loop that works for static environments. We then show how to modify the basic algorithm to work for dynamic environments.

Updating $\mathcal{M}$ in Static Environments

Let $\mathcal{O}$ denote the set of all objects in the environment. Each character maintains a set $\mathcal{M}$ of *observations* built incrementally from the output of the vision module.

Observations are represented as tuples $\langle objID_i, \mathcal{P}_i, T_i, v_i, t\rangle$, with components:

$objID_i$	the object ID of object i
$\mathcal{P}_i$	properties of object i
T_i	3D transformation of object i
v_i	velocities of object i
t	observation time

The set $\mathcal{P}_i$ contains properties of the object, including semantic information about the object, or characteristics pertinent to its current state. For example, if the observed object is a door, then $\mathcal{P}_i$ might identify the object as something that can be moved (rotated), is currently closed, is currently unlocked, etc. For storage efficiency, permanent (unchanging) object properties can be stored in a global table indexed by objectID, while variable properties can be encoded with observation bit flags.

Object properties are flexible and as explained further in Section 4.5, can be utilized by higher-level reasoning engines to enable the character to make more informed decisions about the world. The transformation T_i is the observed position and orientation of the object represented by $objID_i$. The component v_i contains the observed linear and angular velocities of the object. The last component t is the *time stamp*, or the time that the observation was made.

$\mathcal{M}$ represents the character's visual memory of $\mathcal{O}$. Initially $\mathcal{M}$ is empty. At regular intervals, the character's visual perception is simulated by the vision module, which renders the color-coded instances of the objects in $\mathcal{O}$. After scanning the pixels of the resulting image, the set $\mathcal{V}$ is returned, containing the object IDs of all currently visible objects. Each $objID_i$ in $\mathcal{V}$ is combined with its corresponding object's state information to form the tuple $\Omega = \langle objID_i, \mathcal{P}_i, T_i, v_i, t\rangle$.

If no existing tuple in $\mathcal{M}$ contains $objID_i$, then the object was previously unknown, and Ω is added to $\mathcal{M}$. Otherwise, if there exists a tuple in $\mathcal{M}$ that contains $objID_i$, then this object was previously observed. Hence, its corresponding state information is updated based on the values contained in Ω. After $\mathcal{M}$ has been updated from $\mathcal{V}$, then the navigation path-planning module is invoked using only the objects and transformations in $\mathcal{M}$ as obstacles. Thus, each character plans a path

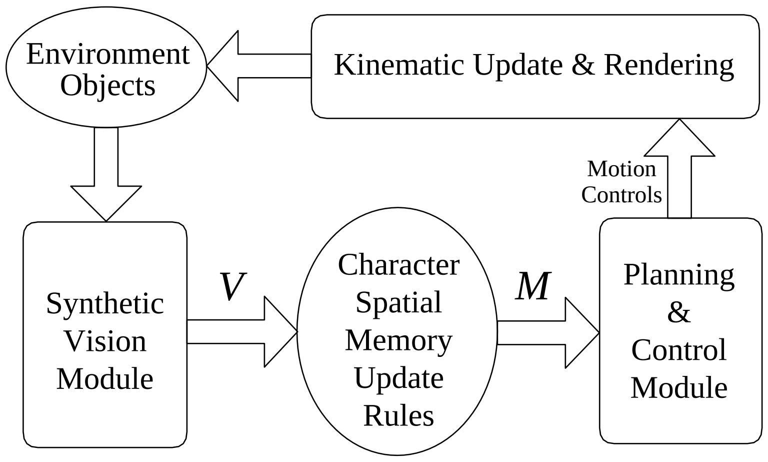
Environment Objects
Kinematic Update & Rendering
Motion Controls
Synthetic Vision Module
V
Character Spatial Memory Update Rules
M
Planning & Control Module

Figure 4.3: The basic sense-plan-control loop for static environments.

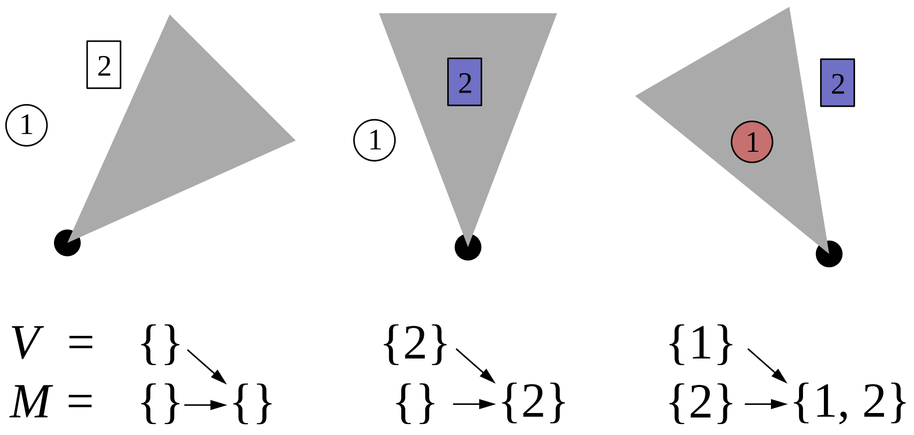

Figure 4.4: Updating $\mathcal{M}$ in static environments

based solely on its own learned model of the world. As the character follows the path, new objects are observed, and $\mathcal{M}$ is updated by repeating the above process, and a new path is computed. The data flow of the algorithm is shown in Figure 4.3. Figure 4.4 illustrates how $\mathcal{M}$ is incrementally updated as previously unknown objects are observed.

Updating $\mathcal{M}$ in Dynamic Environments

The aforementioned procedure works fine for environments with static objects, but we need to make a minor modification to correctly handle changing environments where objects can appear, disappear, or move around unpredictably. The problem lies in recognizing when a previously observed object has disappeared (or moved) from its previously observed location. As an example, let us consider the following two scenarios:

- A character observes an object o_2 at a location T_2. Later, o_2 moves to a new location T_2', and is again observed. In this case, the state information for o_2 in $\mathcal{M}$ should be simply updated to reflect the new location.
- A character observes an object o_1 at a location T_1. Later, o_1 moves to a new location T_1'. However, before the new location is observed, the character observes its former location (o_1 is now missing). In this case, the previous observation of o_1 in $\mathcal{M}$ should be deleted or invalidated.

These two scenarios are depicted in Figure 4.5. The left side illustrates the case where a previously observed object(2) moves to a new location, and the new location is observed before the former location. The state information in $\mathcal{M}$ for the object is updated to account for the new observation(2*). The right side of Figure 4.5 depicts the case where a previously observed object(1) moves to a new location, and its former location is observed prior to its new location. The character should realize that the object is now missing and remove or invalidate the observation in $\mathcal{M}$.

How can the vision module determine whether an object is truly missing, rather than simply obscured by another object? The solution is to re-run the vision module after $\mathcal{M}$ has been updated using only the objects contained in $\mathcal{M}$, along with their corresponding transformations. The result is a set $\mathcal{V}_M$ of object IDs that corresponds to the set of objects the character *expects* to see based on $\mathcal{M}$. By comparing $\mathcal{V}$ and $\mathcal{V}_M$, we can distinguish the above case. Specifically, let $\mathcal{X} = \mathcal{V}_M - \mathcal{V}$ be the set of all object IDs contained in $\mathcal{V}_M$ but not in $\mathcal{V}$. Thus, $\mathcal{X}$ corresponds to objects that the character concludes have disappeared or moved from their previously observed locations, but their new locations are unknown. Consequently, all observation tuples in $\mathcal{M}$ containing object IDs in $\mathcal{X}$ must be removed or invalidated.

Incorporating these modifications into the synthetic vision algorithm facilitates its use in arbitrarily changing environments. The data flow diagram of the modified

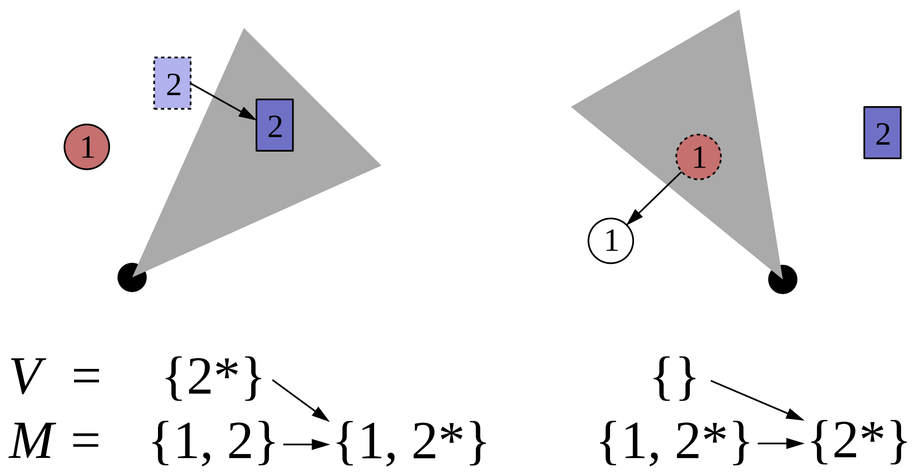

Figure 4.5: Updating $\mathcal{M}$ in dynamic environments

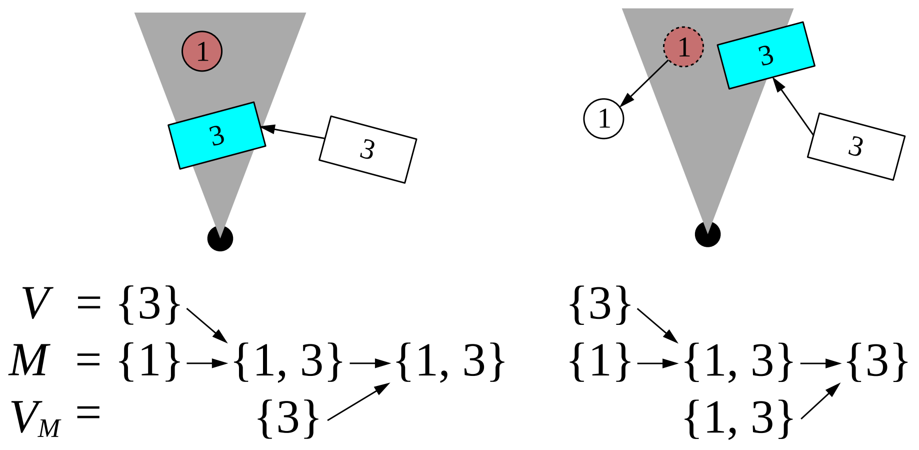

Figure 4.6: “Obscured” vs. “Missing” scenarios requiring $\mathcal{V}_M$ in order to distinguish between them and update $\mathcal{M}$ properly.

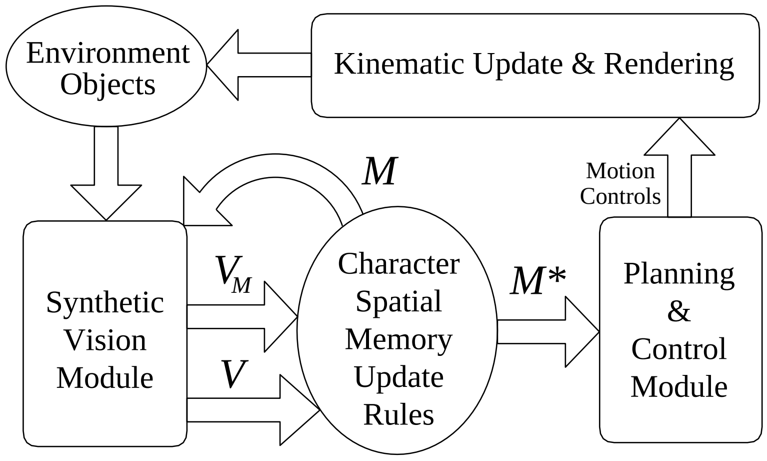
Environment Objects
Kinematic Update & Rendering
M
Motion Controls
V_M
Synthetic Vision Module
V
Character Spatial Memory Update Rules
M*
Planning & Control Module

Figure 4.7: The revised sense-plan-control loop for dynamic environments.

algorithm is shown in Figure 4.7. Two scenarios where the lists of visible objects($\mathcal{V}$) are the same, but the final updates to $\mathcal{M}$ are different, are depicted in Figure 4.6. The scenario on the left corresponds to the *obscured* case, where the previously-observed object(1) is hidden by the appearance of a new object(3). $\mathcal{V}$ and $\mathcal{V}_M$ are equal in this case, so $\mathcal{M}$ retains its previous observation of object(1). The scenario on the right corresponds to the *missing* case, where the previously-observed object(1) has moved to an unknown location while a new object(3) has appeared. Since $\mathcal{V}$ and $\mathcal{V}_M$ differ, the previous observation of object(1) is removed from $\mathcal{M}$.

4.5 Learning and Forgetting

Memory Types

The memory update rules described in Section 4.4 represent a very simple model that remembers all observations until sensing contradicts them. We refer to this as the *basic model*. However, the framework proposed allows several possible memory models to be used. For example, one can imagine a *temporal model* that remembers observations only for a certain period of time. In this case, old observations are periodically deleted from the character's memory. Alternatively, deleting old observations may depend upon the properties of the object. For example, if a non-movable object such as a wall or a floor were observed, we may wish to always retain that observation, while if a moving object such as a vehicle or animal were observed, we may decide to expire such observations as time passes. This means that there can be different types of objects in the world, and different memory rules for each type. We refer to this type of model as a *rule-based model*. The rules governing the update of the character's memory determine how to handle observations of certain objects or object types.

Rule-Based Models

Above low-level navigation planning, there is a layer of reasoning that makes decisions about the goals and beliefs of the character based on its internal model of the world. There are many proposed architectures for implementing rule-based models in the artificial intelligence literature, each with advantages and disadvantages (for an overview and examples, see [RN95, JSW98]). For the purpose of creating autonomous characters, any suitable AI architecture will suffice given that is operates efficiently and provides the ability to encode memory rules. The rules can then be made arbitrarily complex. For example, suppose that a character is exploring an unknown maze of connecting rooms with the goal of finding an exit. After some time, suppose the character concludes that no exit exists based on what it has previously observed (i.e. the low-level navigation planner fails). This event may then trigger some higher-level reasoning that will allow the character to continue to explore (e.g. there is no path, but some doors which have been seen closed or locked recently may have been opened in the meantime. Therefore, observations about closed doors may be deleted for the purposes of navigation planning, despite the fact that they are not yet old enough to discard/forget, etc).

Probabilistic Models

There are no limits imposed upon the complexity of the rule-based models chosen other than practical memory-usage limitations and the time-constraints of the application. For instance, one could potentially adopt a probabilistic approach to handle the uncertainty inherent in making plans based only upon a partially-observed representation of the environment. Almost all the reasoning that real humans do is based on imperfect knowledge. We would like our animated agents to also behave appropriately under these circumstances.

In the artificial intelligence literature, there has been a growing movement towards knowledge representation languages that support an explicit representation of uncertainty[KP97]. The emphasis has been on probabilistic representations, which allow the agent's reasoning process to utilize such techniques as *conditioning* for incorporating new information, and *expected utility maximization* for making decisions [Jen96, CDLS99]. Concepts such as Bayesian belief networks that provide a compact representation of complex probability distributions could be used to allow the agent to make inferences based on observations of the environment. These probability distributions could even be generated automatically using machine learning techniques[Mit97]. This research is beyond the scope of this thesis, but could potentially provide very useful means for designing intelligent animated agents in the future.

4.6 Perception-Based Grasping and Manipulation

The synthetic vision module can be used not only for tasks involving navigation, but for a variety of tasks that require visual coordination. This includes grasping and manipulation tasks (see Chapter 5). For preliminary research on perception-based grasping and manipulation for autonomous agents, see the U.Penn papers[GLM94, DLB96], and the sensor-based grasping model of Huang[HBTT95].

In the examples shown in Figure 4.8 and Figure 4.9, the synthetic vision module is used in order to verify that a particular target object is visible prior to attempting to grasp it. If the object is visible, the grasping and manipulation planner is invoked. If the target object is not visible, the character could try to reposition itself, or initiate a searching behavior in an attempt to find the missing object.

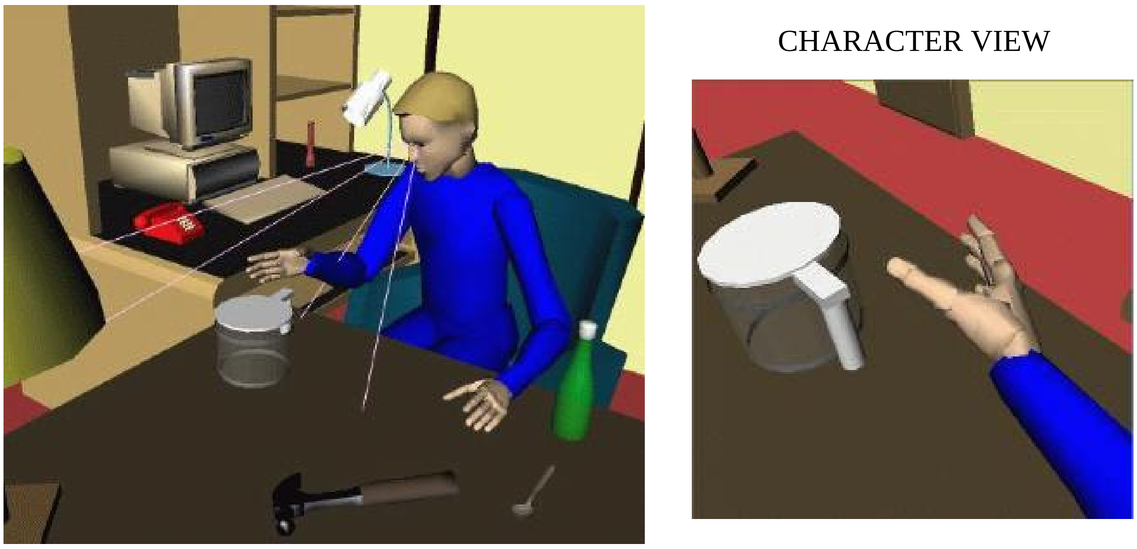

Figure 4.8: Grasping a coffee pot

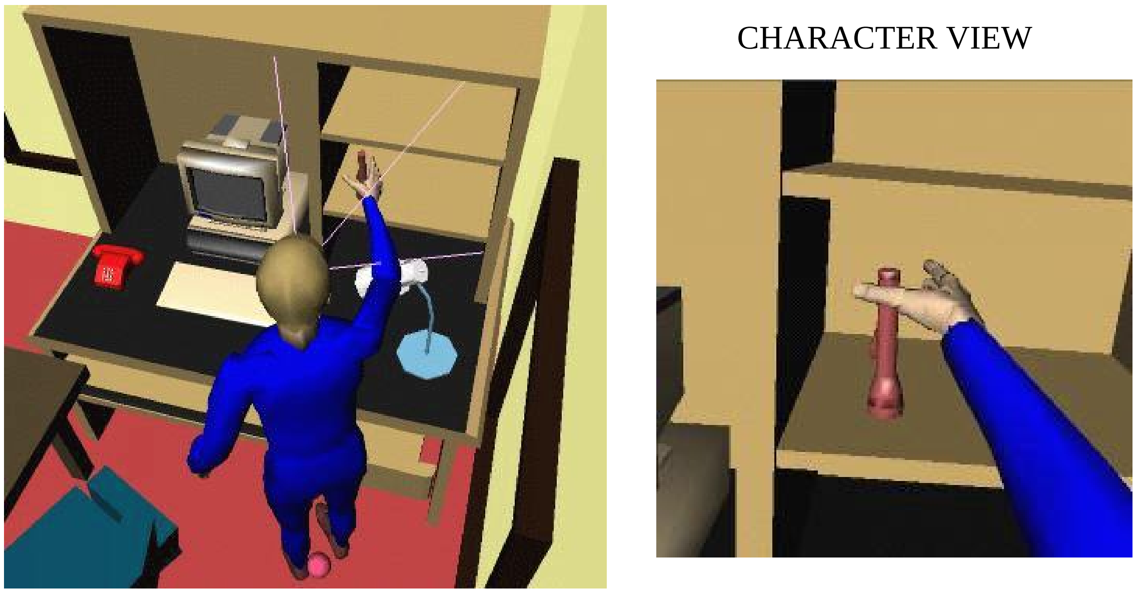

Figure 4.9: Grasping a flashlight

4.7 Other Forms of Perception

Although synthetic vision is the most immediately useful form of simulated perception for navigation and manipulation tasks, other synthetic senses could potentially be incorporated. Although we have not performed any experiments with these other forms of perception, we feel that they are worth mentioning.

Simulated Aural Sensing

Simulated aural sensing or *synthetic audition* is one possibility. An autonomous character should react based not only on what it sees, but also what it hears. Real humans perceive much about their surrounding environment through hearing (e.g. possible approaching danger, relative motions of sound sources via the doppler effect, the size of a space based on its acoustical properties, and listening to the spoken words of others). In virtual reality systems, users tend to rate the immersive quality of their experience much higher if 3D sound effects are included in the simulation.

Animated agents that react to the sounds around them will likely appear more lifelike and believable. Noser and Thalmann proposed ideas about how synthetic audition might be used for autonomous animated characters[NT95]. There has also been active research on effectively modeling the acoustics of virtual environments. A clear and comprehensive presentation of 3-D audio principles and technology can be found in Begault's book[Beg94].

In order to facilitate the interaction of human characters, simulation of explicit verbal communication between characters is needed. Thus, knowing whether or not one character can hear the spoken words of another will require some kind of simulated aural sensing. Perhaps the most compelling reason to simulate the hearing of actors, is to facilitate natural communication with real human users. Indeed, viewing the autonomous animated character as a virtual robot means that ultimately, it may have software for human speech recognition capabilities built-in. In the future, instead of using a mouse, users may be able to give verbal commands to digital actors in the same way that movie directors give verbal directions to real actors.

Simulated Tactile Sensing

Synthetic tactile feedback is another sense that could potentially be exploited by autonomous characters. In particular, simulating the sense of touch might be particularly useful for object manipulation tasks. Some research has been done on sensor-based object grasping and manipulation[HBTT95], although here simulated sensors are used more for collision-detection in generating a correct grasping posture for the hand, rather than really providing simulated tactile feedback to the agent.

Simulating the sense of touch involves calculating *sets of forces* or other nerve signals (e.g. heat, pain) that are generated when a character makes contact with an object in the environment. Recently, there has been great interest in *haptic rendering* or *force display*, which aims to compute the forces generated when a live user interacts with a virtual object[RKK97]. The user obtains real-time force-feedback through the use of a *haptic device*, such as a tendon-driven glove, a robot arm, or even full-body feedback via a motorized exoskeleton. By removing the haptic display hardware altogether, and replacing it with a software interface to an animated agent, one could conceivable use existing technology to provide tactile feedback to the agent.

Given a set of forces, the character would have to interpret the simulated sensations. For example, there might be a pattern-recognition algorithm based on forces instead of image pixels. Applying haptic display technology to virtual humans is an open area of research.

4.8 Results and Discussion

In this section, we describe a few of the experiments we have conducted using the model of synthetic vision outlined in this chapter. We have implemented and tested the algorithms on an SGI InfiniteReality2 running Irix 6.2. Interactive performance has been achieved on complex scenes with up to three characters running full synthetic vision and memory. More than three characters tended to degrade performance unacceptably on a single machine. In such situations, a distributed approach where each character had its own dedicated hardware Z-buffer card would likely facilitate

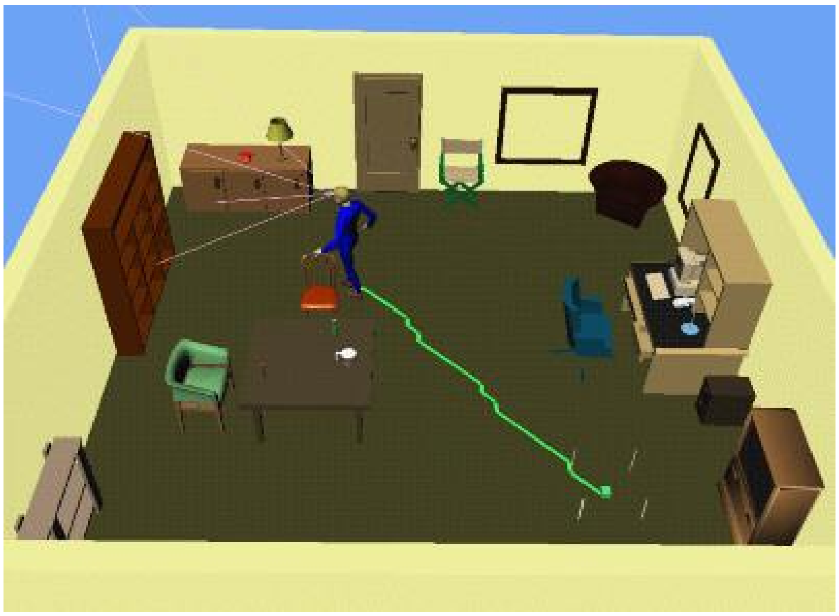

Figure 4.10: After viewing all objects in a room, a character is commanded to walk to one corner.

simulations of many more characters.

Figure 4.10 through Figure 4.13 illustrate a series of snapshots during an interactive simulation involving a human character navigating an a room filled with objects. Figure 4.10 shows a human character after building a complete cognitive map of a room. Figure 4.11 shows a previously-observed table being moved to a new location behind the character's back. Since the character is unaware of the change, its navigation path to the goal location is unaffected. Figure 4.12 illustrates that fact that when the goal location is moved, the character plans a path assuming the table is still at its former location. Finally, Figure 4.13 depicts how when the character turns around to follow the planned path, it notices the table in its new location, and plans accordingly.

Figure 4.14 shows snapshots at various stages of an animation involving a single character exploring an unknown maze environment. The final path is solely the result of the interaction between path planning based in the character's internal model and the visual feedback obtained during exploration. Figure 4.16 shows a trace of the actual motion of the character during exploration. Figure 4.17 shows a trace of the

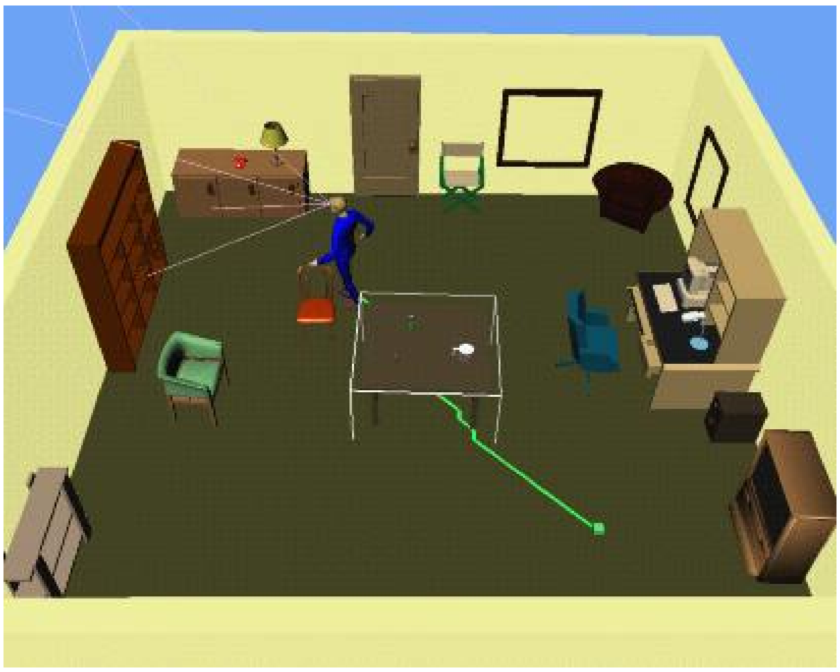

Figure 4.11: A table is repositioned behind the character's back.

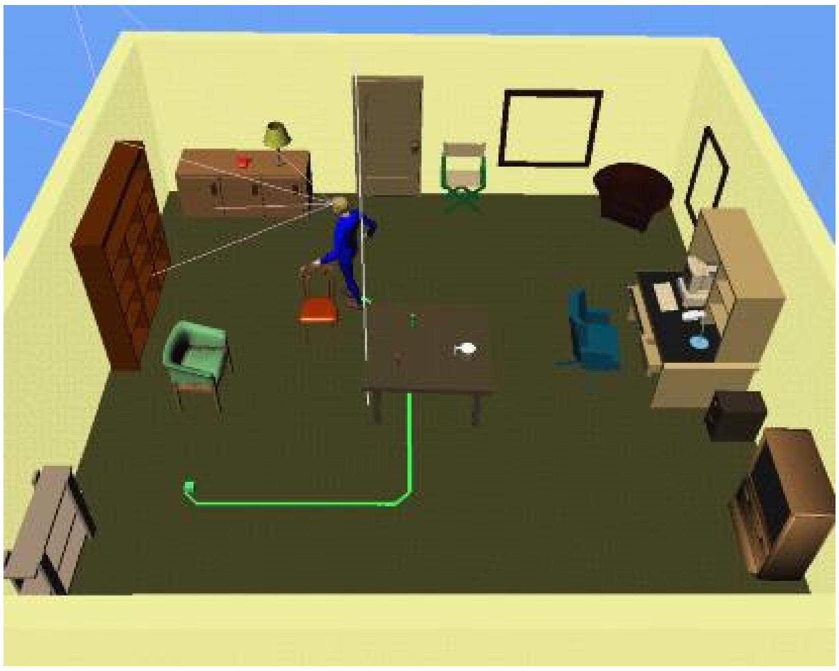

Figure 4.12: As the goal location is moved, the character continues to plan using the table's formerly observed location.

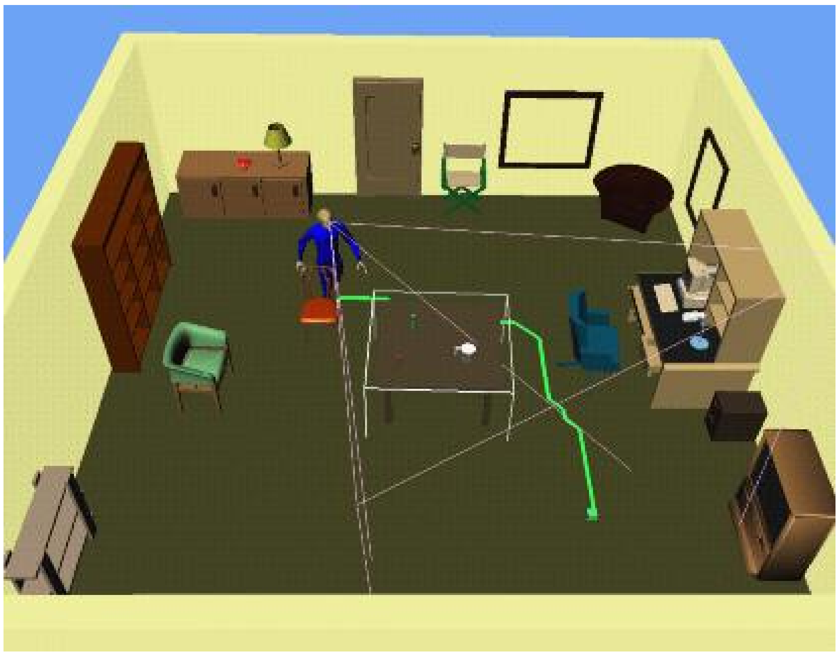
Figure 4.13: After the character turns around and sees the table in its new location, it updates its memory model and replans.

actual motion during exploration of another unknown maze with dead ends that forced the character to backtrack.

Discussion

This chapter presents a simple model of visual perception, memory, and learning for autonomous characters that is suitable for real-time interactive applications. The model is efficient in both storage requirements and update times, and can be flexibly combined with a variety of higher-level reasoning modules or complex memory rules.

Although the synthetic vision module runs fast enough to support a small number of characters simultaneously, it is currently the bottleneck in the computation. This is primarily due to the fact that each character must render the scene *twice* in order to correctly update its cognitive map in the case of dynamic environments. A general memory update scheme for dynamic environments that avoids having to render the scene twice per character would certainly be very useful in improving the efficiency of

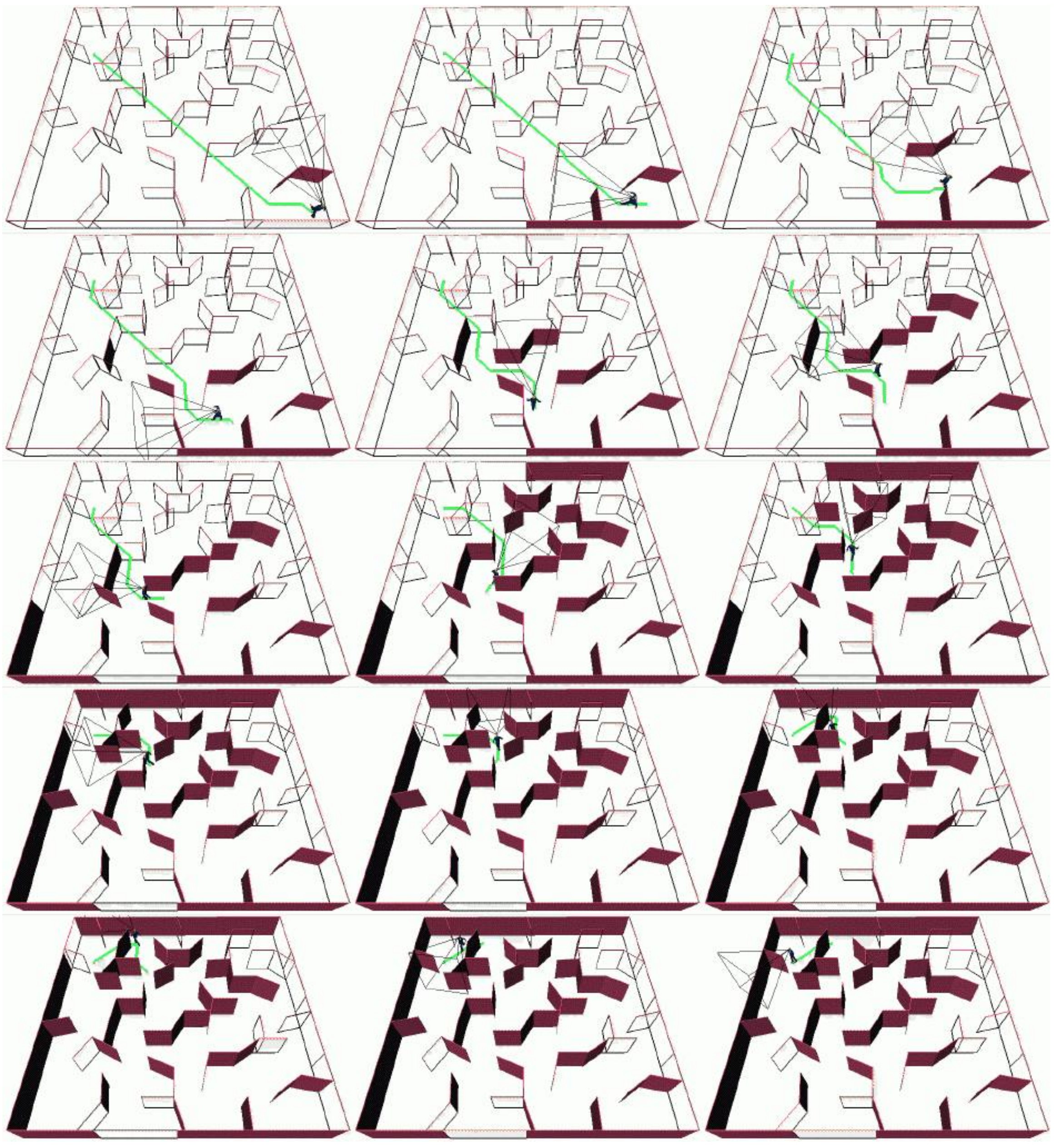
Figure 4.14: Snapshots of a character exploring an unknown maze environment. Objects rendered solid are those contained in $\mathcal{M}$, while those rendered wireframe are unknown. The top-left image shows the initial frame with the character given the task of navigating from the bottom-right corner to the top-left corner of the maze. A portion of the character's viewing frustum, along with the current path is also shown.

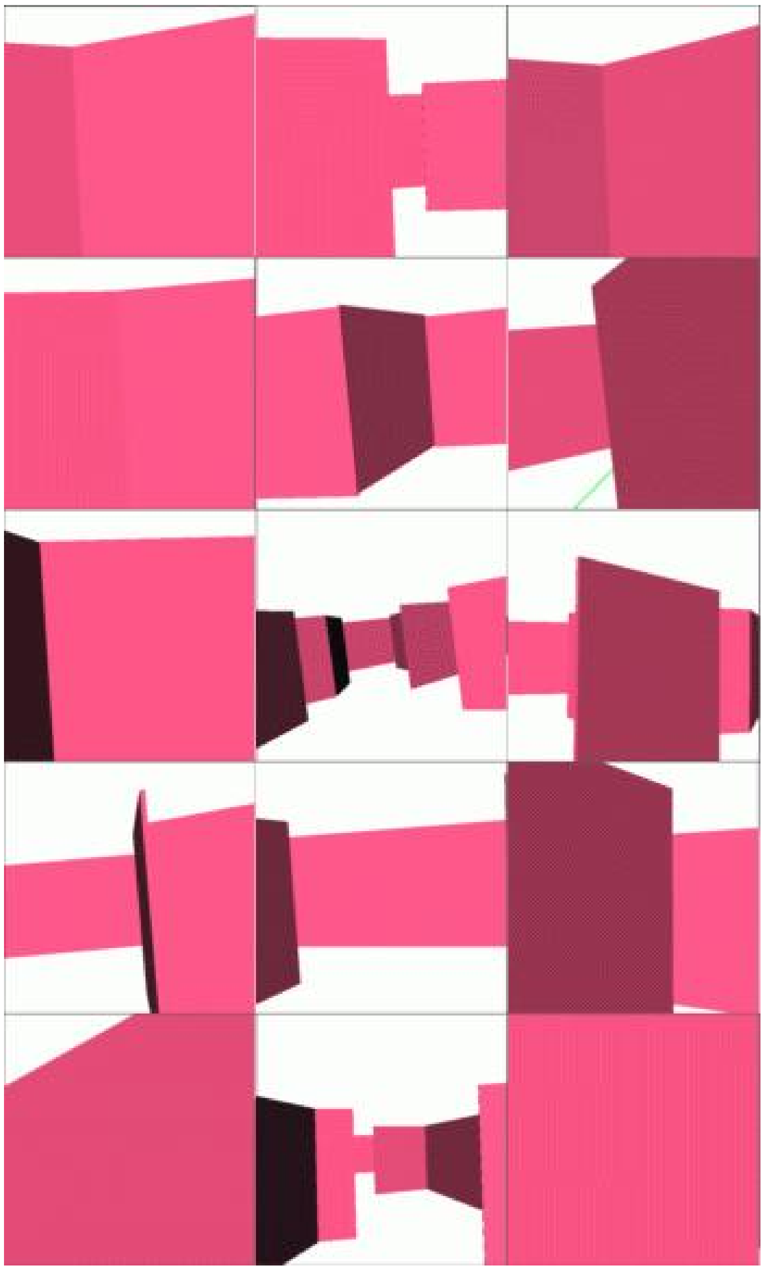

Figure 4.15: Images of the scene rendered from the character's viewpoint during each of the snapshots shown in Figure 4.14.

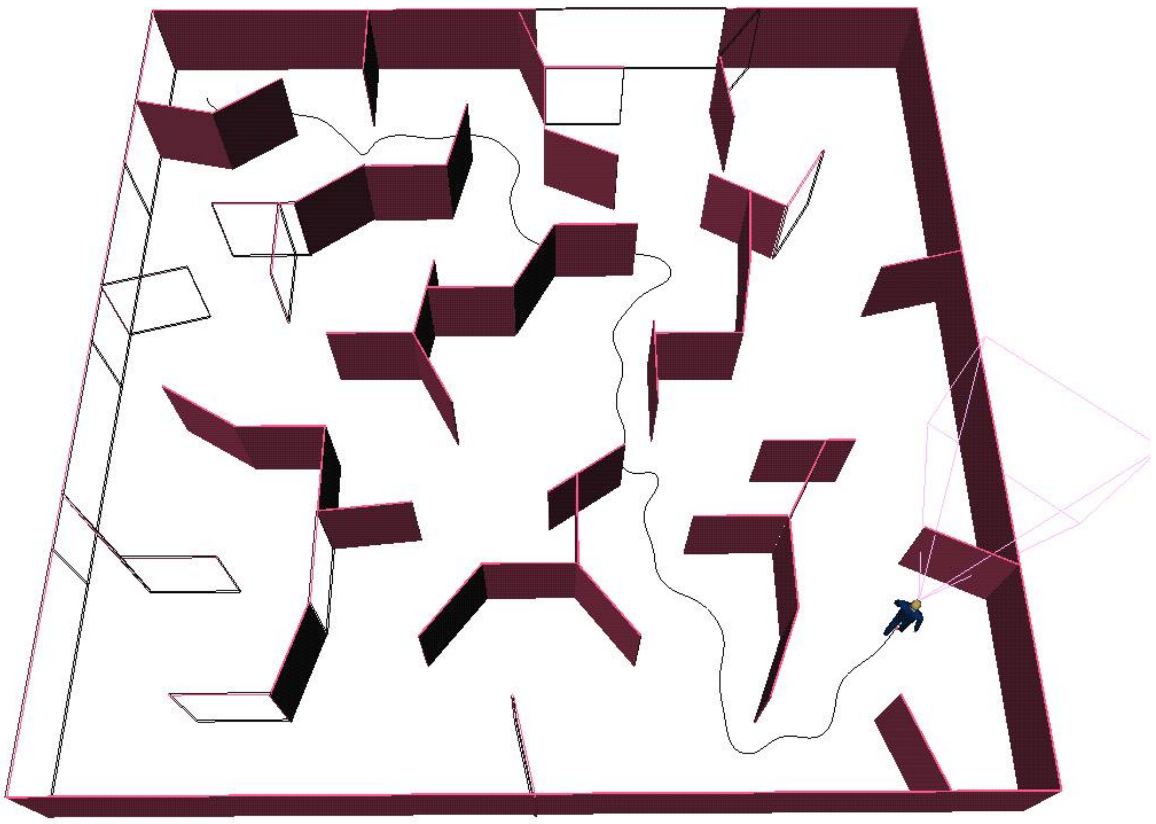

Figure 4.16: Detail of the trajectory followed by the character during the exploration of the unknown maze environment of Figure 4.14.

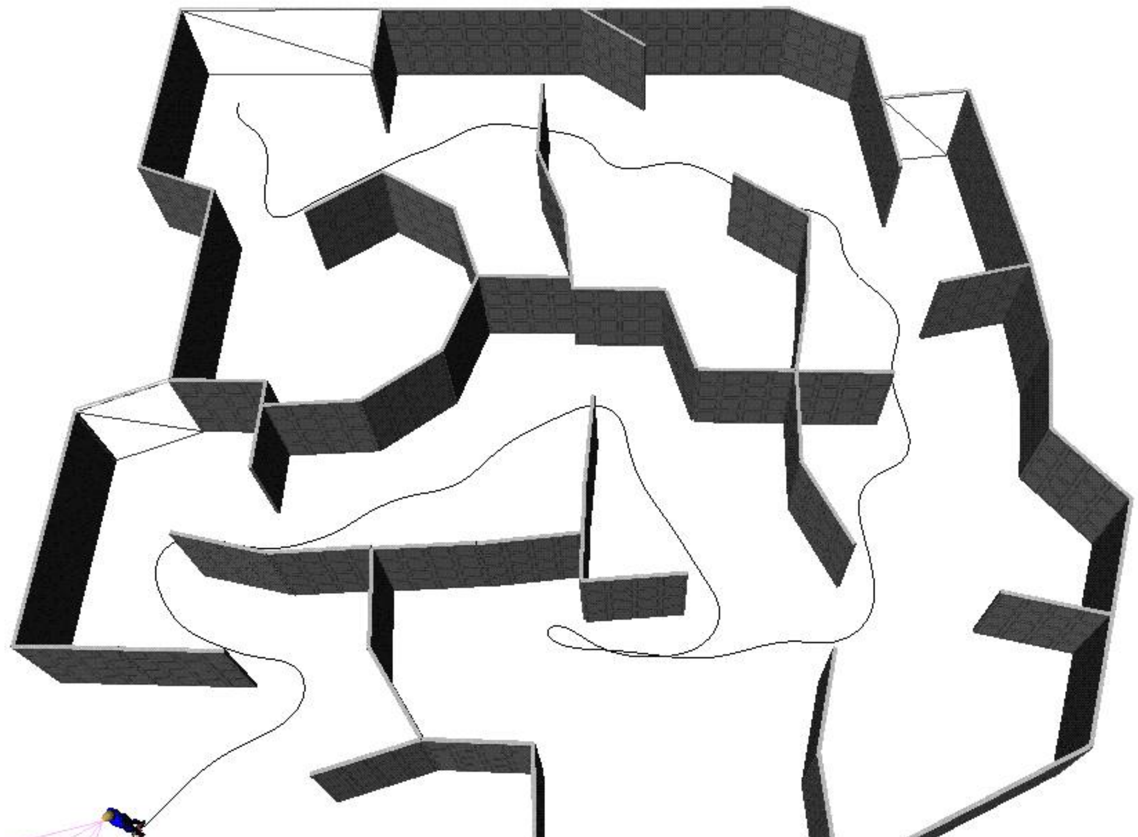

Figure 4.17: The trajectory followed during exploration of another unknown maze environment.

the synthetic vision module, and allowing approximately twice as many characters to be run simultaneously. Other possibilities include invoking the vision module every other frame, or at some regular interval. This could allow one to gain speed (or additional characters) at the expense of accuracy.

One problem with the model of synthetic vision as presented has to do with the fact that only the presence of an object pixel in the character's visual field image is considered, and not its location, or *how many* pixels of the object are visible. If only a tiny fraction of an object is visible, is the object *really* visible to the character?

There are several ways in which this problem could potentially be addressed. One idea is to define a "minimum visible pixel area" that must be met prior to declaring an object as being actually perceived by the character. This could be done by comparing the *estimated* screen area of an object to the total number of pixels of that object that were observed. The object would only be "recognized" if a certain number of pixels were visible at a given distance.

An alternative is to define a mandatory feature that must be seen in order for a character to recognize an object (e.g. if the object is another character, we could require that the character's face be seen before it is recognized). Until that particular feature (or set of features) is seen, the object remains an "anonymous" obstacle.

At a more conceptual level, the model of synthetic vision presented here uses a very simple method of visual data interpretation (using the object ID). Although this method has the advantage of being relatively fast for the purposes of interactive applications, it has a number of important limitations.

For instance, suppose a character observes a lecture hall filled with hundreds of similar chairs. After the character leaves the room, suppose one particular chair (say chair #61) is removed and observed. Using the object ID, the character will automatically know that this is chair #61, which is clearly unrealistic. Special observation rules could be potentially used to govern sets of seemingly identical objects, but doing this in a general way may not be easy.

Another limitation of the synthetic vision model is its inability to handle object transparency or reflections. This stems from the fact that the actual images seen by the character are bypassed, and only uniform projections of object geometry is used.

For example, suppose a character observes an object through a window, or sees the reflection of an object in a mirror. While image-based pattern-matching synthetic vision techniques might correctly classify the object as being observed, the method adopted here fails. Graphic subsystems capable of "screen-door" transparency rendering could enable the method to correctly "see through" transparent objects, but this does not provide a general solution.

As mentioned previously, in the interests of realism, it is important to model the capabilities as well as the limitations of a character. In this chapter, we have assumed that a character is always perfectly "self-localized", meaning that it always knows its own relative location in the environment exactly. However, real humans can become confused or even lost in certain environments, such as a hallway of twisting corridors or a room of mirrors. A different kind of confusion occurs when a human mistakes a tiger skin for a real tiger. Ultimately, these and other issues will have to be addressed in order to create more realistic models of simulated perception for animated agents.

Chapter 5

Manipulation Tasks

5.1 Introduction

This chapter describes a motion generation strategy that combines inverse kinematics and path planning to animate *reaching and object manipulation tasks*. Object manipulation is an important class of motions for animated characters. Like navigation tasks, manipulation tasks can encompass a virtually unlimited combination of object and obstacle geometries. Thus, it seems unlikely that one would be able to successfully enumerate all possibilities and simply store thousands of clip motions. Instead, a flexible strategy that can accommodate a wide range of situations is needed.

Related Work

Computing motions that solve manipulation tasks is related to research in robotics, computer animation, and neurophysiology. We give a brief overview of some of this related work in the following paragraphs:

Robotics: Generating motions for robot manipulator arms is a major area of research in the robotics literature. Path planning techniques have been applied to this problem since Lozano-Perez popularized the configuration-space approach that is now widely used in planning for manipulator arms[LP83]. Faverjon and Tournassoud developed a manipulation planner to plan motions for a robot arm moving among vertical pipes[FT87]. Manipulation planning for repositioning movable objects in 2D workspaces was investigated by Wilfong, and later by Alami, *et al.*[ASL90]. Wilfong showed that manipulation planning for a single movable object is PSPACE hard[Wil88]. Khatib developed a potential-field approach for collision avoidance in [Kha86], which was later used by Quinlan for modifying collision-free paths for manipulator arms in real-time[Qui94]. Lynch implemented a system for manipulating objects by pushing[Lyn93]. Koga developed a multi-arm manipulation planner for repositioning objects in a 3D workspace[Kog94]. A number of randomized path planning techniques have been developed that are suitable for searching high-DOF configuration spaces such as human arms (see Section 5.4).

Computer Animation: For animating the movements of human limbs, a variety of kinematic and inverse kinematic techniques have been proposed [KB82, TT90, BPW92], though relatively few consider complex object manipulation tasks. Lee, *et al.* simulated lifting motions using simplified human arm muscle models [LWZB90]. The use of path planning to automatically generate graphic animation was suggested in [LRDG90]. Subsequently, motion planning was used for animating transitions between body postures in [CB92], [JBN94], and [BMT97]. Koga, *et al.*[KKKL94a] used motion planning to automatically generate multi-arm manipulation motions for human figures. This planner was used to generate a rather complex animation clip involving a human character playing chess with a robot arm[KKKL94b]. Bandi used workspace discretization and inverse kinematics for quickly planning collision-free reaching motions[Ban98]. Other related research includes task-level grasping[DLB96], and computing human hand and finger postures for object grasping using collision-detection[HBTT95].

Neurophysiology: There are many studies in psychology and neurophysiology aimed at determining how the central nervous system of a human manages to coordinate the motion of its limbs. The problem was formulated and investigated in the 1930s by Bernstein[Ber67]. It is now widely agreed in the medical community that human arm movement is represented kinematically[S^+91], and the dynamics or muscle activation patterns are generated as a post-processing step. A good survey of this and other results can be found in the book by Latash[Lat93], or among the collections of articles in [MT86] or [Wal89]. In particular, a numerical model derived from actual human pointing tasks is described in [SF89], and forms the basis of the human arm inverse kinematics algorithm described in [Kon94]. This algorithm was used for the experiments in [KKKL94a], as well as this chapter (see Section 5.3).

Animating Manipulation Tasks

As discussed in Chapter 2, much of the difficulty in synthesizing manipulation motions for character animation stems from the near infinite number of possible goal locations and obstacle arrangements in the environment. This combinatorial explosion of possibilities currently prohibits the direct use of pre-recorded motion sequences.

Motion planning techniques are particularly well-suited for solving problems that involve computing collision-free motions amidst obstacles. In this chapter, we develop a *manipulation planner* for a single human arm. The human arm is modeled as a kinematic chain with 7 degrees of freedom, and motion trajectories are computed directly within the configuration space. Because of high-dimensionality, the space cannot be explicitly represented (as was the case for navigation in Chapter 3). Instead, the space is sampled using a randomized planner that has been specifically tailored to quickly solving common planning queries involving human arms.

In the sections that follow, we give an overview of the problem of manipulation planning for human arms, and discuss its unique challenges. We then discuss randomized path planning and introduce a new manipulation planner designed for efficiently generating collision-free motions for single-arm manipulation tasks. A technique for coordinating the head and eye movements in conjunction with the arm motions is presented in order to increase naturalness. Experimental results are presented along with a summary discussion.

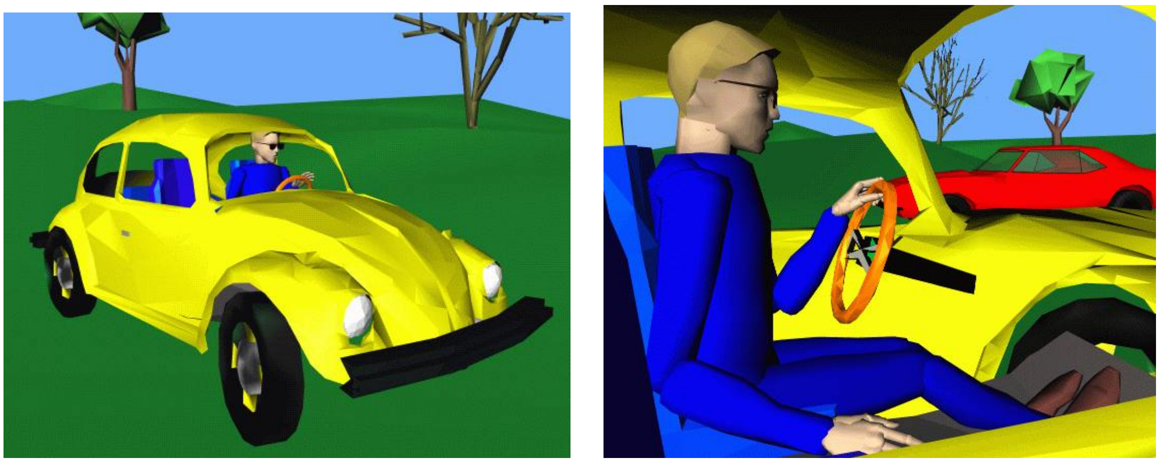

Figure 5.1: Manipulating the controls of a virtual car.

5.2 Manipulation Tasks for Humans

Overview

Object manipulation tasks are ubiquitous in everyday human life. Tasks such as combing one's hair, washing dishes, driving a car, eating, drinking, and playing board games all involve precisely coordinated arm movements. How the human central nervous system (CNS) solves such tasks seemingly effortlessly has been the subject of study in both neurophysiology and robotics. Neurophysiologists have sought to understand more about the human brain and its functions. Robotics researchers look for clues as to how to program robotic arms to solve manipulation tasks, especially in the context of designing humanoid robots for general-purpose use. For computer animation, our goal is to create software for automatically generating manipulation motions for human characters.

Neurophysiological Results

Neurophysiological studies have shown that the coordination of arm motions in real humans is primarily derived from the precise control of a *working point*[Lat93]. In the robotics literature, this point is sometimes referred to as the *tool frame* or *operational space frame*, and consists of both a position and rotation relative to a reference frame[Cra89, Kha95]. Here, we will adopt the robotics convention of specifying both a position and rotation which we call the *task frame*.

The task frame identifies the most important geometric surfaces for executing a certain task. For example, for grasping an object, the task frame may lie on a fingertip or palm, whereas for hitting a nail with a hammer, the task frame will likely be located at the tip of the hammer. Presumably, this is the location about which the human central nervous system is mostly concerned, since its trajectory is vital for completing the task successfully. Neurophysiological studies have shown that the trajectory of the origin of the task frame is better reproduced in repeated trials than trajectories of individual joints[BFF86, Soe84].

Human Figure Animation

For generating reaching, grasping, or manipulation motions for animated characters, we can similarly define a task frame. For instance, the task frame for picking up a coffee pot might be the character's palm. A *grasp transformation* defines the geometric relationship between the task frame and the object being manipulated. A *grasp* specifies a grasp transformation along with a configuration of the character's hand and finger joints.

There may be many possible grasps associated with an object, some possibly involving multiple arms. Calculating a set of possible grasps automatically from an object's geometry is a complicated problem, and has been addressed in the robotics literature (see [PT89] for a survey of this work). For human arms, studies in neurophysiology suggest that humans select grasps based on prior experience or intention[AW89].

In this chapter, we shall regard the selection of a grasp as dependent primarily upon an object's geometric and physical properties. Thus, we assume that a valid set of grasps for a particular character morphology is pre-specified by an animator and associated with an individual object (or class of objects with similar geometry). Other researchers have also proposed storing information about how a virtual object should be manipulated (such as grasp locations) along with the object itself, in order to facilitate object-character interactions [GLM94, KT99]. We also consider only single-arm grasps, since we focus on manipulation tasks that can be accomplished by a single arm. However, objects that require multiple arms for grasping could potentially be handled by incorporating techniques from Koga, *et al.*[KKKL94a].

Given the current configuration of a character, we can compute a *goal configuration* for the character to grasp a target object using the following:

1. The relative position and orientation of the object.
2. A valid grasp (i.e. a grasp transformation, plus a configuration for the hand).
3. An *inverse kinematics* algorithm.

The goal configuration is then passed as input to a motion synthesis strategy used to compute a motion for the character to reach and grasp the object. Once the object has been grasped, a similar procedure can be used to move the object to a target location (see Section 5.5).

5.3 Inverse Kinematics

Overview

Researchers in the field of human motor control and analysis have long puzzled at how the central nervous system selects a particular combination of joint angles to achieve a given task frame position in the workspace. For a good overview of some of the theories and experimental results, see [Lat93].

In robotics, this problem is known as a problem of *inverse kinematics* (IK). Mathematically, we desire a function which maps a given global transformation G_j for a link frame $\mathcal{F}_j$ to a set $\mathcal{Q}$ of values for $\mathbf{q}$ (see Section 2.5). Each configuration $\mathbf{q} \in \mathcal{Q}$, represents a valid inverse kinematic solution. The forward kinematics function $\textsc{Forward}(\mathbf{q})$ positions the link $\mathcal{L}_j$, such that the frame $\mathcal{F}_j$ has a global transformation of G_j relative to the reference frame $\mathcal{F}_{world}$. For a more thorough treatment of inverse kinematics issues and methods, please consult a robotics textbook (for example [Cra89]).

Since there are no general algorithms for directly solving a set of nonlinear equations, most inverse kinematic problems do not have general solution methods. There are two basic approaches to solving inverse kinematics problems: *numerical* methods, and *analytical* (exact) methods. Numerical methods typically employ an iterative procedure to search for solutions to a set of kinematic equations. Analytical methods use closed-form algebraic expressions derived from the kinematic equations.

For any inverse kinematics problem, there are two fundamental issues that must be dealt with:

Existence of solutions: Due to limitations on the link geometry and valid joint ranges, not all positions and orientations in the workspace can be achieved by the task frame. For these cases, no valid solutions exists ($\mathcal{Q}$ is the empty set).

Multiplicity of solutions: If a solution *does* exist, it may not be unique. Due to kinematic redundancies, there may be any number of valid joint configurations that will place the task frame at a particular position and orientation. In some cases, there are infinite number of such configurations ($\mathcal{Q}$ is an infinite set).

Analytic solution methods are typically much faster than numerical methods and often return all possible solutions if a solution exists. Unfortunately, analytic solution methods only exist for a subset of possible kinematic arrangements of links.

Human Arm Inverse Kinematics

A simplified kinematic model of a human arm usually consists of three links (upper arm, forearm, hand) and three joints (shoulder, elbow, wrist). The shoulder joint and the wrist joints are typically modeled as spherical joints having three rotational DOFs each, with the elbow joint having one rotational DOF. This yields a combined total of seven DOFs (see Figure 5.2).

Since an arbitrary position and orientation in 3D specifies only six constraints (three for position and three for orientation), the inverse kinematics problem for the simplified model of a human arm involves an underconstrained system of nonlinear

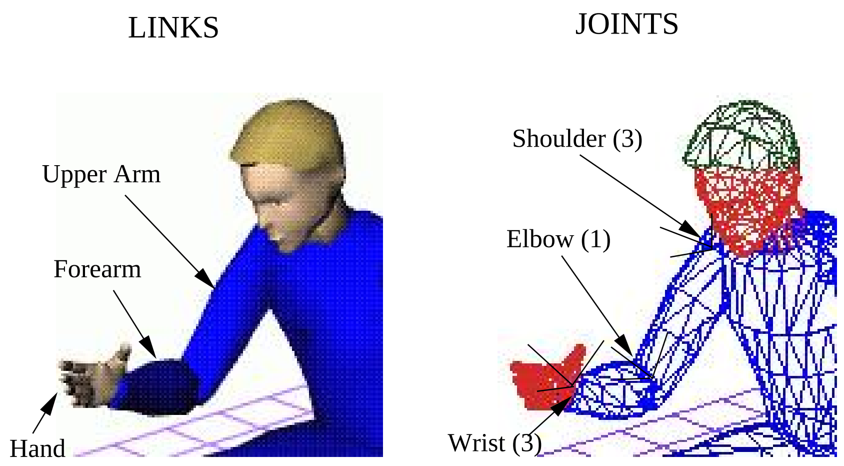

Figure 5.2: Simplified kinematic model of a human arm (7 DOF).

equations. This means that in general, there are many possible arm configurations for a given hand position and orientation. For the purposes of graphic animation, we would like our inverse kinematics algorithm to select a "natural" arm posture when faced with a range of possible inverse kinematic solutions.

For the experiments described in this chapter, we have adapted an inverse kinematics algorithm for human arms that was originally developed by Kondo[Kon94]. The algorithm is based on the sensorimotor transformation model of Soechting and Flanders[SF89]. This model was derived from a set of experiments conducted using human subjects who where instructed to move a pen-sized stylus to various targets in their vicinity. Analysis of their recorded arm postures revealed that the joint angles for the arm can be approximated by a linear mapping from the spherical coordinates of the stylus frame relative to the shoulder. The resulting equations resolve the redundancy in the human arm and identify a single "natural" arm posture based on the arm postures observed in the experiments. For a detailed description of the basic Kondo inverse kinematic algorithm for human arms, the reader is referred to [Kon94] or [KKKL94a]. One of the nice features of this algorithm is that it uses analytic equations to find a natural approximate posture. Thus, the algorithm is both consistently fast, and *repeatable* (i.e it doesn't depend upon an initial guess, as is the case for most iterative algorithms).

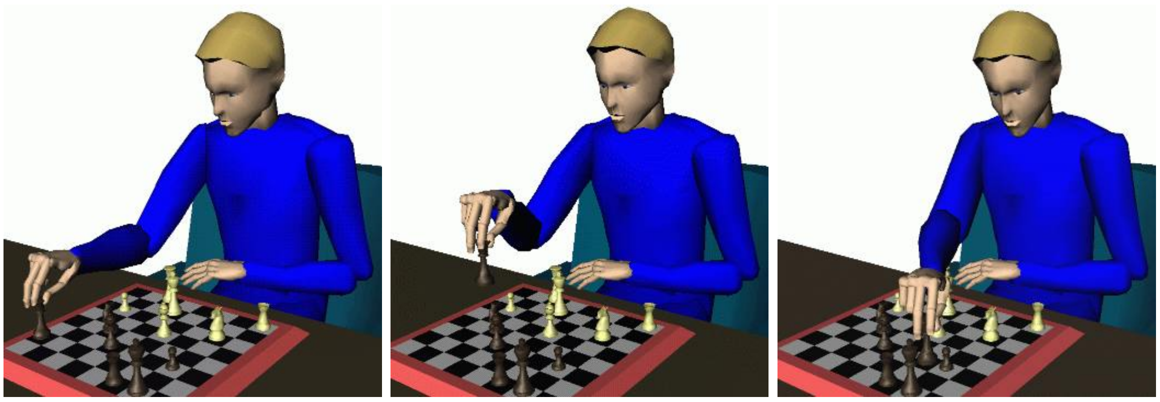

Figure 5.3: Path planning for a 7-DOF human arm

In our experiments, we assume that the torso and shoulder positions remain fixed for the duration of the manipulation motion. This assumption is reasonable for manipulation tasks involving small, nearby objects. Other tasks may require the use of additional degrees of freedom in the shoulder, torso, or legs. See Section 5.6 for a brief discussion of possible ways to incorporate these additional degrees of freedom into the planning process.

5.4 Path Planning

Introduction

Path planning problems arise in such diverse fields as robotics, assembly analysis, virtual prototyping, pharmaceutical drug design, manufacturing, and computer animation. Such problems involve searching the system configuration space for a collision-free path connecting a given start and goal configuration[Lat91]. For problems in low dimensions, the configuration space can often be explicitly represented as exemplified in Chapter 3. For high-dimensional configuration spaces however, it is typically impractical to explicitly represent the configuration space. Instead, the space is *sampled* in order to discover free configurations. Here, the fundamental challenge lies in devising a practical and efficient sampling strategy.

This section describes a randomized path planner with a sampling heuristic specifically designed for computing collision-free manipulation motions for human arms. The sampling heuristic is based on Rapidly-Exploring Random Trees (RRTs)[LaV99, LK99]. The heuristic is optimized to quickly handle single-query path planning problems without any preprocessing of the configuration space. Although designed with human figure animation applications in mind, the path planner has also been demonstrated to be both efficient and practical for a variety of planning problems. Referred to in this thesis as the *RRT-Connect* heuristic, the method exhibits rapid convergence for simple spaces and relative immunity to pathological cases. The rest of Section 5.4 gives a broad overview of previous work in randomized path planning, and then describes the RRT-Connect path planner. Section 5.5 explains how to use the RRT-Connect planner for animating single-arm manipulation tasks. Section 5.6 show several computed examples along with a summary discussion.

Background

Randomized algorithms for path planning have enjoyed success and popularity in the last several years due to their efficiency in handling problems with many degrees of freedom[BKL+97]. The randomized path planner of Barraquand and Latombe[BL90] was an early attempt to practically solve problems with high-dimensional configuration spaces. Their search technique alternated between following the gradient of an artificial potential field[Kha86], and utilizing random walks in order to escape the basin of attraction of any local minima encountered. Variations of this planner were used to solve complex single and multi-arm manipulation tasks[Kog94, KKKL94a]. Unfortunately, pathological cases involving local minima can exist such that the probability of escaping them via a random walk is extremely small [CG93, KŠLO96].

In order to avoid the problems of local minima inherent with artificial potential fields, new sampling strategies were devised. *Probabilistic roadmap* methods build a network of randomly-sampled free configurations in the configuration space [Ove92, KL93, Sve93, Kav95]. After the network (roadmap) has been built, a path may be searched for using standard graph-search algorithms.

Theoretical results show that under reasonable assumptions on the geometry of the configuration space, a relatively small roadmap can correctly capture the connectivity of the free space with high probability[HLM97]. More precisely, the probability that a roadmap incompletely represents the connectivity of the free space decreases exponentially with the number of sample points in the roadmap. The main issue affecting coverage of the free space is the presence of *narrow passages* in the configuration space[HKL$^+$98].

There are numerous variations on the basic roadmap strategy, most of which rely on different sampling techniques in an effort to reduce the computational costs[HST94, AW96, HKL$^+$98, BOvdS99]. The computed roadmaps are especially suitable when multiple path-planning queries are given for a robot in the same static environment, since searching a roadmap is very fast. However, the overhead associated with building the roadmap is often too large for single-query planning problems in interactive environments.

Hsu, *et al.* developed a variant of the probabilistic roadmap planner that is better-suited for solving single-query path planning problems[HLM97]. It avoids the initial cost of preprocessing by incrementally building two trees respectively rooted at the start and the goal. As the trees grow, the planner periodically attempts to join them together to form a path. The idea of constructing search trees from the initial and goal configurations comes from classical AI bidirectional search, and an overview of its use in motion planning methods appears in [HA92]. In order to avoid oversampling any region of the configuration space, the planner incrementally samples around nodes in the trees according to a weighted probability that favors nodes that have few neighbors.

Recently, LaValle introduced the concept of Rapidly-exploring Random Trees (RRTs) [LaV99], a randomized sampling scheme originally designed for nonholonomic motion planning. RRTs have also been applied to kinodynamic planning problems in configuration spaces of up to 12 dimensions[LK99]. For both holonomic and nonholonomic planning, the sampling technique exhibits several desirable properties. Similar to the planner in [HLM97], the goal is to incrementally build a tree of free configurations in a way such that the expansion of the tree is heavily biased towards the unexplored regions of the space. Due to the way that RRTs are constructed, the distribution of samples eventually converges toward a uniform distribution over $\mathcal{C}_{free}$ [LaV99].

The algorithm presented here reuses ideas from the planners presented in [HLM97] and [LaV99] in combination with a greedy heuristic aimed at achieving rapid convergence. Our goal in designing this planner was to build a method whose performance scales roughly with our intuitive perception of the difficulty of the problem. Namely, if a simple solution exists to a path planning problem, the planner should find it very quickly (like a potential-field planner). If the problem is difficult (e.g. involves traversing a narrow passage in the configuration space), the planner should still be able to solve it though more computation time might then be required. Preliminary experiments have shown good performance in a variety of planning situations (see Section 5.6).

The RRT-Connect Planner

The path planning queries considered are of the standard form. Namely, we are given an initial configuration q_{init} and a goal configuration q_{goal} in the configuration space $\mathcal{C}$ of a robot or animated character. Our task is to find a collision-free path connecting q_{init} and q_{goal} (i.e. any path lying entirely in $\mathcal{C}_{free}$, the open subset of collision-free configurations in $\mathcal{C}$, is considered a solution path). In this thesis, the RRT-Connect heuristic is described for the holonomic case. However, with minor modification, the same method could be applied to models with nonholonomic constraints by utilizing the more general RRT construction algorithm given in [LaV99, LK99]. In either case, the algorithm requires that a distance metric ρ be defined on $\mathcal{C}$ (i.e. the function $\rho(q_1, q_2)$ returns some measure of the distance between the pair of configurations q_1 and q_2). Some axes in $\mathcal{C}$ may be weighted relative to each other, but the general idea is to measure the "closeness" of pairs of configurations with a scalar function.

Growing a Tree

The fundamental operation of the algorithm is to grow a branch on a tree of connected free configurations. The *nodes* in a tree lie at either a branching point, or at the end of a terminal branch. Two nodes *connected* by a branch are connected by a collision-free path. For simplicity, a straight-line path in $\mathcal{C}$ is used to make connections between nodes, but any local connection method that is efficient to store and easy to compute may be used.

At every iteration, the planner picks a configuration in $\mathcal{C}$ called the target configuration q_{target}. Given q_{target} and a tree $\mathcal{T}$ of connected free configurations, a branch is grown on $\mathcal{T}$ towards q_{target}. This operation is outlined in pseudocode in Procedure 1. First, the node q_{near} in $\mathcal{T}$ that is "closest" to q_{target} according the the distance metric ρ is found. This is a standard *nearest-neighbor query* in computational geometry. A brute-force approach might simply compute ρ relative to q_{target} for all nodes in $\mathcal{T}$ and return the minimum. This is a linear-time operation (at every iteration of the planner) relative to the number of nodes in $\mathcal{T}$. However, more efficient algorithms exist, such as multi-dimensional k-d trees (for complete references, see [AMN+]).

Procedure 1 GrowTree($\mathcal{T}$, q_{target})

```
$q_{near} \leftarrow$ NearestNeighbor($\mathcal{T}$, $q_{target}$)
**if** $\rho(q_{near}, q_{target}) < d_{max}$ **then**
  **if** Connect($q_{near}$, $q_{target}$) **then**
    AddBranch($\mathcal{T}$, $q_{near}$, $q$)
    **return** CONNECTION
  **end if**
**else**
  $q \leftarrow$ GenerateNewNode($q_{target}$, $q_{near}$)
  **if** Connect($q_{near}$, $q$) **then**
    AddBranch($\mathcal{T}$, $q_{near}$, $q$)
    **return** SUCCESS
  **end if**
**end if**
**return** FAILED
```

The node q_{near} becomes the root of a potential new branch in $\mathcal{T}$. The endpoint of the new branch is either q_{target}, or an intermediate node along the straight-line

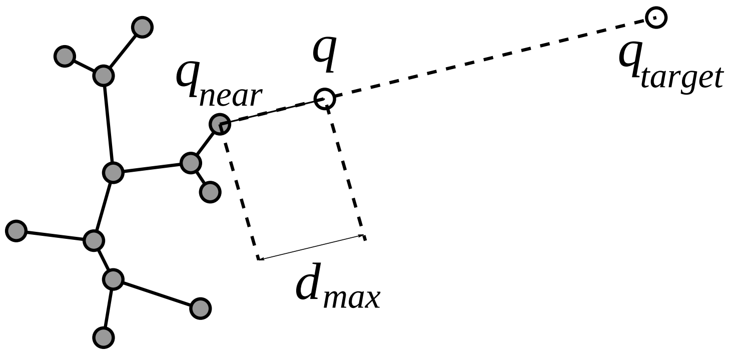

Figure 5.4: Growing a branch towards q_{target}.

path between q_{near} and q_{target}, depending upon whether the distance between q_{near} and q_{target} is less than some maximum distance d_{max}. The constant d_{max}, is used to prevent the algorithm from trying to locally connect very distant configurations, since such an operation is expensive and will likely fail.[1]

If $\rho(q_{near}, q_{target})$ is less than d_{max}, then the procedure *Connect*(q_{near}, q_{target}) is invoked in an attempt to build a straight-line path connecting q_{target} directly to $\mathcal{T}$. If a path is found, a branch is added to $\mathcal{T}$ connecting q_{near} and q_{target}. The result CONNECTION is returned, indicating that a direct connection was successfully made between q_{target} and $\mathcal{T}$. Otherwise, the result FAILED is returned.

If $\rho(q_{near}, q_{target})$ is greater than d_{max}, then the function *GenerateNewNode*(q_{near}, q_{target}) makes a motion toward q_{target} at the fixed incremental distance d_{max} (Figure 5.4). This produces an intermediate node q lying along the straight-line path in $\mathcal{C}$ between q_{near} and q_{target}. The node q becomes the endpoint of the potential new branch in $\mathcal{T}$. The procedure *Connect*(q_{near}, q) verifies that the path between q_{near} and q is collision-free. If so, a branch is added to $\mathcal{T}$ connecting q_{near} and q, and the result code SUCCESS is returned. Otherwise, the result code FAILED is returned.

[1]The constant d_{max} is the only parameter the planner requires. Its value should not be too small nor too large relative to the size of an axis of $\mathcal{C}$. In our experiments, we found that assigning d_{max} a value between 3 to 5 percent of the width of an axis of $\mathcal{C}$ yielded satisfactory results.

The Basic Planner

In order to generate a Rapidly-Exploring Random Tree (RRT), we simply initialize $\mathcal{T}$ with a single node and repeatedly call *GrowTree* using target configurations that are uniform samples of $\mathcal{C}$. This simple procedure will create a tree that probabilistically converges towards a uniform exploration of $\mathcal{C}$[LaV99]. For path planning, we grow two trees starting from both q_{init} and q_{goal} until a connection between the trees is established. We will refer to this as the *basic planner*, which is outlined in Procedure 2.

Procedure 2 BasicPlanner(q_{init}, q_{goal})

```
$\mathcal{T}_{init} \leftarrow$ Initialize($q_{init}$)
$\mathcal{T}_{goal} \leftarrow$ Initialize($q_{goal}$)
**while** $time < T_{max}$ **do**
  $q \leftarrow$ RandomConfig()
  $r_{init} \leftarrow$ GrowTree($\mathcal{T}_{init}, q$)
  $r_{goal} \leftarrow$ GrowTree($\mathcal{T}_{init}, q$)
  **if** $r_{init}$ = CONNECTION **then**
    **if** $r_{goal}$ = CONNECTION **then**
      $path \leftarrow$ BuildPath($\mathcal{T}_{init}, \mathcal{T}_{goal}, q$)
      **return** $path$
    **end if**
  **end if**
**end while**
**return** NULL_PATH
```

We begin by initializing the tree $\mathcal{T}_{init}$ with the single node q_{init}, and the tree $\mathcal{T}_{goal}$ with the single node q_{goal}. We then invoke the main planning loop which consists of repeatedly generating a random target configuration q by uniform sampling of $\mathcal{C}$ and growing both trees towards the sample. The loop will terminate in one of two ways: either the time limit will expire (in which case, the planner fails and no path is returned), or a connection between the trees is established (and the path connecting q_{init} and q_{goal} is returned). A connection between the trees occurs when the calls to *GrowTree*($\mathcal{T}_{init}, q$) and *GrowTree*($\mathcal{T}_{goal}, q$) both return CONNECTION, indicating that the sample q was directly connected to both trees. This means that $\mathcal{T}_{init}$ and $\mathcal{T}_{goal}$ are now connected via the node q. The planner succeeds and the path between q_{init} and q_{goal} through q is returned.

The growth of $\mathcal{T}_{init}$ and $\mathcal{T}_{goal}$ can be thought of as two expanding wavefronts of explored $\mathcal{C}$-space. A path is formed between them whenever a sample is generated that is within a distance d_{max} from both trees and both local connection attempts succeed.

A desirable property of this planner, is that the distribution of nodes in the trees eventually converges toward the sampling distribution (which is uniform here) [LaV99]. This means that the planner will eventually arrive at a uniform coverage of $\mathcal{C}_{free}$, which is also a desirable property of the probabilistic roadmap planner. This property implies that if a path exists between q_{init} and q_{goal}, the planner will eventually find it. Unfortunately, we do not yet have a theoretical analysis of the convergence rate (which is observed to be fast in practice).

The RRT-Connect Heuristic

This section modifies the basic planner described in the previous section by introducing a simple greedy heuristic which we call the *RRT-Connect* heuristic. The goal of the heuristic is to bias the expansion of nodes in the trees towards rapidly establishing a connection between them.

There are two fundamental planner characteristics that we will focus on in designing our heuristic:

1. For queries that admit simple solutions, the planner should solve them very quickly.
2. Overall probabilistic global convergence must be maintained (e.g. the heuristic should not inadvertently cause the planner to become "trapped" by pathological cases.)

The key idea behind the RRT-Connect planner is to alter the basic planner as follows: whenever a node is added to either of the trees, the planner attempts to continually *grow the other tree towards that node*. As we shall soon see, this simple scheme serves to draw the trees closer to each other and consequently more quickly find a path on average, while preserving the required planning characteristics outlined previously.

Pseudocode for the algorithm is given in Procedure 3. Just as with the basic planner, we begin by initializing the tree $\mathcal{T}_{init}$ with the single node q_{init}, and the tree $\mathcal{T}_{goal}$ with the single node q_{goal}. We also initialize three variables, which will encode the current state of the planner:

- q_{target} keeps track of the current target configuration used for expanding the currently active tree.

Procedure 3 RRTConnectPlanner(q_{init}, q_{goal})

```
/* initialization */
$\mathcal{T}_{init} \leftarrow$ Initialize($q_{init}$)
$\mathcal{T}_{goal} \leftarrow$ Initialize($q_{goal}$)
$q_{target} \leftarrow q_{goal}$
$\mathcal{T}_{active} \leftarrow \mathcal{T}_{init}$
$bias \leftarrow$ TRUE
/* main planning loop */
**while** $time < T_{max}$ **do**
  **if** $bias$ **then**
    $result \leftarrow$ GrowTree($\mathcal{T}_{active}, q_{target}$)
    **if** $result =$ CONNECTION **then**
      $path \leftarrow$ BuildPath($\mathcal{T}_{init}, \mathcal{T}_{goal}, q_{target}$)
      **return** $path$
    **end if**
    **if** $result =$ FAILED **then**
      $bias \leftarrow$ FALSE
    **end if**
  **else**
    $q_{target} \leftarrow$ RandomConfig()
    $result \leftarrow$ GrowTree($\mathcal{T}_{active}, q_{target}$)
    **if** $result \neq$ FAILED **then**
      $q_{target} \leftarrow$ NodeAdded($\mathcal{T}_{active}$)
      $bias \leftarrow$ TRUE
    **end if**
    /* switch currently active tree */
    **if** $\mathcal{T}_{active} = \mathcal{T}_{init}$ **then**
      $\mathcal{T}_{active} \leftarrow \mathcal{T}_{goal}$
    **else**
      $\mathcal{T}_{active} \leftarrow \mathcal{T}_{init}$
    **end if**
  **end if**
**end while**
/* planner failed */
**return** NULL_PATH
```

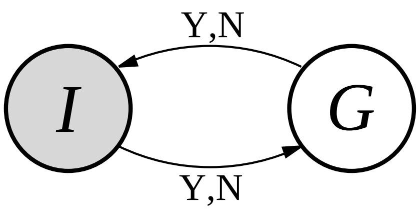

Figure 5.5: Basic Planner state diagram

- $\mathcal{T}_{active}$ points to the currently active tree (either $\mathcal{T}_{init}$ or $\mathcal{T}_{goal}$).
- *bias* is a boolean flag indicating whether the planner is currently growing towards a recently added node in the opposite tree.

q_{target} is initialized to the goal configuration, $\mathcal{T}_{active}$ is initialized to point to $\mathcal{T}_{init}$, and the flag *bias* is set to TRUE. Intuitively, these settings mean that the planner is initialized to try and grow $\mathcal{T}_{init}$ towards q_{goal}.

We then invoke the main planning loop which repeatedly performs one of two things depending upon the value of the *bias* flag:

If *bias* is TRUE: We attempt to grow the currently active tree $\mathcal{T}_{active}$ towards the current target configuration q_{target}. If a direct connection to q_{target} is made (CONNECTION is returned by *GrowTree*), then we have successfully connected $\mathcal{T}_{init}$ and $\mathcal{T}_{goal}$, so we assemble and return the path through q_{target}. If *GrowTree* fails to grow a branch (FAILED is returned), then *bias* is set to FALSE. Otherwise, if a branch was added (but not a direct connection to q_{target}), then by default, we continue to grow the active tree towards q_{target}.

If *bias* is FALSE: We generate a random target configuration by uniformly sampling $\mathcal{C}$. If *GrowTree* succeeds in adding a branch (either CONNECTION or SUCCESS is returned), then q_{target} is set to the node just added and the *bias* flag is set to TRUE. Regardless of whether *GrowTree* succeeded or failed, we toggle the currently active tree $\mathcal{T}_{active}$.

Another way to understand how the RRT-Connect planner works is to examine a state diagram depicting its operation. First, consider the simple state diagram for the basic planner shown in Figure 5.5. There are two states labeled I and G, corresponding to whether $\mathcal{T}_{init}$ or $\mathcal{T}_{goal}$ is currently being expanded respectively. Regardless of whether the attempt

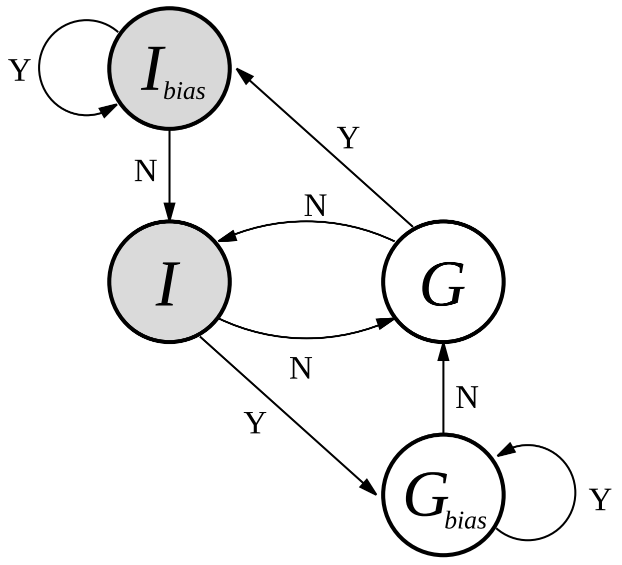

Figure 5.6: RRT-Connect Planner state diagram

to grow a branch on the currently active tree is successful or not (Y or N), a transition is made to the other state. Thus, the planner always alternates between attempting to grow $\mathcal{T}_{init}$ and $\mathcal{T}_{goal}$ towards a randomly sampled configuration.

Now consider the state diagram for the RRT-Connect planner shown in Figure 5.6. The same states I and G are present, but different transitions have been added along with two new states I_{bias} and G_{bias}. These new states correspond to the *bias* flag being TRUE while expanding $\mathcal{T}_{init}$ or $\mathcal{T}_{goal}$ respectively. The planner begins in state I_{bias} (expanding $\mathcal{T}_{init}$ with the *bias* flag TRUE and the target configuration equal to q_{goal}). If *GrowTree* successfully adds a branch (the Y transition), then the planner remains in state I_{bias} and continues to grow toward the current target configuration q_{target}. Otherwise, it transitions to state I (the N transition), which picks a random target configuration and attempts to grow a branch. If it succeeds, it sets q_{target} to be the recently added node, changes the active tree to $\mathcal{T}_{goal}$, and transitions to state G_{bias}. Otherwise, it changes the active tree to $\mathcal{T}_{goal}$, and transitions to state G, which attempts to grow towards a random target configuration, etc. The other transitions in the diagram can be explained similarly.

Intuitively, for planning queries that admit a simple solution, the RRT-Connect planner will spend most of its time in states I_{bias} and G_{bias}. For queries which encounter a lot obstacles, the planner will spend more of its time in states I and G, thus behaving more like the basic planner randomly exploring $\mathcal{C}_{free}$. It should also be pointed out that there is a computational advantage to being in either of the states I_{bias} and G_{bias}. Namely, whenever a self-transition is made from either of these states, we can eliminate the need for the nearest-neighbor calculation. The reason is that we already know that the node at the end of the branch just added must be the closest node to q_{target}. This usually provides a modest savings depending upon the cost of computing ρ.

Path Smoothing

Both the basic and the RRT-Connect planner return the first path they find connecting q_{init} and q_{goal}. These initial paths returned are usually not optimal with respect to ρ. For example, if the initial path is used directly to animate the joints of the character, although the motion will be continuous, it may appear slightly jerky. Any number of techniques may be used to optimize or "smooth out" the initial solution path returned by the planner (for example, see [Lat91]). In the experiments described here, a very simple smoothing technique was used that iteratively selects random pairs of nearby configurations along the path and attempts to connect them with a straight line in $\mathcal{C}$. The process stops after a set number of iterations fails to shorten the path. This method gave satisfactory results for the purposes of experimentation, but other more sophisticated path optimization schemes could be used in place of this method.

5.5 Task Animation

Arm Motions

This section describes how we utilize the RRT-Connect path planner from the previous section to animate single-arm manipulation tasks for human figures. Through a graphical user interface, an operator can interactively select an object, specify a target location, and issue a *move* command. The planner will then attempt to compute three trajectories:

- *Reach*: Move the arm in position to grasp the object.
- *Transfer*: After grasping, move the object to the target location.
- *Return*: Once the object has been placed at the target location, release it and return the arm to its rest position.

If any of these steps should fail, the software sends an appropriate signal to execute a contingency plan (e.g repositioning the torso and shoulder, or attempting to use the other arm to complete the task).

Let us consider the steps necessary for generating a *reaching* motion (the *transfer* and *return* motions involve almost the same set of steps). Each of the steps is listed along with a brief description.

1. **Task specification:** A reaching task for a character $\mathcal{A}$ is specified by identifying a target object $\mathcal{O}$ whose local reference frame $\mathcal{F}_{obj}$ has a global transformation G_{obj} relative to $\mathcal{F}_{world}$.

2. **Grasp Selection:** A valid grasp $g = (T_{grasp}, q_{hand})$ is chosen from the set of grasps for $\mathcal{O}$ that apply to the morphology of $\mathcal{A}$. If more than one valid grasp for $\mathcal{O}$ exists, all are evaluated and ranked according to accessibility and convenience (proximity to the arms of $\mathcal{A}$), and g is chosen as the highest ranking grasp. If the manipulation plan fails using g, a plan using the next highest-ranking valid grasp is attempted.

3. **Grasp Frame Computation:** The local grasp transformation T_{grasp} associated with g is used to compute the global transformation $G_{grasp} = G_{obj}T_{grasp}$ of the *grasp frame*, $\mathcal{F}_{grasp}$. $\mathcal{F}_{grasp}$ marks the precise location on the geometry of $\mathcal{O}$ where the task frame of the arm should be placed (see Figure 5.7).

4. **Arm Selection:** The arm of $\mathcal{A}$ whose task frame $\mathcal{F}_{task}$ is nearest to $\mathcal{F}_{grasp}$ is designated for planning. If the plan fails, a plan using either the other arm or the next valid grasp attempted.

5. **Goal Configuration Computation:** The inverse kinematics algorithm is invoked to attempt to compute a collision-free arm configuration $\mathbf{q}_{goal}$ that aligns the task frame $\mathcal{F}_{task}$ with $\mathcal{F}_{grasp}$.

6. **Path Planning:** The current arm configuration is used as the initial configuration $\mathbf{q}_{init}$. The path planner is invoked to attempt to find a collision-free path connecting $\mathbf{q}_{init}$ and $\mathbf{q}_{goal}$. The hand configuration q_{hand} associated with the grasp g is used for collision-checking during planning.

7. **Trajectory Generation:** If the planner is successful, a motion trajectory τ for the arm is generated by time parameterizing the path using an appropriate velocity

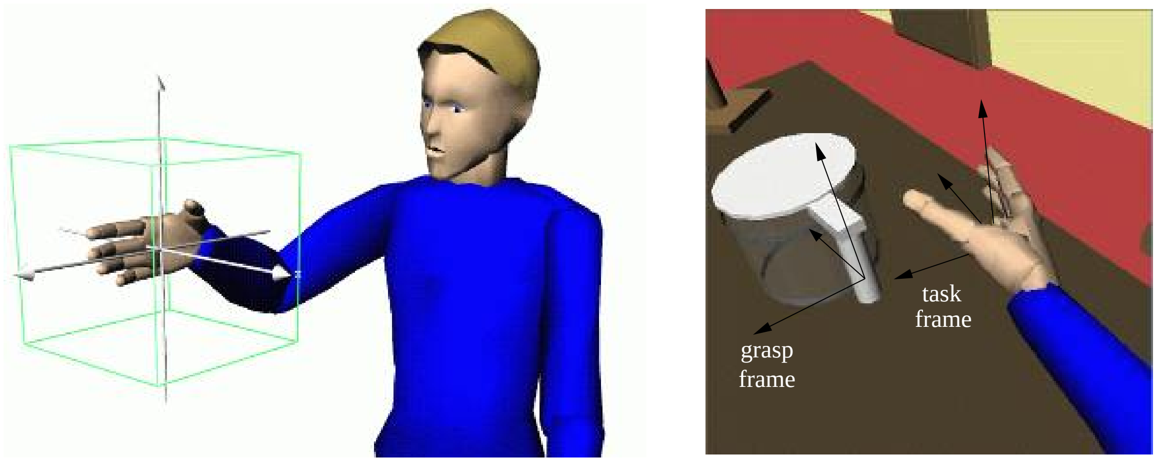

Figure 5.7: An example *task frame* (left). The *grasp frame* for a coffee pot (right).

profile. In our implementation, we use a spline function for a velocity profile curve that attempts to mimic the *ease-in* and *ease-out* concepts commonly used in traditional animation [TJ95].

8. **Final Grasp Adjustment:** After the arm has reached $\mathbf{q}_{goal}$, the finger joints are iteratively modified to cause the hand to actually grasp the object (the joints are incremented until the finger geometry makes contact with $\mathcal{O}$). For example, Figure 5.8 contains snapshots of a human character grasping a flashlight on a shelf. The left image shows the reaching motion and grasping approach at the end of the computed path. The right image shows the grasping configuration after iteratively closing the hand and finger joints.

Computing the *transfer* and *return* motions involve nearly the same set of steps as the *reaching* motion, except for the following differences:

Transfer Motion: Move the object $\mathcal{O}$ to a target location.

1. The hand configuration remains fixed (grasping $\mathcal{O}$) for the duration of the motion.

2. The geometry of $\mathcal{O}$ is added to the collision model of the hand for collision-checking during path planning.

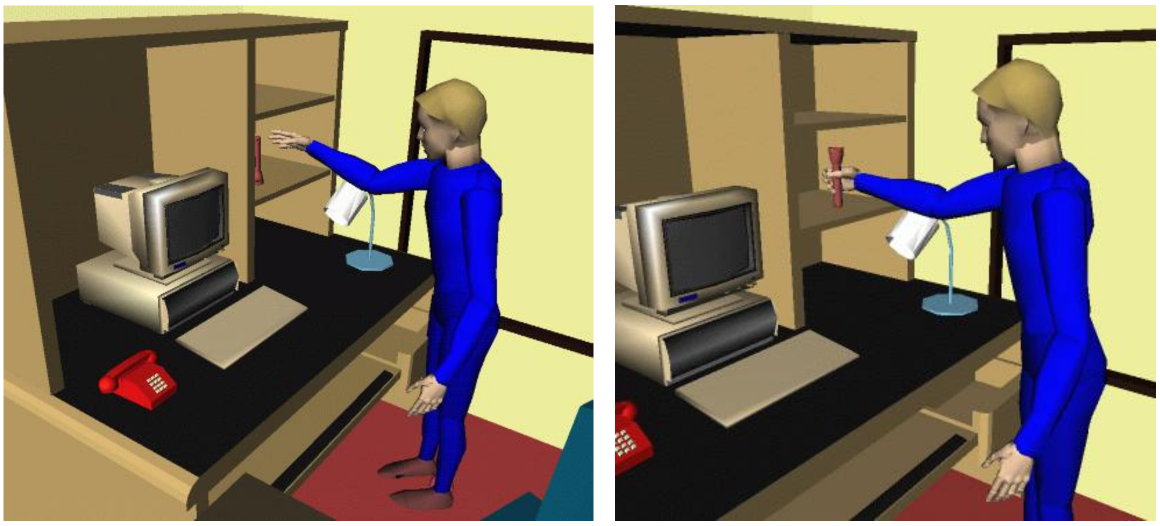

Figure 5.8: Reaching for a flashlight (left), and finalizing the grasp (right).

3. The global transformation $G_{grasp} = G_{target}T_{grasp}$ of $\mathcal{F}_{grasp}$ is computed using the global target transformation G_{target}. G_{target} represents the desired new location for the object $\mathcal{O}$.

Return Motion: Release $\mathcal{O}$ and return the arm to a resting position.

1. Prior to moving the arm, the joints of the fingers are iteratively modified to open the hand, thereby releasing $\mathcal{O}$.

2. The goal configuration $\mathbf{q}_{goal}$ is set to be a default rest configuration for the arm $\mathbf{q}_{rest}$, rather than being computed via inverse kinematics.

Eye-Head Movements

Although the joints of the arm are of primary importance to the animation of grasping and manipulation tasks, the other joints of the character cannot be simply ignored. Computed motions look stiff and unnatural if the other joints of the body remain fixed. This is particularly true if the neck, head, and eyes remain fixed. Based on the experiments we have conducted, the naturalness of the animation can be greatly improved by using a simple gaze function to coordinate the movements of the neck, head, and eyes.

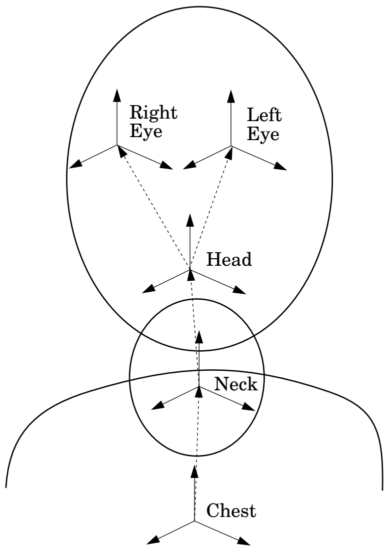

Figure 5.9: The eye-neck kinematic chain

Neurophysiological Studies

Research in neurophysiology has revealed unique patterns of motor control techniques that humans employ in coordinating eye and head movements. Given a visual target, the eyes and head move simultaneously to form a stable directed gaze. Specifically, the eye movements have the function of rotating the optic axis with respect to the head, such that the visual target is either acquired or maintained in the central area of the retina[MT86].

The neck, head, and eyes form a kinematic chain as illustrated in Figure 5.9. The motion of the eyes with respect to the environment is the cumulative transformation formed by the sum of the eye rotation (with respect to the head), the head rotation (with respect to the neck), and the neck rotation (with respect to the torso).

The movement of the eye-neck chain serves to facilitate the visual feedback necessary for accuracy in executing a given task (hand-eye coordination). This implies a geometric mapping between the "global" task point trajectory and the "internal" joint variables of the eye-neck chain. In general, such a mapping is non-linear, but observations of real humans have revealed an approximately linear mapping[MT86]. In addition, humans are endowed with a special reflex called the *vestibulo-ocular reflex* (VOR), which is an automatic mechanism that controls the eye rotations to compensate for voluntary or involuntary head movements[MBD73]. This reflex reduces the mechanical coupling of the eye-neck chain,

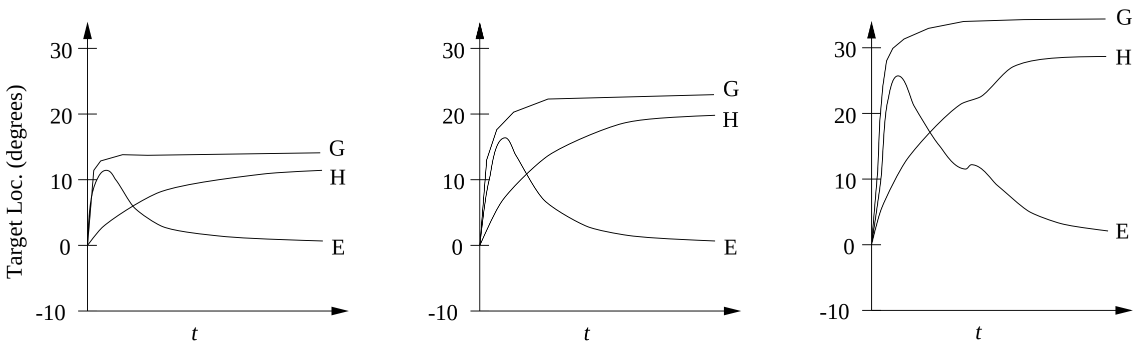

Figure 5.10: Eye and head coordinated movements for different relative target displacements. The curves represent eye rotations (E), head rotations (H), and cumulative gaze rotations (G = E + H). (Adapted from [Rob64])

thereby reducing the brain's task of stabilizing the retinal images, and allowing the brain to more effectively target the eyes towards a location in space.

Eye-Head Trajectories

When humans gaze at a visual target, the eyes quickly rotate to focus on the object while the head (with the larger mass, and hence greater inertia) rotates towards the target to align the visual baseline perpendicular to the line of sight. Perpendicular alignment tends to maximize the accuracy of depth estimation by human stereo vision through the relative increase in the effective length of the visual baseline. The VOR rotates the eyes to compensate for the head movement. The initial small, rapid, jerky movements of the eye are known as *saccadic motion*. The duration of human eye saccades, unlike motions in most other motor subsystems, are not independent of the overall movement amplitude. Instead, they increase monotonically with the amplitude[Rob64]. Figure 5.10 illustrates the relative timing of simultaneous eye and head movements.

An Approximate Model

Based on the studies found in the neurophysiological literature, we have devised a simple *gaze function* model to automatically compute appropriate eye-neck motions for grasping and manipulation tasks. The gaze function is essentially an inverse kinematics algorithm that attempts to mimic natural human gazing behavior for the purposes of animation.

We will denote the gaze function by $gaze(\mathbf{p})$, where $\mathbf{p}$ is the *point of interest*, or the

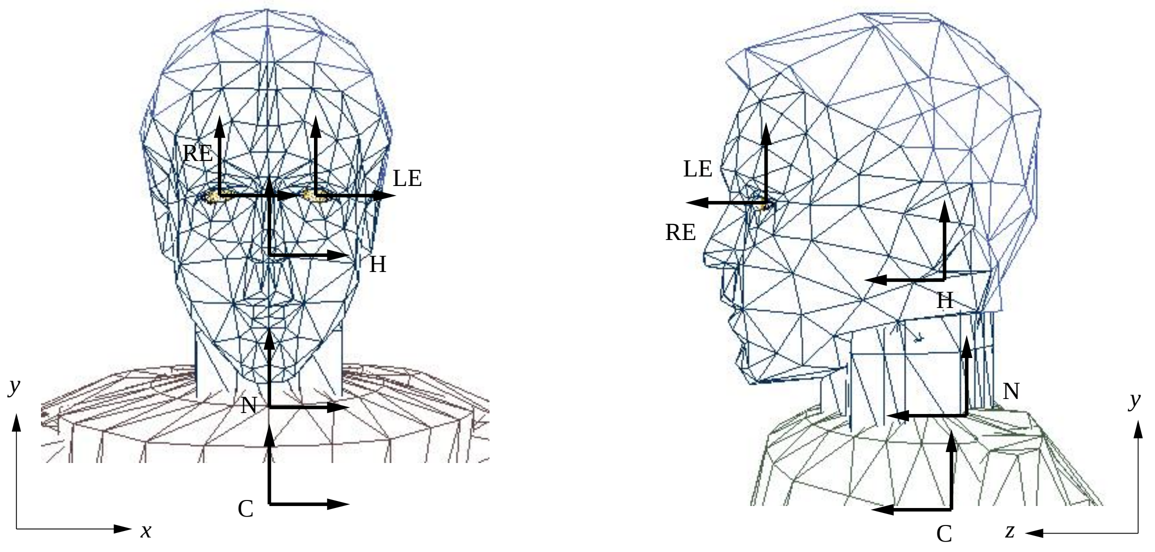

Figure 5.11: Frames of reference for the eye-neck chain.

vector location of the gaze target in the global environment frame. $gaze(\mathbf{p})$ computes the relative transformations of the neck, head, right eye, and left eye. Figure 5.11 depicts the local frames for each of these links. For simplicity, all joints are modeled as spherical joints.

Given a point of interest $\mathbf{p}$, the neck and head links are rotated to face $\mathbf{p}$ by a proportion derived from the timing curves of Figure 5.10. At the start of the gaze motion, the neck and the head are positioned at their current rotation (used as a reference rotation). The neck frame N is rotated relative to the chest frame C, the head frame H is rotated relative to N, and each of the eye frames (RE and LE) are rotated relative to H. The rotations are computed as follows:

$$\begin{aligned}
R_N &= \eta(w_{neck}, R_{ref}, \gamma(C_z, \mathbf{p} - N_o)) \\
R_H &= \eta(w_{head}, R_{ref}, \gamma(N_z, \mathbf{p} - H_o)) \\
R_{LE} &= \gamma(H_z, \mathbf{p} - LE_o) \\
R_{RE} &= \gamma(H_z, \mathbf{p} - RE_o)
\end{aligned}$$

R_N, R_H, R_{LE}, and R_{RE} are the rotations of the neck, head, left eye, and right eye frames respectively. $\eta(w, R_0, R_1)$ is a function that returns an interpolated rotation between the rotations R_0 and R_1 by the weighting factor $w \in [0, 1]$. Since our implementation uses quaternions to represent rotations, a spherical linear interpolation (slerp) function is used

[Sho85]. R_{ref} is the reference rotation, and the weights w_{neck} and w_{head} is used to control the amount of relative neck and head rotation. $\gamma(\mathbf{a}, \mathbf{b})$ is a function that returns a rotation which maps the unit vector along $\mathbf{a}$ to the unit vector along $\mathbf{b}$. C_z is the z-axis of the chest frame in global coordinates (N_z and H_z are the z-axes of the neck and head frames respectively). N_o is the position of the origin of the neck frame in global coordinates (H_o, LE_o, and RE_o are the origins of the head, left eye, and right eye frames).

To prevent illegal postures, as each of R_N, R_H, R_{LE}, and R_{RE} are computed, the rotation is checked against the joint limits for the link. If the joint limits are violated, the rotation is snapped to the nearest valid rotation. Joint limit violations typically only occur if $\mathbf{p}$ is placed behind the character, very close to the character's face, or near the vertical or lateral extremes of the visual field.

For animating the head and eye movements of human characters performing manipulation tasks, the gaze function is used with different points of interest for each of the three motion stages. The relative timing of the eye saccades approximates the trajectories in Figure 5.10 by varying the weights w_{neck} and w_{head} from 0 to 1 as the arm is animated. The point of interest used for each stage is as follows:

- *Reach*: The location of the object $\mathcal{O}$.
- *Transfer*: After grasping $\mathcal{O}$, the destination (target) location for $\mathcal{O}$.
- *Return*: Once $\mathcal{O}$ has been placed at the target location, a default point in the scene.

The default point of interest is used to allow the character to assume an idle gazing posture (for example, looking straight ahead).

The motion of the head and eyes overlaps with the motion of the arm. Based on our experiments, the gazing motion should begin at the same time or slightly precede the arm motion for a more natural effect. Example images of manipulation task animation for human characters using this gazing model are shown in the next section.

5.6 Results and Discussion

This section presents the results of an experimental implementation of the manipulation task motion generation strategy for human characters presented in this chapter. We tested the inverse kinematics, grasping, path planning, and human gaze model in an interactive

	Time (seconds)		H/B
Example	Basic	Heuristic	Ratio
Figure 5.12a	0.071	0.016	0.23
Figure 5.12b	0.262	0.174	0.66
Figure 5.12c	0.274	0.228	0.83
Figure 5.12d	0.552	0.534	0.96
Figure 5.13a	1.694	1.840	1.08
Figure 5.13b	1.817	1.628	0.90
Figure 5.13c	6.644	5.940	0.89
Figure 5.13d	23.751	26.339	1.10

Table 5.1: Planner comparison ($N = 100$)

application involving human characters. For path planning, we show experimental results using both the basic planner and the RRT-Connect planner with a 7-DOF human arm model, as well as a variety of other robots and environment geometries. The majority of these experiments were conducted on a 200 MHz SGI Inidigo2 running Irix 6.2.

Basic Examples

For testing the path planning heuristic, we initially experimented by planning motions for rigid objects in 2D. Both the basic planner and the RRT-Connect planner were given the same queries and subjected to 100 trials each. The results were averaged and compared. Representative runs for eight different examples based on the average performance statistics are shown in Figure 5.12 and Figure 5.13. All of the examples in Figure 5.12 were solved in less than one second, while those in Figure 5.13 took slightly longer[2] (see Table 5.1 for complete timing results).

Utilizing the heuristic resulted in as much as a 75 percent reduction in total collision checks and nodes explored for simple queries. This translated into a similar percentage decrease in total execution time, since collision checking and nearest-neighbor calculations dominate the computation. As expected, the advantage of using the heuristic is especially apparent for queries that admit a very simple solution. This is the primary motivation for

[2]For uniformity in answering planning queries, our current implementation performs all collision checking in 3D. Thus, collision checking in the 2D examples is actually performed using 3D models that are constrained to lie in a plane. Using 2D polygon-based collision checking software could have significantly improved our raw performance data, but these experiments were designed primarily for making a relative comparison test.

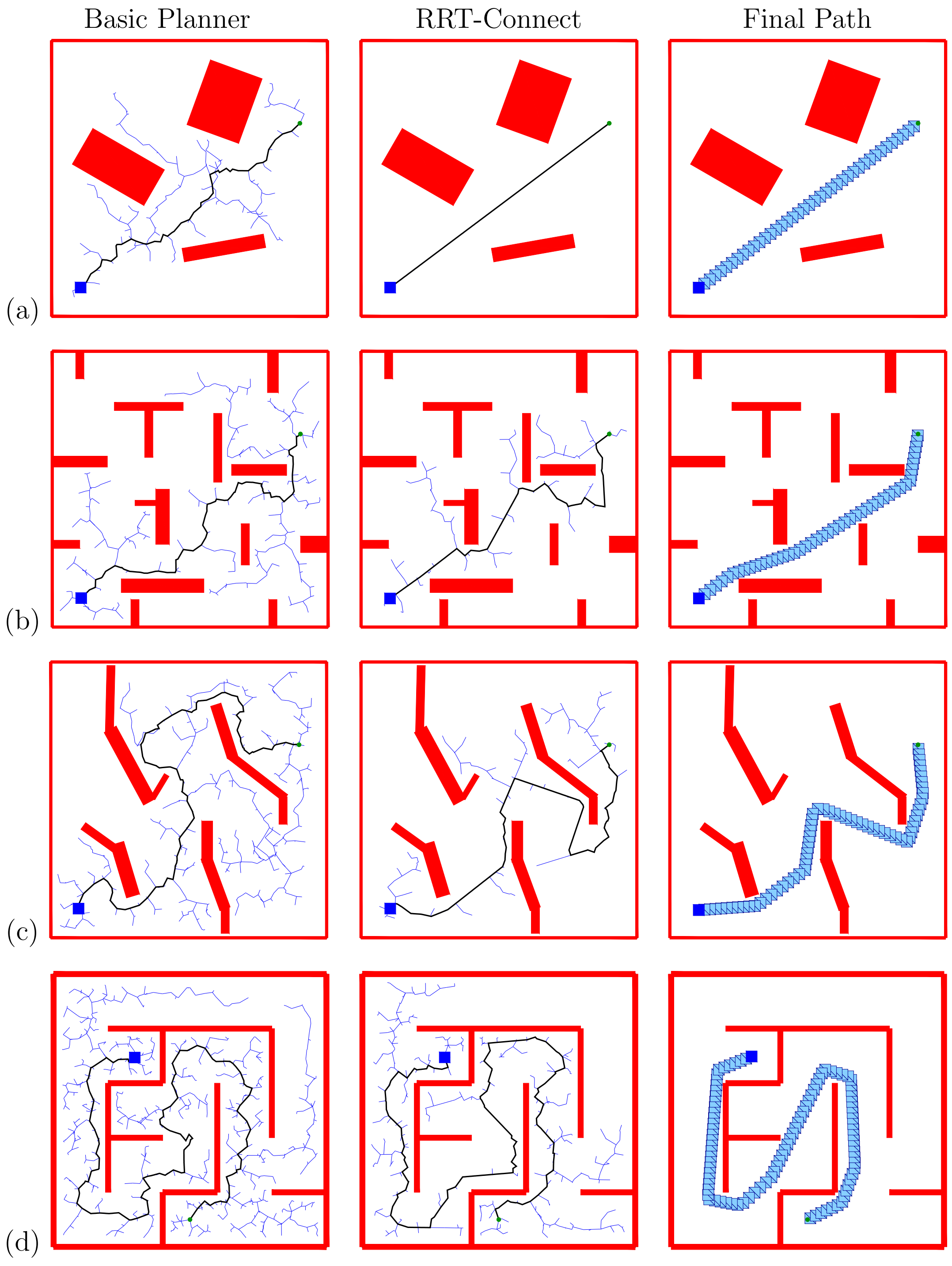

Figure 5.12: Basic vs. RRT-Connect heuristic. Simple queries for a translating 2D rigid body.

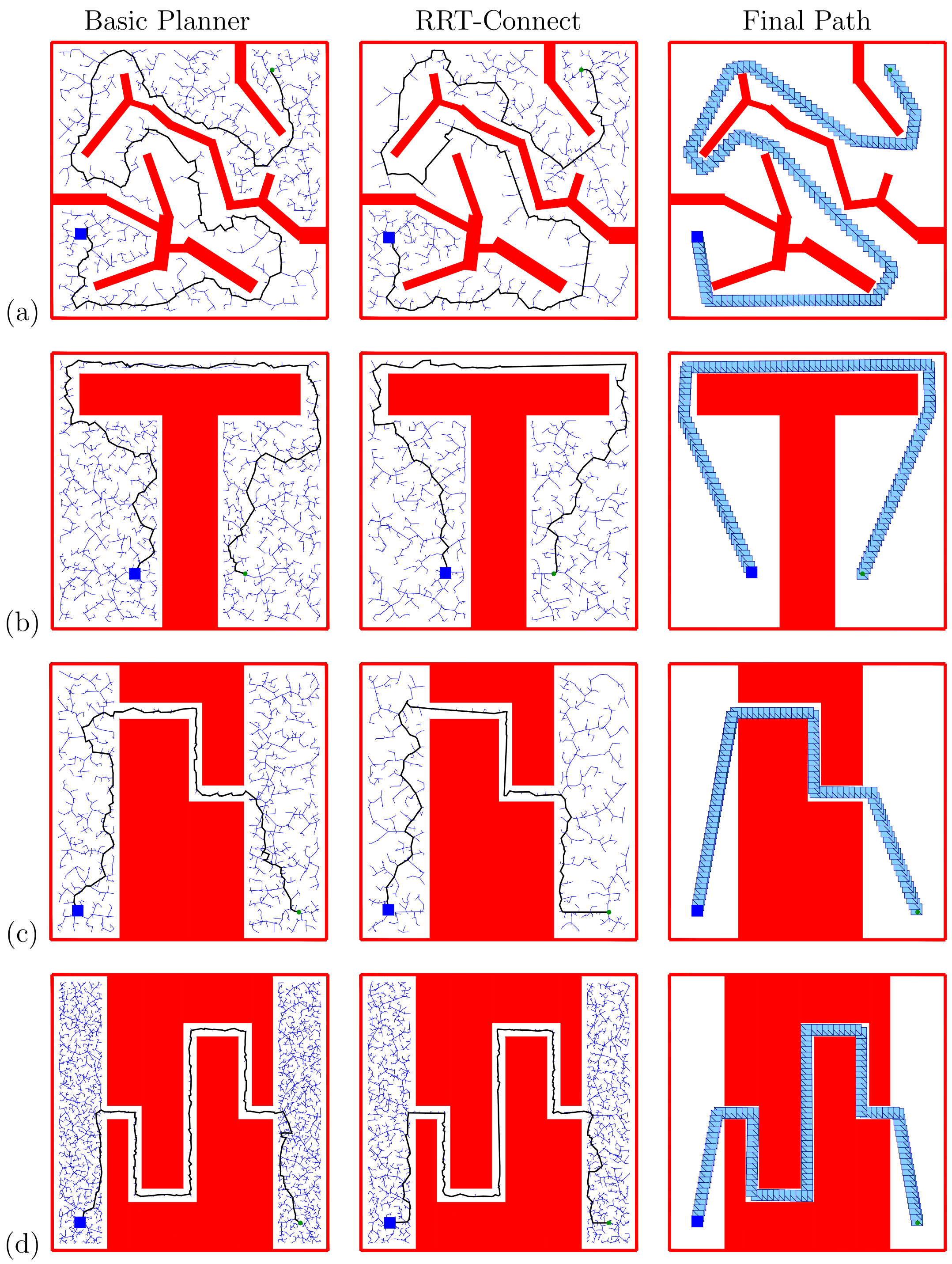

Figure 5.13: More challenging queries for a translating 2D rigid body.

developing this heuristic in the context of animating manipulation motions. The majority of grasping and manipulation tasks typically involve fairly large regions of free space. It is these types of spaces that the heuristic generally performs well in.

Consider example (a) in Figure 5.12 involving a wide-open space. Like most planners based on potential-fields, the heuristic immediately connects the start and the goal. Examples (b) and (c) involve slightly more obstructing obstacles. The planner begins to explore the surrounding space as obstacles are encountered, while simultaneously rapidly converging towards the goal. In addition, the nature of the construction of the RRTs tends to prevent samples from accumulating in regions of previously explored space. Intuitively, RRTs can be thought of as randomized Voronoi diagrams, where large Voronoi regions are more likely to be sampled[LaV99].

More Complex Examples

A few more challenging queries are shown in Figure 5.13. The first example (a) involves a twisty maze with a long solution path, while the next three examples involve various forms of *narrow passages*, which are generally considered to be the most difficult queries for path planning[HKL^{+}98]. In these examples, the robot is a square whose side length measures 0.04 of the diameter of the space. The passage at its narrowest measures 0.05 of the diameter. In examples (a) and (d), the basic planner performed slightly better on average (between 5 to 10 percent). These examples were purposefully designed to be disadvantageous to the heuristic. Intuitively, the heuristic will likely reduce performance if the solution path is very twisty, or if the trees must initially grow in opposite directions before eventually coming together. Based on our experiments, the potential computational gain by utilizing the heuristic can be quite large, while the potential risk is relatively small.

Figure 5.14 shows the the result of using the RRT-Connect planner to compute a motion to move an L-shaped object through a set of narrow gates. The exploration tree on the left is the 2D projection of the actual tree which resides in a 3-dimensional $\mathcal{C}$-space. The right image shows the final path after smoothing. Figure 5.15 depicts the same planner as applied to a 3D model of a grand piano moving from one room to another amidst walls and low obstacles. Several tricky rotations are required of the piano in order to solve this query.

Before applying the planner to a human arm model, experiments were conducted planning manipulation motions for a model of a 6-DOF Puma industrial manipulator arm. For

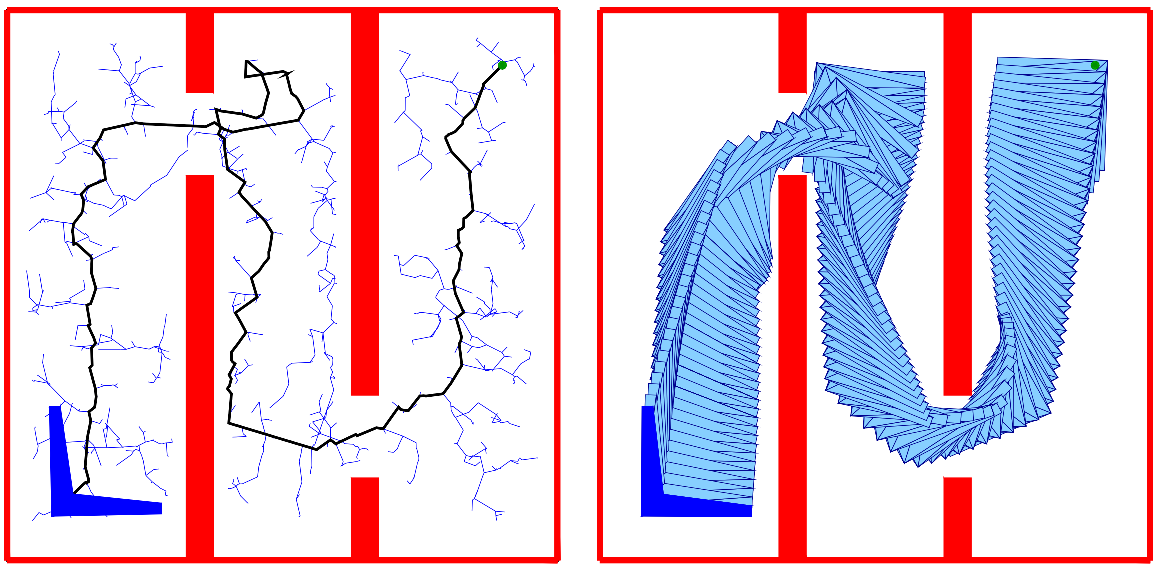

Figure 5.14: The RRTs (left) and computed path (right) for an L-shaped rigid body in 2D.

computing goal configurations, an analytical inverse kinematics algorithm is used. Several snapshots of a path to move a book from the middle shelf to the bottom shelf of a desk is shown in Figure 5.16.

For the queries we tested, motions were computed in a few seconds on average. Interactive applications such as this could become a potential useful tool for the visualization of robotic systems, as well as an automated, task-level programming interface for manipulator robots.

Human Arm Animation

Experiments using the planner to generate manipulation motions for human arms were conducted. Through a graphical user interface, an animator can interactively click and drag an object to a target location and issue a *move* command. The planner will then attempt to compute the motions to *reach* the object, grasp it, *transfer* it to the target location, release it, and *return* the arm to its rest state.

Figure 5.17 shows a human character commanded to move a bottle while seated. This sequence of motions involves two calls to the inverse kinematics solver, and three path planning queries. Traces of the computed paths are shown in the images. The motion of

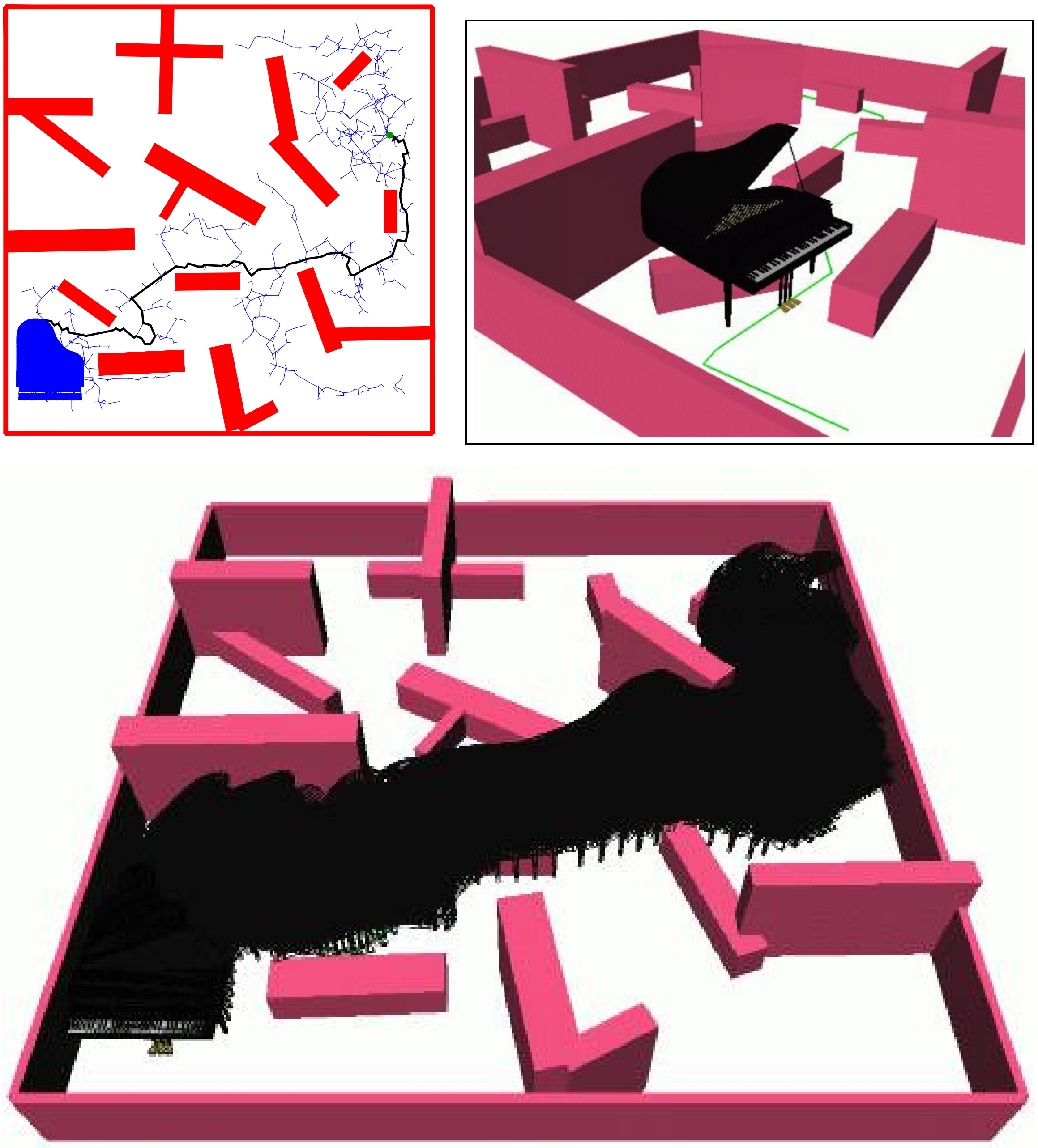
Figure 5.15: Moving a piano in a room with walls and low obstacles.

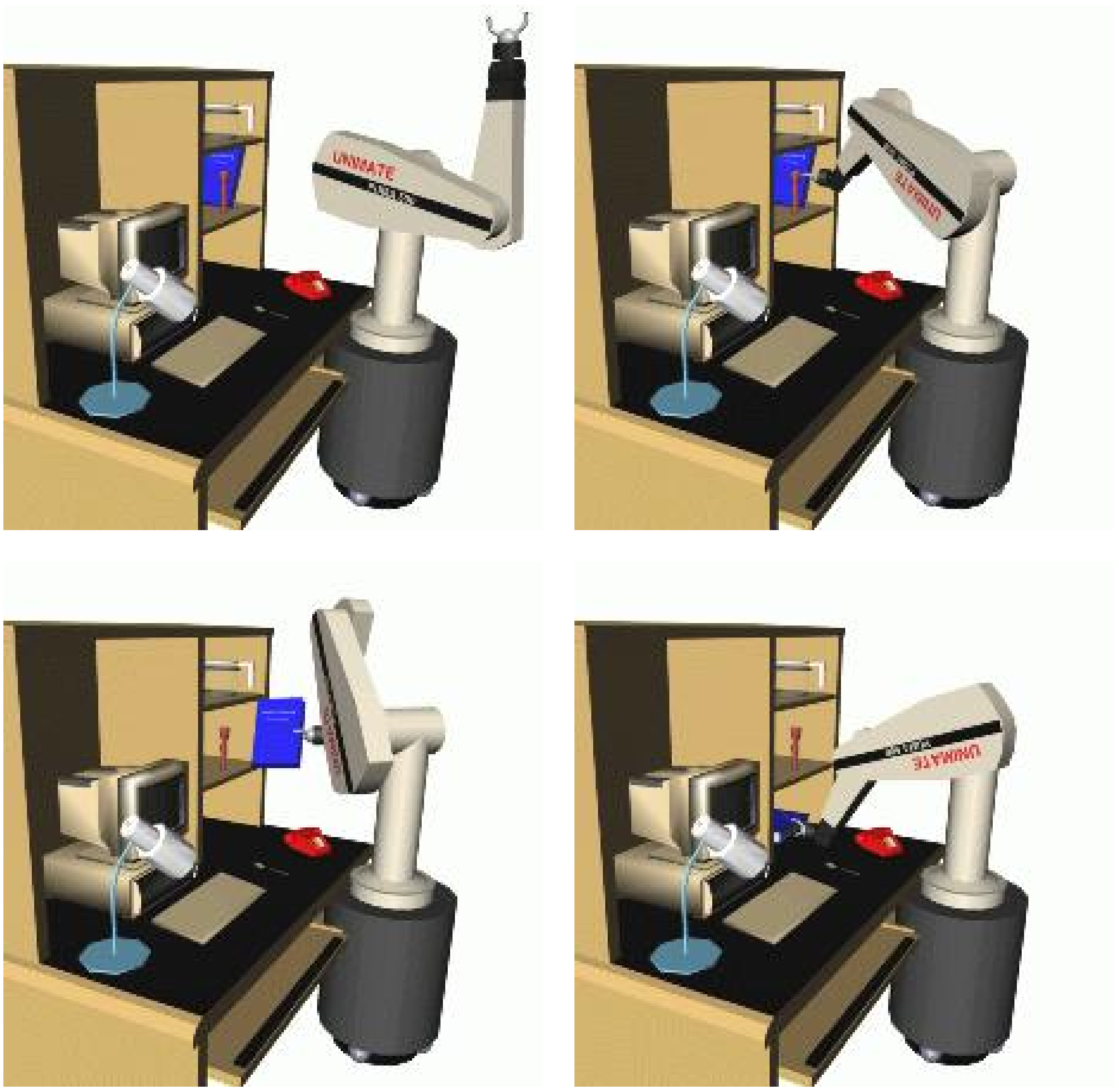

Figure 5.16: A 6-DOF Puma robot moving a book.

the neck, head, and eyes are computed using the approximate gaze model of Section 5.5.

All motions were computed in under 2 seconds on a 200 MHz SGI Indigo2 running Irix 6.2. The human arm is modeled as a 7-DOF kinematic chain, and the entire scene contains over 12,000 triangle primitives. The 3D collision checking software used for these experiments was the RAPID library based on OBB-Trees developed by the University of North Carolina[GLM96].

As mentioned previously, the RRT-Connect heuristic works most effectively when one can expect relatively open spaces for the majority of the planning queries. Figure 5.18 shows a human character playing chess. Each of the motions necessary to reach, grasp, and reposition a game piece on the virtual chessboard were generated using the RRT-Connect planner in an average of 2 seconds on a 200 MHz SGI Inidigo2. The scene contains over 8,000 triangle primitives. The user can interact with the character in real-time, and even engage in a game of "virtual chess", since the manipulation task commands can be constructed automatically from moves issued by chess-playing software.

This planner works extremely well for tasks such as moving game pieces around, since there is a relatively large amount of free space in which to move the arm. However, it can also handle more complicated queries with narrow passages in $\mathcal{C}$-space, such as the assembly maintenance scene depicted in Figure 5.19. Here, the task is to grasp the tool from within the box and place it inside the tractor wheel housing. Solving this particular set of queries takes an average of 80 seconds on an SGI Indigo2, and about 15 seconds on a high-end SGI (R10000 processor). The scene contains over 13,000 triangles.

Discussion

For planning single-arm manipulation tasks for human characters, we have proposed a motion generation strategy that relies primarily on inverse kinematics and path planning software. For path planning, two planner variants are proposed. In particular, we have developed the RRT-Connect heuristic which improves the overall convergence and speed at which we can answer many types of path planning queries. There are several important characteristics of this method that are worth considering:

1. **No pre-processing:** The planner was developed to be able to quickly answer single queries in dynamic virtual environments where both obstacles and the character itself

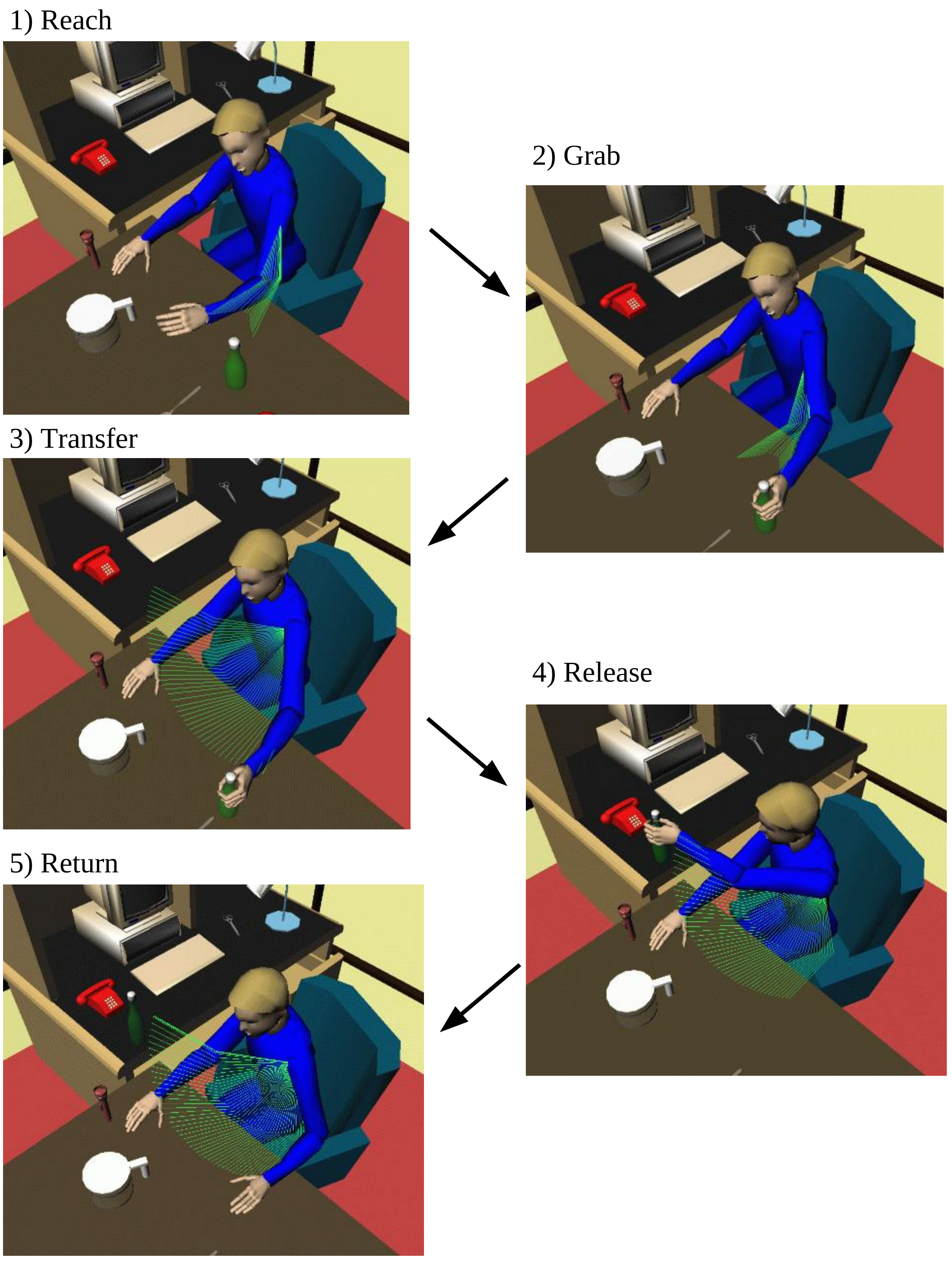

Figure 5.17: A human character repositioning a bottle.

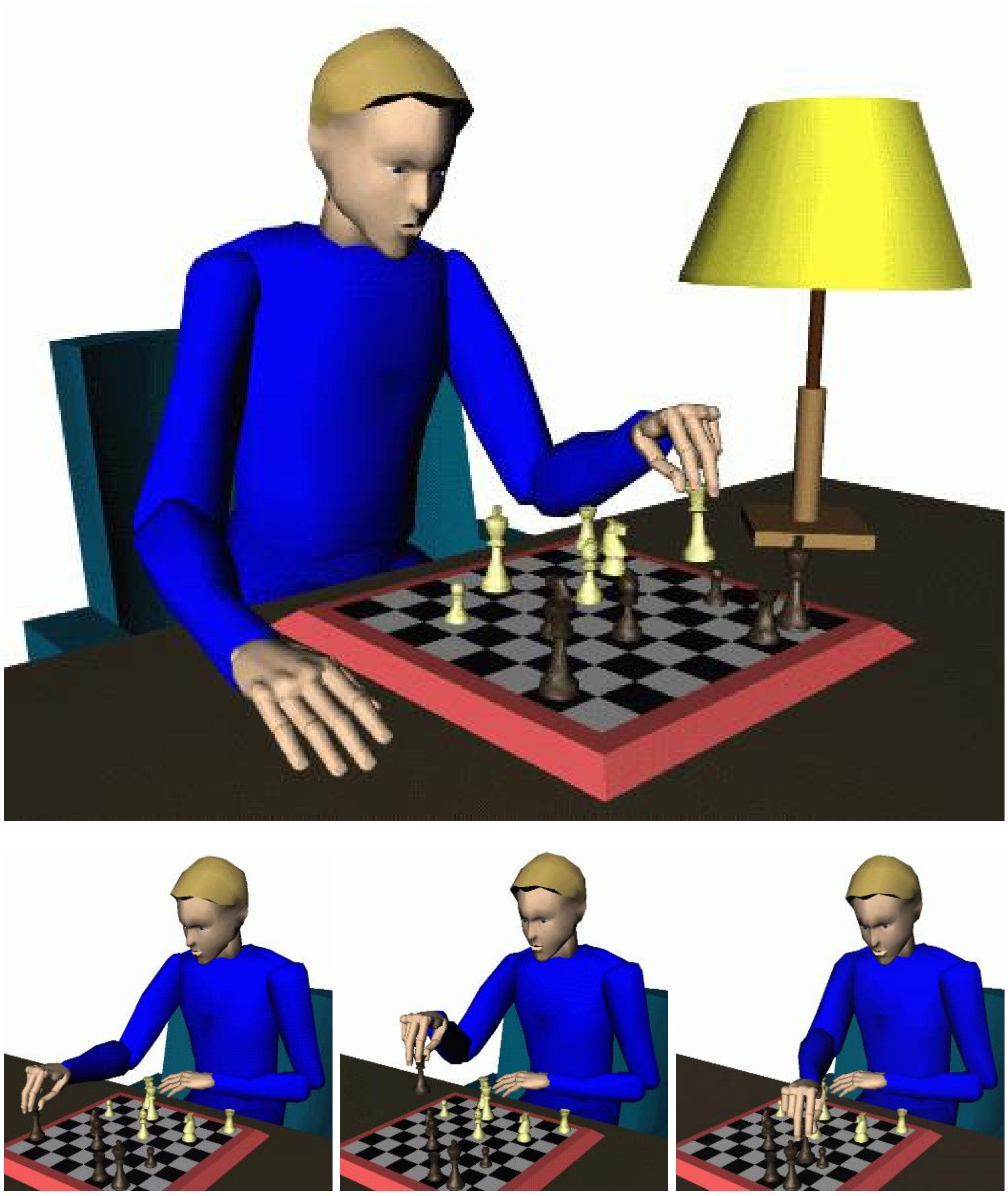

Figure 5.18: Playing a game of virtual chess.

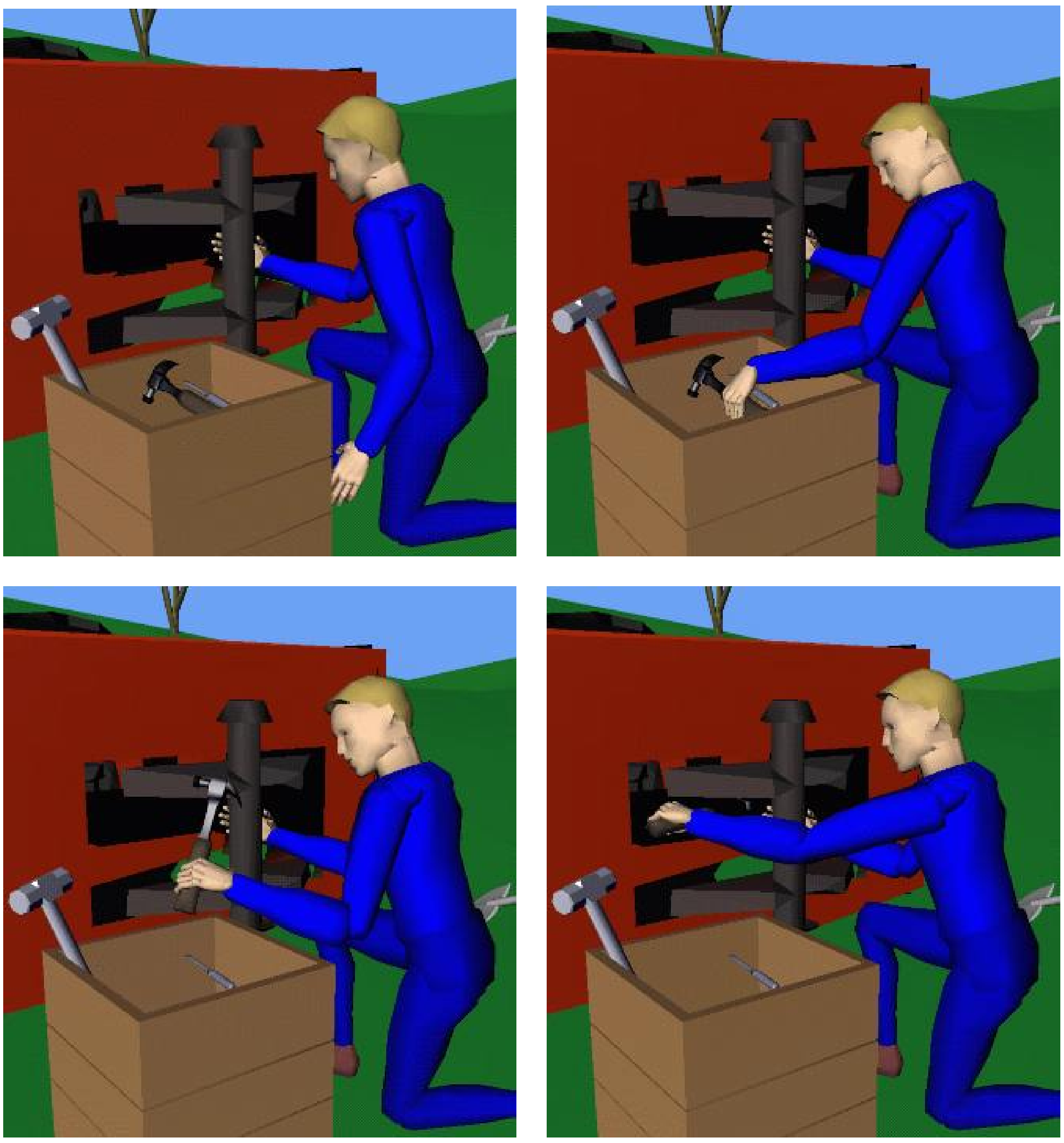

Figure 5.19: A more difficult planning query.

are non-stationary (thus changing the shape of $\mathcal{C}_{free}$ and precluding the use of preprocessing).

2. **Rapid Convergence:** The heuristic was designed to attempt to quickly establish a connection between the two search trees. The time taken to solve a particular query generally scales with the relative complexity of the solution path induced by the geometry of $\mathcal{C}_{free}$.

3. **Uniform Exploration:** The planner utilizes RRTs to explore the surrounding free space, while simultaneously being pulled towards establishing a connection between the trees. Intuitively, being drawn towards different individual nodes in the opposing tree at different times, is like having a potential field that moves around to different areas of $\mathcal{C}_{free}$ rather than remaining fixed at the goal.

4. **No "magic" numbers:** Rather than having multiple parameters to tweak, this planner is simple and requires only a single parameter to be specified (the value of d_{max}). This facilitates ease of implementation and consistency in performance evaluation. In fact, it may even be possible to slightly modify the planner to adaptively select the value of d_{max} according to how successful the planner is at adding branches. If it is often successful, it may try to increase d_{max} (optimism), while if it often fails, it may decide to decrease d_{max} (cautiousness).

5. **Relatively short paths:** The raw paths (prior to smoothing) generated by the planner, though slightly jagged, are relatively short. This is likely due to the fact that RRTs grow with a very low probability of producing paths that spiral or cross themselves.

In the context of the particular application that this planner was developed for (the automatic animation of manipulation tasks), it is now possible to generate complex manipulation motions for animated human figures at interactive rates.

Although the planner has been observed to be efficient and reliable in practice, theoretical analysis of the convergence rate remains to be done. In our current implementation, a predefined maximum execution time limit is used after which the planner returns failure. Thus, the planner is not *complete*, meaning that when it fails, it cannot guarantee that a path does not exist, only that it was unable to find one in the time allotted. Designing a complete planner is possible [Can88], but known *practical* planning methods for high-DOF configuration spaces are not complete. Analyzing the conditions under which the planner performs poorly (as in [BKL$^+$97]), is an area of future research.

Overall, we have found the motions generated by the planner to be quite satisfactory. However, the motion generation strategy could be improved in a number of important ways. Each of the following paragraphs outlines one of several of these issues, along with a discussion of potential methods for improvement.

More Flexible Inverse Kinematics

In our experiments, we assume that the torso and shoulder position remains fixed for the duration of the manipulation motion. This assumption is reasonable for manipulation tasks involving relatively small, nearby objects. However, other tasks require the use of additional degrees of freedom in the shoulder, torso, or legs.

Incorporating these additional degrees of freedom into the path planner itself is relatively straightforward, as we expect the RRT-Connect heuristic planner to scale well to higher-DOF configuration spaces. The primary effort that must be made involves modifying the inverse kinematics algorithm to take these additional degrees of freedom into account. Having such capability in the IK solver would certainly be a useful extension to the current implementation.

Multi-arm Manipulation Tasks

This chapter considers only manipulation tasks that can be accomplished by a single arm. However, tasks involving objects that require multiple arms are numerous. Extending the planner to compute multi-arm motions is a necessary next step to enlarging the set of tasks that can be handled by the planner. Incorporating techniques from Koga, *et al.*[KKKL94a] is one possibility.

Automatic Grasp Generation

Currently, we assume that a valid set of grasps for a particular character morphology is pre-specified by an animator and associated with an individual object (or class of objects with similar geometry). Other similar methods for storing information about how a virtual object should be manipulated (such as grasp locations) along with the object itself have been developed [GLM94, KT99]. Although the labor of specifying valid grasps for an object need only be performed once, it is a requirement that we would like to avoid.

Calculating a set of possible grasps automatically from an object's geometry is a complicated problem, and has been addressed in the robotics literature (see [PT89] for a survey of this work). Adapting some of these techniques to automate the generation of grasp sets for animated human figures based on the geometric or physical properties of a virtual object would be very useful.

"Naturalness" Constraints

The planner described in this chapter enforces no "naturalness" constraints during planning in order to guarantee that the generated motion appears natural. The naturalness of the goal pose is primarily due to the inverse kinematics algorithm. The intermediate poses, although unconstrained, generally appear natural perhaps because the path optimization technique shortens paths in the configuration space according to the distance metric. Depending upon the nature of the distance metric, this may tend to generate paths that seem to be fairly "efficient", and therefore perhaps likely to be close to how an actual human might move in a similar situation. However, more experimental work is needed to determine better methods of producing natural-looking trajectories (for example, see [FH85].)

One fairly simple improvement to the planner would be to bias the sampling of $\mathcal{C}_{free}$ to favor arm configurations that appear natural. For example, the IK algorithm could potentially be used to rate a sampled configuration in terms of "naturalness". The probability of retaining a sample during planning would depend on its rating. Thus, one could penalize unnatural postures during the search.

Task-Based Constraints

In addition to maintaining a natural arm posture, other constraints could potentially be taken into account during planning. For example, suppose the task is to move a glass of water. This task has the added constraint that the glass should be kept vertical during the arm motion to prevent the contents from spilling. As with naturalness constraints, the planner could be biased to favor arm configurations that result in vertically upright glass positions. Sampled configurations that result in horizontal or inverted glass positions would be either discarded or retained with very low probability.

Physically-Based Constraints

The majority of animation research for human characters has been performed at the level of kinematics. Recently, there has been an upsurge in interest in physically-based models as the increase in computing power has allowed fairly sophisticated simulations to be calculated at interactive rates. Although some preliminary work has been done to apply physically-based or biomechanical models to animating human figures [LWZB90, PW99, LM99], such techniques have not yet been applied to generating physically-correct motion for complex object manipulation tasks.

As with task-based constraints, the physics of the arm motions and the strength limitations of the arm muscles could potentially be used to formulate additional search criteria during planning. Motion planning that considers the dynamics of the system is sometimes referred to as *kinodynamic planning.* Though some work has been done on kinodynamic planning[O'D87, DXCR93, DX95, LK99], none has considered complex models such as human figures. Given a biomechanical model of a human arm, it may be possible to automatically plan physically-correct motions.

Researchers in human motor control and analysis have had limited success in devising general models of human muscle activation patterns. There is even more ambiguity in relating patterns of muscle forces to task frame trajectories than in relating kinematic trajectories in the joint space to task frame trajectories [JvdGG89, ZG89]. The problem is ill-posed since the degrees of freedom (muscles controls) are even more excessive (redundant) than in the strictly kinematic case. For example, there are two major muscle groups whose contraction affects the elbow flexion degree of freedom. Moreover, the moment arm lengths and lines of force for these muscle groups actually vary as the elbow bends[CA91]. Despite this, it may be possible to devise an approximate model for producing reasonable motions.

More experimental work is needed to determine whether kinodynamic planning techniques using biomechanical models can be adapted for the purposes of computer animation.

Realistic Velocity Profiles

In our current implementation, we use a simple spline function to generate a time parameterized trajectory from the path returned by the planner. However, one could potentially generate more realistic velocity profiles based on studies of human movement [AH84]. Some have suggested generating expressive animation using velocity profiles based on Laban movement analysis[BCC99]. Alternatively, using an appropriate physically-based model of the human arm, a physically-correct time-optimal trajectory along the planned path can be computed [BDG85]. In combination with a variational technique and an appropriate cost function, a purely kinematic path can be made to satisfy dynamic or naturalness constraints [SD91]. These or similar methods could potentially be used to increase the realism of the computed motion, especially if a manipulation task involves moving a heavy object.

Visual and Tactile Feedback

The motion generation strategy presented in this chapter does not utilize any sensory information during planning. It is a purely geometric planner that considers only the shape of the character, the object, and the environment when constructing a motion plan. However, real humans use both visual and tactile feedback extensively when performing manipulation tasks. By combining the gaze function derived in this chapter with the simulated vision module of Chapter 4, one could potentially design a manipulation task planner that incorporated rudimentary visual feedback. At the beginning, the planner could verify that the object is actually visible before attempting to grasp it (see Section 4.6). If the object is not visible, the character could attempt to reposition itself or initiate a searching behavior. During the arm motion itself, simulated visual feedback could be used to simulate the hand-eye coordination of real humans.

Simulated tactile sensing is appropriate for manipulation tasks which fundamentally involve contacts between a character and an object. Ultimately, a character could utilize both visual and tactile feedback for manipulating objects, as do some industrial robots. For example, a physically-based muscle model of the arm could be used to servo the hand using visual feedback until the simulated sense of touch indicates contact with the object. Adopting this kind of manipulation model could also potentially automate the generation of natural motions for the head and eyes, as the character moves to maintain the visibility of the object in the center of its visual field.

Chapter 6

High-Level Behaviors

6.1 Introduction

For the purposes of creating interesting animations for autonomous characters, some degree of high-level scripting of a character's behavior is needed. The general idea is to aggregate several lower-level task commands using a logical scripting language, in order to create a script (program) that defines a higher-level behavior.

This chapter describes how more complex high-level behaviors for animated characters can be constructed. The concepts are illustrated with simple scripted examples that have been implemented and integrated with the task-level navigation and manipulation planners presented in Chapter 3 and Chapter 5 respectively. The examples are discussed along with a summary discussion.

6.2 Programming Behaviors

Scripting Languages

High-level behaviors for animated characters are typically defined via scripts written in a scripting language. Many commercial applications (such as video games) employ some kind of scripting system for animating complex behaviors. Various researchers have also proposed different kinds of scripting systems for designing animated agents (for example, the PAT-nets and SCA loops of U. Penn [BWB+95], Funge's CML scripts [Fun98], and the Improv system at NYU [PG96] which uses non-deterministic scripts). Although some systems make complex behaviors easier to program than other, all generally offer the same functionality. This is evident by the fact that any given scripting system can typically be programmed to emulate (simulate the behavior of) any other scripting system.

At a minimum, a scripting language should provide the ability to *loop*, perform *conditional branches*, and invoke commands for executing *atomic tasks*. The ability to *invoke other scripts* (possibly passing parameters to them) is not necessary, but very convenient for modularizing behaviors.

Atomic tasks are either low-level motor commands, high-level task commands (such as "moveTo", "getObject", "touchObject", etc), or commands to initiate the playback of a pre-defined motion, such as waving hello to another character (see Section 2.4). A script can define not only a sequence of atomic tasks that make up a more complex behavior, but also the logical relationships and conditions under which the atomic tasks should be performed.

System Integration

Communication between a scripting language and an animation system is accomplished via what we shall refer to here as *sensors* ("inputs") and *actions* ("outputs"). Sensors serve as feedback channels, providing abstract representations of what the character perceives in the environment. Actions correspond to atomic actions that can be performed by the character.

Using the data provided by sensors in conditional branches facilitates the ability to alter a character's actions depending upon what the character perceives in its environment. As a consequence, even very simple scripts with sensing feedback can yield quite complex behaviors and interactions between characters. This feedback loop corresponds to the practical implementation of the control loop of the "virtual robot".

The Generality of Scripting

With the necessary sensors and actions accessible through a scripting language, scripts are a fully-general method of programming a character. There is often conceptually no limit to the complexity of a script that may be defined (though limitations to their ability to be practically implemented may exist). Thus, even a very complicated behavior-based model can generally be simulated using a simple scripting language.

Some researchers have argued that parallel computation will fundamentally alter the abilities of software agents. In fact, parallelism itself does not provide any computational advantage, since any parallel machine can be simulated by a single-processor serial machine. However, the improved performance due to parallelism may enable algorithms that are too slow for serial machines to become practical.

6.3 Following and Pursuit Behaviors

Using a navigation algorithm as a scriptable action, interesting animations involving multiple characters can be created. For example, simply setting the navigation goal of one character to be the sensed location of another character immediately yields following or pursuit behavior.

We have created several example scripted behaviors using the navigation strategy presented in Chapter 3. Figure 6.1 shows an example involving two characters that have been scripted to follow another character that is under user control (in the foreground). The actual script executed is shown in Figure 6.2. Because path planning is used for navigation, the character will naturally circumvent obstacles while following the target character.

There are several possible strategies for avoiding collisions between characters. One simple idea is have each character "plan around" the others. For example, we can project the geometry of the other characters at their current locations prior to obstacle growth. Since each character may change its location over time, replanning may be necessary periodically in order to avoid the possibility of a collision between two characters. One simple coordination scheme might replan whenever other moving characters cross a character's current path.

More sophisticated schemes that account for character velocities are also possible. Instead of planning around the other characters at their *current positions*, each character could plan around the other characters' *predicted future positions*. A character's future position could be calculated by a simple extrapolation of its current actual or estimated velocity. The examples shown in Figure 3.12, Figure 3.13, and those in Figure 6.1 and Figure 6.4 utilize this technique in order to avoid potential inter-character collisions.

Increasingly complex animations can be constructed by building upon existing behaviors. For example, given the following (pursuit) behavior, it is quite simple to script several characters to play a game of tag, where one character who is "it" pursues nearby characters,

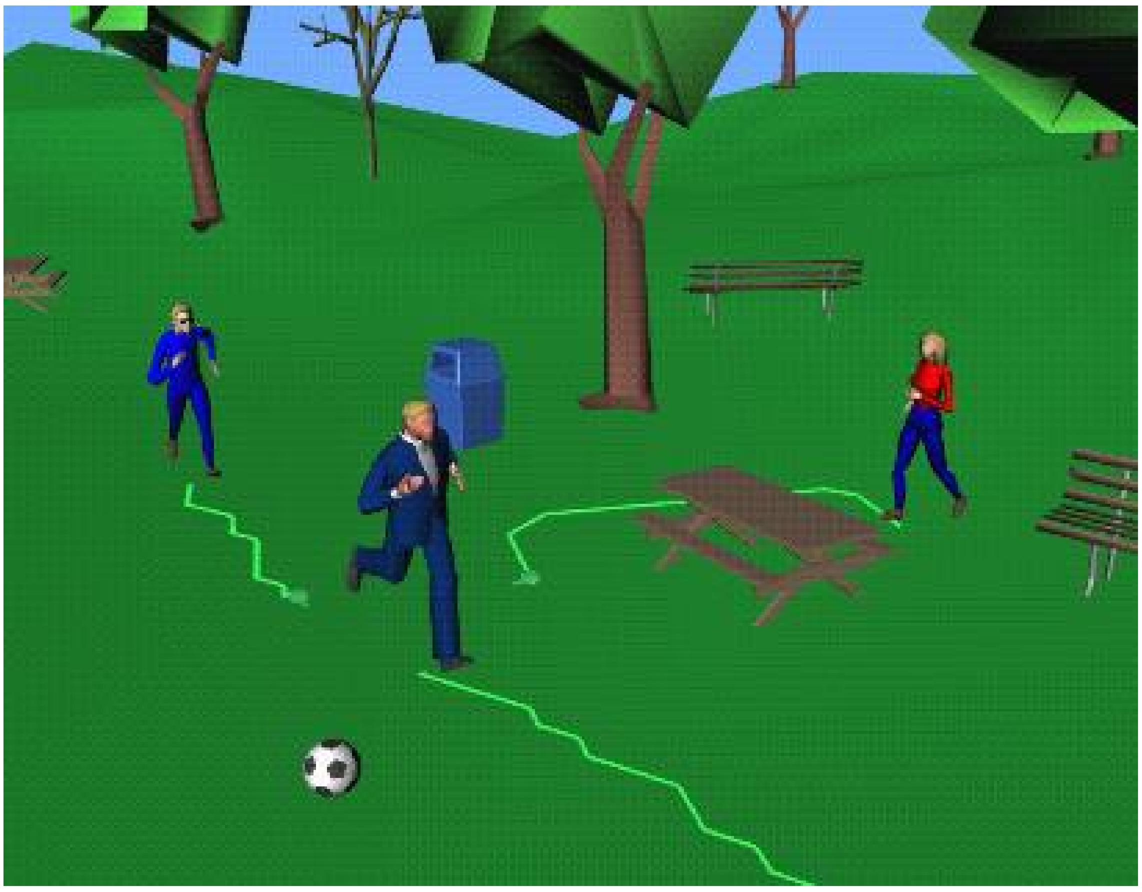

Figure 6.1: Two human characters scripted to follow another human character under user control.

```
while (TRUE) {
 if (! nearCharacter(target, 1.7))
  moveTo(target, 1.5);
 else
  stop();
}
```

Figure 6.2: An example following (pursuit) script.

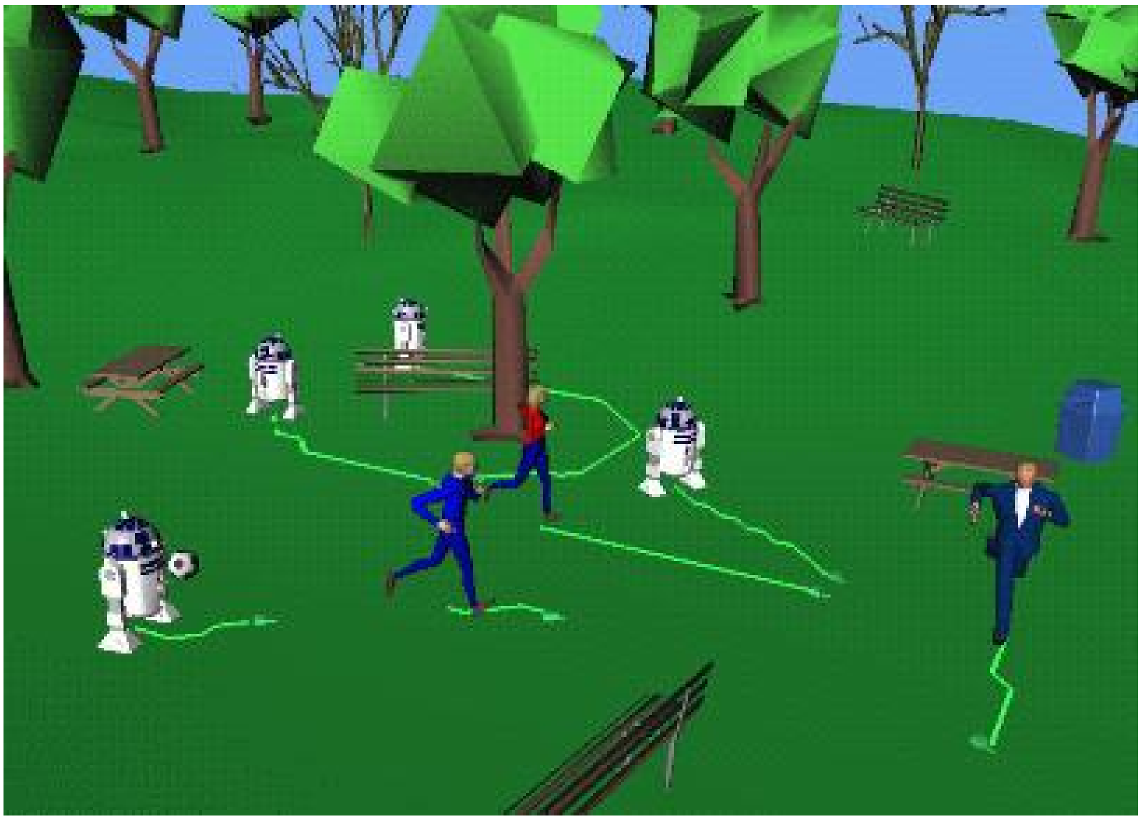

Figure 6.3: An scenario involving wandering and following behaviors for human and robot characters.

while the others attempt to flee.

Figure 6.3 and Figure 6.4 show examples involving seven characters (three humans and four robots) in different scenarios. In Figure 6.3, two human character are scripted to follow the other human character (which is under user control), while the robots are instructed to wander and follow any human that passes nearby. In Figure 6.4, all of the characters are instructed to pursue a single character under user control. Besides being applicable as a general pursuit behavior for video games, it can be used to animate a crowd following a leader (e.g. a virtual tour guide leading a group of virtual tourists).

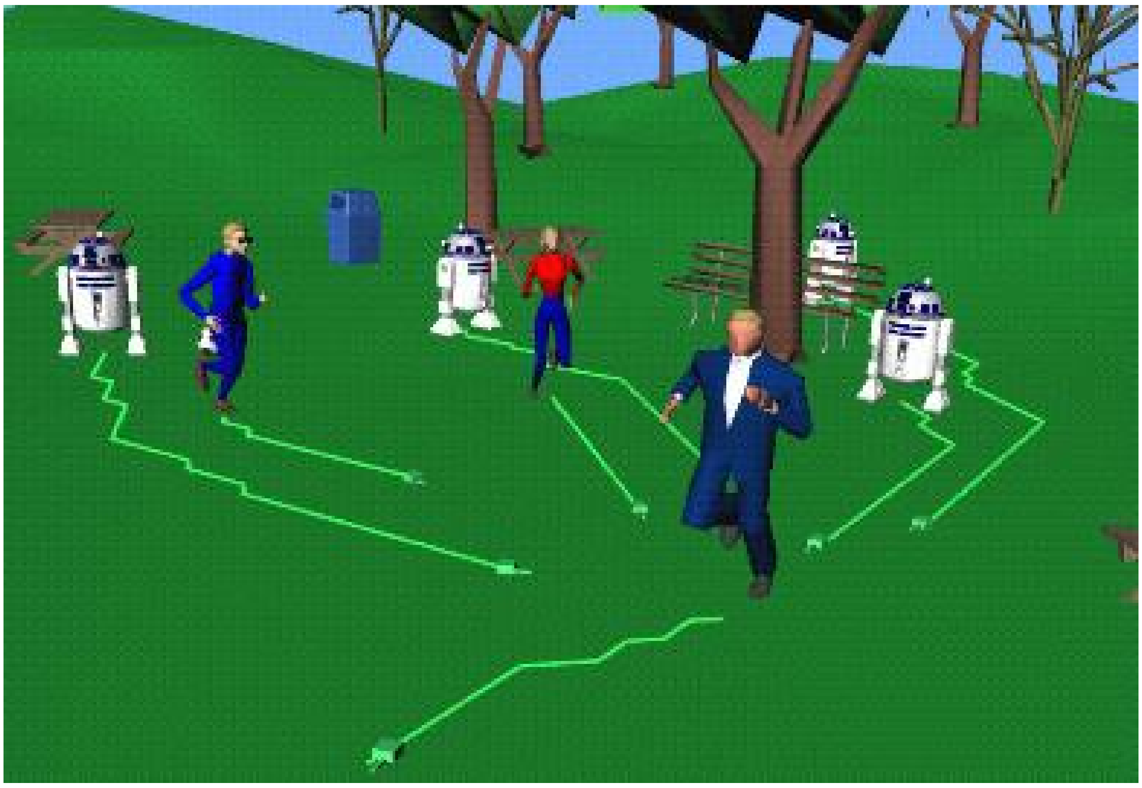

Figure 6.4: A group of characters scripted to pursue a user-controlled character.

6.4 Integrating Navigation and Manipulation

Given the two fundamental atomic task commands "moveTo" (navigation) and "getObject" (manipulation), one can script high-level behaviors that involve picking up and moving objects around. First, the navigation planner can be used to plan a motion for the character to move near a desired object. Then, the manipulation planner can be invoked to grasp and pick up the object.

For example, one could create a *cleaning behavior* for a virtual park employee in which the character wanders around picking up any litter it walks near. By utilizing the synthetic vision module of Chapter 4, the basic cleaning behavior could be modified to only pick up objects that the character *actually sees*. Synthetic vision could also be used to create a script that controls the character to actively look for litter instead of pure wandering. Modularizing behaviors (such a cleaning behaviors), allows us to integrate the behavior as a subgoal for more complex actions. For example, an autonomous waiter in a virtual cafe could roam around tables filling the patrons' cups of coffee, but if he happens to see some litter, the cleaning behavior can be invoked.

In the same way that a vast library of clip motions could potentially be used to animate specific tasks, a vast *library of behaviors* could be developed over time and provided as a resource to animators. As the number and complexity of the behaviors increases, the perceived "intelligence" of the the animated agents will likely also increase accordingly.

Chapter 7

Conclusion

The fundamental challenges in computer graphics and animation lie primarily in having to deal with complex geometric, kinematic, and physical models. The development of better software tools has advanced the state of the art in the modeling and rendering of these models. However, software tools for the automatic generation of motion are still relatively scarce. In particular, techniques are needed to animate both autonomous and user-controlled human figures naturally and realistically in response to high-level task commands.

The preceding chapters of this thesis have presented a research framework aimed at facilitating the high-level control of animated characters in interactive virtual environments. The next section briefly summarizes the key ideas contained in this thesis, and the following section contains a concluding discussion.

Summary

We approach the problem of automatically synthesizing motions for animated characters from the standpoint of modelling and controlling a "virtual robot" (an *autonomous animated agent*). We propose a general architecture for autonomous animated agents based on *planning*, *sensing*, and *control*. To test the viability of this approach, we have developed techniques for automatically generating the gross body motions for animated human figures given high-level navigation or manipulation task commands. Although we have concentrated on the animation of human-like character models, the basic ideas are applicable to other types of characters.

Goal-directed Navigation: We present a technique for quickly synthesizing collision-free motions for animated human figures given high-level navigation tasks amidst obstacles in changing virtual environments. The method combines a fast 2D path planner, a path-following controller, and uses cyclic motion capture data to generate the underlying animation. In order to provide a feedback loop to the overall navigation strategy, we incorporate an approximate synthetic vision module for simulating the visual perception of the character. It uses rendering hardware to quickly identify the objects visible to the character, along with a simple model for representing and updating a character's spatial memory. The model is efficient in both storage requirements and update times, and can be flexibly combined with a variety of higher-level reasoning modules or complex memory rules. Using such a sensing and memory model, a character can be made to explore in real-time an unknown environment, and incrementally build its own internal model of the world while it navigates toward a goal location.

Object Manipulation: We introduce a new manipulation planner designed for efficiently generating collision-free motions for single-arm manipulation tasks given high-level commands. For moving an object, the planner automatically generates the motions necessary for a human arm to reach and grasp the object, reposition it, and return the arm to rest. The planner searches the configuration space of the arm, modeled as a kinematic chain with seven degrees of freedom. Goal configurations for the arm are computed using an inverse kinematics algorithm that attempts to select a natural posture. To compute a collision-free path connecting the arm initial configuration to the goal configuration, we present an efficient, general path planning approach *RRT-Connect.* The planner grows two rapidly-exploring random trees (RRTs) in the configuration space and employs a simple greedy heuristic that aggressively attempts to establish a connection between the trees. This heuristic is demonstrated to be well-suited to generating motions for manipulation tasks commonly faced by human characters. For increased naturalness, we also present an approximate technique for coordinating the head and eye movements in conjunction with the arm motions.

For testing and evaluation purposes, we have designed an experimental implementation of the navigation and manipulation motion generation strategies and integrated them into an interactive application involving human characters. Though many improvements can be made, the generated animation looks fairly realistic.

Discussion

Nearly all known motion generation techniques developed in the graphics and robotics literature can be unified under a common framework by viewing the motion synthesis problem as one whose solution involves two fundamental tools: *model-building* and *search*. Specifically, solving a motion synthesis problem almost always involves *constructing a suitable model* and *searching an appropriate space* of possibilities. At present, no single motion synthesis strategy can adequately handle the variety of tasks that can arise. Instead, different tools and techniques can be used in combination and tailored to meet the requirements of specific tasks. We believe that assembling a library of algorithms and motion data libraries under the unifying framework of an animated agent will help to facilitate the development of autonomous characters with realistic motions and behaviors.

In the future, interactive virtual environments will be populated by lifelike autonomous characters. For offline animation applications, smarter software will alleviate the heavy burden of key-framing motions for complex characters. Next generation cinematographers will direct virtual humans in virtual scenes in much the same way as they direct real human actors. Ultimately, the dream of *virtual reality* may be fully realized, in which artificial scenes can be created that are indistinguishable from real life. In a sense, virtual reality defines a kind of "Turing Test" for animated agents (i.e. whether or not a user can distinguish animated agents from living people).

Let us consider for a moment some of the software technology and components necessary for realizing truly lifelike characters. Clearly, visual realism will involve both sophisticated graphical models as well as motions. For human characters, *expressive facial animation* using realistic skin models, and flexible models of *hair* and *clothing* that move in response to body motions will likely be important. In addition to having a realistic appearance, lifelike human characters will move fluidly and naturally, and exhibit believable behavior.

Besides the characters themselves, the virtual world they inhabit will also be lifelike. Since the real world is driven by the laws of physics, physically-based models will form an integral part of any attempt to realistically simulate humans in the real world. Even now, increases in computing power have allowed fairly sophisticated dynamic simulations to be calculated at interactive rates. Assuming that Moore's Law holds true, we can expect that future animation software will embody the laws of physics, and facilitate complex dynamic simulations. Given these advances, it is therefore natural to ask whether the fundamental problem of task-level motion control for human figures should be reformulated. Instead of trying to find a set of joint functions that solve a particular task, we should find a set of *controls* (i.e. muscle activation patterns) that solve the task. Ultimately, this could lead to a new representation of motions: *control functions* (forces and torques), instead of purely kinematic functions of joint variables.

For true believability, an autonomous character should exhibit realistic behavior. This implies that the character is capable of emulating some of the higher-level cognitive functions of real humans. This includes logical deduction and reasoning, adapting to changes in the environment, and learning from mistakes. Thus, an artificial life approach to animation that integrates perception, machine learning, and knowledge representation will likely be needed.

As mentioned in Chapter 2, the goal of building believable autonomous animated agents (virtual robots) shares much in common with efforts to build intelligent physical robots. However, a virtual robot has numerous advantages over a physical robot. While a physical robot must contend with the problems of uncertainty and errors in sensing and control, a virtual robot enjoys "perfect" control and sensing. This means that it should be *easier* to design an animated agent that behaves intelligently, than a physical agent that does so. In some sense, creating an intelligent autonomous animated agent can be viewed as a necessary step along the way towards creating an intelligent autonomous physical robot. If we cannot create a robot that behaves intelligently in simulation, it is unlikely that we will be able to create a robot that behaves intelligently in the real world. Thus, the "grand vision" for animated agents encompasses many of the long-term goals for artificial intelligence.

At the present time however, we have a more modest objective: the animation of lifelike autonomous characters for video games, animated films, and interactive simulations. Since these applications do not always require perfectly realistic motions, this more modest goal should conceivably be easier to achieve relative to the "grand vision". In any case, we believe that applications involving autonomous characters will continue to increase both in number and in sophistication as software and hardware technology improves. Clearly, many challenging research issues must be addressed before the dream of fully autonomous animated agents can be realized.

References

[AG85] W.W. Armstrong and M.W. Green. The dynamics of articulated rigid bodies for the purposes of animation. *The Visual Computer*, 1:231–240, 1985.

[AH84] C.G. Atkeson and J.M. Hollerbach. Kinematic features of unrestrained arm movements. MIT AI Memo 790, 1984.

[AMN$^+$] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching. To appear in the Journal of the ACM.

[AN99] Y. Aydin and M. Nakajima. Realistic articulated character positioning and balance control in interactive environments. In *In Proceedings of CA '99 : IEEE International Conference on Computer Animation.*, pages 160–168, Geneva, Switzerland, May 1999.

[Ark92] R. C. Arkin. Cooperation without communication: Multiagent schema based robot navigation. *Journal of Robotic Systems*, pages 351–364, 1992.

[Ark98] R. C. Arkin. *Behavior-based Robotics.* MIT Press, 1998.

[ASL90] R. Alami, T. Simeon, and J.P. Laumond. A geometrical approach to planning manipulation tasks: The case of discrete placements and grasps. In H. Miura and S. Arimoto, editors, *Robotics Research 5*, pages 453–459. MIT Press, 1990.

[AW89] S. Athenes and A.M. Wing. Knowlege-directed coordination in reaching for objects in the environment. In S.A. Wallace, editor, *Perspectives on the coordination of movement*, pages 285–301. Elsevier Science Publishers, 1989.

[AW96] N. Amato and Y. Wu. A randomized roadmap method for path and manipuation planning. In *Proc. of IEEE Int. Conf. Robotics and Automation*, pages 113–120, Minneapolis, MN, 1996.

[Bad97] N. Badler. Real-time virtual humans. *Pacific Graphics*, 1997.

[Ban98] Srikanth Bandi. *Discrete Object Space Methods for Computer Animation.* PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998.

[Bar89] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Proc. SIGGRAPH '89*, pages 223–231, 1989.

[BB98] R. Bindiganavale and N. Badler. Motion abstraction and mapping with spatial constraints. In *Proc. of CAPTECH '98 : Workshop on Modelling and Motion Capture Techniques for Virtual Environments.* Springer-Verlag, November 1998.

[BC89] A. Bruderlin and T. Calvert. Goal-directed dynamic animation of human walking. In *Proc. SIGGRAPH '89*, pages 233–242, 1989.

[BCC99] N. Badler, D. Chi, and S. Chopra. Virtual human animation based on movement observation and cognitive behavior models. In *In Proceedings of CA '99 : IEEE International Conference on Computer Animation.*, pages 128–137, Geneva, Switzerland, May 1999.

[BDG85] J. Bobrow, S. Dubowsky, and J. Gibson. Time-optimal control of robotic manipulators. *Int. Journal of Robotics Research*, 4(3), 1985.

[Beg94] D. Begault. *3-D Sound for Virtual Reality and Multimedia.* Academic Press, Boston, MA, 1994.

[Ber67] N. Bernstein. *The coordination and regulation of movements.* Pergamon, Oxford, 1967.

[BFF86] M.B. Berkinblit, A.G. Feldman, and O.I. Fukson. Adaptability of innate motor patterns and motor control mechanisms. *Behavioral and Brain Sciences*, 9:585–638, 1986.

[BG95] B. M. Blumberg and T. A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In Robert Cook, editor, *Proc. SIGGRAPH '95*, Annual Conference Series, pages 47–54, August 1995.

[BGW$^+$94] R. Bindiganavale, J. Granieri, S. Wei, X. Zhao, and N. Badler. Posture interpolation with collision avoidance. In *Proc. of Computer Animation '94*, pages 13–20, 1994.

[BH95] D. C. Brogan and J. K. Hodgins. Group behaviors with significant dynamics. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1995.

[BKL$^+$97] J. Barraquand, L.E. Kavraki, J.C. Latombe, T.Y. Li, R. Motwani, and P. Raghavan. A random sampling scheme for path planning. *Int. J. of Robotics Research*, 16(6):759–774, 1997.

[BL90] J. Barraquand and J.-C. Latombe. Robot motion planning: A distributed representation approach. *Int. J. of Robotics Research*, 10(6):628–649, December 1990.

[BL93] J. Barraquand and J.C. Latombe. Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. *Algorithmica*, 10:121–155, 1993.

[BLR94] J. Bates, A. B. Loyall, and W. S. Reilly. An architecture for action, emotion, and social behavior. In *Artificial Social Systems : Proc of 4th European Wkshp on Modeling Autonomous Agents in a Multi-Agent World*. Springer-Verlag, 1994.

[Blu96] B. M. Blumberg. *Old Tricks, New Dogs : Ethology and Interactive Creatures.* PhD thesis, MIT Media Laboratory, Boston, MA, 1996.

[BMT97] R. Boulic, R. Mas, and D. Thalmann. Complex character positioning based on a compatible flow model of multiple supports. *IEEE Transactions on Visualization and Computer Graphics*, pages 245–261, July-Sept 1997.

[BOvdS99] V. Boor, M. Overmars, and A.F. van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In *Proc. of IEEE Int. Conf. Robotics and Automation*, Detroit, MI, 1999.

[BPW92] N. Badler, C. Phillips, and B. Webber. *Simulating Humans: Computer Graphics, Animation, and Control.* Oxford University Press, 1992.

[Bro85] R. A. Brooks. A layered intelligent control system for a mobile robot. In *Robotics Research The Third International Symposium*, pages 365–372. MIT Press, Cambridge, MA, 1985.

[BRRP97] B. Bodenheimer, C. Rose, S. Rosenthal, and J. Pella. The process of motion capture: Dealing with the data. In *Proc. of the Eurographics Workshop on Computer Animation and Simulation*, 1997.

[BT97] S. Bandi and D. Thalmann. A configuration space approach for efficient animation of human figures. In *Proc. of IEEE Nonrigid and articulated motion workshop*, Puerto Rico, 1997.

[BT98] S. Bandi and D. Thalmann. Space discretization for efficient human navigation. In *Proc. of EUROGRAPHICS annual conference*, Lisbon, Portugal, 1998.

[BTMT90] R. Boulic, D. Thalmann, and N. Magnenat-Thalmann. A global human walking model with real time kinematic personification. *The Visual Computer*, 6(6), December 1990.

[BW95] A. Bruderlin and L. Williams. Motion signal processing. In Robert Cook, editor, *Proc. SIGGRAPH '95*, Annual Conference Series, pages 97–104. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[BW98] D. Baraff and A. Witkin. Large steps in cloth simulation. In *Proc. SIGGRAPH '98*, pages 43–54, 1998.

[BWB$^+$95] N. Badler, B. Webber, W. Becket, C. Geib, M. Moore, C. Pelachaud, B. Reich, and M. Stone. Planning and parallel transition networks: Animation's new frontiers. In *Proc. of Pacific Graphics '95*, pages 101–117. World Scientific Publishing, 1995.

[CA91] D. Chaffin and G. Andersson. *Occupational Biomechanics.* John Wiley and Sons, Inc., second edition, 1991.

[Can88] J.F. Canny. *The Complexity of Robot Motion Planning.* MIT Press, Cambridge, MA, 1988.

[CB92] W. Ching and N. Badler. Fast motion planning for anthropometric figures with many degrees of freedom. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 2340–2345, 1992.

[CDLS99] R.G. Cowell, A.P. Dawid, S.L. Lauritzen, and D.J. Spiegelhalter. *Probabilistic Networks and Expert Systems.* Springer-Verlag, 1999.

[CG93] D. Chalou and M. Gini. Parallel robot motion planning. In *Proc. of IEEE Int. Conf. Robotics and Automation*, pages 24–51, Atlanta, GA, 1993.

[CH99] S.-K. Chung and J.K. Hahn. Animation of human walking in virutal environments. In *In Proceedings of CA '99 : IEEE International Conference on Computer Animation.*, pages 4–15, Geneva, Switzerland, May 1999.

[Che96] S. Chenney. Sensing for autonomous agents in virtual environments. http://http.cs.berkeley.edu/~schenney/autonomous/sensing.html, 1996.

[Com99] Joao L.D. Comba. *Kinetic Vertical Decomposition Trees.* PhD thesis, Stanford University, Dept. of Computer Science, 1999.

[Cra89] John J. Craig. *Introduction to Robotics : Mechanics and Control.* Addison-Wesley, 1989.

[CS92] Y. Chrysanthou and M. Slater. Computing dynamic changes to bsp trees. In *Computer Graphics Forum (EUROGRAPHICS '92 Proceedings)*, volume 11, pages 321–332, September 1992.

[DLB96] B. Douville, L. Levison, and N. Badler. Task level object grasping for simulated agents. *Presence*, 5(4):416–430, 1996.

[dPF93] M. Van de Panne and E. Fiume. Sensor-actuator networks. In *Proc. SIGGRAPH '93*, pages 225–234, 1993.

[dPFV90] M. Van de Panne, E. Fiume, and Z. Vranesic. Reusable motion synthesis using state-space controllers. In *Proc. SIGGRAPH '90*, pages 225–234, 1990.

[DX95] B. Donald and P. Xavier. Provably good approximation algorithms for optimal kinodynamic planning for cartesian robots and open chain manipulators. *Algorithmica*, 14(6):480–530, 1995.

[DXCR93] B. Donald, P. Xavier, J. Canny, and J. Reif. Kinodynamic motion planning. *Journal of the ACM*, 40(5):1048–1066, November 1993.

[FDHF90] J. Foley, A. Van Dam, J. Hughes, and S. Feiner. *Computer Graphics: Principles and Practice.* Addison Wesley, Reading, MA, 1990.

[Fer96] P. Ferbach. A method of progressive constraints for nonholonomic motion planning. In *Proc. of the IEEE International Conf. on Robotics and Automation (ICRA'96)*, pages 1637–1642, Minneapolis, MN, April 1996.

[FG97] S. Franklin and A. Graesser. Is it an agent , or just a program? In *Intelligent Agents III (LNAI Volume1193)*, pages 21–36, Berlin, Germany, 1997. Springer Verlag.

[FH85] T. Flash and N. Hogan. The coordination of arm movements: an experimentally confirmed mathematical model. *The Journal of Neuroscience*, 5(7):1688–1703, 1985.

[FKN80] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. In *Proc. of SIGGRAPH '80*, volume 14. ACM SIGGRAPH, July 1980.

[FT87] B. Faverjon and P. Tournassoud. A local based approach for path planning of manipulators with a high number of degrees of freedom. In *Proc of IEEE Int. Conf. Robotics and Automation*, pages 1152–1159, 1987.

[Fun98] John Funge. *Making the Behave: Cognitive models for computer animation.* PhD thesis, University of Toronto, 1998.

[GBR$^+$95] J. P. Granieri, W. Becket, B. D. Reich, J. Crabtree, and N. L. Badler. Behavioral control for real-time simulated human agents. In Pat Hanrahan and Jim

Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 173–180. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[Gle98] M. Gleicher. Retargetting motion to new characters. In *SIGGRAPH '98 Proc.*, 1998.

[GLM94] C. Geib, L. Levison, and M.B. Moore. Sodajack: an architecture for agents that search for and manipulate objects. Technical report, Univ. of Pennsylvania, 1994. IRCS Report 94-31.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *SIGGRAPH '96 Proc.*, 1996.

[GM85] M. Girard and A Maciejewski. Computational modeling for the computer animation of legged figures. In *Proc. of SIGGRAPH '85*, 1985.

[GRBC96] B. Guenter, C. Rose, B. Bodenheimer, and M. F. Cohen. Efficient generation of motion transitions using spacetime constraints. In *SIGGRAPH '96 Proc.*, 1996.

[GT95] R. Grzeszczuk and D. Terzopoulos. Automated learning of muscle-actuated locomotion through control abstraction. In *SIGGRAPH '95 Proc.*, pages 63–70, 1995.

[GTH98] R. Grzeszczuk, D. Terzopoulos, and G. Hinton. Neuroanimator: Fast neural network emulation and control of physics-based models. In *SIGGRAPH '98 Proc.*, 1998.

[HA92] Y.K. Hwang and N. Ahuja. Gross motion planning: A survey. *ACM Computing Surveys*, 24(3):219–291, 1992.

[HBTT95] Z. Huang, R. Boulic, N. Magnenat Thalmann, and D. Thalmann. A multi-sensor approach for grasping and 3d interaction. In *Proc. Computer Graphics International '95*, Leeds, 1995.

[HC98] D. Hsu and M. Cohen. Task-level motion control for human figure animation. Unpublished Manuscript, 1998.

[HKL$^+$98] D. Hsu, L.E. Kavraki, J.C. Latombe, R. Motwani, and S. Sorkin. On finding narrow passages with probabilistic roadmap planners. In P.K. Agarwal, L.E. Kavraki, and M.T. Mason, editors, *Robotics: The Algorithmic Perspective, Workshop on Algorithmic Foundations of Robotics*, pages 141–153. A. K. Peters, Natick, MA, 1998.

[HKR97] M.R. Henzinger, P. Klein, and S. Rao. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci*, 55:3–23, 1997.

[HLM97] D. Hsu, J.-C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. *Int. J. of Computational Geometry and Applications*, 9(4-5):495–512, 1997.

[Hod96] J. K. Hodgins. Three-dimensional human running. In *Proc. IEEE Int. Conf. on Robotics and Automation*, 1996.

[HP97] J.K. Hodgins and N.S. Pollard. Adapting simulated behaviors for new characters. In *Proc. SIGGRAPH '97*, 1997.

[HST94] T. Horsch, F. Schwarz, and H. Tolle. Motion planning for many degrees of freedom : Random reflections at c-space obstacles. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA'94)*, pages 3318–3323, San Diego, CA, April 1994.

[HW98] J.K. Hodgins and W.L. Wooten. Animating human athletes. In *Robotics Research: The Eighth International Symposium*, pages 356–367. Springer-Verlag, 1998.

[IC88] P. M. Isaacs and M. F. Cohen. Mixed methods for complex kinematic constraints in dynamic figure animation. *The Visual Computer*, 4(6):296–305, 1988.

[Isi95] A. Isidori. *Nonlinear Control Systems.* Springer-Verlag, New York, NY, 3rd edition, 1995.

[JBN94] M. R. Jung, N. Badler, and T. Noma. Animated human agents with motion planning capability for 3D-space postural goals. *The Journal of Visualization and Computer Animation*, 5(4):225–246, October 1994.

[Jen96] F. Jensen. *An Introduction to Bayesian Networks*. Springer Verlag, 1996.

[Jon97] F.M. Jonsson. An optimal pathfinder for vehicles in real-world digital terrain maps. Master's thesis, The Royal Institute of Science, Stockholm, Sweden, 1997.

[JSW98] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[JvdGG89] H.A.H. Jongen, D. van der Gon, and J.J. Gielen. Activation of human arm muscles furing flexion/extension and supination/pronation tasks. *Biological Cybernetics*, 61:1–9, 1989.

[Kav95] L.E. Kavraki. *Random networks in configuraiton space for fast path planning*. PhD thesis, Dept. of Computer Science, Stanford University, January 1995.

[KB82] J.U. Korein and N.I. Badler. Techniques for generating the goal-directed motion of articulated structures. *IEEE Computer Graphics and Applications*, pages 71–81, 1982.

[KB96] H. Ko and N. Badler. Animating human locomotion in real-time using inverse dynamics, balance and comfort control. *IEEE Computer Graphics and Applications*, 16(2):50–59, March 1996.

[Kha86] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. of Robotics Research*, 5:90–98, 1986.

[Kha95] O. Khatib. Inertial properties in robotic manipulation: An object-level framework. *International Journal of Robotics Research*, 14(1):19–36, Feb 1995.

[KK88] L. Kanal and V. Kumar, editors. *Search in Artificial Intelligence*. Springer-Verlag, New York, 1988.

[KKKL94a] Y. Koga, K. Kondo, J. Kuffner, and J.-C. Latombe. Planning motions with intentions. In *Proc. SIGGRAPH '94*, pages 395–408, 1994.

[KKKL94b] J. Kuffner, K. Kondo, Y. Koga, and J.C. Latombe. Endgame. Video transactions of the Electronic Theater of SIGGRAPH '94, July 1994.

[KL93] L.E. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration space for fast path planning. Technical report, Dept. of Computer Science, Stanford University, September 1993.

[KL99] J.J. Kuffner and J.C. Latombe. Fast synthetic vision, memory, and learning models for virtual humans. In *In Proceedings of CA '99 : IEEE International Conference on Computer Animation.*, pages 118–127, Geneva, Switzerland, May 1999.

[KMB95] E. Kokkevis, D. Metaxas, and N. I. Badler. Autonomous animation and control of four-legged animals. In Wayne A. Davis and Przemyslaw Prusinkiewicz, editors, *Graphics Interface '95*, pages 10–17. Canadian Human-Computer Communications Society, May 1995. ISBN 0-9695338-4-5.

[Kog94] Y. Koga. *On Multi-Arm Manipulation Planning.* PhD thesis, Stanford University, Stanford, CA, 1994.

[Kon94] K. Kondo. Inverse kinematics of a human arm. Technical Report STAN-CS-TR-94-1508, Dept. of Computer Science, Stanford University, Stanford, CA, 1994.

[KP97] D. Koller and A. Pfeffer. Object-oriented bayesian networks. In *Proc. of 13th Annual Conference on Uncertainty in AI (UAI)*, Providence, Rhode Island, August 1997.

[KŠLO96] L. Kavraki, P. Švestka, J. C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration space. *IEEE Trans. on Robotics and Automation*, 12(4):566–580, 1996.

[KT99] M. Kallmann and D. Thalmann. A behavioral interface to simulate agent-object interactions in real time. In *In Proceedings of CA '99 : IEEE International Conference on Computer Animation.*, pages 138–146, Geneva, Switzerland, May 1999.

[Kuf98] J. J. Kuffner, Jr. Goal-directed navigation for animated characters using real-time path planning and control. In *Proc. of CAPTECH '98 : Workshop on Modelling and Motion Capture Techniques for Virtual Environments.* Springer-Verlag, November 1998.

[Lat91] J. C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, MA, 1991.

[Lat93] Mark L. Latash. *Control of Human Movement.* Human Kinetics Publishers, Champaign, IL, 1993.

[LaV99] S.M. LaValle. Rapidly-exploring random trees : A new tool for path planning. Preliminary manuscript available at http://janowiec.cs.iastate.edu/~lavalle/, 1999.

[LGC94] Z. Liu, S. J. Gortler, and F. C. Cohen. Hierachical spacetime control. In *Proc. SIGGRAPH '94*, pages 35–42, 1994.

[LK99] S.M. Lavalle and J.J Kuffner. Randomized kinodynamic planning. In *Proc. of the IEEE International Conf. on Robotics and Automation (ICRA'99)*, Detroit, MI, May 1999.

[LM99] J. Lo and D. Metaxas. Recursive dynamics and optimal control techniques for human motion planning. In *In Proceedings of CA '99 : IEEE International Conference on Computer Animation.*, pages 220–234, Geneva, Switzerland, May 1999.

[LP83] T. Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, February 1983.

[LPW79] T. Lozano-Perez and M.A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, October 1979.

[LRDG90] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. In *Proc. SIGGRAPH '90*, 1990.

[LWZB90] P. Lee, S. Wei, J. Zhao, and N. Badler. Strength guided motion. In *Proc SIGGRAPH '90*, volume 24, pages 253–262, 1990.

[Lyn93] K.M. Lynch. Planning pushing paths. In *In Proc. of JSME Int. Conf. Advanced Mechatronics*, pages 451–456, Tokyo, Japan, 1993.

[Mai96] R. Maiocchi. *3-D character animation using motion capture*, chapter 1, pages 10–39. Prentice-Hall, London, 1996.

[MBD73] P. Morasso, E. Bizzi, and J. Dichgans. Adjustment of saccade characteristics during eye head movements. *Exper. Brain Research*, 16:548–562, 1973.

[Mir96] B. Mirtich. *Impulse-Based Dynamic Simulation of Rigid Body Systems.* PhD thesis, University of California, Berkeley, CA, 1996.

[Mit97] Tom Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[Mit98] J.S.B. Mitchell. *Handbook of Computational Geometry*, chapter Geometric Shortest Paths and Network Optimization. Elsevier Science, 1998.

[MT86] P. Morasso and V. Tagliasco, editors. *Human Movement Understanding: from computational geometry to artificial intelligence.* North-Holland, 1986.

[MTBP95] P. Maes, D. Trevor, B. Blumberg, and A. Pentland. The ALIVE system full-body interaction with autonomous agents. In *Computer Animation '95*, April 1995.

[MZ90] M. McKenna and D. Zeltzer. Dynamic simulation of autonomous legged locomotion. In *Proc. SIGGRAPH '90*, pages 29–38, 1990.

[Nay92] B. Naylor. Partitioning tree image representation and generation from 3d geometric models. In *Proceedings of Graphics Interface '92*, pages 201–212, May 1992.

[Nil80] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

[NM93] J. T. Ngo and J. Marks. Spacetime constraints revisited. In *Proc. SIGGRAPH '93*, pages 343–350, 1993.

[NRTT95] H. Noser, O. Renault, D. Thalmann, and N. Magnenat Thalmann. Navigation for digital actors based on synthetic vision, memory and learning. *Comput. Graphics*, 19:7–19, 1995.

[NT95] H. Noser and D. Thalmann. Synthetic vision and audition for digital actors. In *Proc. Eurographics '95*, 1995.

[O'D87] C. O'Dunlaing. Motion planning with inertial constraints. *Algorithmica*, 2(4):431–475, 1987.

[OK94] C.W.A.M. Van Overveld and H. Ko. Small steps for mankind: Towards a kinematically-driven dynamic simulation of curved path walking. *The Journal of Visualization and Computer Animation*, 5:143–165, 1994.

[Ove92] M. Overmars. A random approach to motion planning. Technical report, Dept. Computer Science, Utrect University, Utrect, The Netherlands, October 1992.

[Per95] K. Perlin. Real time responsive animation with personality. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):5–15, March 1995. ISSN 1077-2626.

[PG96] K. Perlin and A. Goldberg. IMPROV: A system for scripting interactive actors in virtual worlds. In Holly Rushmeier, editor, *Proc. SIGGRAPH '96*, Annual Conference Series, pages 205–216. ACM SIGGRAPH, Addison Wesley, 1996.

[PT89] J. Pertin-Troccaz. Grasping: A state of the art. In O. Khatib, J.J. Craig, and T. Lozano-Perez, editors, *Robotics Review 1*, pages 71–98. MIT Press, Cambridge, MA, 1989.

[PW99] Z. Popovic and A. Witkin. Physically based motion transformation. In *Proc. SIGGRAPH '99*, Annual Conference Series. ACM SIGGRAPH, 1999.

[PY90] M. Paterson and F. Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete and Computational Geometry*, 5(5):485–503, 1990.

[Qui94] S. Quinlan. *Real-time Modification of Collision-free Paths.* PhD thesis, Stanford University, Stanford, CA, 1994.

[RBKB94] B. Reich, N. Badler, H. Ko, and W. Becket. Terrain reasoning for human locomotion. In *Computer Animation '94*, pages 76–82, Geneva, Switz., 1994.

[Rei79] J. H. Reif. Complexity of the mover's problem and generalizations. In *Proc. 20th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 421–427, 1979.

[Rey87] C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[Rey99] C. W. Reynolds. Steering behaviors for autonomous characters. In *Proc. of 1999 Game Developers Conference*, 1999. http://hmt.com/cwr/steer/.

[RH91] M. Raibert and J. Hodgins. Animation of dynamic legged locomotion. In *Proc. SIGGRAPH '91*, pages 349–358, 1991.

[RHC86] G. Ridsdale, S. Hewitt, and T. W. Calvert. The interactive specification of human animation. In M. Green, editor, *Proc. of Graphics Interface '86*, pages 121–130, May 1986.

[RKK97] D. C. Ruspini, K. Kolarov, and O. Khatib. The haptic display of complex graphical environments. In *Proc. SIGGRAPH '97*, pages 345–352. ACM SIGGRAPH, August 1997. ISBN 0-89791-896-7.

[RN95] S. Russel and P. Norvig. *Artificial Intelligence: a Modern Approach.* Prentice-Hall, 1995.

[Rob64] D.A. Robinson. The mechanics of human saccadic eye movements. *Journal of Physiology*, 174:245–264, 1964.

[RTT90] O. Renault, N. M. Thalmann, and D. Thalmann. A vision-based approach to behavioral animation. *Visualization and Computer Animation*, 1:18–21, 1990.

[S$^+$91] A.M. Smith et al. Group report: What do studies of specific motor acts such as reaching and grasping tell us about the general principles of goal-directed motor behavior? In D.R. Humphrey and H.J. Freund, editors, *Motor Control: Concepts and Issues*, pages 357–381. John Wiley and Sons, New York, 1991.

[SD91] Z. Shiller and S. Dubowsky. On computing time-optimal motions of robotic manipulators in the presence of obstacles. *IEEE Trans. on Robotics and Automation*, 7(7), December 1991.

[SF89] J.F. Soechting and M. Flanders. Sensorimotor representations for pointing to targets in three dimensional space. *Journal of Neurophysiology*, 62(2):582–594, 1989.

[Sho85] K. Shoemake. Animating rotation with quaternion curves. In *Proc. of SIGGRAPH '85*, pages 245–254, 1985.

[Sim94] K. Sims. Evolving virtual creatures. In *Proc. SIGGRAPH '94*, pages 15–22, 1994.

[Soe84] J. F. Soechting. Effect of target size on spatial and temporal characteristics of a pointing movement in man. *Exp. Brain Research*, 54:121–132, 1984.

[SOH99] R. W. Sumner, J.F. O'Brien, and J. K. Hodgins. Animating sand, mud, and snow. *Computer Graphics Forum*, 18(1), January 1999.

[SS83] J. T. Schwartz and M. Sharir. On the 'piano movers' problem: Ii. general techniques for computing topological properties of real algebraic manifolds. *Advances in applied Mathematics*, 4:298–351, 1983.

[Ste95] A Stentz. Optimal and efficient path planning for unknown and dynamic environments. *Int. Journal of Robotics and Automation*, 10(3), 1995.

[Str91] S. Strassmann. *Desktop Theater: Automating the Generation of Expressive Animation.* PhD thesis, M.I.T. Media Arts and Sciences Program, Toronto, Canada, 1991.

[Str93] P. S. Strauss. Iris inventor, a 3d graphics toolkit. In *ACM SIGPLAN Notices (OOPSLA '93 Proc.)*, volume 28, pages 192–200, October 1993.

[Str94] S. Strassmann. Semi-autonomous animated actors. In *Proc. AAAI '94*, pages 128–134, 1994.

[Sve93] P. Svestka. A probabilistic approcach to motion planning for car-like robots. Technical report, Dept. Computer Science, Utrect Univ., Utrect, The Netherlands, April 1993.

[TDT96] N. M. Thalmann and *eds.* D. Thalmann. *Interactive Computer Animation.* Prentice Hall Europe, London, 1996.

[Tel92] Seth Teller. *Visibility Computation in Densely Occluded Polyhedral Environments.* PhD thesis, UC Berkeley CS Department, 1992. TR 92/708.

[TF88] D. Terzopoulos and K. Fleischer. Deformable models. *The Visual Computer*, 4:306–331, 1988.

[TJ95] F. Thomas and O. Johnston. *The Illusion of Life: Disney Animation*. Hyperion, 1995.

[TNH96] D. Thalmann, H. Noser, and Z. Huang. *Interactive Animation*, chapter 11, pages 263–291. Springer-Verlag, 1996.

[TR95] D. Terzopoulos and T. Rabie. Animat vision: Active vision in artificial animals. In *Proc. Fifth Int. Conf. on Computer Vision (ICCV'95)*, pages 801–808, Cambridge, MA, June 1995.

[TT90] N. M. Thalmann and D. Thalmann. *Computer Animation*. Springer-Verlag, New York, second revised edition, 1990.

[TT94] X. Tu and D. Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *Proc. SIGGRAPH '94*, pages 43–50. ACM SIGGRAPH, July 1994. ISBN 0-89791-667-0.

[Tu96] X. Tu. *Artificial Animals for Computer Animation: Biomechanics, Locomotion, Perception, and Behavior*. PhD thesis, University of Toronto, Toronto, Canada, 1996.

[UAT95] M. Unuma, K. Anjyo, and R. Takeuchi. Fourier principles for emotion-based human figure animation. In *Proc. SIGGRAPH '95*, 1995.

[Udu77] S. Udupa. *Collision Detection and Avoidance in Computer Controlled Manipulators*. PhD thesis, Dept. of Electrical Engineering, California Institute of Technology, 1977.

[VCNT95] P. Volino, M. Courchesne, and N.Magnenet-Thalmann. Versitile and efficient techniques for simulating cloth and other deformable objects. In *Proc. SIGGRAPH '95*, pages 137–144, 1995.

[vdP97] M. van de Panne. From footprints to animation. *Computer Graphics Forum*, 16(4):211–223, October 1997.

[Wal89] Stephen A. Wallace, editor. *Perspectives on the coordination of movement.* Elsevier Science Publishers, Amsterdam, 1989.

[Wil88] G. Wilfong. Motion planning in the presence of movable obstacles. In *Proc. of 4th ACM Symp. Computational Geometry*, pages 279–288, 1988.

[WM88] A. Witkin and Kass M. Spacetime constraints. In *Proc. SIGGRAPH '88*, pages 159–168, 1988.

[WP95] A. Witkin and Z. Popovic. Motion warping. In *Proc. SIGGRAPH '95*, 1995.

[ZB94] J. Zhao and N. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics*, 13(4):313–336, October 1994.

[ZG89] F.E. Zajac and M.E. Gordon. Determining muscle's force and action in multi-articular movements. *Exer Sport Sci Rev*, 17:187–230, 1989.