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.

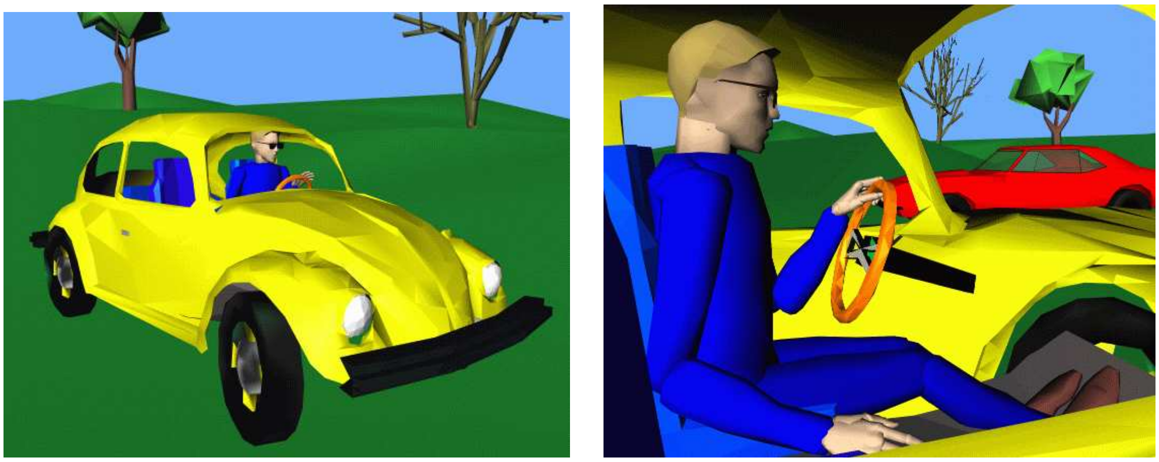

Figure 5.1: Manipulating the controls of a virtual car.

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 $\text{Forward}(\mathbf{q})$ 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

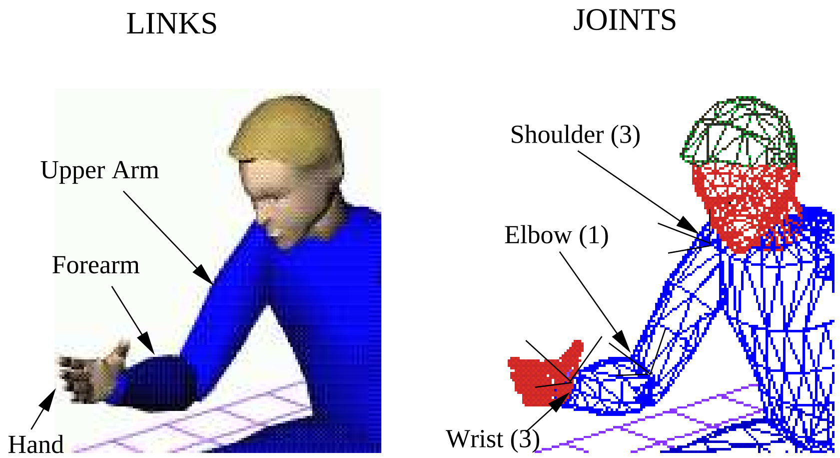

Figure 5.2: Simplified kinematic model of a human arm (7 DOF).

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).

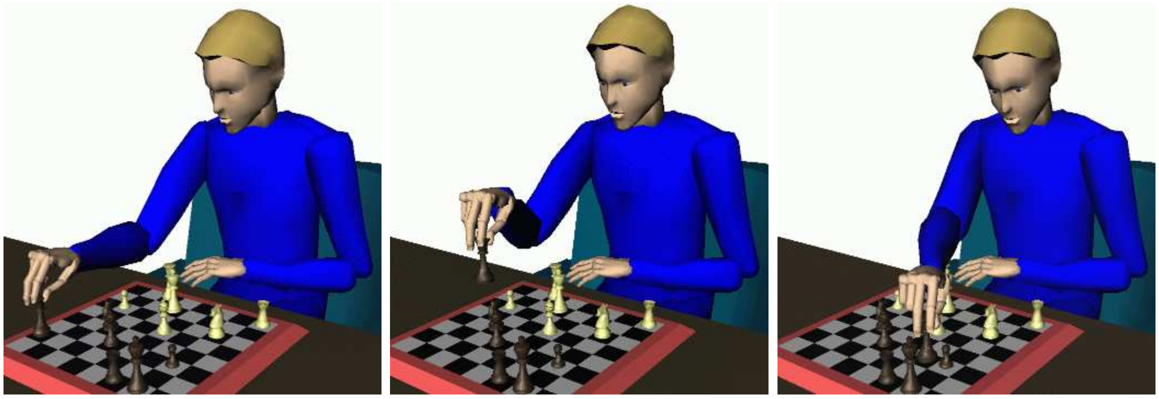

Figure 5.3: Path planning for a 7-DOF human arm

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 ← NearestNeighbor(T, q_target)
if ρ(q_near, q_target) < d_max then
  if Connect(q_near, q_target) then
    AddBranch(T, q_near, q)
    return CONNECTION
  end if
else
  q ← GenerateNewNode(q_target, q_near)
  if Connect(q_near, q) then
    AddBranch(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

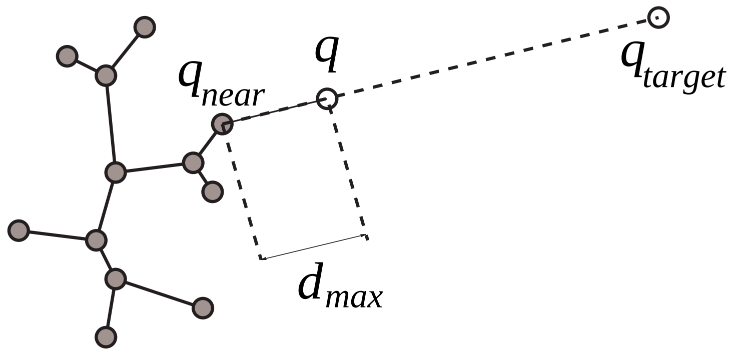

Figure 5.4: Growing a branch towards q_{target}.

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 */
T_init ← Initialize(q_init)
T_goal ← Initialize(q_goal)
q_target ← q_goal
T_active ← T_init
bias ← TRUE
/* main planning loop */
while time < T_max do
  if bias then
    result ← GrowTree(T_active, q_target)
    if result = CONNECTION then
      path ← BuildPath(T_init, T_goal, q_target)
      return path
    end if
    if result = FAILED then
      bias ← FALSE
    end if
  else
    q_target ← RandomConfig()
    result ← GrowTree(T_active, q_target)
    if result ≠ FAILED then
      q_target ← NodeAdded(T_active)
      bias ← TRUE
    end if
    /* switch currently active tree */
    if T_active = T_init then
      T_active ← T_goal
    else
      T_active ← T_init
    end if
  end if
end while
/* planner failed */
return NULL_PATH
```

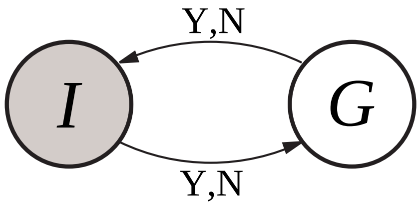

Figure 5.5: Basic Planner state diagram

- $\mathcal{T}_{active}$ 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

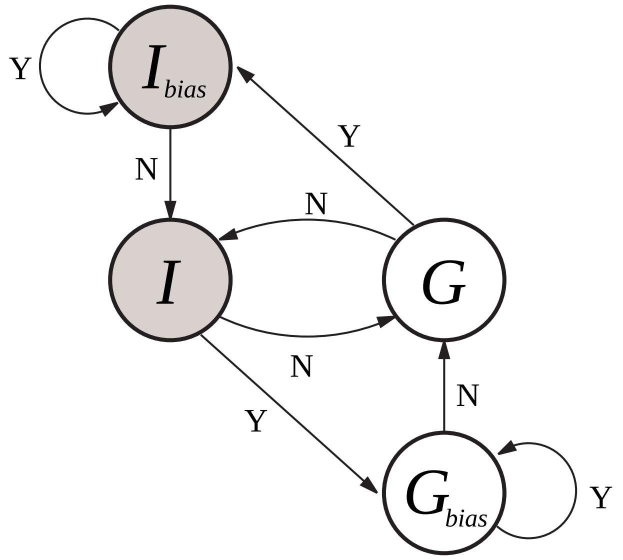

Figure 5.6: RRT-Connect Planner state diagram

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

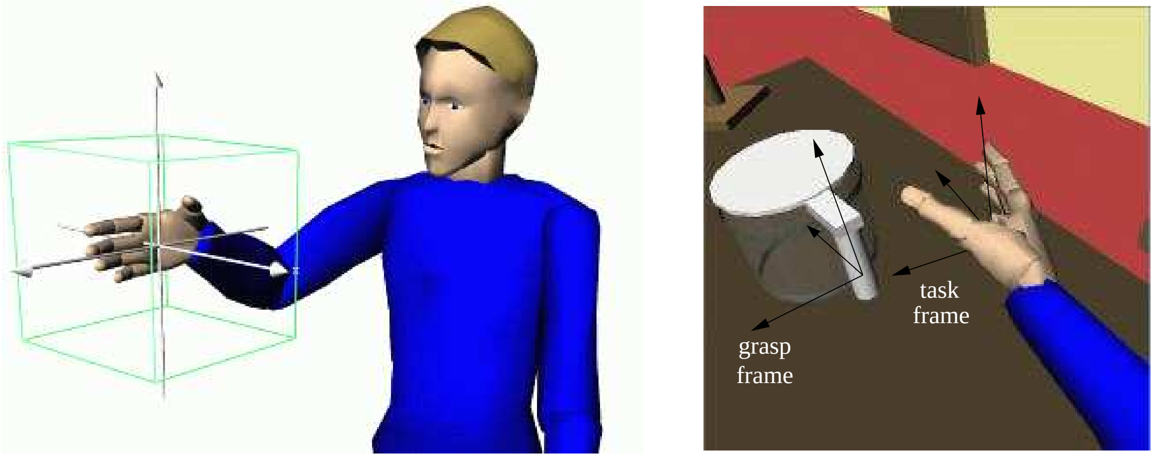

Figure 5.7: An example *task frame* (left). The *grasp frame* for a coffee pot (right).

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.

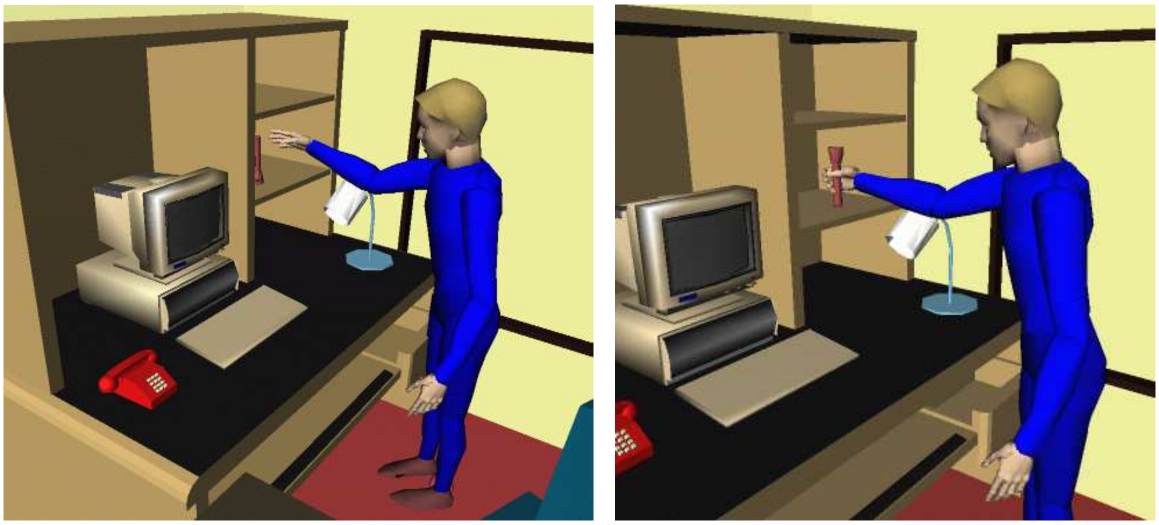

Figure 5.8: Reaching for a flashlight (left), and finalizing the grasp (right).

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.

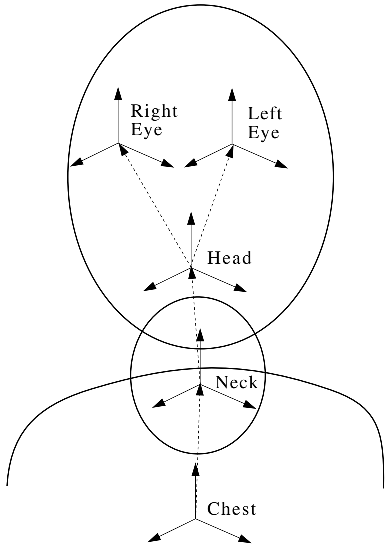

Figure 5.9: The eye-neck kinematic chain

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,

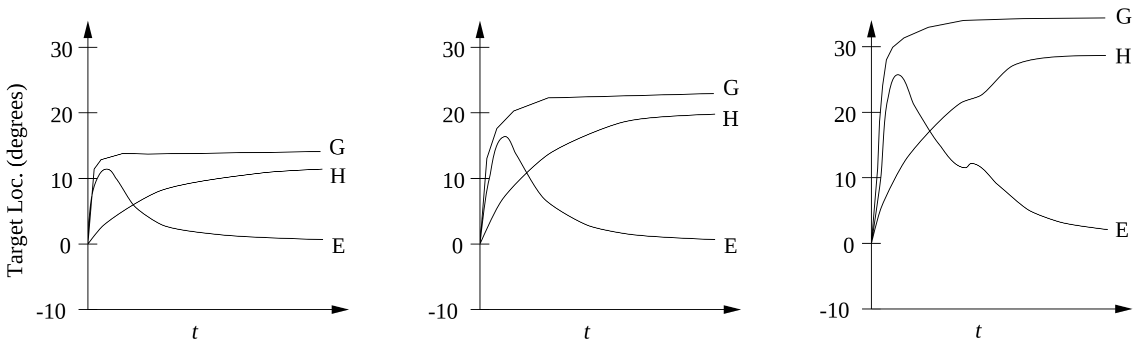

Figure 5.10: Eye and head coordinated movements for different relative target displacements. The curves represent eye rotations (E), head rotations (H), and cumulative gaze rotations (G = E + H). (Adapted from [Rob64])

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

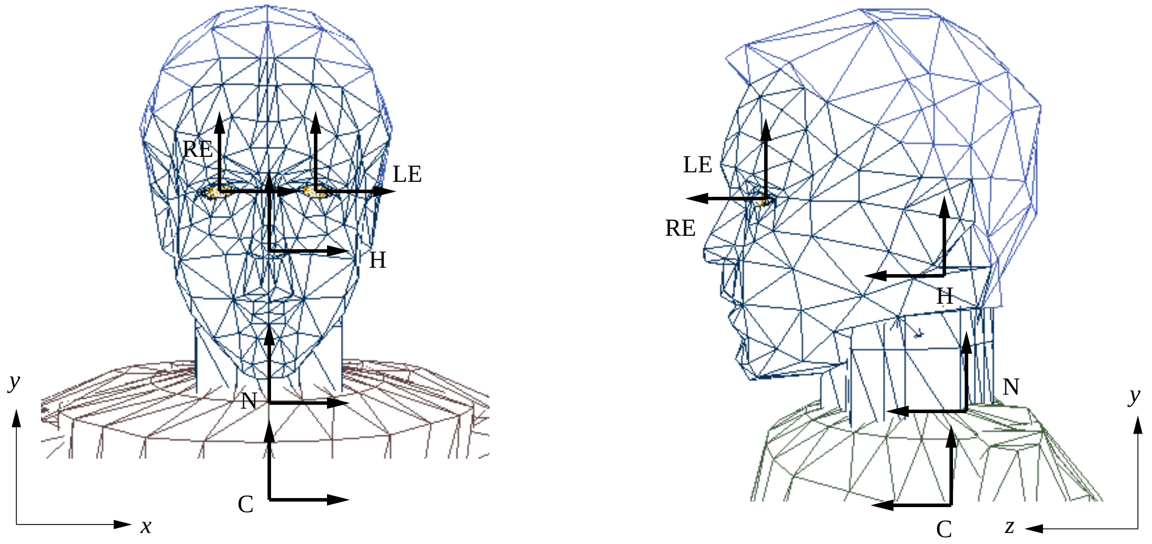

Figure 5.11: Frames of reference for the eye-neck chain.

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.

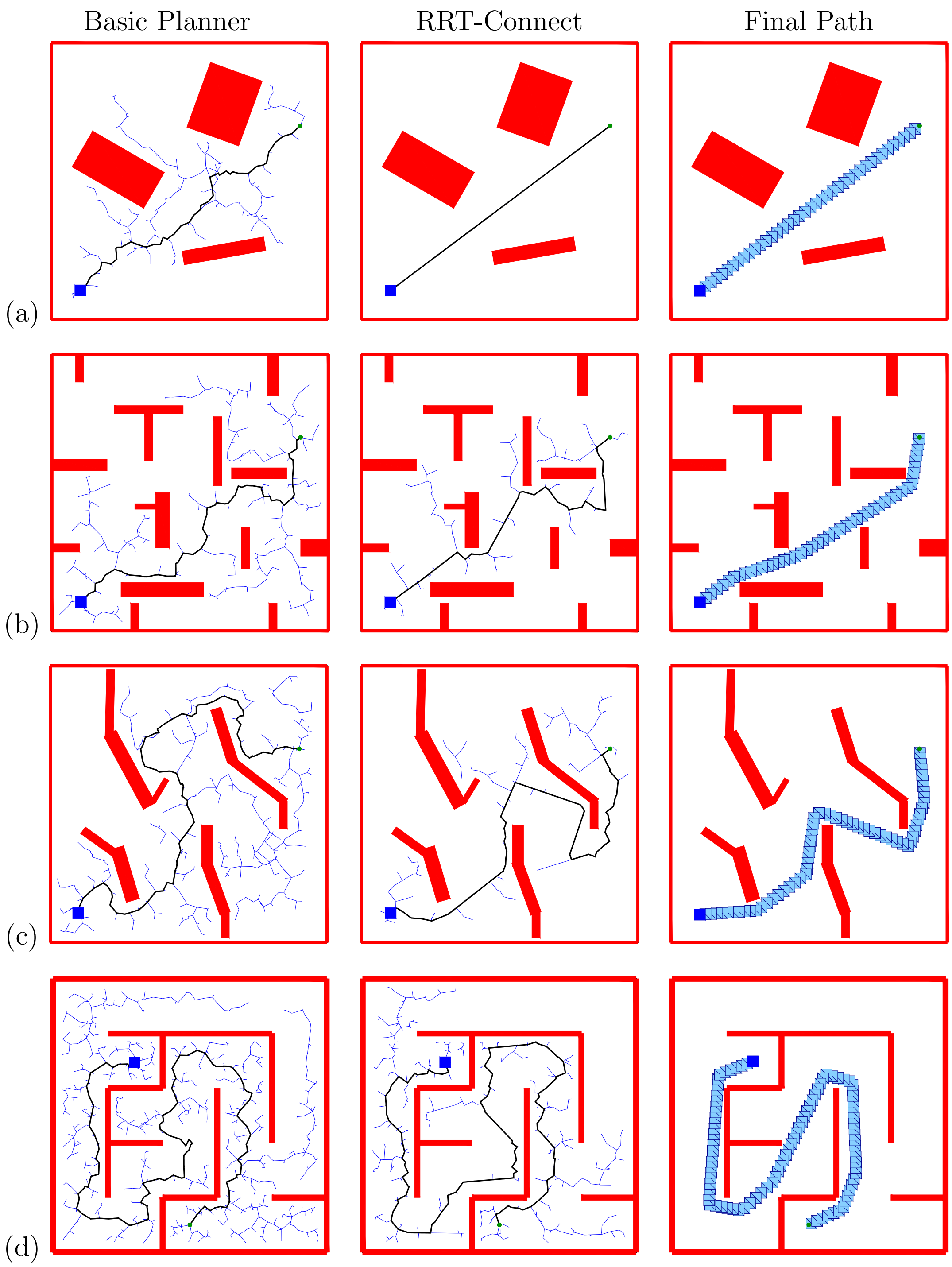

Figure 5.12: Basic vs. RRT-Connect heuristic. Simple queries for a translating 2D rigid body.

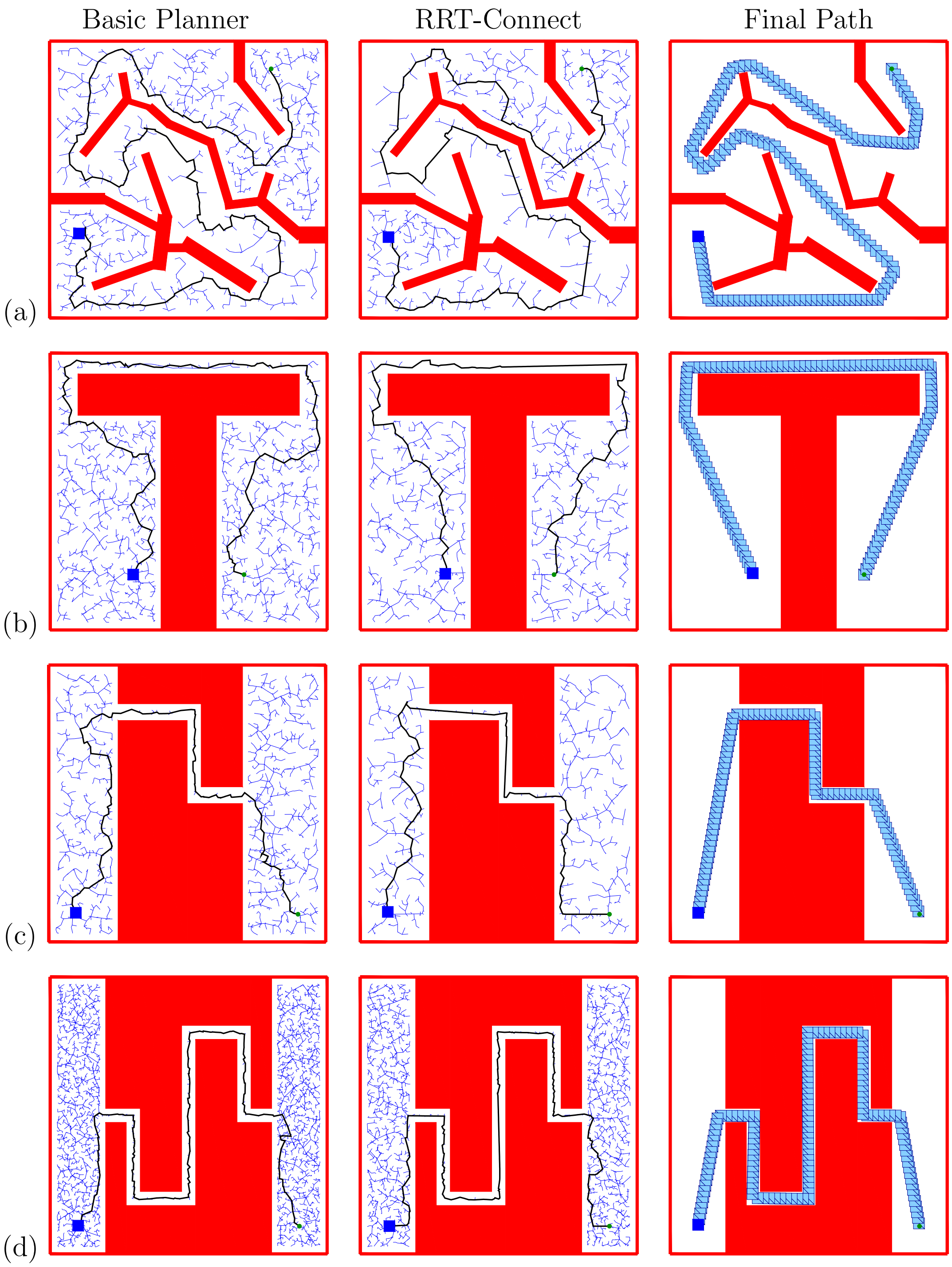

Figure 5.13: More challenging queries for a translating 2D rigid body.

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

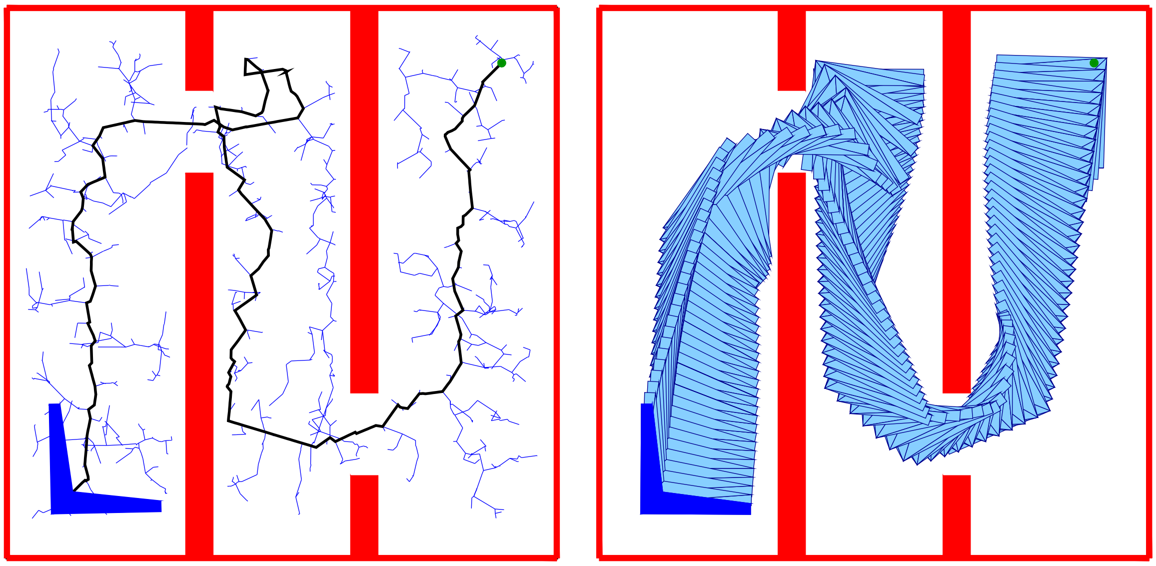

Figure 5.14: The RRTs (left) and computed path (right) for an L-shaped rigid body in 2D.

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

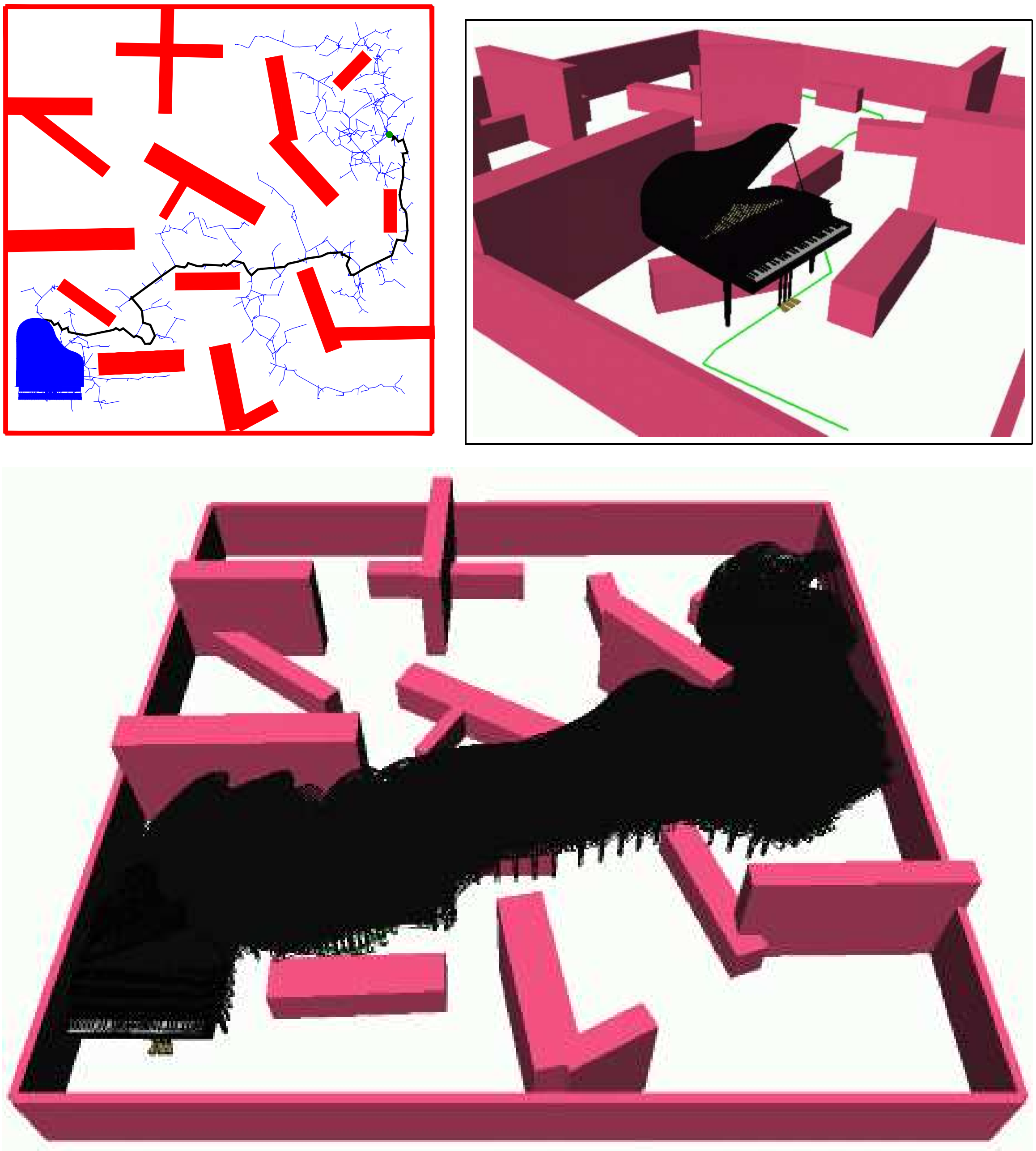

Figure 5.15: Moving a piano in a room with walls and low obstacles.

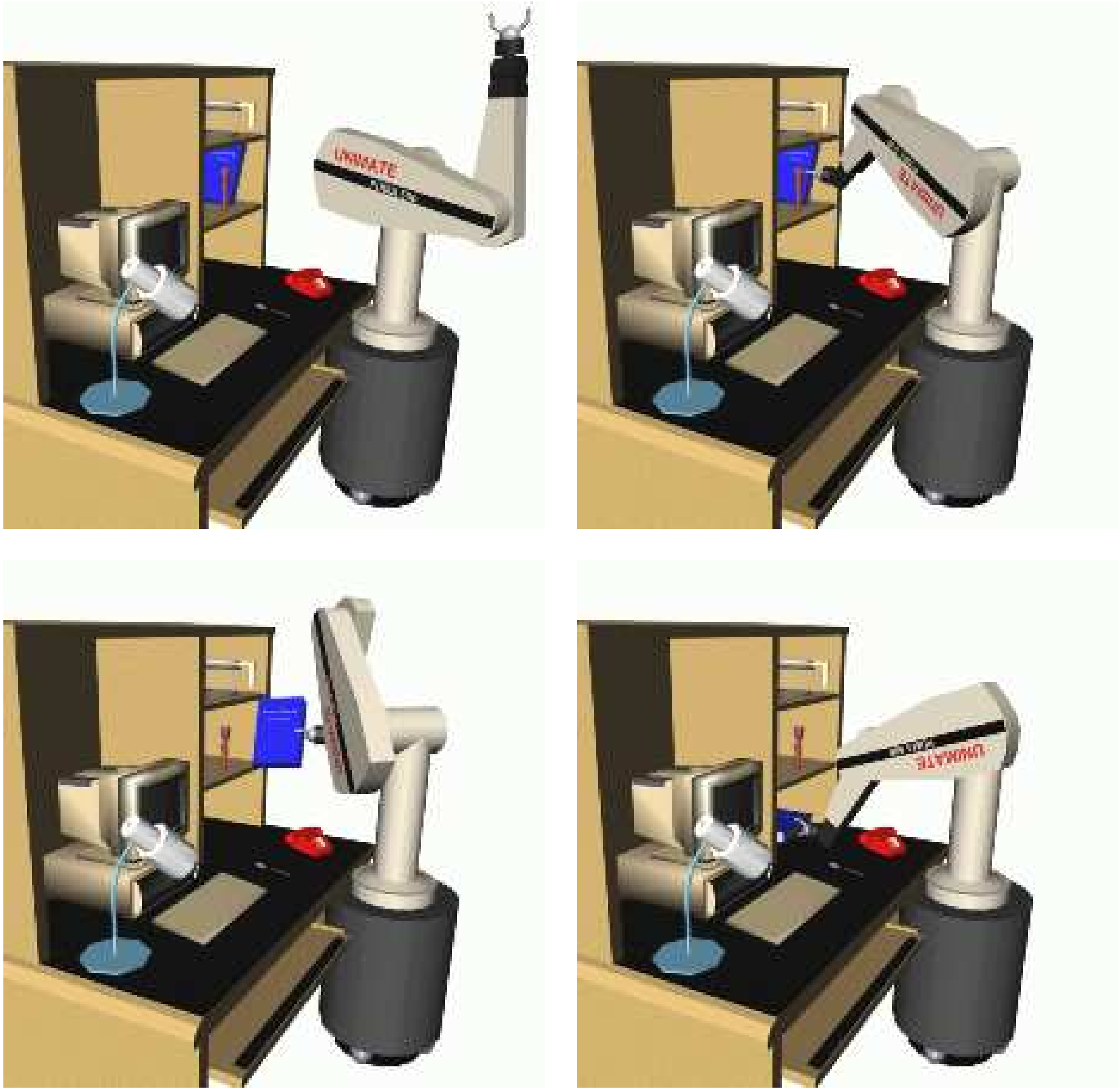

Figure 5.16: A 6-DOF Puma robot moving a book.

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

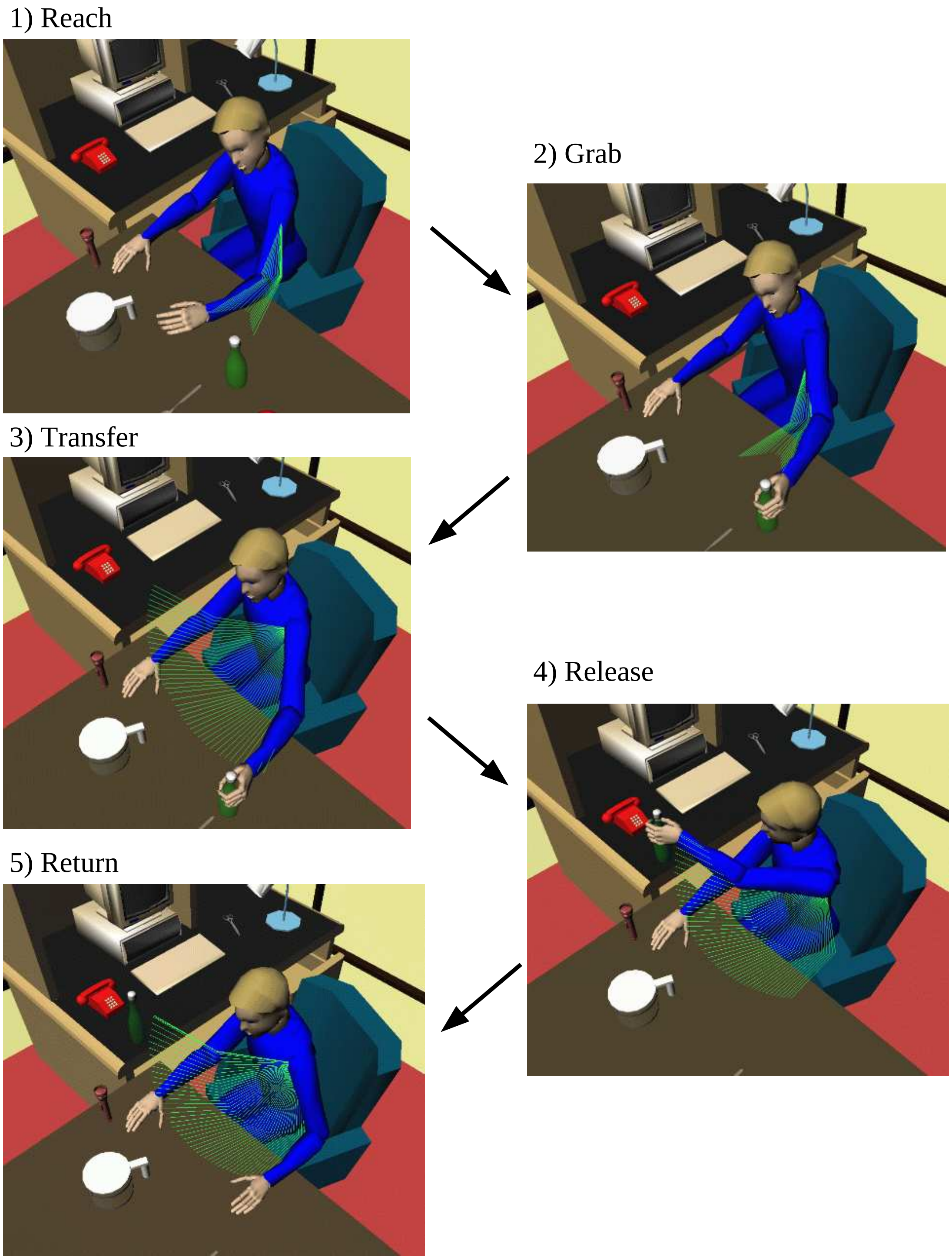

Figure 5.17: A human character repositioning a bottle.

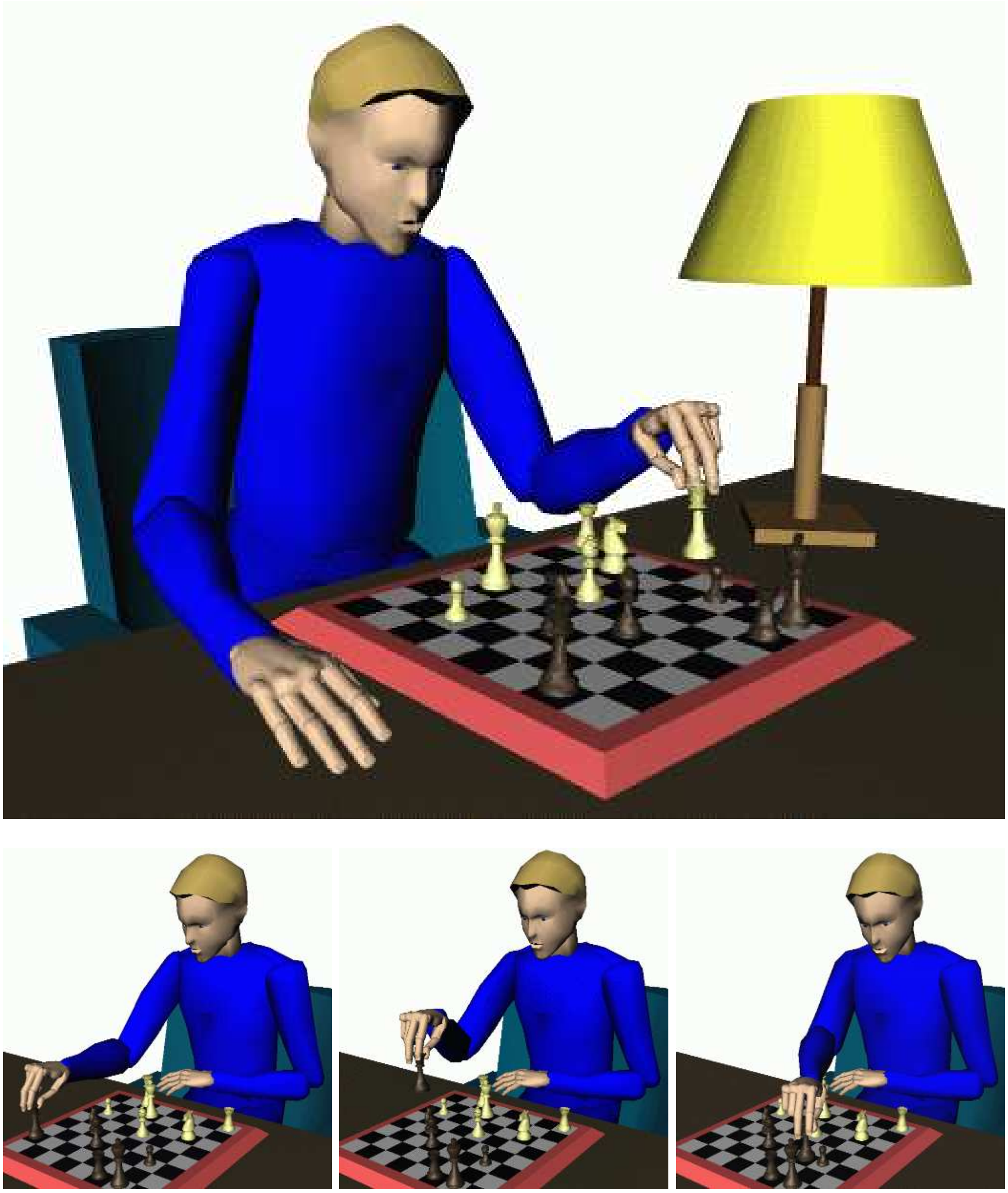

Figure 5.18: Playing a game of virtual chess.

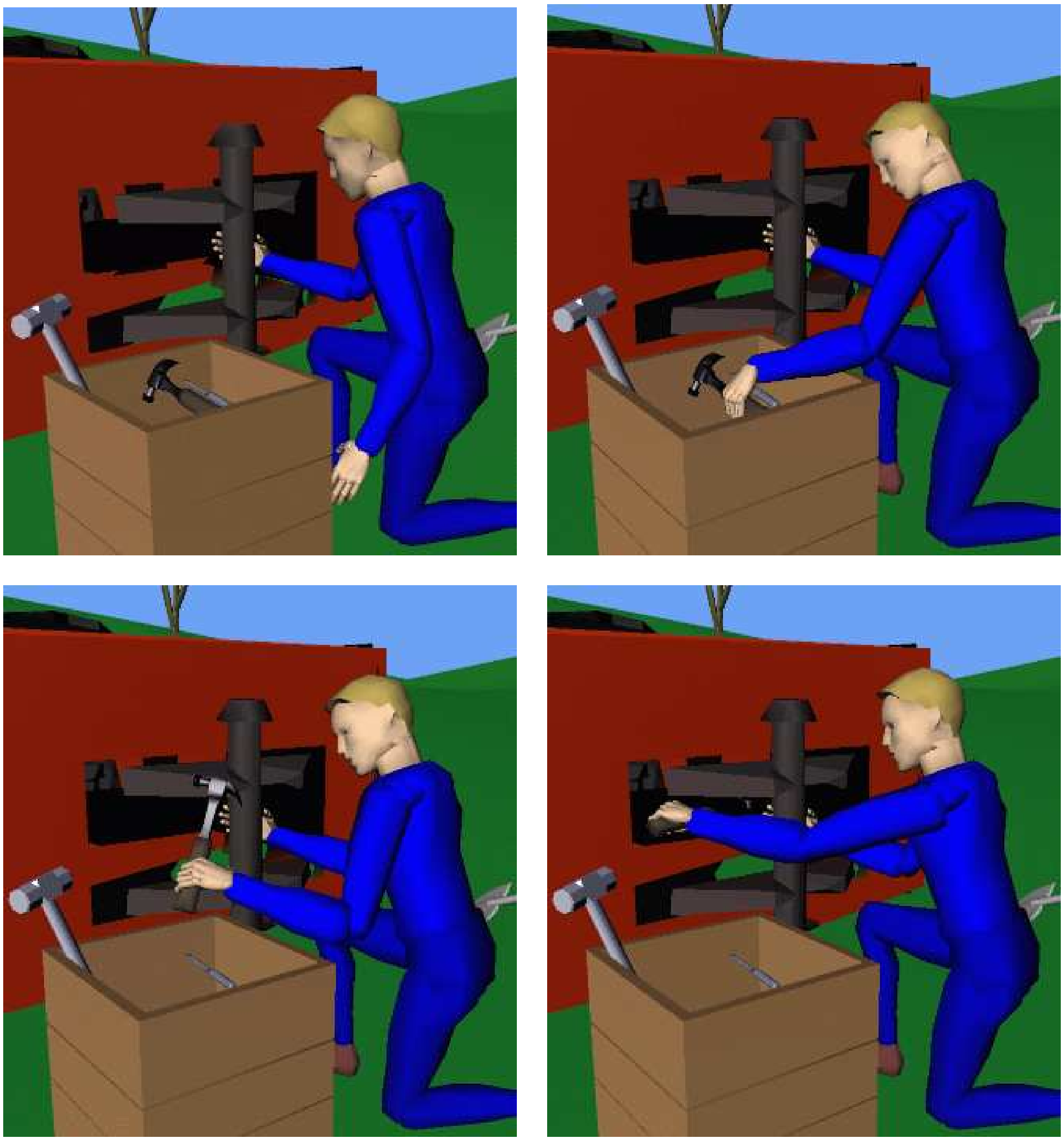

Figure 5.19: A more difficult planning query.

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

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.