A Grasp-based Motion Planning Algorithm for Intelligent Character Animation

by

Maciej Kalisiak

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical & Computer Engineering
University of Toronto

Abstract

A Grasp-based Motion Planning Algorithm for Intelligent Character Animation

Maciej Kalisiak
Master of Applied Science
Graduate Department of Electrical & Computer Engineering
University of Toronto
1999

The automated animation of human characters continues to be a challenge in computer graphics. We present a novel kinematic motion planning algorithm for character animation which addresses some of the outstanding problems. The problem domain for our algorithm is as follows: given an environment with designated handholds and footholds, determine the motion as an optimization problem. The algorithm exploits a combination of geometric constraints, posture heuristics, and gradient descent optimization in order to arrive at an appropriate motion sequence. The method provides a single framework for animated characters capable of animating the model using an assortment of modes of locomotion and capable of solving complex constrained locomotion tasks. We illustrate our results with demonstrations of a human character using walking, swinging, climbing, and crawling in order to navigate in a variety of environments.

Acknowledgements

I owe a great debt of gratitude to a number of people for having directly or indirectly helped this thesis come to fruition: to Michiel, my supervisor, whose great enthusiasm and tireless scrutiny of the thesis while on sabbatical were irreplaceable; to Cathy, my wife, for coping with a missing husband with such understanding, especially the last few months; to my parents, who have fostered my curiousity, interests and education from the very beginning; to my committee members who graciously donated their time; to Steve Hranilovic, without whose exemplary influence I might have not pursued graduate studies and possibly not even successfully finished the undergraduate ones; and to the DGP bunch, whose helpfulness, amicability, and the occasional BZ session helped to make this a thoroughly enjoyable experience.

This thesis was further made possible by funding from the University of Toronto and the Province of Ontario through the University of Toronto Fellowship and the OGSST grants.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The animation of human figures has been a challenge since the advent of computer graphics, and thus has seen the evolution of many tools, operating at several different levels of abstraction and user-control. A method which consistently animates the human form in a natural fashion across a wide range of scenarios is still lacking.

Human motion is very complex, as demonstrated by the duration of time it takes a human baby to learn how to walk on its own, with more complex motions taking even longer. From a computational point of view, the difficulty of locomotion stems partly from the many potentially feasible solutions, from which the most natural variation must be chosen. In choosing a particular motion, a character must strike a balance between a number of complex constraints, which include balance, comfort, and joint limits. This motivates the casting of character animation as an optimization problem, and is also the approach we take in this thesis.

Methods for generating human motion can be subdivided into a number of categories or specializations. Some are intended for general locomotion in simple terrains, while others are more proficient in constrained environments. Certain algorithms are limited to reproducing unique movements, such as diving. Others specialize in dealing with secondary tasks, such as object manipulation. Yet others concern themselves with generating expressive gestures. Of all these categories, our planner deals only with the first two: human locomotion through free and constrained environments.

To date, the majority of character animation techniques use local methods which focus on what to do here and now in order to keep a character balanced or moving forward. By their nature, these solutions tend to be limited to a specific mode of locomotion and do not help to plan the motion over larger distances or constrained situations. Intelligent motions in such environments require a planning process in order to properly anticipate obstacles and potential dead-ends. Once a plan has been formulated, a motion needs to be synthesized which follows the plan closely. It is this planning, in combination with a method for generating path-adhering motion, which forms the focus of our work.

Motion planning in the context of human animation is not new but the current state of the art does not deal adequately with general environments and motions. Current motion synthesis tools are predominantly focused on walking, while treating other motions, such as bending to pick up an object, as secondary motions to be executed when the character is stationary. Yet other motions requiring movement in tightly constrained environments, are not handled at all.

An example of the type of problem that we would like to solve is presented in Figure 1.1. We would like to be able to automatically navigate a character through such varied environments by using a number of different modes of locomotion. The small boxes on the obstacle surfaces represent points which the character is allowed to grasp, pull, or step on.

The intended audience of our animation tool consists of animators who require a more abstract method of animating characters, one which will alleviate the need to fully specify the motion, requiring only high level constraints such as starting and ending configurations[†]. This work would therefore be of use in computer games, simulators, virtual worlds, as well as secondary character animation for production animations, i.e., film and video.

[†]A character's *configuration* describes the position as well as the arrangement of the body.

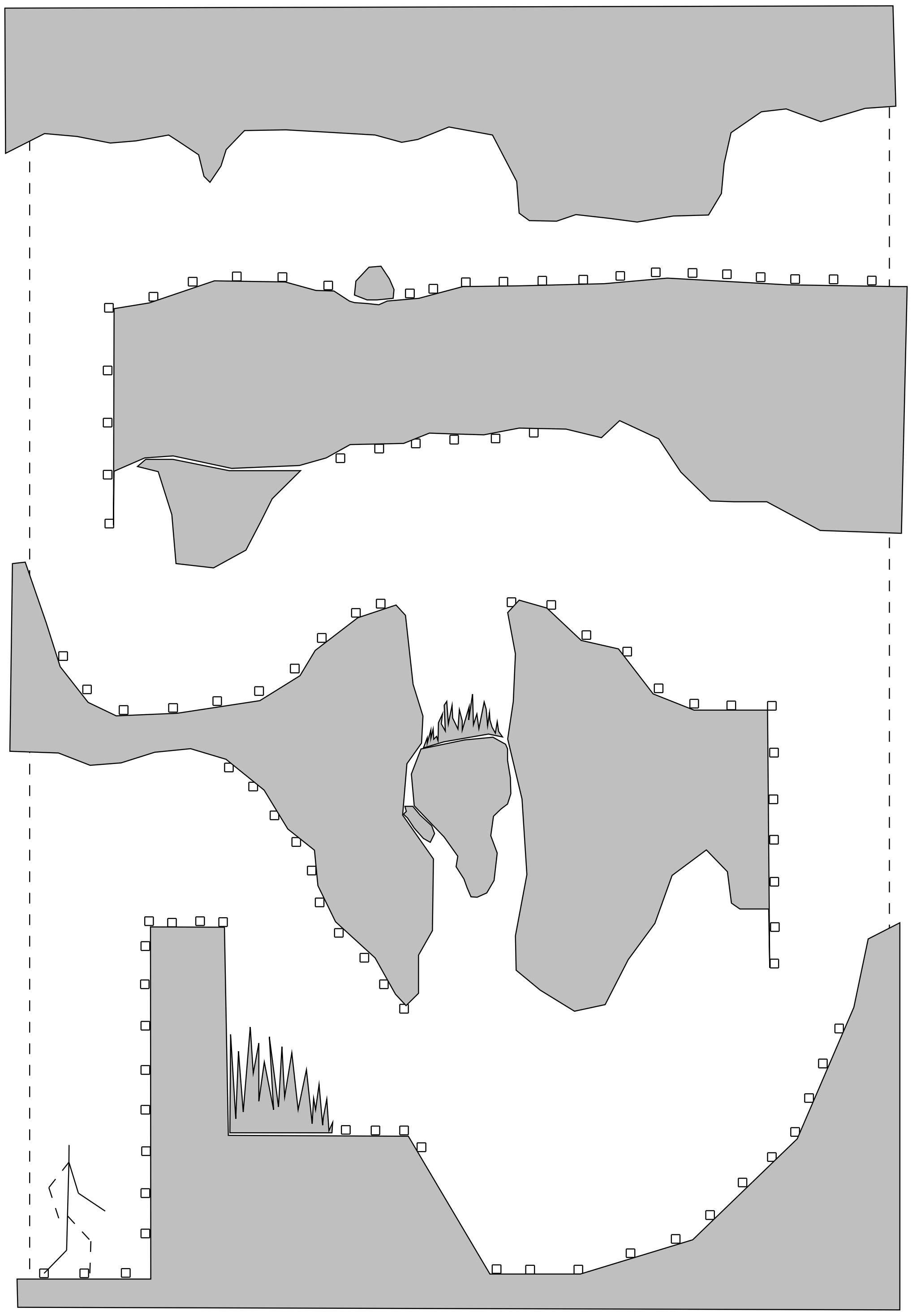


Figure 1.1: A sample environment we would like to be able to solve

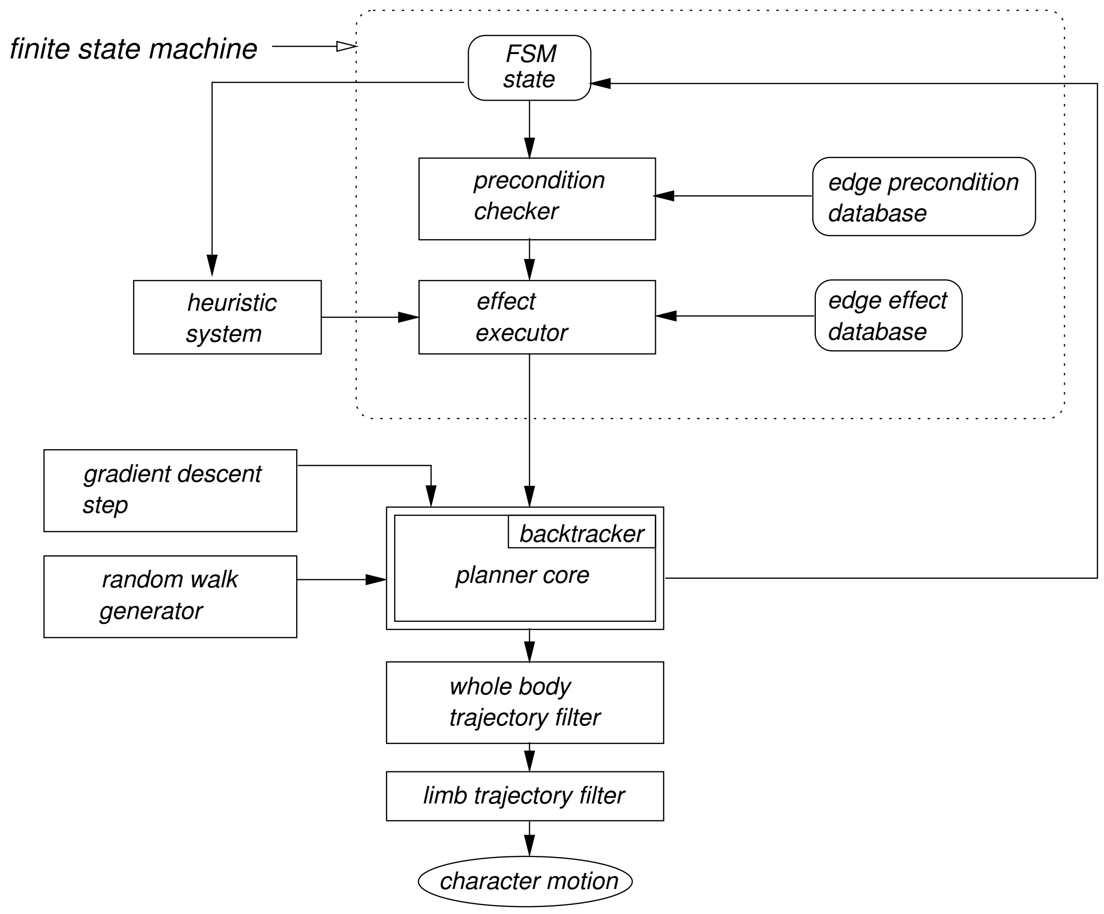


Figure 1.2: A block diagram of our planner

1.2 High-level overview of work

The block diagram in Figure 1.2 outlines the structure of our approach. The heart of the planner is the *planner core*, also sometimes referred to as the *scribe*. Its main function is to collate the motion segments handed to it into a single stream which represents an initial motion plan. Each such motion segment consists of a sequence of configurations or poses for the character. The sources providing the core with these segments are the "gradient descent step" module, the random walk generator, and the finite state machine(FSM). We provide a brief overview of each of these below.

The simplest way in which the planner generates motion is through the *gradient descent* of the *configuration space potential*. The configuration space potential, written as $\mathcal{P}(\mathbf{q})$, denotes the shortest collision-free distance of the character's center of mass(COM) from its position in configuration $\mathbf{q}$ to that in the final configuration. The gradient descent employed differs from the traditional algorithm in that we do not compute analytic gradients of the field as this is expensive and complicated. Instead, at each step we sample $\mathcal{P}(\mathbf{q})$ for a number of slightly perturbed versions of the current configuration, and proceed with the one which results in the largest drop in this potential. The "gradient descent step" module performs a single step using this method.

Since gradient descent has the drawback of becoming stuck at local minima, we use *random walks* to escape these. This consists of perturbing the character's configuration for a prespecified duration. In essence, Brownian motion is applied to the figure. The intention of this is that the character will lie outside the influence of the local minimum in the ending configuration, and hence will proceed on a different path to the goal.

Finally, we employ a *finite state machine*(FSM) to model different character gaits. In this FSM each state represents a different locomotion mode while the edges represent transitions between them. Each such edge has some geometric preconditions which must be satisfied prior to traversing it, as well as a number of post-traversal effects, most of which result in a motion segment being generated and passed to the scribe. These effects deal with altering the character's *grasps*, an abstract representation of the attachment of an extremity to a handhold or foothold. Most of these regrasping operations are usually followed by a posture correction step. This consists of a gradient descent through the *discomfort space*. This is a potential field similar to the one used in the "gradient descent step" module, but one that is computed with the use of a heuristic system which implicitly models human comfort. The result of this latter gradient descent is to array the character into the most comfortable pose currently possible.

At every iteration the planner core accepts a motion segment from one of these components. The source is chosen based on the following order: FSM, gradient descent step module, random walk generator. If a particular component cannot provide a meaningful motion segment to follow the current solution then the next module is consulted. The random walk generator can always provide a motion segment. If the planner core notices that the character is not making any progress it assumes it is stuck and performs a backtracking operation. This consists of regressing to a previous point in the solution, throwing the subsequent portion of it away, and restarting the collating process. Once an approximate solution has been found joining the initial configuration to the final one, it is passed in sequence through the "whole body" and the "limb" trajectory filters. The purpose of these filters is to make the character's motion smooth and fluid.

1.3 Layout of thesis

In the next chapter we discuss a collection of methods that have been applied to the problem of human motion synthesis. In Chapter 3 we introduce several of the prerequisite concepts needed for the understanding of our algorithm. Chapter 4 describes the complete planner. We present and discuss our results in Chapter 5. Finally, we summarize our contributions and propose future work in Chapter 6.

Chapter 2

Previous Work

In this chapter we review a wide variety of existing character animation techniques. We first discuss keyframing and motion capture methods. Both of these can be classified as *kinematic animation* techniques and they are both widely used in practice. This is followed by a discussion of ongoing research into the use of dynamics for character animation. We conclude by discussing a path planning algorithm best suited to our research and its application to character animation.

Tools and algorithms for character animation can be evaluated according to a number of different criteria, which can include:

- interactivity
- amount of animator control
- amount of automation
- the concepts explicitly modeled
- the time scale on which the method operates
- robustness with respect to variations in environment

Interactivity ranges from complete, such as required in game systems, to none, as in film production. Animator control can likewise be complete, such as afforded by keyframing, to none, as in the case of autonomous creature simulation. Inversely related to that is the amount of automation, which specifies the level of demand for animator's input. Some of the concepts that can be modeled by a given method are physics, sensing, memory, and knowledge. The method can operate either on the global time scale, or locally. Finally, a given approach can be generic and adept at working with environments of varying structure and complexity, or it can be designed and fine-tuned for only a limited subset of terrains.

The primary criteria for our research are the offloading of the work to the computer while providing a useful amount of animator control, and the ability to operate in diverse, and especially constrained environments.

2.1 Kinematic animation

While the human form is difficult to animate, the many possible applications of character animation have led to the creation of a variety of animation tools. These can be broadly categorized as being *dynamic* or *kinematic*. Dynamic approaches model the forces and torques involved in a motion, while kinematic approaches use a combination of constraints and other model-specific rules in order to generate novel motions. It is this latter category of animation methods that we discuss in this section. The two most prominent approaches in this area are *keyframing* and *motion capture*.

2.1.1 Keyframing

Keyframing is perhaps the best known animation technique, one which has its origins in traditional media and one which is still widely used today. In this method the user completely positions the character at key instances of time. Piece-wise continuous splines, which model individual joint angles over time, are then fitted to the provided keyframes. Subsequently, any intermediate frames can be arrived at using interpolation.

In the hands of an experienced animator, keyframing is an effective tool for creating motion, allowing complete control over every detail of the motion. The drawback of keyframing is the effort required in specifying a large amount of information and the considerable skill required to achieve natural-looking motion with this technique. Further discussion on the topic can be found in [Stu84].

2.1.2 Motion capture

Perhaps the most popular and most direct kinematic method of creating animations is the use of motion-capture data. This consists of having a real human actor perform a set of desired motions, which can then be mapped onto a computer-animated character.

There are a variety of ways of recording the motion data. In one approach the actor is outfitted with visual markers at significant points on his or her body, the motion is filmed using multiple cameras, and the character configuration is extracted through computation from the 2D positions of the markers. Another approach requires requires that the actor be outfitted with 3D sensors. The 3D trajectories of the sensors are recorded and inverse kinematic are then applied to derive the corresponding joint trajectories. Yet other methods rely on sensors which are capable of directly measuring the joint angles themselves. Whatever the method used for capturing the data, appropriate noise reduction steps must be taken before it can be used since the data is usually noisy due to the limitations of sensors and extraction methods used.

One key issue in using motion capture data is that the virtual character quite often does not have the same dimension as the original actor. Applying the captured motion unmodified violates important constraints, resulting for example in feet not touching the ground or the character "skating". Gleicher [Gle98] as well as Ko and Badler [KB92] both present a method for adapting motion data to differently proportioned characters.

Another serious limitation of motion capture is that any character animated with this approach is only capable of the motions which have been previously recorded;

there is no simple way to generate novel motions. Furthermore, even though additional motions could be captured for any lacking movement types that have been identified, one is limited to motions which are recordable. For example, it is not obvious how to obtain motion capture data for imaginary creatures such as a dragon.

Recent work attempts to address the reusability of captured motions. Bruderlin and Williams present in [BW95] a method for applying signal processing methods to the character's joint motion curves, allowing the user to amplify or attenuate different portions of the frequency spectrum of these signals, resulting in the emphasis of different characteristics of the original motion. [UAT95], on the other hand, outlines a method for interpolating as well as extrapolating between two related motion sequences to smoothly blend between the two, or to exaggerate the characteristics of one of them with respect to the other. Witkin and Popović propose in [WP95] a method to alter or *warp* a given captured motion by accepting keyframe-like constraints which dictate a smooth deformations of the original motion curves. The main limitation of these methods is that the amount of deformation or extrapolation that can be applied is limited as large warps will cause the characteristics of the motion to be lost, and will result in a pronounced artificial look. Furthermore, these methods are not well suited to complex environments since collision avoidance and planning are not easily integrated.

2.2 Dynamic animation

In this section we present methods which fall under the dynamic animation category. In general, these methods focus more on the local planning aspect of motion, concentrating on achieving physically correct movement. As the problem they undertake is a much more difficult one than that of kinematic methods, none of them take global planning into consideration, a trait crucial to the task that we are studying. These methods are therefore presented for completeness, as their resemblance and applicability to our work is very limited.

2.2.1 Controller-based methods

The principal idea behind the dynamic approach to human animation is to employ physical simulation for the purpose of obtaining physically correct motions. The use of physics imposes a constraint on the possible motion, but in the case of character animation, it does not serve to uniquely define the motion. Instead, the use of a physical simulation transforms the animation problem into a control problem, namely one of determining the muscle actions which are necessary in order to achieve desired movements. The simulated muscles are typically abstracted as being rotary actuators capable of exerting torques at the joints. Designing appropriate control functions is a difficult task for an animator, however, in large part because the relationship between the applied control and the resulting motion is indirect. A torque applied at a particular joint produces a resulting set of accelerations, which need to be integrated twice in order to generate the resulting change in position. Hence, one cannot easily modify the torques to incrementally modify some existing motion. An animator using torque as a function of time to provide control is also further handicapped because these functions cannot typically be copied for reuse in another situation. This is because the motion is a function of both the initial state and the applied control inputs.

Torque based controllers are further plagued by usually being capable of only a very limited class of motions. To make a character capable of a substantial repertoire of motions one has to endow it with a large collection of controllers, but this leads to the problem of deciding the timing of transitions from one controller to another. Furthermore, controllers are usually specific to the creature they were designed for, although a method is proposed by Hodgins and Pollard in [HP97] for adapting existing controllers to new models, ones that are similar to some extent.

Closed-loop

Controllers can be roughly categorized as either open- or closed-loop. A closed-loop controller differs from an open one in that it uses feedback to monitor its progress, and if need be, adjust its operation.

One of the most successful applications of closed-loop controllers to the animation of locomotion are the hopping bipeds, quadrupeds and kangaroos of Raibert [RH91]. Here the animator provides high level input, such as the desired speed, direction, and gait type, which the controller uses, along with the current state of the subject, to compute the necessary forces and torques that need to be applied at the actuators to keep the character balanced and moving in the desired direction. This approach has also been successfully applied to robots.

[Sim94] presents a method for simultaneously evolving a creature's anatomy and the corresponding controller through the use of genetic algorithms. Although the results are interesting, this method has limited use since it does require a large amount of computing power, and the virtual creatures produced are highly abstract, of little resemblance to any real life forms.

Sensor Actuator Networks (SANs) [vF93] are a collection of units whose outputs are a non-linear function of the sum of the weighted inputs. The parameters of the SAN are chosen using a variant of simulated annealing in order to optimize the performance as measured by fixed-duration simulations of the SAN providing control to a given creature.

Open-loop

Open-loop controllers tend to be rare since they are very delicate and unstable. They are specific not only to the character they were designed for, but also to the environment and initial state. Any perturbation or anomaly in the system risks upsetting the motion and rendering the controller output ineffective.

One dynamic method which can be considered open-loop is that of Pose Control Graphs [vKF94, van96]. This approach consists of a finite state machine(FSM) which specifies a number of poses as well as the hold time for each. The FSM then uses PD controllers to drive the character's joint angles during a physical simulation. Although the PD controllers themselves are closed loop, the overall FSM approach does not make use of any feedback. [Las96] and [LvF96] further introduce a method for adding a layer of control which attempts to improve balance in unstable PCG controllers.

2.2.2 Trajectory-based methods

Spacetime Constraints(SC) [WK88] is an alternate approach to incorporating physics into motions. It involves directly optimizing motion trajectories in order to satisfy physics constraints as well as user constraints and objectives. This approach fits nicely into the keyframe paradigm; the user specifies an initial motion by hand and then lets the algorithm optimize this approximation to yield a physically plausible motion. This approach has a further beneficial property that if no such solution could be found, the motion which most resembles a physically-correct one is returned.

The SC method has several potential disadvantages. The approach can often lead to an enormous optimization problem prone to local minima. As a result, most implementations make concessions and adopt certain limitations. As applied to human motion, this approach suffers from requiring complex and unintuitive input on the part of the user, since numerous constraints are needed to keep the motion in line with what a human would produce in reality, as well as the daunting task of trying to express all the traits and eccentricities of human motion in the *objective* function. This approach also has the undesirable quality of providing perfect anticipation due to the global nature of the optimization and the absence of a sensing model.

In [Coh92], Cohen extends the SC approach by using a B-spline representation for the joint trajectories as well as by localizing the optimization through the use of time windows. This was further improved upon through the use wavelets in [LGC94]. Also, in [LC95] Cohen and Liu allow the timing of the keyframes to be relaxed. This approach is based on the observation that although most users have a good idea what poses should be assumed in a desired animation, they often do not have a good intuitive feel as to **when** these poses should be assumed. The work presents a method for generating motion from a set of keyframes in which most lack timing information.

A trajectory-based representation is also adopted by [van97, Tv98, Tor97] in their work on a footprint-driven motion generation method. In this case the problem size and the potential for local minima is reduced through the use of a simplified physical model. Once an optimized trajectory has been obtained for the simplified model, the remaining details of the motion are filled in by taking advantage of a priori knowledge of the footprint locations and footprint timing, which together form the animator's motion specification. The specification of footprints can also be automated with the use of a simple path-planning algorithm.

2.3 Strongly related work

In the following two subsections we discuss the two kinematic systems which most resemble our own work in terms of motivation and desired goals, although they do so through different methods. They are the *Motivate 3D Game System* by Motion Factory, and the *Jack* system from the University of Pennsylvania.

2.3.1 The *Motivate* system

The "Motivate 3D Game System" [Fac, KAB+] is a commercial 3D game development system. It offers a novel way for generating game content through the use of real-time motion synthesis and dynamic event-based programming. The production cycle usually consists of importing externally generated 3D models for the characters, the definition of character skills, and the specification of *behaviours*, hierarchial finite state machines with embedded procedural code describing the actions or tasks the character is to perform in particular situations.

The real-time motion synthesis component presents an interesting mix of the previously described kinematic methods and motion warping. Here, the animator first builds up a database of atomic actions for the character using keyframing or motion capture. These are referred to as *skills* or *actions*. Once a sufficiently large repertoire has been gathered, the solution is assembled by stringing together a sequence of these actions. Each of these motion segments is chosen for its closeness in fit to the motion requirements at the current point in the solution. A form of motion warping is subsequently applied so that these motion segments meet the constraints precisely.

Although this method performs admirably for simple scenarios, significant discontinuities in the motion can be seen whenever two substantially different atomic actions are put together, or when the database lacks a suitable skill for a particular required movement. In essence, the result is highly dependent on the quality and diversity of the "skill" database, which in turn depends on the experience and expertise of the animator. The available demonstrations of this technique depict the characters using only variations on walking gaits, such as tiptoeing, running, bounding, and jumping. The actions are usually performed in relatively unconstrained spaces. It is therefore uncertain how well this system would cope with constrained environments, and whether it could be used equally well for other forms of locomotion, such as crawling or climbing.

This system differs from ours not only in the method employed, but also in the goals and requirements it is trying to satisfy. Whereas we concentrate on fluid navigation through constrained environments, the Motivate system places the emphasis on real-time character animation at the expense of motion continuity and sophistication, as the former are demanded by game playing environments, which are the target application for this planning system. Motivate is also concerned with object manipulation, which subsequently affects the capabilities of a path planner.

2.3.2 The *Jack* system

The other kinematic approach that aims to achieve some of the same goals as our method is the Jack system [BPW93, PB91, LWZB90]. This is a very complex, multi-faceted system, mainly used for ergonomic studies. It allows the user to perform field-of-vision analysis, comfort assessment, as well as testing reachability. Recently it has been further outfitted with strength modeling and collision avoidance. Like the Motivate system [Fac], the Jack system is capable of grasping objects. Although developed at University of Pennsylvania, it is now a commercial product made available by Transom Technologies [Tra].

Ergonomic analysis can be performed on static postures of the character, as well as when it is in motion. Such dynamic behaviour study first requires the user to script a *task* – the movements and operations which are to be performed. Motion is generated by way of dragging body segments to new locations. A natural and aesthetically pleasing look is achieved by employing or following *behaviours* during the generation of limb and body motion. These are procedural implementations of empirical properties observed in humans. One such behaviour is that of balance: in order to keep the character balanced, the behaviour attempts to keep the character's projection of its center of mass within the support polygon. It does so by adjusting the character's posture, perhaps even taking a step forward or backward if necessary. A task may consist of any number of such generated motions combined. Once the scripting is complete, the task can be saved and the simulation rerun any number of times, with different parameters. In this way the user can study the task's ergonomic qualities for varying body types and sizes.

Although the Jack system works quite well in solving the local motion planning problems found in ergonomic studies, it does not solve the problem that we have undertaken to research. The main deficiency is the low level of automation: the user needs to manually input the motions that make up the task, whereas we would like to input only high level motion constraints, such as initial and final configurations. Even if the system was capable of such planning, and the motions in a given task were generated procedurally, the resulting locomotion would probably be limited to walking-type gaits only. This is due to most of the behaviours having been designed based on this assumption.

In contrast to Jack, the approach that we are proposing is a motion-planning method suitable for constrained environments, one capable of autonomously making use of many types of gaits, one that can use search techniques to escape possible local minima, and one that can handle motion transitions as well as being able to determine when a transition might be warranted. With the single framework outlined in our approach the character can be automatically animated over a wide range of complex landscapes, using a variety of human motions, limited only by the number of motions built into the implementation.

2.4 Path planning

The path planning problem has some interesting qualities to it. On the one hand it can be viewed as a discrete decision problem: the choosing of which handholds and footholds to use, as well as the matching of limbs to the said points of contact. On the other hand, it also has a significant continuous component: the finding of a smooth motion for the body and limbs in between the discrete decisions.

[BL91], as well as parts of [LPW91] describe the Randomized Path Planner(RPP). This approach consists of denoting certain key points on the object being animated as *control points*. A *navigation function* potential field, spanning the environment, or *workspace*, is constructed for each such control point. A linear combination of these potential fields are then combined into a configuration space potential function, denoted by U. This potential function essentially can be considered a rough metric of how far the character is from the goal configuration. The navigation functions are specially chosen so that the occurrence of local minima in U are kept low. The path planning then consists of performing a gradient descent through U, using random walks as a means of escape from local minima. This method has many benefits: for simpler problems it is several orders of magnitude faster than previous methods, it works well with obstacles of arbitrary complexity, and is capable of solving problems for robots with large number of DOFs as well as multirobot scenarios.[†]

This RPP algorithm has been extended to deal with 3D manipulation tasks in [KKKL94], which, at least in part, has later been incorporated into the Motivate system [Fac] that we have already discussed. This work concentrates on the problem of cooperative multiarm manipulation of objects, suited especially well to tasks which require regrasping of the object being manipulated. Even though we draw on some RPP concepts in our work, this particular extension does not have much in common with our research since the robots here are fixed to the ground; robot locomotion is not addressed. Tasks illustrated in the [KKKL94] paper are the human manipulation of a chessboard with the aid of a third robotic arm, as well as the task of picking up and putting on a pair of sunglasses using two arms.

[†] [BL91] describes a 3D solution to 31 DOF problem, as well as some 2D problems with 2 robots, 3 DOFs each.

Our work in the following chapters takes its inspiration in part from the Randomized Path Planner algorithm. We found it's ability to handle models with many degrees of freedom (DOFs) very appropriate for our work, and the research showed that it was quite fast. Both these factors made it ideally suited for our purposes. A number of adaptations were necessary to this algorithm, since our requirements for the path planner were slightly different from those originally intended. The main addition was the ability of the character to intentionally come into contact and "attach" to the environment (signifying a foot- or hand-hold), which resulted in the character always traveling in close proximity of some surface in the environment, which is in direct contrast with the RPP algorithm where the subject tries to stay equidistant from all surrounding walls and obstacles.

In the following chapter we will take a deeper look at some of the basic concepts and methods involved in RPP and its predecessors. We will also present a simplified version of the RPP planner, which forms the basis of our planner, which we present in chapter 4.

Chapter 3

Underlying Concepts

In this chapter we describe the fundamental concepts of the class of randomized path planning algorithms upon which we base our own work. We further illustrate these concepts by describing a simple, precursory planner for solving the free-space motion problem for a human character.

This chapter does not present any significant novel contributions, although occasionally it might apply previous work in new ways. Much of this material is a review of ideas presented by Latombe et al. [LPW91] and Barraquand and Latombe [BL91] in their work on randomized path planning. As our work builds on theirs, it is necessary to review the concepts of this particular prior art in some detail.

3.1 Notation

This section provides a concise overview of the notation that we will be using throughout this work. It is borrowed directly from the work on the Randomized Path Planner algorithm in [LPW91, BL91]. Additional notation will be introduced and explained as the need for it arises.

- We represent the character being animated with $\mathcal{A}$.
- The character's environment, also called the *workspace*, we denote with $\mathcal{W}$.

- $\mathcal{B}_i$ refers to a particular obstacles in $\mathcal{W}$.
- $\mathcal{B} = \bigcup\limits_i \mathcal{B}_i$, also called the *obstacle region*.
- $\mathbf{p}$ represents a specific point on $\mathcal{A}$.
- A particular *configuration*(§1.1) of $\mathcal{A}$ we denote with $\mathbf{q}$.
- A sequence of $\mathbf{q}$ represents a *path*, and is usually denoted by τ.
- When $\mathcal{A}$ has the configuration $\mathbf{q}$, $\mathcal{A}(\mathbf{q})$ denotes the region that the character occupies in $\mathcal{W}$.
- $\mathcal{C}$ denotes the *configuration space* of $\mathcal{A}$,
  i.e. the space described by the union of all the possible $\mathbf{q}$.
- $\mathcal{C}_{free}$ denotes the free space in $\mathcal{C}$,
  i.e. $\mathcal{C}_{free} = \{\mathbf{q} \in \mathcal{C} \mid \mathcal{A}(\mathbf{q}) \cap \mathcal{B} = 0\}$.
- When given a configuration $\mathbf{q}$, $\mathcal{X}$, the *forward kinematic map*, maps a particular $\mathbf{p}$ to its position in $\mathcal{W}$ ($\mathcal{X} : \mathcal{A} \times \mathcal{C} \to \mathcal{W}$)

3.2 Overview of method

Figure 3.1 illustrates what kind of problem we want our simplified planner to be able to solve. We want it to fly the character (with the handy jet-pack strapped to his back) from the starting point to the finish, as suggested by Figure 3.2. This is the free-space motion problem, akin to that of the "piano-mover". We can characterize this task as finding a free path through the workspace $\mathcal{W}$, preferably of minimal distance[†], that avoids collisions with obstacles. This can be equivalently stated as the problem of finding the shortest path from $\mathbf{q}_{start}$ to $\mathbf{q}_{finish}$ through $\mathcal{C}_{free}$. One

[†]Since the configuration includes both measures of linear and angular distance, "minimal distance" is ambiguous. In this thesis the term always pertains strictly to the linear distance metric, unless explicitly stated otherwise.

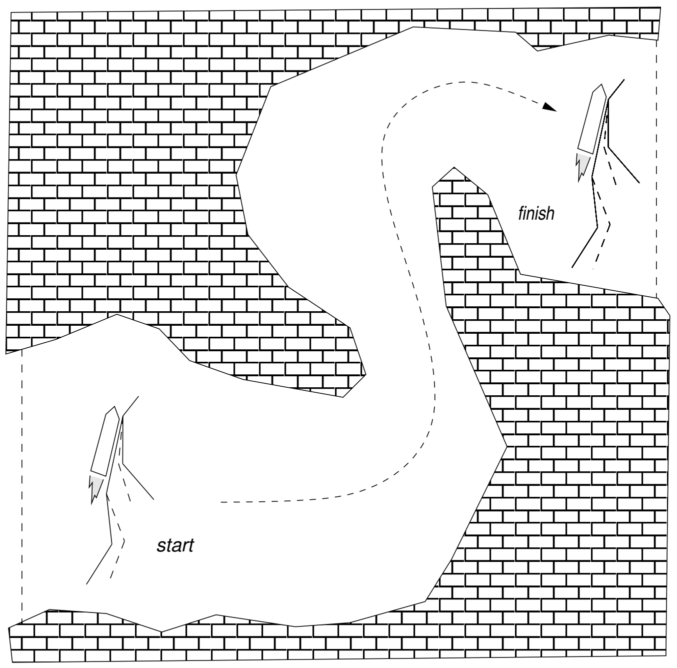


Figure 3.1: A sample problem for the simplified planner of this chapter

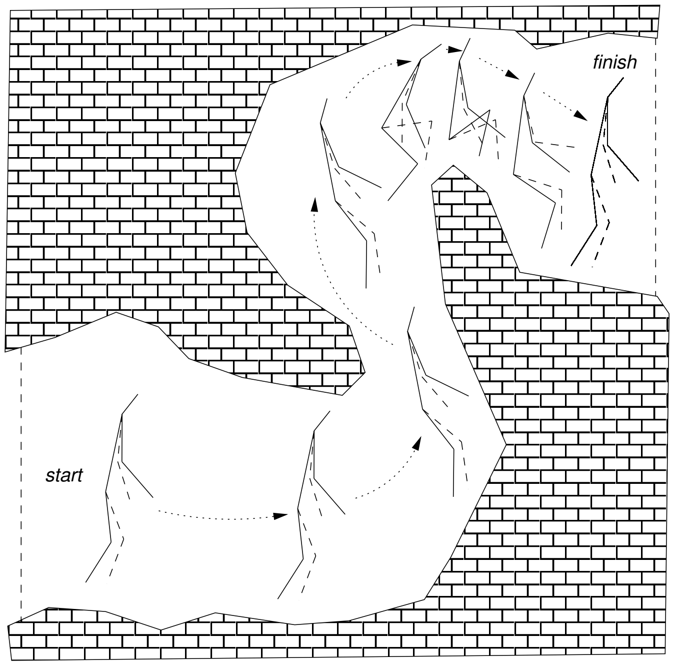


Figure 3.2: One possible solution; jet-pack not shown for clarity

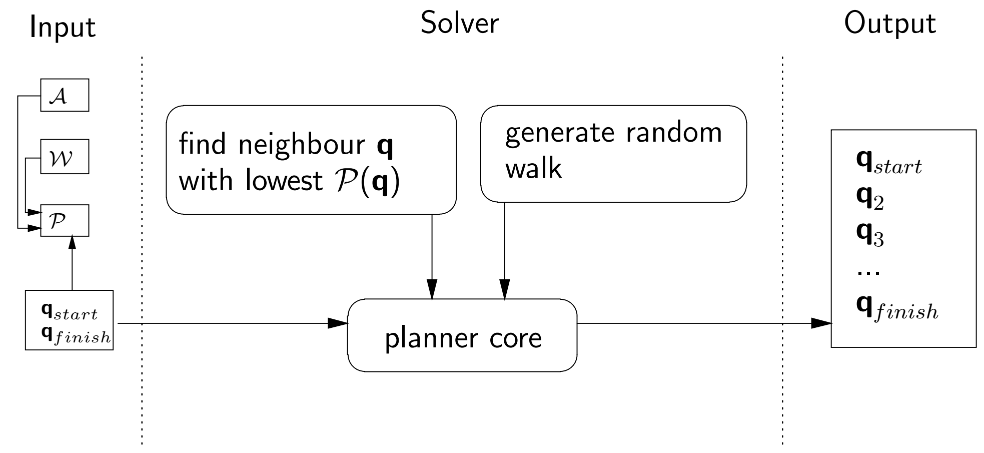

Input
Solver
Output
$\mathcal{A}$
$\mathcal{W}$
$\mathcal{P}$
$\mathbf{q}_{start}$ $\mathbf{q}_{finish}$
find neighbour $\mathbf{q}$ with lowest $\mathcal{P}(\mathbf{q})$
generate random walk
planner core
$\mathbf{q}_{start}$ $\mathbf{q}_2$ $\mathbf{q}_3$ ... $\mathbf{q}_{finish}$

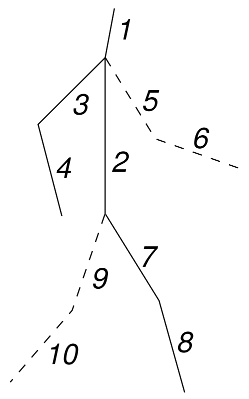
Figure 3.3: System block diagram of the precursory planner

way to do this is to compute a potential field[‡] for $\mathcal{C}$, which we will refer to as $\mathcal{P}$, and then to apply the gradient descent algorithm. Unfortunately this approach, like most methods which rely on potential fields, is prone to become trapped in local minima. A practical solution to this problem is to apply a random walk through $\mathcal{C}_{free}$ whenever this happens, as outlined in the work on RPP [BL91].

The remaining sections of this chapter describe the concepts outlined in the overview above in greater detail. Figure 3.3 shows a block diagram of the simplified planning system. The details of this diagram will be discussed shortly, and the reader may wish to refer back to it.

The algorithms in this and the following chapter deal only with 2D motion planning problems, instead of our desired goal of 3D animation. This lends clarity to the explanations, as well as reflecting the current state of our implementation. We believe that our method as described within these pages is easily extendible to three dimensions, hence the concession made does not limit the scope of application of the algorithm. We further discuss the extension to three dimensions in §6.2.1.

[‡]We will examine this potential field shortly. For readers familiar with the concept it should be noted that this potential field differs slightly from the traditional one in that ours does not include an obstacle-repelling component.



link	length
1	0.4
2	1.25
3	0.75
4	0.75
5	0.75
6	0.75
7	0.75
8	0.8
9	0.75
10	0.8

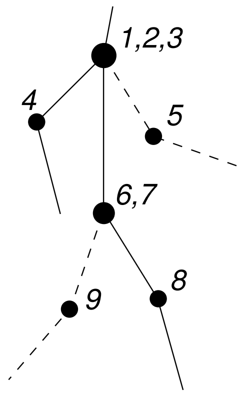
Figure 3.4: The 10 link skeleton

3.3 Skeletal description

Figure 3.4 illustrates the skeletal representation that we have chosen for the character. Since we are working in two dimensions, we have naturally adopted the side view. An evident flaw is that we have no links representing hands and feet, which should be explicitly modeled for purposes of animation. This choice is a compromise, allowing for the reduction of the configuration space (and hence the search space) by four dimensions, one dimension discarded per each extremity, with negligible influence on the final solution. The missing extremities could be reinserted by post-processing the solution obtained using the simplified figure. We have not yet implemented such a post-processing stage, instead focusing our efforts on the more fundamental elements of planning motions in constrained environments.

Figure 3.5 displays the joints in the model and specifies the joint limits which constrain their motion. As we are working in two dimensions for the time being, we use joints with a single rotational degree of freedom, which thus restrict the skeleton to motion in the xy-plane. Figure 3.6 illustrates the convention that we have adopted for representing joint angles.

There are many possible choices for representing $\mathbf{q}$, the skeleton's configuration. Our representation will build upon one which is well known in the kinematic literature. It consists of the x and y coordinates of a specific point on the skeleton, which we will



joint	link_a	link_b	θ_{min}	θ_{max}
1	1	2	-20	90
2	2	3	90	360
3	2	5	90	360
4	3	4	0	170
5	5	6	0	170
6	2	7	-10	170
7	2	9	-10	170
8	7	8	-160	0
9	9	10	-160	0

Figure 3.5: The joints of the skeleton and their limits

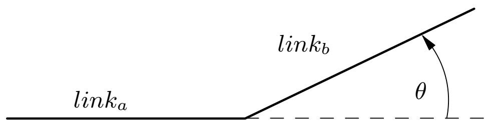


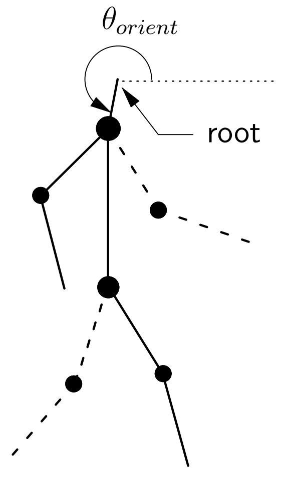
Figure 3.6: The convention used for joint angles

refer to as the *root*[§], a related *orientation angle*, and all the joint angles. We illustrate this in Figure 3.7. Since we have left the choice of the root point unspecified, this leaves us with an infinite number of possible representations of $\mathbf{q}$. This flexibility is useful, as it allows us to vary the representation of $\mathbf{q}$ over time. This proves to be convenient when constraining a foot or hand to maintain contact with the environment while the rest of the figure is allowed to move. (§4.2, page 40)

The *orientation angle* denotes the orientation of some arbitrarily chosen link in the skeleton, usually the root link[¶], with respect to the horizontal, or some other imaginary line which is used as reference. The purpose of this angle is to describe the global orientation of the skeleton given the remaining DOFs.

[§] It is often useful to visualize the skeleton as a tree.

[¶] The root point of the configuration is constrained to lie on one of the links; we refer to this link as the *root link*.



$$\mathbf{q} = (x_{root}, y_{root}, \theta_{orient}, \theta_1, ..., \theta_9)$$

Figure 3.7: Composition of $\mathbf{q}$, for a particular choice of *root*

3.4 Environment description

The space through which the model is to navigate is the *workspace* $\mathcal{W}$. In our work we have found it necessary to augment this concept in a number of places with additional information. We shall refer to this augmented workspace as the *world*. Specifically, the *world* is the rectangular region of space in which the navigation task is to be solved, together with the description and relevant characteristics of the obstacles found within. We will use the $\mathcal{W}$ notation for both, workspace and world, except in situations where this results in ambiguity, in which case we will use the explicit terms.

For other reasons which will become apparent later, our planner works with a discretized approximation of the workspace, which we shall refer to as the *bitmap*. As the name implies, the bitmap is a map of binary values spanning the rectangular region of $\mathcal{W}$, where each bit is set to false if the the corresponding cell lies in free-space, and true if occupied by an obstacle. There are two ways of judging cell occupancy: a cell can be considered to be occupied when the overlap between the cell and $\mathcal{B}$, denoted by λ, is either $\lambda > 0$ or when $\lambda > 0.5$ of the cell area. The former would be preferable for physical applications where collisions must be avoided at all costs.

The latter is the one that we use in our implementation. It allows us to use coarser grids at the expense of occasional shallow limb-obstacle inter-penetration. Figure 3.8 illustrates the relationship between the world and its bitmap.

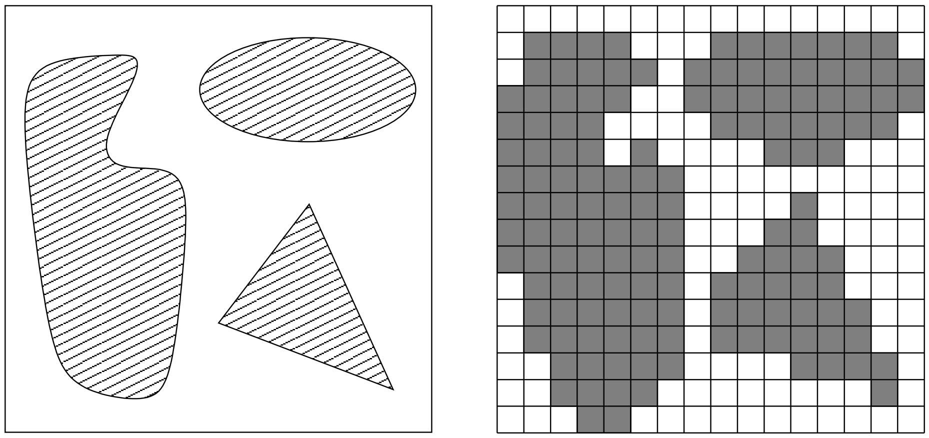


Figure 3.8: $\mathcal{W}$ and it's *bitmap*, at a resolution of 16x16

Constructing a *bitmap* requires the choice of an appropriate resolution. This is a function of the desired quality of motion, its accuracy, as well as the relative size of the world to the size of the character. For a workspace and character dimensions such as those in Figure 3.1, a reasonable resolution would be 64x64. For rectangular workspaces the key issue is to keep an aspect ratio of 1.

There is a compromise involved in choosing the bitmap resolution. Fine grids lead to large memory requirements and can greatly slow down the progress of the planner. Coarse grids can fail to capture the detail of the terrain for the $\lambda > 0.5$ cell occupancy case, causing the character to occasionally move limbs through narrow obstacles which have not registered on the bitmap. An additional problem with coarse grids is that footholds and handholds will appear with increasing frequency to be completely engulfed by obstacles on whose surface they lie. For optimum efficiency the resolution should be therefore chosen as low as possible, without incurring the coarse grid penalties.

3.5 $\mathcal{P}$ and the distance map

One of the primary functions of the bitmap described in the previous section is to aid in the construction of $\mathcal{P}$, a potential field over the configuration space $\mathcal{C}$. The planning process uses this field's gradient to determine the direction in which it should proceed. Instead of building the complete field ahead of time, it is convenient to compute particular values of it on the fly, as needed. We use partial configuration information to measure the character's linear distance to the goal, and this is then used as the value of $\mathcal{P}$. This partial configuration specification is a projection of the full configuration, and in our case it is chosen to be the center of mass(COM).

A particular value of $\mathcal{P}$ is computed by first building the *distance map*, which is constructed using the "wavefront expansion" algorithm described in [LPW91]. The distance map is discretized over the workspace, much like the bitmap. Every cell in this map denotes the L^1, or "Manhattan" distance of the character's COM to the *goal* cell, the one occupied by the COM in $\mathbf{q}_{finish}$. This distance measures the shortest path one would have to travel, in terms of cells traversed, in order to reach the goal cell by either vertical or horizontal single-cell hops through the free-space. The value of $\mathcal{P}$ for a particular $\mathbf{q}$ is then equal to the value of the cell in distance map in which its COM lies.

Figure 3.9 shows an example of a distance map as well as how it is built. The construction process consists of assigning the goal cell a value of zero, and then assigning a value one greater to all the unvisited 1-neighbours,[‖] recursively and in a breadth-first fashion, but only for cells which lie in free-space. Equivalently, if one considers the discretized map as a graph of free-space cells, with adjacent cells connected by arcs (only the 1-neighbours) then the generation of the distance map is analogous to Dijkstra's "shortest path" algorithm, using arc weights of 1.

[‖]A *1-neighbour* is one that differs only by a single coordinate; in like fashion, an *n-neighbour* is one which can differ in all 'n' coordinates.

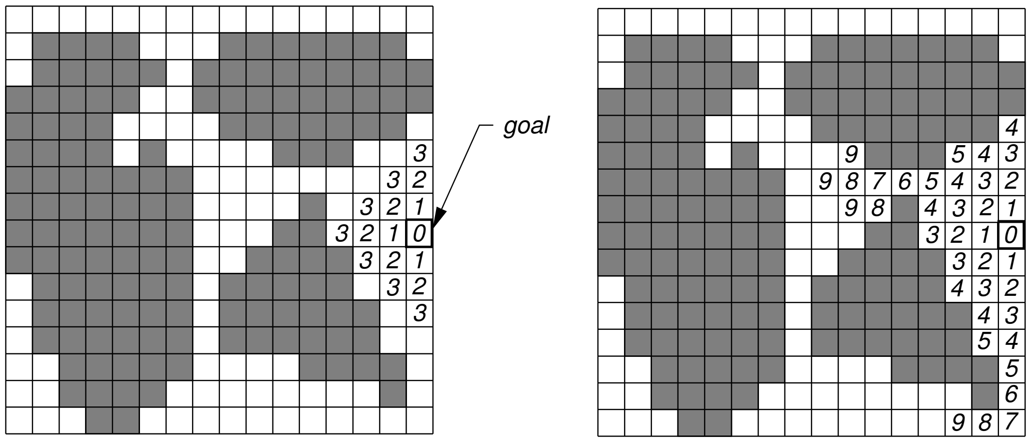


Figure 3.9: The wave propagation process for a distance map; the state of the map after three, and then nine iterations

3.6 A precursory planner

3.6.1 Gradient descent

Given the bitmap, distance map, and $\mathcal{P}$ as building blocks, we now examine the algorithm for a simplified planner. The backbone of this planner is an optimization technique known as the *gradient descent*, which is augmented by several refinements borrowed from previous work [BL91, LPW91].

Intuitively, this process consists of iteratively improving the character's posture using small variations in its configuration until the destination is reached. At each step of the process the variation is chosen such that $\mathcal{P}(\mathbf{q})$ is minimized. Formally, a gradient descent consists of iteratively evaluating the subject's position in the potential field, computing the corresponding gradient ∇, and advancing in the direction of the steepest descent, namely $-\nabla$. We stop this process once the global minimum is reached, which corresponds to the goal configuration. Figure 3.10 provides a visual abstraction of the gradient descent algorithm, using an improvised three dimensional $\mathcal{P}$.

The choice of step size can vary over time. At any point in time during the descent the character is allowed to alter any of the coordinates in $\mathbf{q}$ forward or backward by

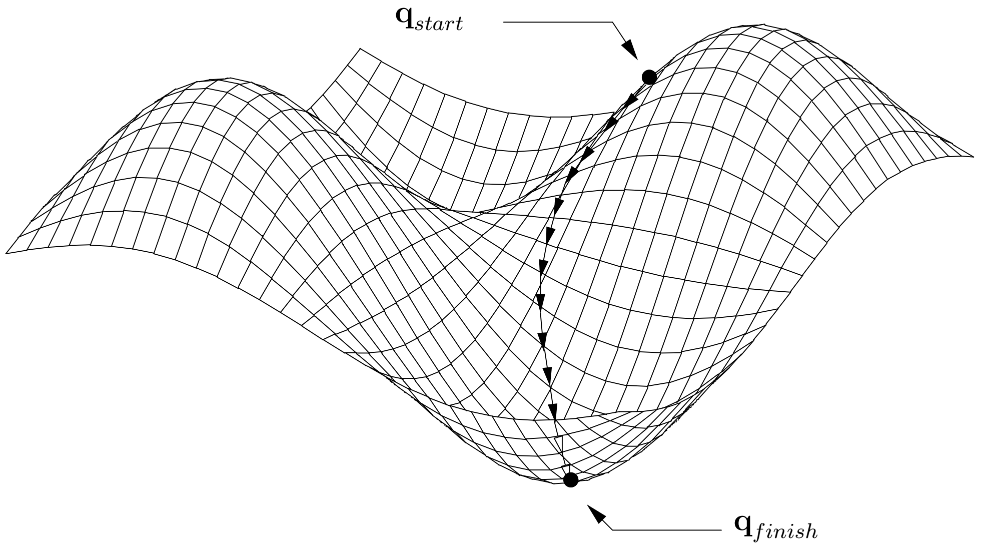


Figure 3.10: Gradient descent in an imaginary two dimensional potential field

a set amount, which we will refer to as Δq_j, where j denotes one of the coordinates of $\mathbf{q}$. Usually this amount is calculated to be such that all points on the character traverse at most a distance of 2 cells in the bitmap. The purpose of this is to make any transition between two consecutive configurations, $\mathbf{q}_n$ and $\mathbf{q}_{n+1}$, small enough so that we can be assured that this will not result in parts of the character traversing through obstacles. The reader is referred to [BL91] for a more detailed explanation and the calculations involved.

The direct application of the above algorithm to the task of manipulating the character with the jet-pack does not work very well for several reasons. The first problem comes from our formulation of $\mathcal{P}$. The gradient descent will stop when $\mathcal{P}(\mathbf{q}) = 0$, i.e. when the location of the character's center of mass matches the one that would result from the character being in the $\mathbf{q}_{finish}$ configuration. This does **not** guarantee though that $\mathbf{q} = \mathbf{q}_{finish}$ when we finish the descent. A practical solution to this is to simply do a linear interpolation from $\mathbf{q}$ to $\mathbf{q}_{finish}$.

Another problem that we have to contend with is the high dimensionality of $\mathcal{C}$. In an ideal situation, one would compute a gradient and follow it. However, for high dimensional problems with potential non-linearities, the random sampling approach is in fact one of the few workable solutions [LPW91,BL91]. Here we evaluate $\mathcal{P}(\mathbf{q})$ for a number of randomly selected neighbouring configurations and choose the one with the lowest potential as the successor. Although the resulting path will not strictly follow the gradient, it typically provides a suitably close approximation.

The remaining issue that needs to be addressed is the classical problem of the local minima. Due to the imposed constraint that the character's body is not to penetrate obstacles, our gradient descent is limited to inspecting and using only configurations which lie in $\mathcal{C}_{free}$. As the planner follows the negative gradient it will occasionally come to a point where this constraint invalidates the use of any of the neighbouring configurations that have a lower potential value. Since any further progress is impossible without violating the "no-collision" constraint, this constitutes a local minimum.

3.6.2 Random walks

To continue solving after reaching a local minimum, our planner needs to resort to any of the local minimum escaping methods. The approach that we take is the random walk, as detailed in [LPW91, BL91]: we apply Brownian motion to the character's configuration for a prespecified duration. The random walk might not be successful in escaping the minimum on the first attempt so it might have to be performed a number of times. For a thorough discussion of Brownian motion in the context of RPP we refer the reader to [LPW91,BL91]. Our implementation of the random walk is as follows: at each step of the walk each coordinate j of the current configuration has a uniform chance $(\frac{1}{3})$ of being either increased by Δq_j, decreased by the same, or left alone. If the resulting configuration results in a collision with $\mathcal{B}$ then we discard this latest $\mathbf{q}$ and try again.

In the case of deep local minima this tactic can sometimes still prove ineffective. These can occur in constrained environments where the inter-obstacle distances are smaller than the subject being animated, as in the case of a deep cave with a small aperture at the back of the cavern, through to the other side of the mountain. Although in theory one could extend the length of the random walks with the expectation that this would enhance the chances of success of escape, this probability does not scale linearly with respect to the walk length. Hence we need a better, more efficient way to deal with the problem. We therefore resort to *backtracking*, as outlined in the RPP algorithm [BL91, LPW91].

3.6.3 Backtracking

Backtracking consists of restarting the planner at an earlier point along the solution trajectory we have already determined. The choice is performed randomly with a uniform distribution over the domain of all randomly generated configurations in the current solution, i.e. ones derived from a previous random walk. The rationale for choosing from these is that the complement of this set consists of configurations generated by a gradient descent; these are more likely to lie near local minima as each gradient descent unfailingly ends in one. By choosing from the randomly generated set we therefore increase the probability of a successful escape. If no random walks have been yet undertaken, we use the whole solution as the domain for randomly choosing a point.

Once the character is placed in the configuration as chosen by the backtracking code, a new random walk is performed. The purpose of this is that hopefully the above process will place the character on an alternative down slope of $\mathcal{P}$, one which will ultimately lead to a different path taken towards the goal. The probability of difficult-to-escape local minima is a function of the frequency of sub-character sized inter-obstacle distances, as well as the degree of environment confinement.

Figure 3.11 illustrates a scenario where backtracking might be likely. The character starts at point #1. He flies towards the cave, passing through point #2, and ends up stuck in a deep local minimum at point #3. A number of random walks followed by gradient descents still do not yield any progress. The solver then backtracks, randomly choosing point #2. A random walk is performed which happens to succeed in escaping the local minimum of the cave (point #4). The character continues using gradient descent until he arrives at the finish, point #5.

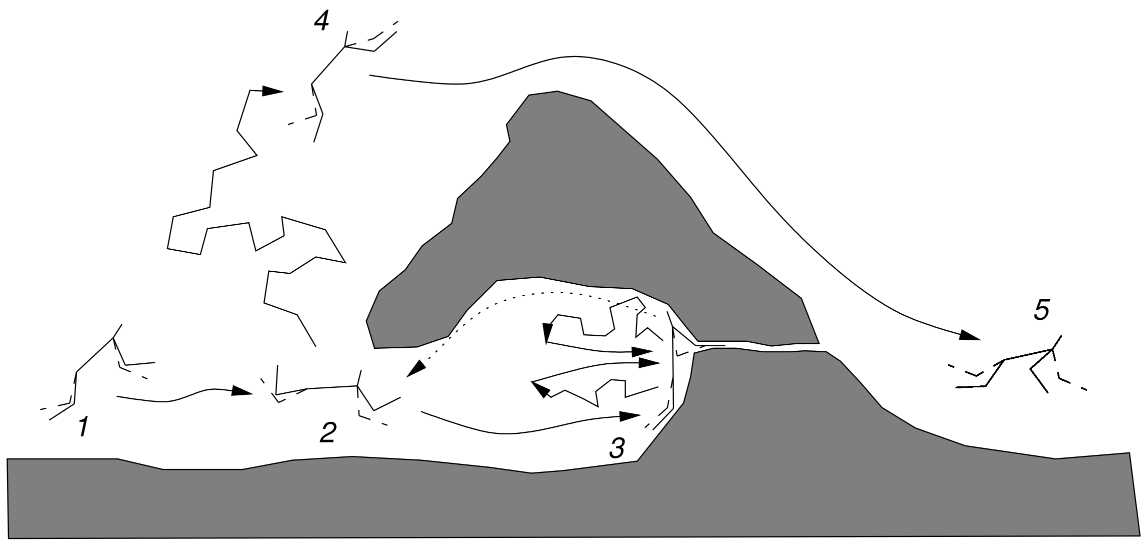


Figure 3.11: Backtracking example, using a jet-pack (not shown). 1) start; 2) intermediate configuration; 3) stuck in a deep local minima, despite a number of random walks; we backtrack to 2 and perform a random walk there; 4) the walk succeeds in escaping the minimum; 5) finish

3.6.4 Planner summary

We summarize the operation of our precursory planner using pseudocode in the following figures. Figure 3.12 shows the pseudocode for our now complete simple planner, with some supporting functions shown in Figure 3.13. Table 3.1 describes the remaining subroutines not covered by the two figures.

The simple planner that we have presented in this chapter is capable of solving the free-space motion problem for a human character in arbitrary environments. Its solving process consists of an alternating sequence of gradient descents and random walks, with an occasional backtracking operation. In the following chapter we adapt and extend this planner to deal with locomotion modes other than flight.

```
// the 'precursory' planner
function simp_plan($\mathcal{A}$, $\mathcal{W}$, $q_{start}$, $q_{finish}$) {
    dist_map ← preprocess($\mathcal{A}$, $\mathcal{W}$, $q_{finish}$);
    $\tau_{sol}$ ← $q_{start}$;
    $q_{cur}$ ← $q_{start}$;

    // while not at the global minimum
    while $\mathcal{P}$($q_{cur}$, dist_map) > 0 {
        $\mathcal{P}_{cur}$ ← $\mathcal{P}$($q_{cur}$, dist_map);

        q ← grad_desc($\mathcal{A}$, $q_{cur}$, dist_map);
        if q
            $\tau_{sol}$ ← $\tau_{sol}$, q;
        else {
            // local minimum hit; try some random walks
            for i = 1 to max_walks {
                $\tau_i$ ← rand_walk($q_{cur}$, $\mathcal{P}$, max_walk_len);
                if $\mathcal{P}$(last($\tau_i$), dist_map) < $\mathcal{P}_{cur}$ {
                    walk_success ← true;
                    break;
                }
            }
            if walk_success
                $\tau_{sol}$ ← $\tau_{sol}$, $\tau_i$;
            else {
                // random walks failed; backtrack
                $\tau_{sol}$ ← backtrack($\tau_{sol}$);
                $\tau_{sol}$ ← $\tau_{sol}$, blind_rand_walk(last($\tau_{sol}$), max_walk_len);
            }
        $q_{cur}$ ← last($\tau_{sol}$);
        }
    }
    // solution found; now smoothly interpolate to final q
    $\tau_{sol}$ ← $\tau_{sol}$, linear_interp($q_{cur}$, $q_{finish}$);
    return $\tau_{sol}$;
}
```

Figure 3.12: Pseudocode for the “precursory” planner

```
// configuration space potential
function $\mathcal{P}$(q$_{cur}$, dist_map) {
    (x$_{COM}$, y$_{COM}$) $\leftarrow$ cent_of_mass(q$_{cur}$);
    return dist_map[x$_{COM}$, y$_{COM}$];
}

// single step of gradient descent
function grad_desc($\mathcal{A}$, q$_{cur}$, dist_map) {
    $\mathcal{P}_{cur}$ $\leftarrow$ $\mathcal{P}$(q$_{cur}$, dist_map);
    $\mathcal{P}_{min}$ $\leftarrow$ $\infty$;

    // stochastic sampling of configuration neighbourhood
    for i = 1 to max_neighs {
        q$_i$ $\leftarrow$ rand_neigh(q$_{cur}$);
        if $\mathcal{P}$(q$_i$, dist_map) < $\mathcal{P}_{min}$ {
             q$_{min}$ $\leftarrow$ q$_i$;
             $\mathcal{P}_{min}$ $\leftarrow$ $\mathcal{P}$(q$_i$, dist_map);
        }
    }
    if $\mathcal{P}_{min}$ $<$ $\mathcal{P}_{cur}$
        return q$_{min}$;
    else
        return $\emptyset$;
}

// random walk over a given potential field;
// stops early if it reaches a q of lower potential than q$_{cur}$
function rand_walk(q$_{cur}$, pot_field, max_walk_len) {
    $\tau$ $\leftarrow$ $\emptyset$;
    q $\leftarrow$ q$_{cur}$;
    for i = 1 to max_walk_len {
        q $\leftarrow$ rand_neigh(q);
        $\tau$ $\leftarrow$ $\tau$, q;
        if pot_field(q) < pot_field(q$_{cur}$)
            return $\tau$;
    }
    return $\tau$;
}
```

Figure 3.13: Pseudocode of supporting functions

`preprocess()`	Given the description of the character and the workspace, as well as the goal configuration, this routine computes the bitmap and the distance map.
`last()`	This routine returns the last configuration of the supplied path.
`backtrack()`	Given a path, chooses a point using the method described in the text and discards the portion from that point to the end.
`blind_rand_walk()`	This routine returns a random walk path of the specified length, starting at the required configuration. It differs from `rand_walk()` in that it does not check any potential field values for the purposes of a premature termination of walk.
`linear_interp()`	This routine performs a linear interpolation between the first and second configuration it has been handed, and returns the resulting path.
`cent_of_mass()`	Returns the center of mass of the character for a particular configuration.
`rand_neigh()`	Returns a random neighbour of the supplied configuration.

Table 3.1: Description of the remaining "precursory" planner functions

Chapter 4

The Path Planner

In order to apply the path-planning algorithm to character animation, it needs to be augmented in several ways. These additions are the core contributions of our thesis and are described in detail in this chapter

The precursory planner is deficient in several ways. The primary inadequacy is the lack of the ability to come in contact with obstacles; human locomotion requires us to apply forces to counter the effect of gravity as well as to propel us forward. The secondary quality missing from the simplified planner is the notion of aesthetic and comfortable postures. Our character needs to use hand and foot holds, and do so in a manner befitting typical human motion.

In this chapter we will first define the concept of *grasp points* and how they influence the character's interaction with its environment. We then proceed to look at the application of heuristics in order to infuse the character's motion with the desired natural form. A second means of refining the synthesized motions involves the use of a smoothing post-process. Finally, we describe the use of a locomotion-mode finite-state machine that is used to endow the character with an arbitrary number of distinct behaviours or locomotion modes.

4.1 Grasp points

Realistic locomotion of any kind, whether walking, climbing, or crawling, involves contact with the environment. *Grasp points* located in the environment are introduced in our algorithm as an explicit means of specifying allowable points of contact. They may be specified by the animator, or may be created by an automatic process. Grasp points are not an absolute specification of where a hand or foot will be at some point during an animation — each one represents a potential contact that may or may-not be used by a synthesized motion. Grasp points have the important property that they reduce the number of ways in which a character can interact with its environment, which helps to shape a complex motion planning problem into one which is more tractable.

Koga et al. [KKKL94] also make use of grasps but their use of the term differs from ours. Like us they limit all grasping to a predefined set of allowed grasps, called the *grasp set*, but in their context, a *grasp* describes which arms of the robot make contact with the object being manipulated, as well as at which points. In contrast, using our terminology, a character making contact with the environment in n points has n grasps, not 1.

We define three types of grasp points: *pendent*, *load-bearing*, and *hybrid*. Pendent grasp points represent points which the character can grab with his hands and hang from. Load-bearing grasps are points on which the character can stand and are therefore the most common type of grasp point. Finally, hybrid grasp points allow for both types of contact. Figure 4.1 demonstrates representative examples of the grasp point types.

The classification of grasp points proves to be useful in several ways. First, it immediately eliminates wildly unrealistic scenarios, such as a character deciding to walk on its hands.[†] Secondly , grasp point properties provide useful information to the locomotion-mode finite state machine, discussed later, about the immediate environment. This is more expedient than directly analyzing the surrounding obstacles.

[†]...unless this is desired; these types of decisions are handled by the LFSM described in §4.7.

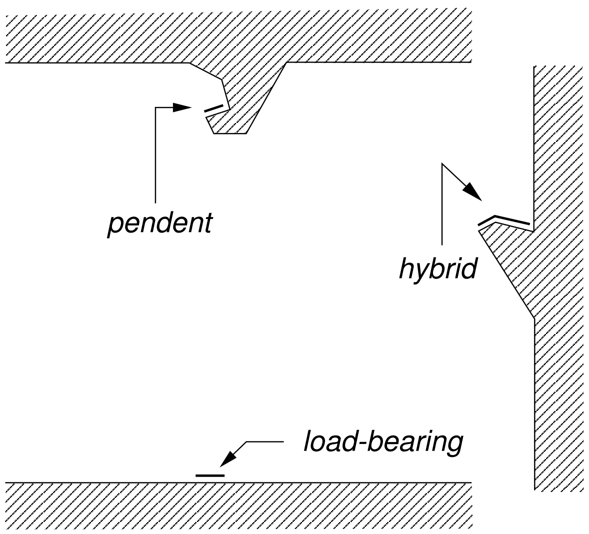


Figure 4.1: Classification of grasp points

Finally, by having these different types of grasp points we give the animator a bit more control over the planning process by allowing him to restrict the possible grasp types at different points in the environment.

Although in this work we assume and implement a system where the grasp points are hand positioned by the animator, it is foreseeable that this process could be automated. A simple method to accomplish this would be to distribute a number of grasp points equidistantly over all surface segments, and to select the point type based on the surface orientation. Figure 4.2 illustrates the scheme used for type selection. Specifically, if we define $\vec{u}$ to be the up vector, $\vec{u} = (0, 1)$, and $\vec{N}$ the surface normal, then the grasp point type choice is made as follows:

- load-bearing: $\vec{u} \cdot \vec{N} > \frac{1}{\sqrt{2}}$
- pendent: $\vec{u} \cdot \vec{N} < -\frac{1}{\sqrt{2}}$
- hybrid: $-\frac{1}{\sqrt{2}} > \vec{u} \cdot \vec{N} > \frac{1}{\sqrt{2}}$

During locomotion, the hands and feet of the skeleton attach and detach from the grasp points as needed. This allows a rich set of motions to be generated, although grasp points do not directly allow for certain motions, such as sitting, because other

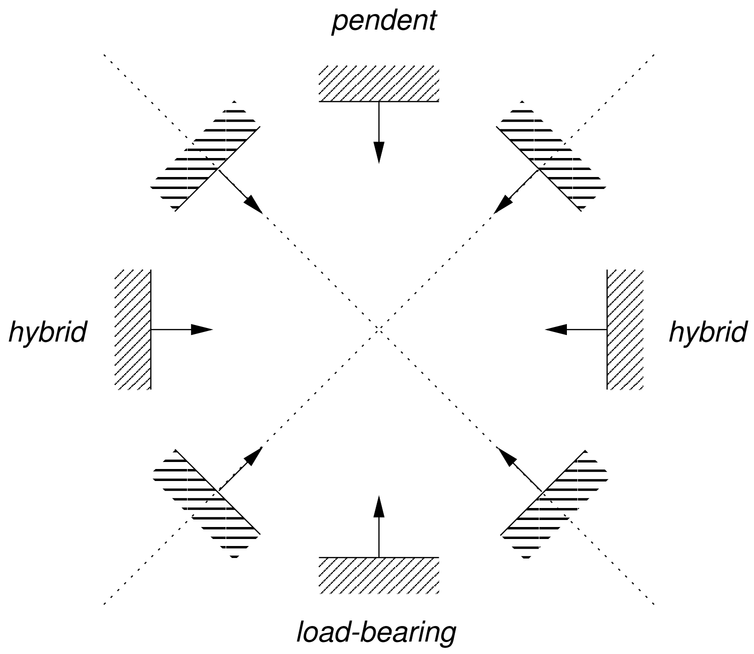


Figure 4.2: Selection of grasp point type for populating a surface based on its orientation

points of contact are involved. In such circumstances it is nonetheless usually possible to adjust the planner to achieve the same result through creative use of the *heuristics* (see §4.3).

4.2 Local planning

The algorithm described in the previous chapter focused on global motion planning. In the following two sections, we consider planning problems of a more local nature, focusing on the details required for the motion to have appropriately human characteristics. Although we use walking as the example motion, one should keep in mind that the methods described below are applicable to all classes of movement.

The use of recognizable gaits is an important characteristic of natural locomotion. A first approximation to walking can be achieved by requiring that one foot be in contact with the ground at all times. This is introduced into the motion planning algorithm by constructing the character's potential movements using the grasp points.

Foot contact is initiated by *grasping*, attaching the foot to a suitable grasp point. Figure 4.3 illustrates the walking process. The decision to switch the grasps is made when the airborne foot has a suitable grasp point within reach, one that is slightly ahead. Since it is highly unlikely that the gradient descent will produce configurations in which the free foot conveniently passes through the desired grasp points, we employ rudimentary inverse kinematics on just the two links of the free limb to place it's extremity at the required position, if possible.

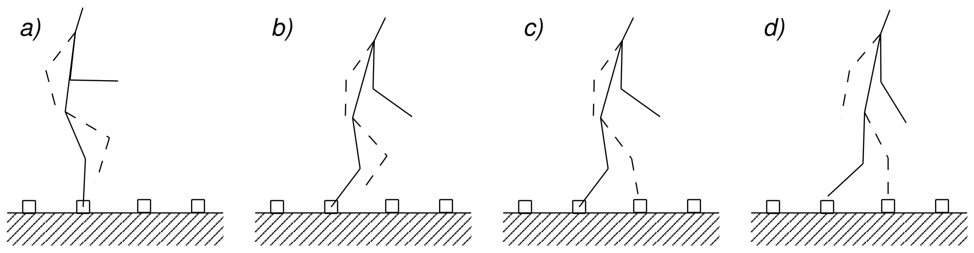


Figure 4.3: The walking cycle; a) starting posture; b) after a few gradient descent steps; c) IK used to reach the next grasp point; d) grasp switched to other leg and gradient descent continued

The representation of the character's configuration, $\mathbf{q}$, is adapted according to the current grasp point(s). The root is chosen to be the grasping extremity, and the random neighbour generating routine, `rand_neigh()`(see Figure 3.13 and Table 3.1), is modified to only vary the configuration in DOFs other than x_{root} and y_{root}. This allows for easy enforcement of the grasp constraint, as changes to the joint angles will not affect the position of the constrained point on the root link.

This method requires a slight extension though to handle configurations having more than one grasp. In this case we set the root to be one of the grasps, and `rand_neigh()` is augmented to use rudimentary, two-link inverse kinematics to bring the remaining grasping extremities to their corresponding grasp points in the new configuration. Figure 4.4 gives the pseudocode of the modified `rand_neigh()`. In this pseudocode fragment, the function `simp_IK()` takes as parameters an extremity of the character as well as two configurations, and returns the result of applying simple inverse kinematics on the the first configuration to obtain the same extremity position as in the second.

```
// the ‘‘grasp-aware’’ version of rand_neigh()
function rand_neigh_wgrasps($q_{cur}$, grasps) {
    // guaranteed to terminate: in worst case it will find $q_{cur}$
    forever {
        // call the older version to get a random neighbour
        q ← rand_neigh($q_{cur}$);
        q[0] ← $q_{cur}$[0];
        q[1] ← $q_{cur}$[1];

        foreach grasp in grasps {
            if $\mathcal{X}$(q, $p_{grasp}$) ≠ $\mathcal{X}$($q_{cur}$, $p_{grasp}$) {
                q ← simp_IK(grasp, q, $q_{cur}$);
                if ¬q
                    // couldn’t reinstitute grasp;
                    // choose another random q
                    continue;
            }
        }

        // all the grasps have been restored
        return q;
    }
}
```

Figure 4.4: Pseudocode for `rand_neigh_wgrasps()`, the “grasp-aware” version of `rand_neigh()`

These preceding modifications result in planned motions that advance the character towards the target configuration, $\mathbf{q}_{finish}$, but the motion remains largely unnatural. It is reminiscent of a shaky but unusually nimble and acrobatic person walking against a strong wind. The most glaring problem of the motion planner as described thus far is an absolute disregard for balance and gravity. Figure 4.5 illustrates the dominant postures assumed by the motion planner.

4.3 Heuristics system

In order to achieve more natural motions, we employ a system of heuristics to guide the character towards the desired postures at key points in the solution. We define

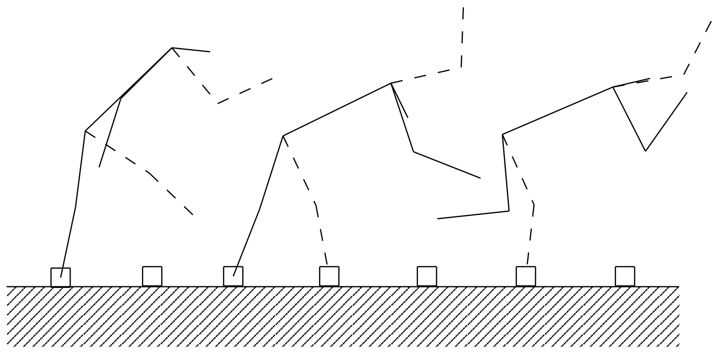


Figure 4.5: Typical postures for a character without the heuristics system

these key points to be the time instances at which any change of grasp occurs. Each heuristic analyzes the character's posture and provides feedback on one particular property or characteristic, returning a value ranging from 1 to $+\infty$, 1 being optimal and $+\infty$ being unacceptable. For a complete discussion and listing of the heuristics that we have implemented please see §5.2. Multiple heuristics are combined into a single discomfort function $\mathcal{D}$ in a multiplicative fashion, as shown in 4.6. This function is similar to $\mathcal{P}$ in that it is also a potential field spanning the configuration space $\mathcal{C}$. To correct a character's posture we perform a gradient descent through $\mathcal{D}$, with the effect of minimizing the character's discomfort level.

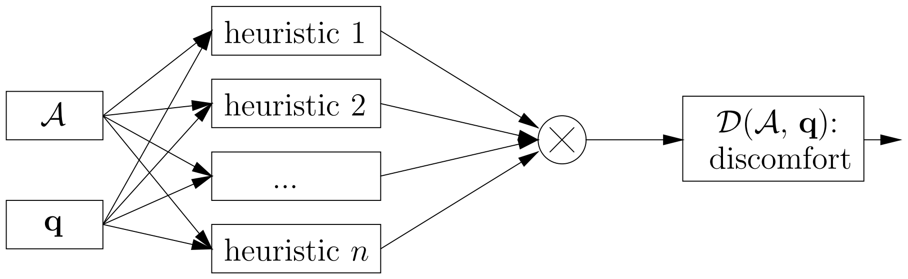


Figure 4.6: Computation of the discomfort function

The optimization of the discomfort function suffers from the same problem as $\mathcal{P}$, namely the existence of local minima. To escape these we therefore use the same approach as in the last chapter: we apply a brief Brownian motion and attempt to continue with the descent. An additional optimization we perform is to halt the gradient descent process before the desired minimum is reached, by stopping when a prespecified comfort threshold is passed. As the stochastic gradient descent nears a minimum, it's progress slows down as lower potential configurations are harder to find, while the visual improvement in comfort is less and less noticeable. The premature halting of the descent therefore saves the planner from unnecessary computation.

Figure 4.7 shows the pseudocode for the gradient descent through the *discomfort space*, while Figure 4.8 presents the complete posture correction routine.

4.4 Interaction between components

Global-local interaction lies at the heart of human motion planning in constrained environments. The best global solutions may require temporary local discomfort. However it is not clear how best to explore this necessary compromise, as the global solution is not known in advance.

It might be useful at this point to step back and analyze how the two components, the global and the local, interact and relate to each other. To reiterate, the global component of our planner consists of the gradient descent through the configuration space $\mathcal{C}$. The local components are twofold: the constraint requiring a grasp be present, and the heuristics, which are used to reposition the character at key instances in time. Note, however, that the grasp constraint is directly integrated into the global planning gradient descent.

To understand how the grasp constraint affects the operation of the gradient descent it is helpful to visualize the addition and removal of grasps as constantly adding and subtracting available regions from the potential field that the global part is working with. At any particular time, the gradient descent does not have the whole $\mathcal{C}$ available to it since the character is not completely free to assume any configuration due to the presence of the grasp constraints. In fact, only a small subregion of $\mathcal{C}$ can be reached. When a new grasp is made and an old one released, this subsequently makes a cluster of new configurations attainable since the character can reach further. At the same time, some of the previous configurations are now unattainable since the character has essentially moved away from them.

```
// the discomfort gradient descent
// returns the resulting path
function discomf_grad_desc(q) {
    $\tau \leftarrow \emptyset$;
    discomf $\leftarrow$ $\mathcal{D}$(q);

    // we return from within when progress stops
    // or the comfort threshold has been passed
    forever {
        // sample neighbourhood
        for i=1 to max_samples {
            q $\leftarrow$ rand_neigh_wgrasps(q);
            if $\mathcal{D}$(q) < discomf
                break;
        }
        if $\mathcal{D}$(q) < discomf {
            // found a q with lower discomfort
            $\tau \leftarrow \tau$, q;
            discomf $\leftarrow$ $\mathcal{D}$(q);
            if discomf < comfort_threshold
                return $\tau$;
        }
        else
            // could not get any more comfortable; done
            return $\tau$;
    }
}
```

Figure 4.7: Pseudocode for the discomfort gradient descent

```
// routine to correct character's posture
// returns the correcting path
function get_comfy(q) {
    $\tau$ $\leftarrow$ discomf_grad_desc(q);

    // will return from within when progress stops
    // or when comfort threshold passed
    forever {
        if $\mathcal{D}$(q) $<$ comfort_threshold
            return $\tau$;
        cur_discomf $\leftarrow$ $\mathcal{D}$(q);

        // try to escape the current local minimum
        for attempt = 1 to max_attempts {
            $\tau_i$ $\leftarrow$ rand_walk(q, $\mathcal{D}$, max_len);
            $\tau_i$ $\leftarrow$ $\tau_i$, discomf_grad_desc(last($\tau_i$));
            if $\mathcal{D}$(last($\tau_i$)) < cur_discomf {
                $\tau$ $\leftarrow$ $\tau$, $\tau_i$;
                break;
            }
        }
        if $\mathcal{D}$(q) $\geq$ cur_discomf
            // could not get any more comfortable
            return $\tau$;
    }
}
```

Figure 4.8: Pseudocode for the complete posture correction process

The heuristic system is not as tightly coupled to the global planner as the grasp constraint, so here there is a potential for contention between the two components, in terms of directing the character. Nonetheless, even though the posture-correcting path suggested by the heuristic system may countermand some of the progress achieved by the global planner, the extent of the heuristic system's ability to modify the path is severely limited by virtue of the character being constrained by the current grasps. Unlike the extended global component, it does not have the ability to change the character's grasps.

4.5 Body trajectory smoothing

The system described thus far produces results which still have a detrimental flaw: the character's motion contains a lot of noise as a result of using stochastic sampling during gradient descent, and worse, any random walks or posture corrections are present in their entirety in the final solution. In short, the motion embodies the history of the search process used to produce it, and as a result does not exhibit the degree of anticipation required to achieve natural fluid motions. A separate process is therefore introduced in order to cull any unwanted motion segments as well as optimize the subsequent trajectory, thereby making the result more fluid. We refer to this process as a "smoothing pass", and it is carried out on the intermediate solution produced by the planner. The smoothing algorithm we present is directly borrowed from the work on RPP [BL91], with some modifications necessitated by the addition of grasps.

The algorithm consists of attempting to substitute sections of the path with linear segments. A particular substitution is only allowed if the linear path segment does not cause the character to collide with the environment at any point. Sections of diminishing length are iteratively considered for substitution, starting with the full length of the path. All subsets of length l of the path are tested before any of length less than l. Figure 4.9 represents the process visually in only two dimensions. The process is identical when applied to the 12 dimensional space of $\mathcal{C}$; the linear segments

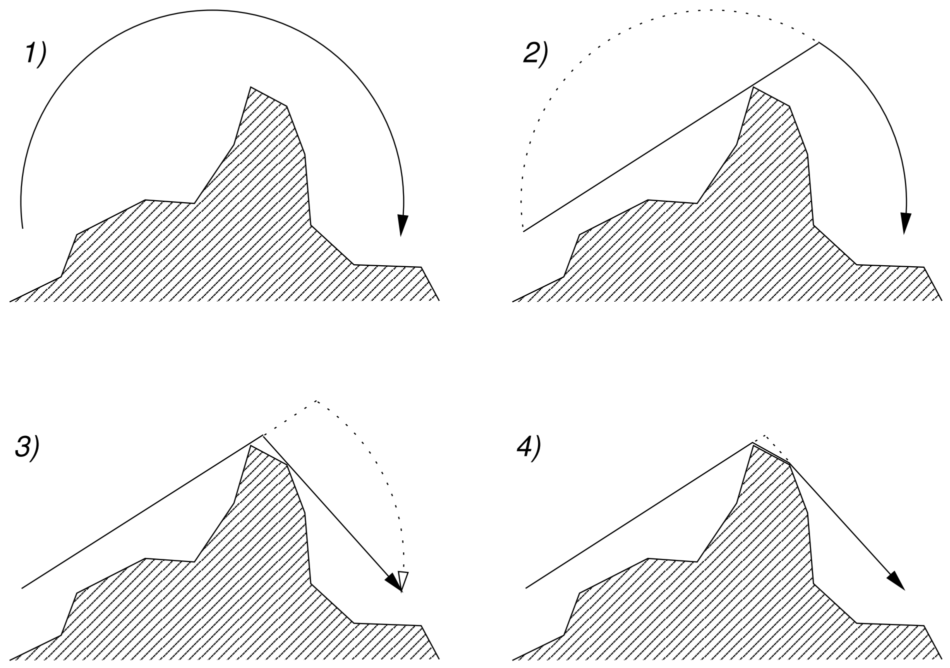


Figure 4.9: The smoothing process illustrated; only three substitutions were needed in this example

correspond to simultaneous linear interpolation of all the DOFs of $\mathbf{q}$. Figure 4.10 presents the pseudocode for the algorithm.

The original algorithm cannot be applied directly because the grasp constraints will be violated by the interpolation process. We therefore restrict the application of the smoothing algorithm to motion segments having no change of grasp configuration. In the case of a single grasp, the interpolation is easily implemented because the root position is defined to be at the grasp point, and is shared by both configurations used as endpoints of the interpolation. The case of multiple grasps requires further consideration in order to properly maintain the current set of grasps. We amend the `linear_interp()` routine so that it returns a path representing the linear interpolation between two configurations, but one where the grasping limbs have been repositioned to maintain their grasps. This is done by using rudimentary 2-link inverse kinematics at every step of the linear interpolation to reposition the grasping extremities to their proper locations. If the IK operation fails at any step, `linear_interp()` fails and returns an empty path.

```
// routine for smoothing a path
function smooth($\tau$) {
    for len=length($\tau$) to 2 {
        // NOTE: the current values of $\tau$ and len are always
        // used in the condition below
        for start=0 to (length($\tau$) - len) {
            $\tau_{lin}$ $\leftarrow$ linear_interp($\tau$[start], $\tau$[start+len]);
            if $\neg$ collides($\tau_{lin}$) and 0 < length($\tau_{lin}$) < len {
                $\tau$ $\leftarrow$ $\tau$[0..start-1], $\tau_{lin}$, $\tau$[start+len+1..length($\tau$)-1];
                if len = length($\tau$)
                    // $\tau$ got smoothed into a single line
                    return $\tau$;
                if len > length($\tau$) {
                    len = length($\tau$);
                    break;
                }
            }
        }
    }
}
```

Figure 4.10: Pseudocode for the smoothing process

4.6 Limb trajectory smoothing

Following the smoothing pass described in the previous section, we apply a separate smoothing pass independently to each limb. Prior to this filtering, limbs not actively involved in grasping at a smoothed segment's endpoints exhibit unnecessary movement. Figure 4.11 illustrates this problem. In this situation the full body trajectory smoother will proceed to smooth the motion segments between the consecutive character configurations shown in the figure. This will result in the right foot (the right leg is drawn with the solid line) to proceed to its next grasp point in a zig-zag pattern. The purpose of this secondary smoothing pass is therefore to make such limb motions fluid.

This secondary smoothing consists of subdividing the path into segments delimited by configurations in which the particular limb assumes a key pose. We define a *key pose* as one in which the limb assumes an important configuration, one which should

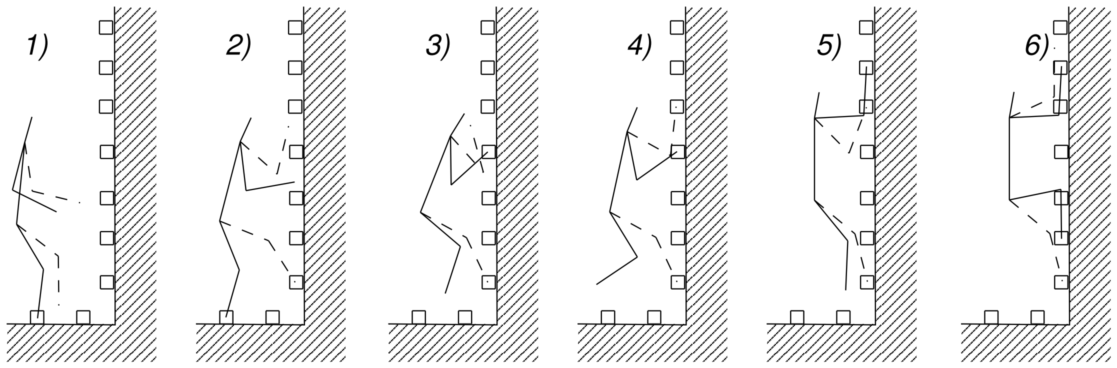


Figure 4.11: An animated climbing motion. The right and left legs are indicated with solid and dashed lines, respectively. We desire the right leg to swing smoothly from its configuration in frame 2 to that in frame 6.

be present in the final smoothed solution, and not optimized away. The trajectories associated with the joints in question are then smoothed over these segments using the method described in the previous section. This process is performed separately for each limb. The most obvious key poses are those in which the limb engages or releases grasps. Other key poses may be introduced by a particular implementation as needed. Using Figure 4.11 as an example, the right leg, drawn with the solid line, assumes key poses in frames 2 and 6, while the right arm, also drawn with the solid line, does so in frames 3 and 5. Figure 4.12 illustrates this limb smoothing process visually.

Generating arm motion for walking requires special attention. This is achieved by employing a heuristic which makes the arms track the motion of the legs, using the observation of footfalls to mimic the typical counter-balancing action of the arms (see §5.3.1). In order to smooth the arm swing motion on a per walking-step basis instead of it being optimized into a single slow arm swing the duration of the full solution, we need to mark the arms' configurations as "key" poses whenever the supporting foot is changed.

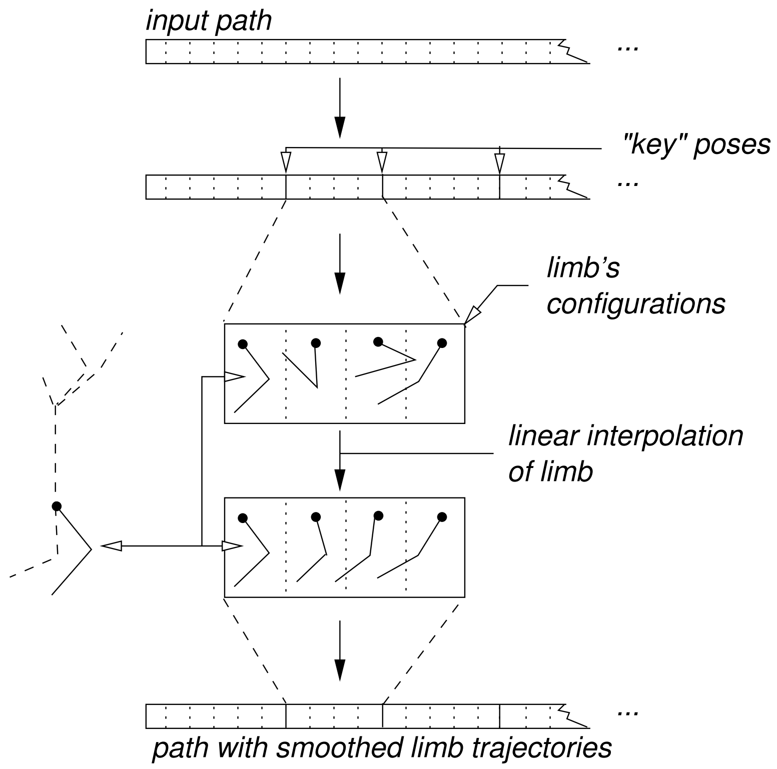

input path
...
"key" poses
...
limb's configurations
linear interpolation of limb
...
path with smoothed limb trajectories

Figure 4.12: The limb smoothing process. Joints associated with the limb are linearly interpolated between the appropriately chosen configurations. All other configuration parameters remain unaltered.

4.7 Locomotion modes

Although our discussion of character animation has thus far centered around walking, we desire our planner to be able to make use of other modes of locomotion when appropriate, such as climbing or crawling. We implement the notion of locomotion modes through the use of a *finite state machine*(FSM). We further represent obvious preferences among locomotion modes, such as a strong preference for walking over other modes, or a preference for climbing with hands and feet over climbing with the hands alone.

Without an explicit model for locomotion modes, the character will typically progress towards its goal in a seemingly haphazard fashion, given the randomized nature of the path planner. For example, in the case of flat terrain where one would

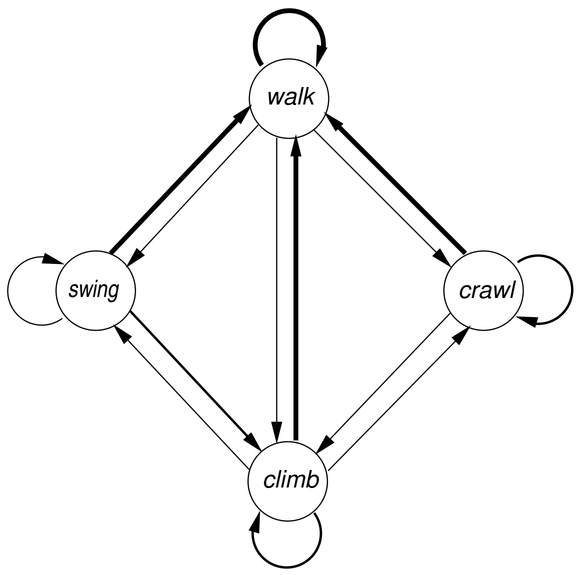


Figure 4.13: Suggested simple locomotion mode finite state machine; precedence in precondition testing is indicated by line thickness

expect the character simply to walk it's length, the subject will likely walk some portions of it normally, some others ape-like, haphazardly propping itself up with a hand or two, sometimes walking like a spider with all four limbs, and sometimes even trying to walk on its hands. This clearly differs from what we might expect in reality, where there is a strong preference to adhere to a number of fixed locomotion modes.

Figure 4.13 presents a robust and straight-forward FSM for the walking, climbing, crawling and swinging modes of locomotion, while Figure 5.4 in the next chapter represents the FSM that we have actually implemented (the disparity arises from historical reasons). In the FSM each state corresponds to a particular mode of locomotion, while the edges between the states represent the transitions between the modes. Each such edge may carry a number of preconditions that must be met for it to be traversed, and its traversal specifies a set of actions which result in the addition to or subtraction from the current solution. The preconditions typically consist of a number of geometrical constraints that must be satisfied. The resultant set of actions can be of varying degree of complexity: for simple effects it can be nothing more than a single grasp switch, while for more complex cases it can consist of a sequence of regrasps and posture corrections. Other cases may make use of unusual functionality, such as backtracking. There are neither guidelines nor limitations for the creation of edge-effects.

The added motion segments consist of the required changes to the character's posture to bring it into compliance with the dominant characteristics of the new locomotion mode. Of particular note are the self-loops in the graph. Even though their transition does not alter the mode of locomotion they provide the necessary regrasping operation which allows the character to keep advancing using that particular mode.

There is no unique choice of FSM. Although we will go into further details on our particular implementation in the next chapter, it is worthwhile to briefly expand on the details of a specific FSM state transition for illustrative purposes. The edge that we will consider is the the self-loop for the walking state. A useful precondition for it might be: "is there a grasp point within reach of the airborne foot that we could attach to, *and* is it also closer to the goal position than the other foot, *and* is it at a comfortable distance away from the other foot?" The corresponding action, executed upon the satisfaction of the precondition and hence the edge traversal, would be to bring the airborne foot down into contact with the newly found grasp point, effect a grasp, and release the other foot.

The FSM is integrated into the existing randomized path-planning framework by being consulted after each gradient descent step. That is, at each such time, all the outgoing edges from the current state are looked at, and the first one whose preconditions are met is traversed. If none of the edges' preconditions can be satisfied then no transition is executed. This is often the case, and results in a string of consecutive gradient descent steps, uninterrupted by any regrasp operations. The relative preference for the different states is instituted by the appropriate sequencing of the precondition tests. These preferences are indicated on the FSM diagram by the varying line weights, with a heavier weight denoting a higher precedence.

4.8 The complete system

Figure 4.14 illustrates our complete system. The current FSM state dictates which heuristics can be used, as well as which set of preconditions need to be checked whenever the FSM is consulted. The "precondition checker" determines which edge should be traversed and appropriately influences the "effect executor" to generate the corresponding sequence of configurations representing the state transition. Since only this component is capable of regrasping, it is the only one that draws on the heuristic system (which is employed to correct posture after most regrasp operations). The planner core, the heart of the system, decides when and in what sequence to draw on the paths provided by the random walk generator, the gradient descent step routine, and the effect executor. It also contains the backtrack mechanism, and because of this it requires the capability of modifying the FSM state. This is because if it decides to backtrack to a previous configuration which was in a different locomotion mode, the state has to be updated. Once the planner core comes up with an answer to the problem, it is further processed by the full-body smoother, and then by the limb smoother.

Figure 4.15 gives the pseudocode for the planner core, the core segment of the planner shown in the former diagram. One will notice that it is nearly identical to our "precursory" planner from the previous chapter. The differences are hilighted with a vertical bar in the first column.

To summarize, in this chapter we have described an algorithm that can enable a character to successfully navigate towards a goal in a wide range of constrained environments. Chapter 5 now provides the set of results which validate the algorithm and illustrate its proficiencies and deficiencies.

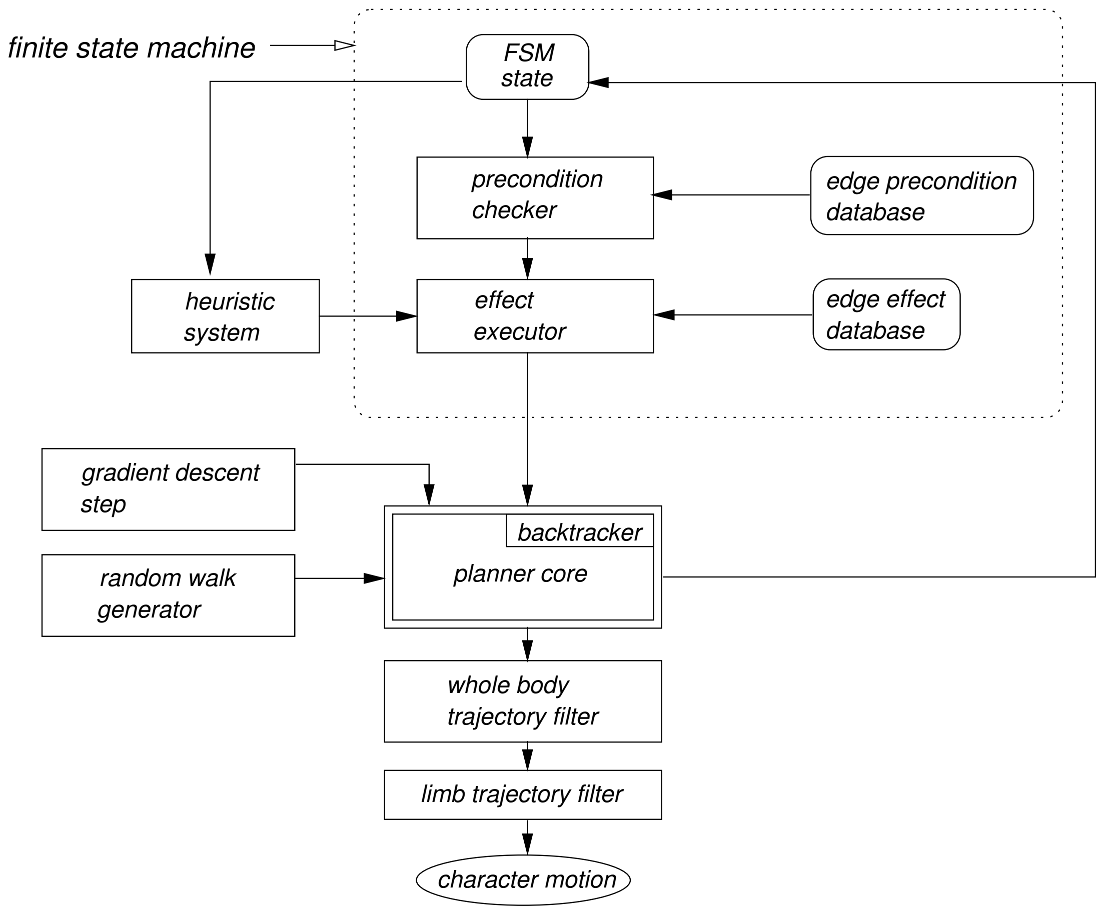


Figure 4.14: The complete system of the planner

```
// the ''heart'' of the planner
function planner_core($\mathcal{A}$, $\mathcal{W}$, $q_{start}$, $q_{finish}$) {
    dist_map ← preprocess($\mathcal{A}$, $\mathcal{W}$, $q_{finish}$);
    $\tau_{sol}$ ← $q_{start}$;
    $q_{cur}$ ← $q_{start}$;

    while $\mathcal{P}$($q_{cur}$, dist_map) > 0 {
        $\mathcal{P}_{cur}$ ← $\mathcal{P}$($q_{cur}$, dist_map);

|       $\tau$ ← effect_executor();
|       $\tau_{sol}$ ← $\tau_{sol}$, $\tau$;
        q ← grad_desc($\mathcal{A}$, $q_{cur}$, dist_map);
        if q
            $\tau_{sol}$ ← $\tau_{sol}$, q;
|       if ¬q and $\tau$ = $\emptyset$ {
            // local minimum hit
            for i = 1 to max_walks {
                $\tau_i$ ← rand_walk($q_{cur}$, $\mathcal{P}$, max_walk_len);
                if $\mathcal{P}$(last($\tau_i$), dist_map) < $\mathcal{P}_{cur}$ {
                    walk_success ← true;
                    break;
                }
            }
            if walk_success
                $\tau_{sol}$ ← $\tau_{sol}$, $\tau_i$;
            else {
                // backtrack
                $\tau_{sol}$ ← backtrack($\tau_{sol}$);
                $\tau_{sol}$ ← $\tau_{sol}$, blind_rand_walk(last($\tau_{sol}$), max_walk_len);
            }
        $q_{cur}$ ← last($\tau_{sol}$);
        }
    }
    $\tau_{sol}$ ← $\tau_{sol}$, linear_interp($q_{cur}$, $q_{finish}$);
    return $\tau_{sol}$;
}
```

Figure 4.15: The pseudocode for the “planner core”

Chapter 5

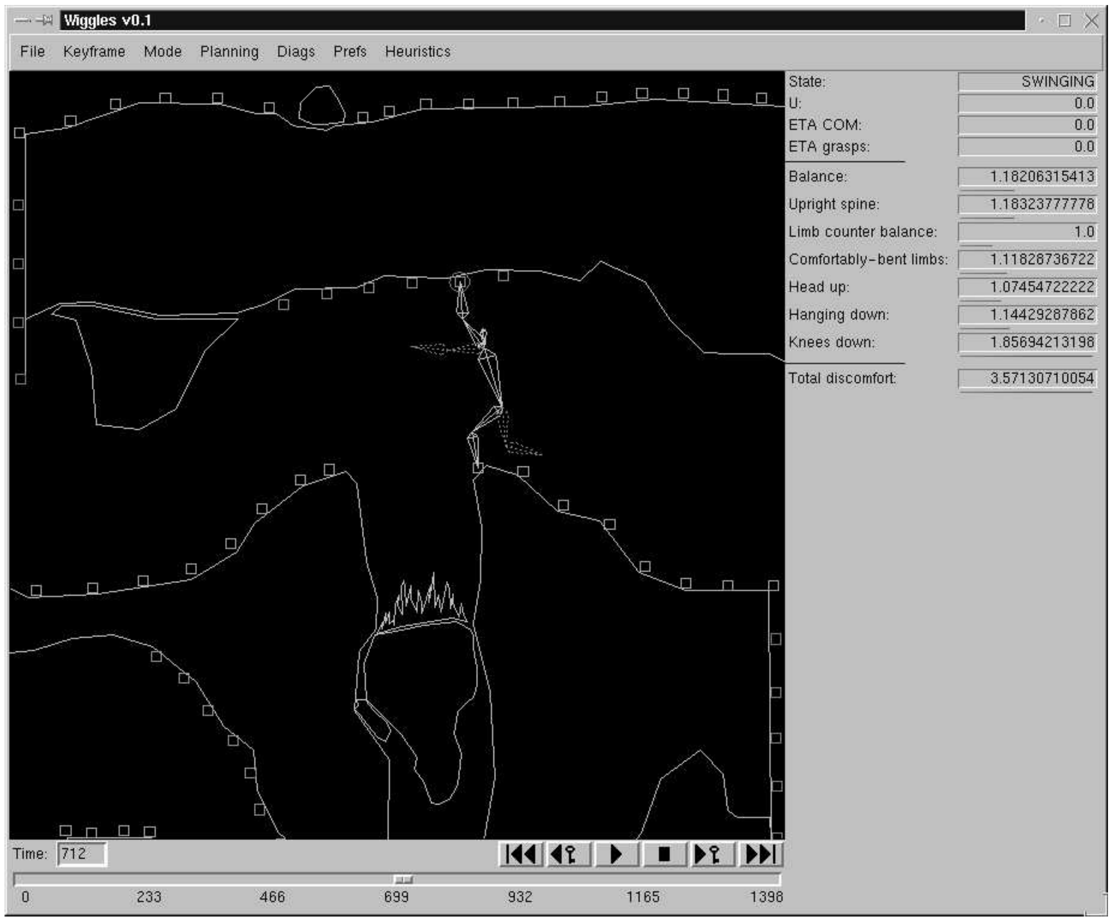
Implementation and Results

In this chapter we review our implementation of the concepts and planner discussed in the previous chapter. We start with a quick overview of the platform used, the outward appearance of the implementation, and a summary of its capabilities. We then describe the details of the implemented heuristics, as well as all the locomotion mode finite state machine(LFSM) edges. Each of these sections is further accompanied by a brief overview of the process involved in adding more of these components. This is followed by a look at some practical issues we found that need to be dealt with when implementing the planner. We then proceed to demonstrate the resulting motions produced for several environments. Finally we discuss these results, their run times, and the weaknesses encountered.

5.1 Implementation overview

We have implemented our system in C++, using egcs, on a 266MHz Pentium II machine running Linux. The system uses Tcl/Tk for the graphical user interface implementation, and the Togl[†] package in conjunction with the Mesa libraries to render the 3D visual representation of the character and its environment. Figure 5.1 shows a snapshot of the user interface.

[†]http://www.ssec.wisc.edu/~brianp/Togl.html



State: SWINGING
U: 0.0
ETA COM: 0.0
ETA grasps: 0.0
Balance: 1.18206315413
Upright spine: 1.18323777778
Limb counter balance: 1.0
Comfortably-bent limbs: 1.11828736722
Head up: 1.07454722222
Hanging down: 1.14429287862
Knees down: 1.85694213198
Total discomfort: 3.57130710054
Time: 712

Figure 5.1: The outward appearance of our planner

Our planner is capable of using walking, climbing, swinging and crawling motions, as the circumstances require. It is capable of performing the transitions between these modes. It has 7 heuristics implemented to aid with posture correction. The finite state machine itself is implemented in a somewhat different structure then presented in the previous chapter, but it is functionally equivalent. For the purposes of clarity and consistency of presentation, we will describe our FSM implementation in terms of the preconditions and effects.

5.2 Heuristics

The heuristics are used for posture correction by the effect executor component whenever it performs a grasp change, although not always. It is up to the particular effect whether it is required or not. The posture correction, as described in the previous chapter, consists of performing a gradient descent through the discomfort space $\mathcal{D}$. Only the heuristics appropriate for the current mode of locomotion are used in the calculation of $\mathcal{D}$. Table 5.1 provides a summary of the heuristics implemented, and in which modes they are employed. Their implementation is described in the following subsections.

MODE	balance	upright_spine	limb_counter.	comfy_limbs
walking	●	●	●	●
climbing				
swinging	●			●
crawling				

MODE	head_up	hang_down	knees_down
walking	●		
climbing	●	●	
swinging	●	●	
crawling	●		●

Table 5.1: Heuristic usage by different locomotion modes.

5.2.1 Common heuristics

heur$_{head\_up}$

This trivial characteristic is meant to keep the character's head lined up with its spine, but with certain flexibility. The desired effect is that of being mounted on a spring, thus allowing for some head movement when trying to clear overhangs, etc. The heuristic's value is directly proportional to the deviation of the head from its "rest" position, when it is parallel with the spine. The particular expression that we used is:

$$\mathbf{heur}_{head\_up} = 1 + \frac{|\theta_{neck}|}{180}$$

It should be noted that we refer to angles in degrees, and that we measure them as demonstrated in Figure 3.6.

$\mathbf{heur}_{comfy\_limbs}$

"Comfortable limbs" refers to giving the limbs a slightly bent, comfortable look. The *rest* position of the arms and legs has been set at 20 degrees from the fully extended orientation. The direction of this 20 degrees is determined by the joint constraints of the limb. The value of the heuristic is simply the sum of the deviation of all the limbs from this "rest" position:

$$\mathbf{heur}_{comfy\_limbs} = 1 + \frac{|\theta_1 - \theta_{rest_1}| + |\theta_2 - \theta_{rest_2}| + |\theta_3 - \theta_{rest_3}| + |\theta_4 - \theta_{rest_4}|}{720}$$

$\mathbf{heur}_{balance}$

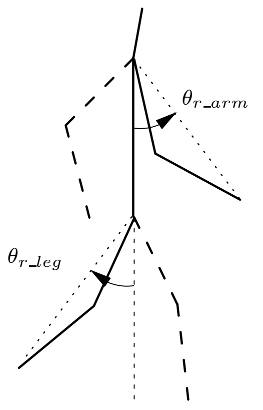
As the name implies, this heuristic reflects how balanced the character is. The value is calculated by first finding the average x coordinate of all the current grasps, and then squaring the difference between this value and the x coordinate of the center of mass of the character.

$$\mathbf{heur}_{balance} = 1 + \left\{ x_{COM} - \frac{\sum_{i=1}^{grasps} x_i}{grasps} \right\}^2$$

$\mathbf{heur}_{upright\_spine}$

The purpose of this heuristic is to keep the character's back in a vertical orientation. The value is simply taken directly from the difference between the actual angle of the spine and the vertical.

$$\mathbf{heur}_{upright\_spine} = 1 + \frac{|\angle(\vec{v}_{hip} - \vec{v}_{neck}) + 90|}{90}$$



$$\theta_{r\_arm} = \theta_{r\_leg}$$

Figure 5.2: The desired arm position during walking

5.2.2 Specialized heuristics

$\mathbf{heur}_{limb\_counter\_balance}$

As we mentioned in the previous chapter, the correct arm swing is essential to the aesthetic appeal of any generated walking motion. This heuristic serves to mimic this arm behaviour. It measures the deviation of the limbs from their required position. The arms are deemed to be in the desired orientation whenever the angle of the arm, between the vertical and the line extending from hand to shoulder, is equal in magnitude but opposite in direction to that of the *opposite* leg angle, the on between the vertical and the line extending from foot to hip. Figure 5.2 illustrates this.

$$\mathbf{heur}_{limb\_counter\_balance} = 1 + \frac{|\theta_{l\_arm} + \theta_{l\_leg}| + |\theta_{r\_arm} + \theta_{r\_leg}|}{90}$$

$\mathbf{heur}_{hang\_down}$

This heuristic is used to give the impression of the body being pulled down by gravity. It is implemented by comparing the vertical distance between the highest grasping hand and the neck to the full length of the whole arm. The smaller the difference, the lower the heuristic value. A similar calculation is done for the distance between the average position of the two feet and the hip. Using this heuristic the discomfort gradient descent will tend to favour poses in which the body extends

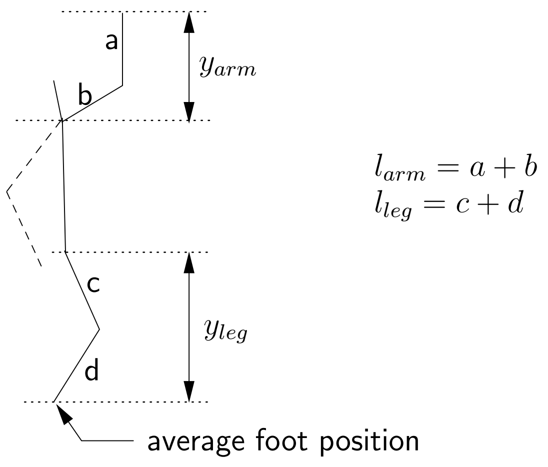


Figure 5.3: Parameters involved in $\mathbf{heur}_{hand\_down}$

downwards from the grasp, with the legs hanging down. Figure 5.3 illustrates the situation.

$$\mathbf{heur}_{hang\_down} = 1 + \frac{(l_{arm} - y_{arm}) + (l_{leg} - y_{leg})}{4}$$

$\mathbf{heur}_{knees\_down}$

The main idea behind this heuristic is to give the impression that the character's shins are in contact with the ground. The lower the positions of the knees, the lower the value. Through the use of this heuristic the gradient descent through $\mathcal{D}$ will favour postures in which the knees are pressed down as far as possible – resulting in the impression that the character is kneeling.

$$\mathbf{heur}_{knees\_down} = 1 + \{|y_{l\_foot} - y_{l\_knee}| + |y_{r\_foot} - y_{r\_knee}|\}^2$$

5.2.3 Adding heuristics

Creating appropriate heuristics can be a non-trivial endeavour as there are no strict guidelines to follow. Heuristics are typically based on geometric considerations. A newly implemented heuristic is usually first tried out in the posture editor, which is used to provide the planner with the input problem postures, and the value is calibrated so that all the heuristics return roughly the same value for similar visual amount of deviation from the "ideal".

5.3 The locomotion mode finite state machine

As described previously, the LFSM performs the vital function of manipulating grasps. It decides when and how grasps should be modified, and implements these changes when needed. The FSM is represented by a weighted directed graph, with the weight of each edge representing the traversal preference relative to the other outbound edges for a given vertex. The vertices in the graph represent the different locomotion modes, while the edges denote transitions. Each edge has a set of preconditions that must be met prior to traversal, and an arbitrary number of effects when it is traversed.

Figure 5.4 demonstrates the FSM that we have implemented. In the following subsections we discuss each of the edges of the FSM, starting with the self-loops. These are the transitions which do not entail a change of state, but rather serve to iteratively perform the grasp modifications that are inherent in traveling continually in that particular mode of locomotion. The purpose of discussing the self-loops first is so that the reader is familiar with the mechanics of each locomotion mode when we come to discuss the transitions between them, represented by the other, remaining edges.

The notation we will use in describing the edge preconditions and effects is as follows:

- Γ refers to the set of all grasp points
- γ refers to a particular grasp point
- $\vec{v}$ refers to the location of some point in $\mathcal{W}$
- two letter subscripts by default have the following meaning:
  - first letters **g**, **f**, and **b** refer respectively to "grasping", "free", and "both".
  - second letters **h**, **f**, and **l** refer respectively to "hand", "foot", and "limb"

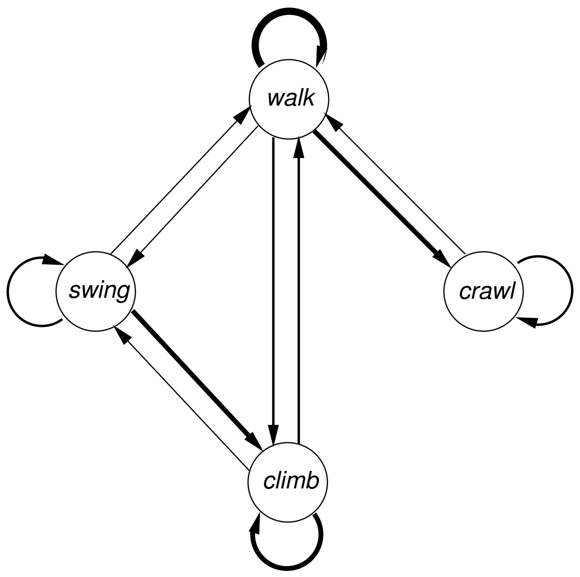


Figure 5.4: The implemented locomotion mode finite state machine; the line thickness indicates higher precedence in precondition checking.

5.3.1 Self-loops

Walking

Walking is the most commonly used mode. This mode is intended for terrains having slope $m, -1 < m < 1$, with a positive upward component to the surface normal. The intuitive purpose of this particular edge is to perform a step.

Preconditions:

- $\exists\, \gamma \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma, \texttt{limb}_{ff}) \\ \texttt{type}(\gamma) = \texttt{load\_bearing} \\ \texttt{dist\_map}(\vec{v}_\gamma) < \texttt{dist\_map}(\vec{v}_{gf}) \\ |\vec{v}_\gamma - \vec{v}_{gf}| > \texttt{min\_limb\_sep} \end{cases}$
- the posture correction after the regrasp leads to a pose in which $x_{foot\_1} \leq x_{COM} \leq x_{foot\_2}$

Effects:

- attach the free foot to γ
- posture correction step
- detach the previously grasping foot

$\boxed{\textbf{Climbing}}$

This mode is intended for scaling vertical walls. Climbing usually requires continuously maintaining two or three points of contact, with the former resulting in a more energetic and nimble appearance. It is this former approach that we adopt in our implementation. Of the two points, one is always grasped with the hand, and the other with a foot.

Preconditions:

- $$\exists\, \gamma_h \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma_h, \texttt{limb}_{fh}) \\ \texttt{dist\_map}(\vec{v}_{\gamma_h}) < \texttt{dist\_map}(\vec{v}_{gh}) \\ |\vec{v}_{\gamma_h} - \vec{v}_{gh}| > \texttt{min\_limb\_sep} \\ |\vec{v}_{\gamma_h} - \vec{v}_{bf}| > \texttt{min\_hand\_foot\_sep} \end{cases}$$
  $$\bigvee$$
  $$\exists\, \gamma_f \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma_f, \texttt{limb}_{ff}) \\ \texttt{dist\_map}(\vec{v}_{\gamma_f}) < \texttt{dist\_map}(\vec{v}_{gf}) \\ |\vec{v}_{\gamma_f} - \vec{v}_{gf}| > \texttt{min\_limb\_sep} \\ |\vec{v}_{\gamma_f} - \vec{v}_{bh}| > \texttt{min\_hand\_foot\_sep} \end{cases}$$

Effects:

- attach free hand to γ_h if it exists, else attach free foot to γ_f
- detach the sibling limb of the one that just effected a grasp

Notes: the poses generated in this matter are of sufficient aesthetic appeal that the posture correction step is not necessary. As this operation is very time consuming, we have omitted it in this case.

$\boxed{\textbf{Swinging}}$

The swinging mode can be thought of as the arm-based analog to walking, at least from the kinematic point of view. The essential quality of the motion produced while in this mode is that the legs take no active part in locomotion, only the arms. This mode is primarily used when traversing by way of a sequence of overhead grasp points, such as can be found in a monkey-bar environment. The FSM also switches to this mode in situations which require the character pulling itself up, as the general behaviour and governing rules are the same.

Preconditions:

- $\exists\, \gamma \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma, \texttt{limb}_{fh}) \\ \texttt{type}(\gamma) = \texttt{pendent} \vee \texttt{type}(\gamma) = \texttt{hybprid} \\ \texttt{dist\_map}(\vec{v}_\gamma) < \texttt{dist\_map}(\vec{v}_{gh}) \\ |\vec{v}_\gamma - \vec{v}_{gh}| > \texttt{min\_limb\_sep} \end{cases}$
- the posture correction after the regrasp leads to a pose in which $x_{hand\_1} \leq x_{COM} \leq x_{hand\_2}$

Effects:

- attach the free hand to γ
- posture correction step
- detach the previously grasping hand

$\boxed{\textbf{Crawling}}$

This self-loop implements crawling on one's hands and knees. It requires three points of contact for the character at all times. The difficult aspect of this transition is the need to give the character the appearance of making full contact with the shin bone. We achieve the desired effect by employing the $\textbf{heur}_{knees\_down}$ heuristic, as discussed earlier in this chapter. Without it, the character crawls like a spider, only touching the ground with the end effectors.

Preconditions:

- $\exists\, \gamma \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma, \texttt{limb}_{fl}) \\ \texttt{type}(\gamma) = \texttt{load\_bearing} \\ \texttt{dist\_map}(\vec{v}_\gamma) < \texttt{dist\_map}(\vec{v}_{gl}) \\ |\vec{v}_\gamma - \vec{v}_{gl}| > \texttt{min\_limb\_sep} \end{cases}$

Effects:

- attach the free limb to γ
- posture correction step
- if character "stretched out", release with the foot furthest from goal, else release with the hand furthest from goal

Notes: the limb denoted with the subscript “gl” refers to the sibling limb of the one that is unattached. Also, the character is “stretched out” if the largest current distance between any foot and any hand is greater than 65% of the maximum possible such length, which is 4.3 for our character as defined in Figure 3.4. The purpose of this selection process is to maintain a relatively comfortable distance between the hands and the feet.

5.3.2 State transitions

In the following we describe the remaining edges, the ones responsible for changes in state. They are sorted by the originating locomotion mode.

$\boxed{\textbf{walking} \rightarrow \textbf{climbing}}$

Preconditions:

- $\exists\, \gamma \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma, \texttt{limb}_{fh}) \\ \texttt{type}(\gamma) = \texttt{hybrid} \\ y_{head} < y_\gamma \end{cases}$

Effects:

- attach the appropriate free hand to γ

Notes: subscript “fh” refers to either free hand.

$\boxed{\textbf{walking} \rightarrow \textbf{swinging}}$

Preconditions:

- character hasn’t advanced in a long while
- $\exists\, \gamma \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma, \texttt{limb}_{fh}) \\ \texttt{type}(\gamma) = \texttt{pendent} \end{cases}$

Effects:

- attach the appropriate free hand to γ
- detach any foot grasps

Notes: subscript “fh” refers to either free hand.

$\boxed{\textbf{walking} \rightarrow \textbf{crawling}}$

Preconditions:

- comfort adjustment using "walking" mode heuristics results in character's center of mass to lie horizontally outside the two points of contact

Effects:

- let γ = the grasp point which the grasping foot is attached to
- backtrack in the solution we have assembled so far until the feet are attached to two other grasp points
- apply Brownian motion until the right hand is able to reach γ using simple IK
- attach the hand to γ

Notes: this is a poor implementation of the transition, but one that works. A much better one would be to have the character squat down and grasp any free grasp point in front of it, but this would require a full IK engine. We did not have one available, but any serious implementation of the planner surely would.

$\boxed{\textbf{climbing} \rightarrow \textbf{walking}}$

Preconditions:

- grasp point type for either the grasping hand or foot is `load_bearing`
- $\exists\, \gamma \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma, \texttt{limb}_{ff}) \\ \texttt{type}(\gamma) = \texttt{load\_bearing} \end{cases}$

Effects:

- attach the free foot to γ
- detach any hand grasps
- apply posture correction using "walking" heuristics
- detach the foot furthest from goal

$\boxed{\textbf{climbing} \rightarrow \textbf{swinging}}$

Preconditions:

- $\exists\, \gamma \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma, \texttt{limb}_{fh}) \\ \texttt{type}(\gamma) = \texttt{pendent} \end{cases}$

Effects:

- attach the free hand to γ

- detach any foot grasps
- apply posture correction using “swinging heuristics
- detach the other hand

$\boxed{\textbf{swinging} \rightarrow \textbf{walking}}$

Preconditions:

- character hasn't advanced in a long while
- $\exists\, \gamma \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma, \texttt{limb}_{ff}) \\ \texttt{type}(\gamma) = \texttt{load\_bearing} \end{cases}$

Effects:

- attach the appropriate free foot to γ
- detach any hand grasps

Notes: subscript “ff” refers to either free foot.

$\boxed{\textbf{swinging} \rightarrow \textbf{climbing}}$

Preconditions:

- $\exists\, \gamma_h \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma_h, \texttt{limb}_{fh}) \\ \texttt{type}(\gamma_h) = \texttt{hybrid} \vee \texttt{type}(\gamma_h) = \texttt{pendent} \\ |\vec{v}_{\gamma_h} - \vec{v}_{gh}| > \texttt{min\_limb\_sep} \end{cases}$
- $\exists\, \gamma_f \in \Gamma \mid \begin{cases} \texttt{reachable}(\gamma_f, \texttt{limb}_{ff}) \\ \texttt{type}(\gamma_f) = \texttt{hybrid} \\ y_{hip} > y_{\gamma_f} \\ y_{hip} < y_{head} \\ |\vec{v}_{\gamma_f} - \vec{v}_{\gamma_h}| > \texttt{min\_hand\_foot\_sep} \end{cases}$

Effects:

- attach the free hand to γ_h
- attach the appropriate free foot to γ_f
- apply posture correction
- release the other hand's grasp

Notes: subscript “ff” refers to either free foot.

$\boxed{\textbf{crawling} \rightarrow \textbf{walking}}$

Preconditions:
- character has advanced in “crawling” mode for a significant distance (this is to prevent the character from squatting and getting right up again)
- there is sufficient room to stand up
- both feet are grasping

Effects:
- detach any hand grasps
- perform posture correction using “walking” heuristics
- detach the foot furthest from goal

5.3.3 Adding new modes of locomotion

The addition of a new mode is a multistep process. We first analyze the grasping behaviour of the desired movement: how many grasps are in effect at a given time, how many are performed with hands and how many with feet, to what kind of grasp points do these attach, under what conditions and in which order are the grasps changed, etc. Using these observations we formalize the self-loop’s preconditions and effects, as well as formulating any new heuristics that might be required. A sample implementation of these specifications is then constructed and tested in an environment appropriate to the new mode. Once satisfied with the self-loop’s performance we consider the state’s connectivity: to which states should we be able to transition, and vice-versa. Each transition is then analyzed and implemented in a manner similar to that of the self-loop.

The preconditions and effects for a particular edge are currently implemented together as a single C++ routine, consisting just of the geometric tests and a sequence of grasp operations. With the necessary “housekeeping” code the routines amount to 20 lines on the average, although this varies with the complexity of the transition. All the edge routines are kept in a single file, although that is not strictly necessary. Any addition of modes and transitions require additional coding in C++, although it is possible to implement the LFSM code such that the edge preconditions and effects are provided in script or data form. This would facilitate the addition of new modes, although any such implementation is inherently less flexible and robust since the data or script capabilities are limited to the fixed functionality that the LFSM provides.

5.4 Other details

5.4.1 Interpolation over distance map

We have found that a fair number of local minima in the gradient descent of $\mathcal{P}$ were due to the too coarse discretization of distance map: any two configurations which are relatively similar result in small variations in the location of the center of mass, which often map to the same cell in distance map. If a particular configuration happens to have its center of mass near the middle of the corresponding cell, it is very likely that all of its neighbours will also map to the same cell. This unnecessarily results in a local minimum. Although one could raise the resolution of distance map, a much better solution is to use bilinear interpolation. Figure 5.5 illustrates the difference.

5.4.2 Restepping

Our planner considers the problem as solved when $\mathcal{P}(\mathbf{q}) = \mathcal{P}(\mathbf{q}_{finish}) = 0$. As we mentioned in Chapter 3, this signifies that the character has reached the destination, but not necessarily $\mathbf{q} = \mathbf{q}_{finish}$. Unlike the "precursory" planner of Chapter 3, a linear interpolation from $\mathbf{q}$ to $\mathbf{q}_{finish}$ is unacceptable since this results in the breaking of any grasps present. The solution we have adopted consists of adding a simple "stepping" planner. This component attempts to make $\mathbf{q}$ match $\mathbf{q}_{finish}$ by manipulating the grasps. In our current implementation this subplanner is limited to working with the foot grasps only, in a "walking" style posture.

There are only two cases that we need to concern ourselves with: "simple" and "wrong-footed". Figure 5.6 shows the two scenarios and how this simple planner deals with them. In the simple case the subplanner grasps the final grasp point with the free foot, sets the root of $\mathbf{q}$ to this foot, and performs a linear interpolation to $\mathbf{q}_{finish}$.

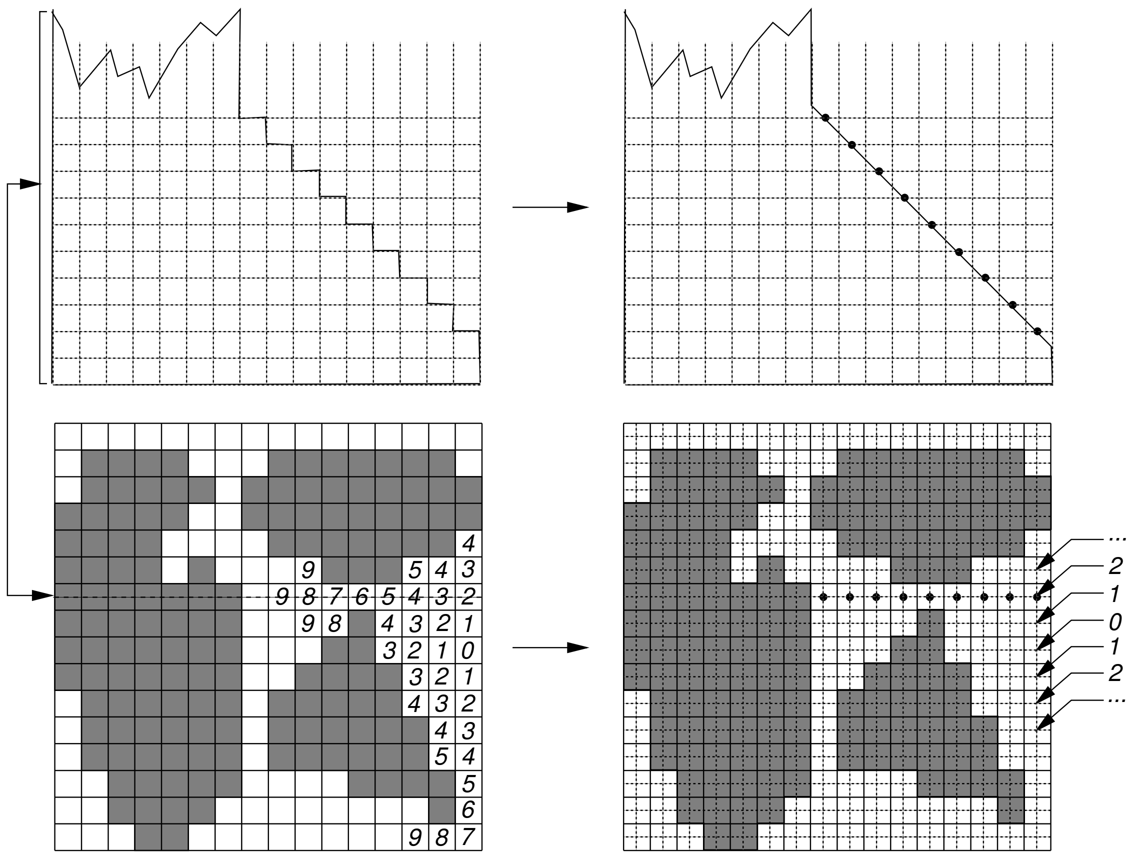


Figure 5.5: The bilinear interpolation of distance map

In the wrong-footed case, were the currently grasping foot is attached to the wrong grasp point, we attach the free foot to the free grasp point, detach the other foot, attach it to the correct grasp point, detach the originally free foot, and once again, linearly interpolate to $\mathbf{q}_{finish}$.

5.4.3 "Turn-around" operation

Navigating through environments such as the one shown in Figure 1.1 requires an operation which would allow the character to turn around. Since we are working in two dimensions and changing direction inherently requires three dimensions, we had to improvise a little bit. We have achieved the desired effect by substituting the character with its mirror image whenever a turn around is required. As the solver

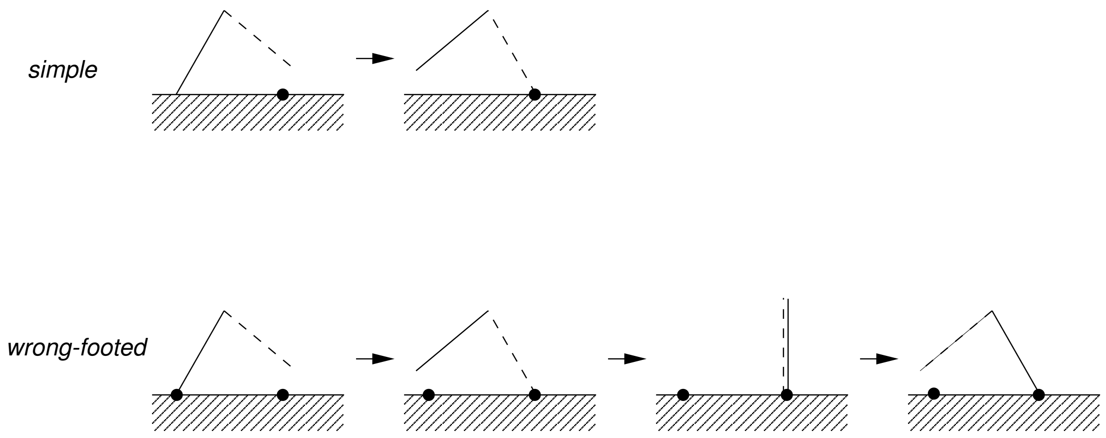


Figure 5.6: The two cases and their handling by the "stepping" planner; we show only the legs, each one represented by a single link

could backtrack at any point, and do so to any previous posture, it is necessary to augment the state information with the character's current direction. The turn-around motion is not required in a 3D implementation, as it will come about on its own from the normal operation of the solver.

5.4.4 Multi-part problems

Throughout this thesis we have always formulated the planning task as a two-point boundary value problem. This approach is too restrictive for all but the simplest planning problems. We have therefore extended the planner's problem specification capabilities: besides providing the starting and finishing configuration one can now also provide an arbitrary number of "milestone" configurations and pregenerated sequence. A "milestone" configuration simply specifies that the solved path muss contain this $\mathbf{q}$ at some point. The same is true of pregenerated sequence, which is just a set of consecutive configurations. This permits our planner to work with other human animation algorithms in constructing the solution, allowing for a much richer potential solution space.

The planner handles a pregenerated sequence by splitting the whole problem into two subproblems: the first consists of finding a path from $\mathbf{q}_{start}$ to the first configu-

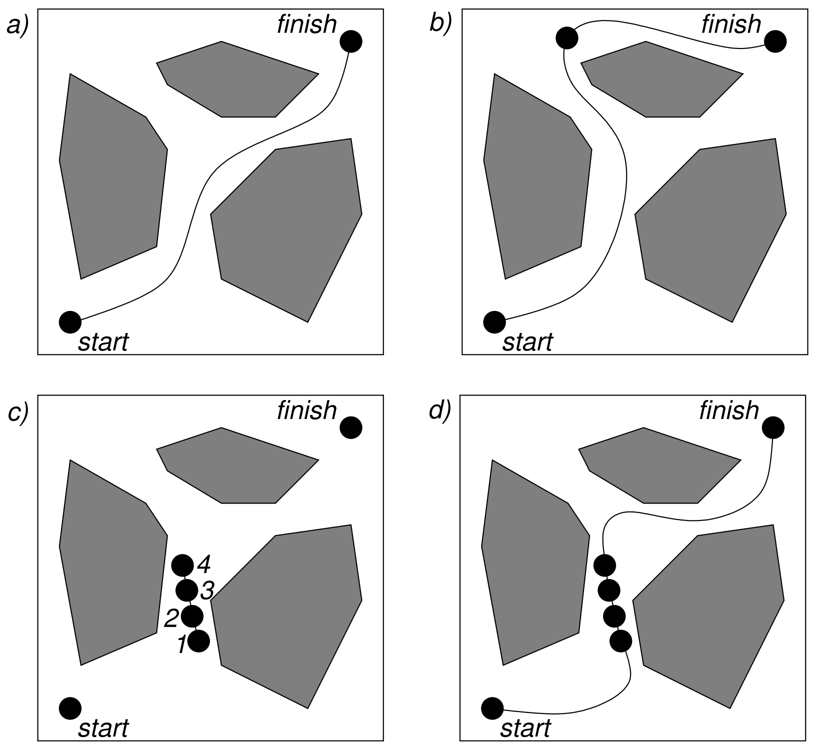


Figure 5.7: Multi-part problems visualized in 2 dimensions; a) a simple two-point boundary value problem; b) the solution after a “milestone” configuration is provided; c) the problem with a pregenerated sequence; and d) the solution

ration of the sequence, while the second looks for a path from the last configuration of the said sequence to $\mathbf{q}_{finish}$. This is illustrated in Figure 5.7. Using this approach every scenario is reduced to a number of two-point boundary value problems, and each one is solved separately.

A “milestone” configuration is treated as a pregenerated sequence of length 1. The main advantage of using one is that it provides the animator with extra control. It can be used for adding gestures to the motion without the need to provide a complete sequence of configurations, for speeding up the solving process by providing direct hints on position and posture, and even for forcing particular routes to be taken.

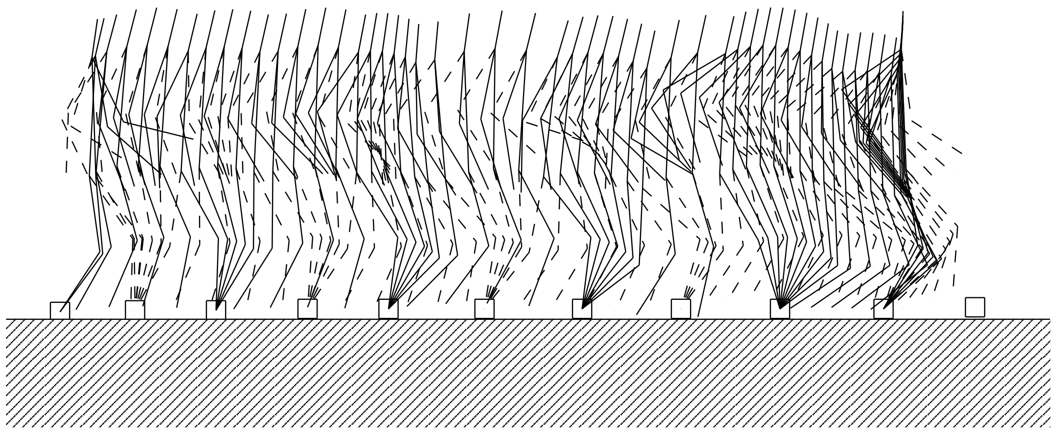


Figure 5.8: Walking mode, with a “restep” at the end to match $\mathbf{q}_{finish}$

5.5 Results

In this section we present some of the results obtained with our implementation. We first present examples of the four locomotion modes on their own, followed by a complex problem which requires the use of them all.

5.5.1 Walking

Figure 5.8 presents a simple flat terrain scenario. The figure at the extreme left is the start configuration, while that at the extreme right is the destination configuration.

Note that although the grasp points are evenly distributed, the resulting motion is somewhat irregular. The stochastic nature of the planner, and in particular, of the posture correction algorithm are the primary reason. This may be considered a weakness, or a strength, depending on the application. Finally, one can see at the end a “shuffle” of the feet; a restep was necessary as the character was wrong-footed. This can be seen by looking closely at the right foot, rendered with a solid line: it grasps both of the last two grasp points in sequence. The solving time for this scenario averages around 15–20 seconds on our system.

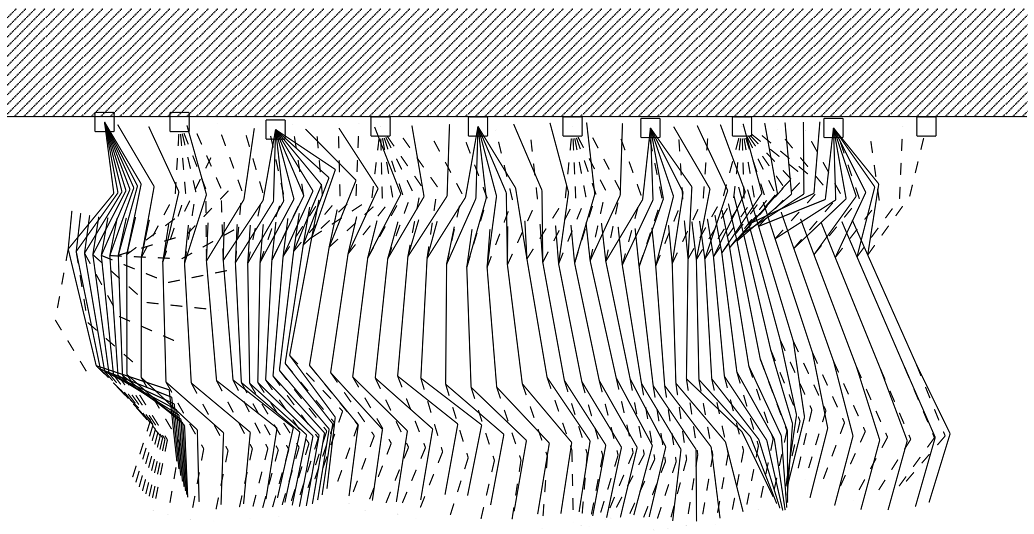


Figure 5.9: Swinging mode

5.5.2 Swinging

Figure 5.9 shows a segment of swinging motion. The mechanics of this locomotion mode are nearly identical to that of walking. The only visible difference, other than the limbs involved in grasping, is that no heuristic analogous to $\mathbf{heur}_{limb\_counter\_balance}$ is used: the legs are free to swing as they like. This planner solves this problem in roughly the same time as the walking scenario above, 15–20 seconds on the average.

5.5.3 Climbing

Figure 5.10 depicts a climbing sequence, as well as the transitions to and from walking. We only present the prominent configurations in which a change of grasps occurs. The first and last frames show the starting and finishing configurations. Running time for this scenario varies, ranging roughly from 30–60 seconds, sometimes even longer. The critical point in the solution is the transition from climbing to walking since this is the point where the character's motion is usually the most constrained: as one can see in the bottom-left frame, placing the feet in new grasps more difficult since the figure is hunched forward. The time required to find a proper transition large depends on the choice of grasps leading up to this point. The flexibility of *grasp surfaces*(see §6.2.2) will greatly improve the speed of this process.

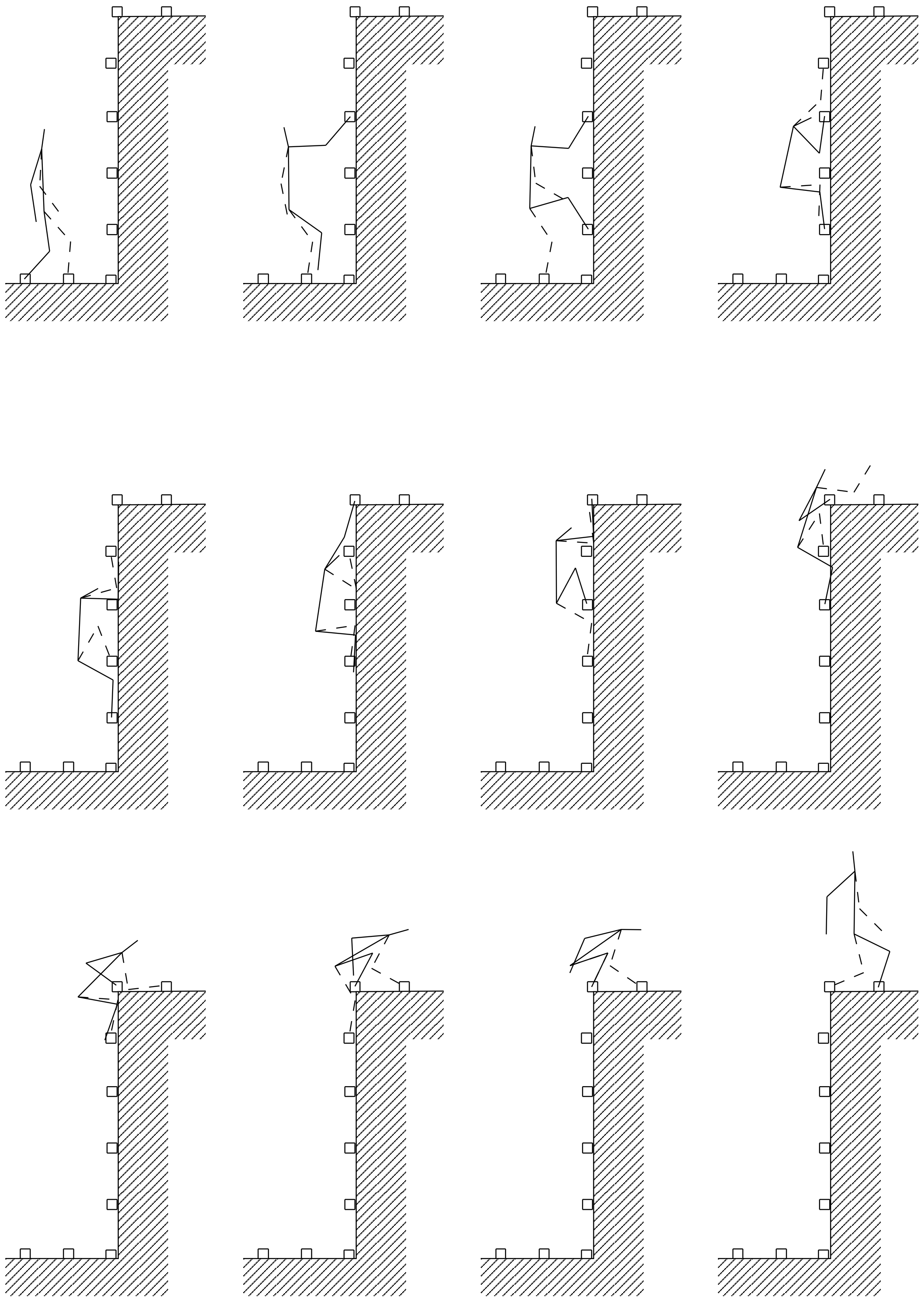


Figure 5.10: Climbing mode (selected frames shown)

5.5.4 Crawling

Figure 5.11 shows the character performing a walking to crawling transition, crawling, and then another transition, from crawling back to walking. Once again we show only important keyframes. One can see here the limitations of our $\mathbf{heur}_{knees\_down}$ heuristic: although the knee is very low to the ground, occasionally it is visibly still not touching it. Our implementation of the planner requires about 2–3 minutes, or occasionally longer to solve this problem. The vast majority of the time is spent on the walking to crawling transition; the LFSM has to exhaust all posture adjustment attempts while trying to walk before deciding that crawling is warranted. A more efficient precondition for this LFSM edge should make the solving time of this scenario comparable to that of walking.

5.5.5 Complex problem

Our last example, shown in Figure 5.12, is the solution to the problem we have introduced in Chapter 1, in Figure 1.1. All four locomotion modes are used, as well as most of the available transitions. Besides providing the initial and final configurations we have also provided a canned sequence of the character sliding down the ramp, which the planner was to insert at the appropriate point. Since, for clarity, only every 20th frame is shown in the solution, the sliding sequence appears only as its first and last frame, with the two character images surrounding the downward ramp. The running time for this problem usually averages between 10–15 minutes, and requires a handful of manual backtrack operations. These are described in §5.6.3.

5.6 Discussion

5.6.1 Run times

Our prototype implementation is largely unoptimized. There is room for optimization both in the code implementation as well as in the formulation of the preconditions

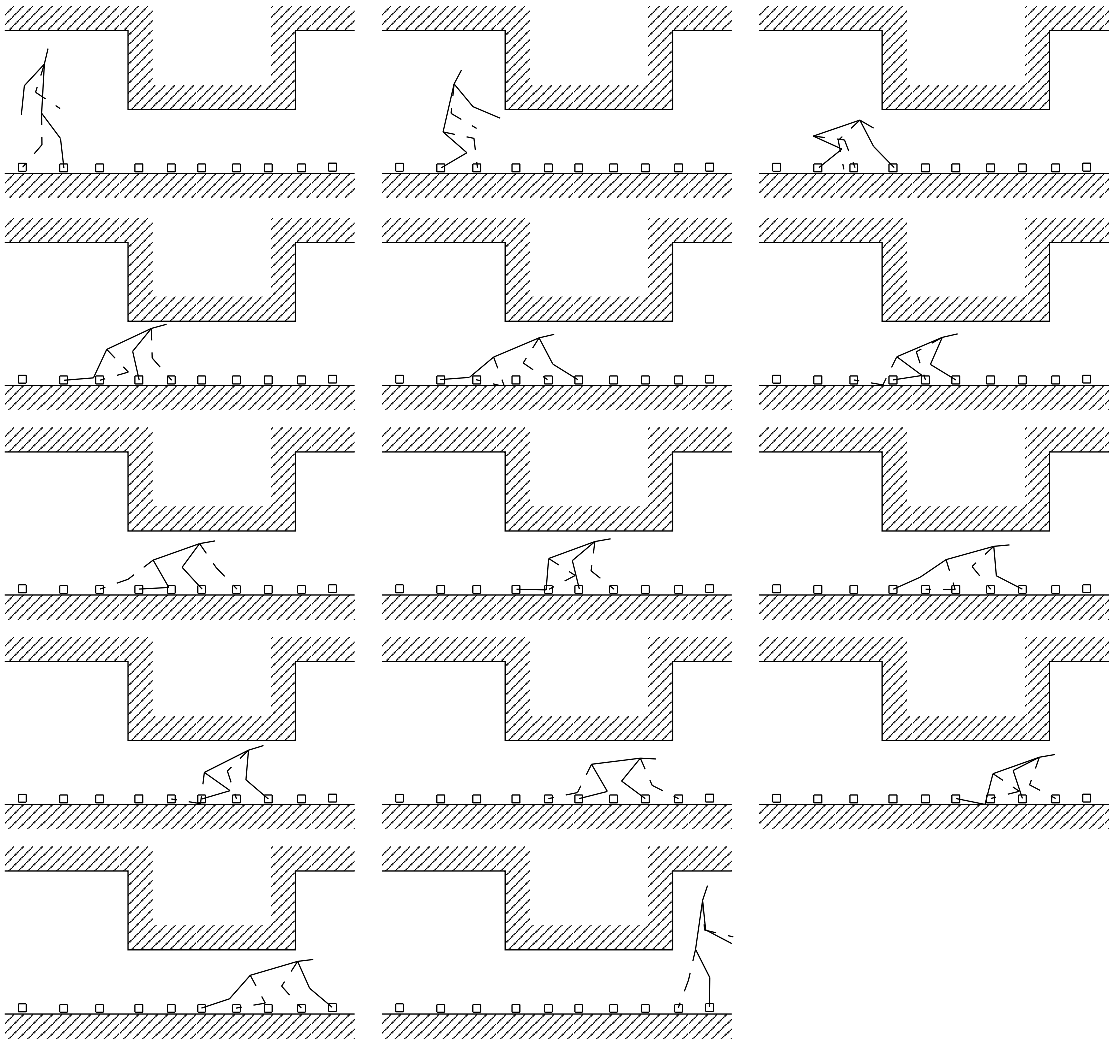


Figure 5.11: Crawling mode (selected frames shown)

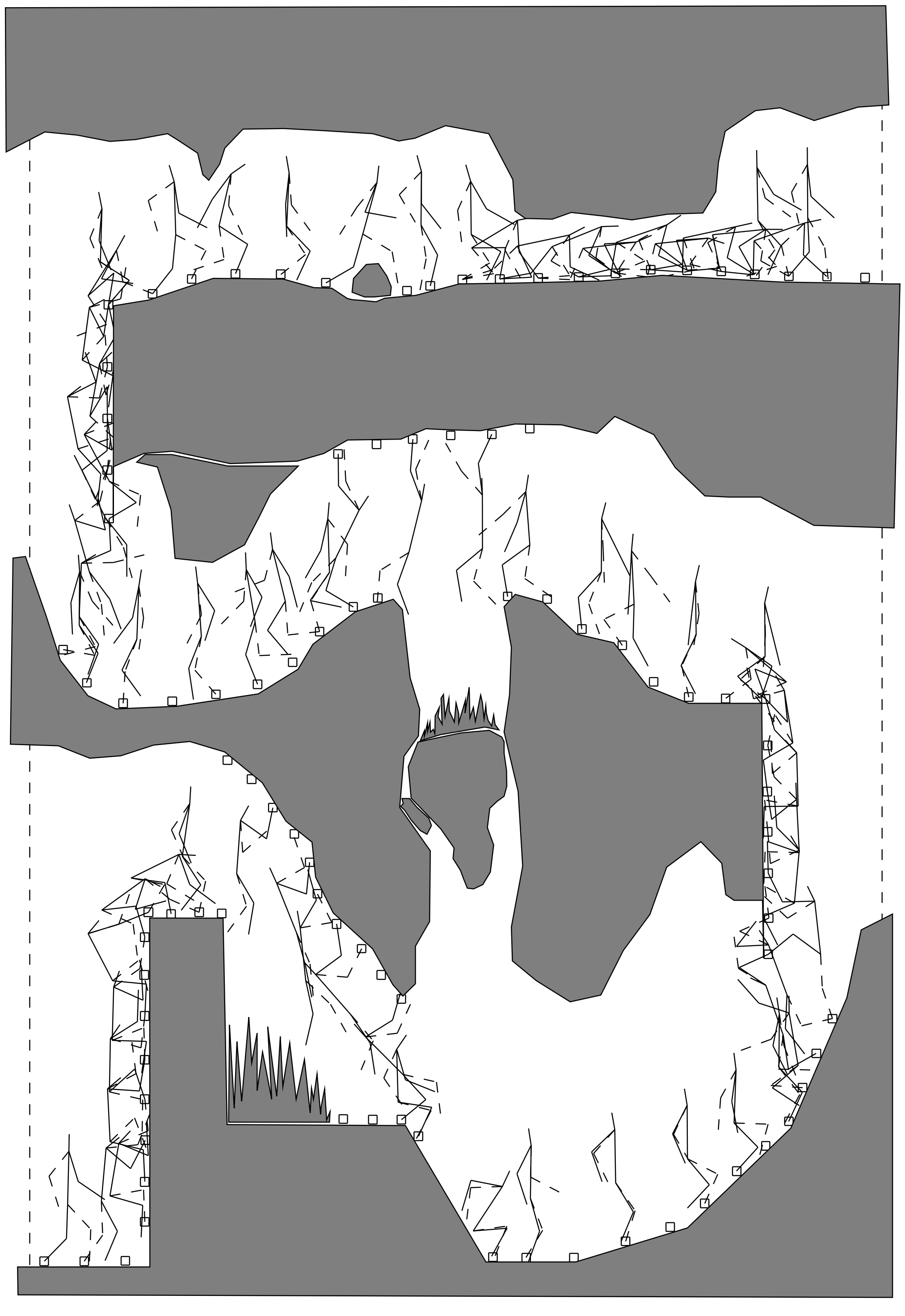


Figure 5.12: A complex problem (every 20th frame of the solution shown)

and effects used. The planner could potentially be improved to be realtime, although this will involve a substantial effort.

Example	Figure	Time
Walking	5.8	15–20 sec.
Swinging	5.9	15–20 sec.
Climbing	5.10	30–60 sec.
Crawling	5.11	2–3 min.
Complex	5.12	10–15 min.

Table 5.2: Summary of run times for the different examples

We summarize the runtimes for all the examples presented in Table 5.2. The run times vary from run to run due to the stochastic nature of the planner, the varying speeds of the different locomotion modes used, and the efficiency, or inefficiency, of their implementations. Furthermore, for a simple walk the length of a room, as demonstrated in Figure 5.8, the planner consumes around 95% of its time in readjusting the character's posture. This seems to be the primary area that needs to be addressed in any optimizations.

As we mentioned in §4.3, one attempt at optimizing the posture adjustment process consists of stopping the discomfort gradient descent whenever a certain acceptable level of comfort has been achieved. This works well when the character is unconstrained and can satisfy all the heuristics to a large extent. In confined spaces however, a tradeoff must be struck since all the heuristics cannot be lowered sufficiently. This results in $\mathcal{D}(\mathbf{q})$ never falling below the `comfort_threshold` and the planner therefore continues exhaustively until a minimum is reached. Making this threshold vary according to the current environment of the character would be ideal, but this appears to us to be difficult to automate.

The posture readjustment could be further improved by requiring it first to try some common postures for the character, or some subpart of it. The posture correction stage would take much less time if, for example, the gradient descent of the discomfort space $\mathcal{D}$ was started with the backbone already vertically oriented, thus eliminating the lengthy process of bringing it upright. These common postures could be provided as *a priori* knowledge, or alternatively the planner could learn them during the solving process itself.

5.6.2 The solver's stochastic nature

The stochastic nature of our motion planner is a double edged sword, especially when dealing with the gradient descent optimizations. It makes optimization possible in the high-dimensional spaces which define a character's motion, but the false identification of local minima can be problematic.

Our use of Monte Carlo sampling during the gradient descent of $\mathcal{D}$ often results in the planner not finding the configuration which *best* satisfies the heuristics in effect. That is, the true absolute minimum is often not reached, despite a number of attempts to escape any encountered local minima, whether true or false. This visually results, for example, in the knees being slightly off the ground when crawling, or the arms not swinging fully to their optimal positions while walking. Although providing the solver, and the gradient descents in particular, with more time generally gives better results, much could be done to improve the situation, considering that a human observer can usually immediately suggest a number of possible modifications leading to lower discomfort when the planner is stumped.

The above weakness occasionally escalates to be an even greater nuisance. Due to the gradient descent's inability to reach the optimal configuration, a different decision is made in the finite state machine, one that puts the planner on a wrong path. Even though the planner will find this out sooner or later and backtrack, it may take some time and this is wasting computing cycles unnecessarily. For example, it may happen that during a walk in an inclined yet unconstrained environment that the solver fails to bring the character to the most comfortable configuration, which may trigger a transition to climbing or crawling. As the space is unconstrained this looks puzzling. Such occurrences are more likely in constrained environments. If care is taken, it should be possible to formulate preconditions for the transition edges which are more discriminating and less often fooled.

5.6.3 Manual backtracking

An alternative and practical solution to the problem mentioned in the previous subsection is to allow the user to stop the solving process at any time, and specify a point to backtrack to. In our implementation we display a push button which stops the solver at any time, and one can then use the time slider and a menu option to provide the point. The solver then throws out any configurations past that point which are not part of the original problem specification, while the configuration marking the backtrack point is marked as $\mathbf{q}_{start}$, so any further runs start at that point.

5.6.4 Simple IK limitations

The use of the simple, 2 link inverse kinematics is quite limiting. First, it adds significantly to the total running time: posture adjustment aside, a full implementation of IK would be able to completely bypass any gradient descents through $\mathcal{C}$ for unconstrained motions such as walking and swinging since the planner could simply step from grasp point to grasp point by the use of IK alone. The more constrained locomotion modes would also benefit in a similar manner but to a lesser extent since some gradient descent steps might be necessary there.

The second limitation is illustrated in Figure 5.13, which depicts a section of the complex problem shown in Figure 5.12, except that one of the obstacles has been removed, which is shown with the dashed outline. In this situation the descent through $\mathcal{C}$ "attracts" the character's center of mass in the direction indicated. This results in the hip being likewise pulled in the same direction, which prevents the establishment of the next grasp since the simple 2 link inverse kinematics cannot make the free foot reach the next grasp point. The only possibility of advancing is afforded by a random walk finishing in a configuration which brings the hip closer to the ground in front of the character, but this usually takes a long amount of time. It is for this reason that we have inserted the additional overhanging obstacle in the final problem. The use of a full IK engine would, once again, remove this weakness since it would be able to drag the character down as needed.

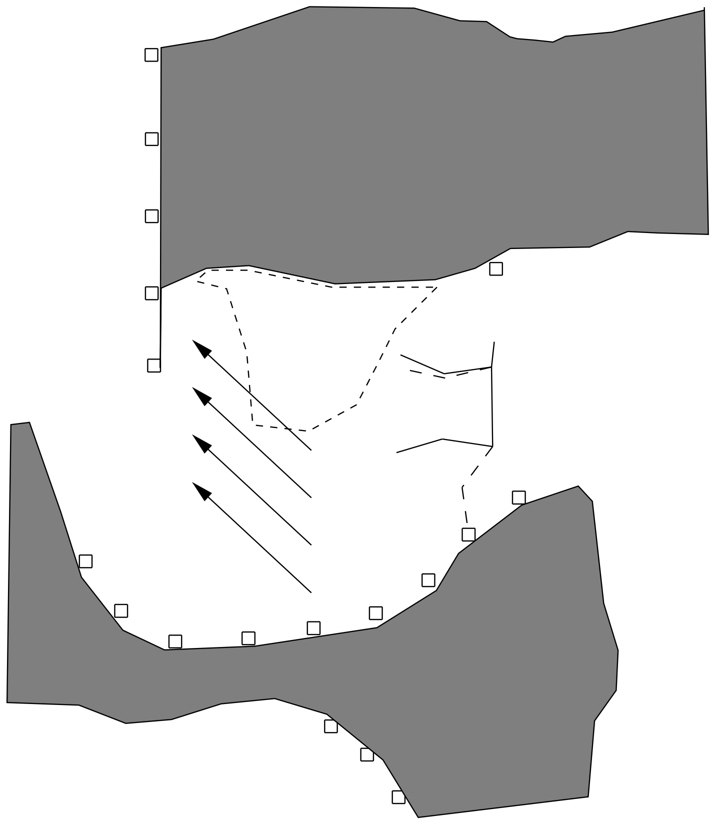


Figure 5.13: A weakness of the planner due to the use of simplified inverse kinematics; the arrows show the gradient of the distance map

Chapter 6

Conclusions

6.1 Summary of work

The algorithms described in this thesis provide a novel planning method for automated character animation. They are particularly well suited for planning motions in unstructured, constrained environments and for generating plausible transitions between various modes of locomotion.

Our work integrates configuration-space planning methods with the requirements of character animation. At the heart of this problem is how to efficiently exploit knowledge of a character's motion preferences while solving potentially complex global motion planning problems. The use of grasp points serves to explicitly model some key aspects of the motion, while a collection of heuristics together implicitly model motion preferences. A finite state machine is used to imitate the polarization of human motion into distinct locomotion modes.

What makes the algorithms interesting is that they must tread the line between discrete and continuous optimization problems because the choice of grasps is discrete while the remainder of the motion is continuous. Yet choices in the continuous domain affect the discrete domain, and vice versa. Thus the algorithm must optimize a combined set of discrete and continuous choices. Secondly, the algorithm exploits both deterministic and stochastic methods. The FSM and heuristics are deterministic, while the core of the planning algorithm has a significant stochastic component.

The previous work whose goals resemble that of our system are the Jack system and Motion Factory's Motivate 3D Game Development System. Our work differs from the former in that it has a strong planning component and is able to handle complex motion in constrained environments and simple ones with equal ease. It differs from the latter in that it does not restrict the character's motions to prerecorded data but is capable of adapting to any terrain, and any required movement mode.

6.2 Future Work

6.2.1 Three dimensions

The most serious limitation of the planner that we have presented is the restriction of the current implementation to motion in two-dimensional environments. However, the extension to three dimensions should entail only a limited amount of additional complexity. The main components which are affected are the procedures that search $\mathcal{P}$ and $\mathcal{W}$. Searching of the configuration space is performed by the gradient descent as it stochastically samples the neighbourhood of a particular $\mathbf{q}$. The additional complexity here would be due to the higher dimensionality of the configuration space which in turn is due to the character's joints now having more degrees of freedom. $\mathcal{P}$ is also searched by the heuristic system when correcting the character's posture. Since both routines use stochastic sampling this should have a limited impact on running times. Searching of the workspace is performed by the finite state machine preconditions which look for suitable new grasp points. The impact here can also be made relatively small through the use of an efficient grasp point searching algorithm.

The distance map likewise gains a dimension, which considerably increases its generation time. However, this is not really critical since the map can be precomputed and the result cached for future runs of the planner. It is also probable that the distance map could in many situations be substituted by the distance-to-the-goal computations having a more local scope.

6.2.2 Grasp surfaces

Although the concept of a grasp point is very practical, there are situations where it would be convenient to be able to mark entire regions as being graspable. The most common example of this is any flat terrain which is meant to be walked upon; since any point on such a surface can be theoretically used to support the figure, it seems awkward to have to restrict grasping to only individual points upon it. This extension has the potential of enabling better quality for motions using such surfaces, given that the character would be free to choose its own, preferred stride length.

A significant problem that would need to be resolved when implementing these grasp surfaces is how the planner should choose the point at which to grasp. We would like to perform the grasp such that the resulting stride length, or arm span, is a comfortable one. A working initial solution consists of choosing at random within the given region, and using an additional heuristic during character posture readjustment which would specify how comfortable the stride and arm span lengths are. The comfort-adjusting gradient descent would then tend to converge on the best stride possible under the given circumstances.

6.2.3 Arbitrary skeletons

The planner as currently implemented is limited to animating anthropomorphic skeletons. Although one can vary the lengths of the links in the skeleton, any vertex-graph alterations, such as appending a third leg for example, are currently not allowed. This constraint arises from the "precondition checker" and the "effect executor" parts of the FSM, which implicitly assume certain geometrical properties of the skeleton, such as the number and type of limbs. After some consideration this seems like an acceptable and even necessary concession: for example, the logistics of walking differ between a human with two legs and a centipede with a hundred. Human motion planning is challenging; the broader problem even more so.

Although any given implementation of the planner assumes certain geometrical properties of the skeleton, the general theoretical approach does not. That is, one can make an implementation for a centipede just as easily as an implementation for a human. The only constraint is that in general the centipede-planner cannot be used with a human character, and vice-versa.

It would be interesting to evolve the planner so that it was more generic in terms of skeletal structure, to allow the use for both, the human and the centipede, as well as any other skeleton. This then would require that the structure-specific parts of the planner be implemented separately, and plugged into the planner as needed, depending on the model currently being animated.

Alternatively, if one is interested in animating imaginary creatures for which we have no preconceived notion of their locomotion dynamics, it might be interesting to adapt the planner to be able to handle arbitrary skeletal topologies without the need for the specialized FSM components. The role of the heuristic system here would only then be to give the appearance of physically-correct motions, and not to give "characteristic" behaviour.

6.2.4 Motion speed control

Since the planner has no explicit notion of time nor speed, we perform a one-to-one mapping between the configurations of the solution's path and the keyframes used in playback. This is not necessarily the best approach. The resulting motions could be made more fluid by altering this mapping so that the speeds of the different parts of the character's body change in a continuous manner. Such a mechanism could also accept a parameter, whereby the animator specifies the desired speed. This parameter could be time varying, allowing the speed to vary throughout the animation.

6.2.5 Complex grasping

A limitation in our planner is that only the end-points of the limbs are allowed to grasp (i.e. hands and feet). Although this is typically sufficient, there are motions which require more complex grasps. Two examples of this are using the buttocks as a support when sliding on the floor, and leaning the back of one's shoulders against a wall also as a means of support. These types of motions cannot be generated by the planner at this point in time. Furthermore, it might be interesting to allow complete links to attach to the environment. Crawling on the hands and knees, and sitting, are examples where such grasps would be useful. A number of solutions are possible but further research is needed into which one would be the most appropriate in terms of flexibility and robustness.

6.2.6 Learning

Further improvements in the planner could perhaps be obtained from the judicious application of machine learning algorithms in different parts of our method. A prime candidate for their use would be the heuristic system. Currently each heuristic routine contains a number of parameters which require tuning when the routine is constructed. Optimal values for these could be easily determined through some form of training. Further learning methods could be used for the purpose of automatic or semi-automatic generation of edge preconditions. In the semi-automatic approach the planner might start with all the preconditions initially prespecified and through time improves upon them based on what it has learned on previous runs. Finally, learning could also be applied to a certain extent in the development of edge effects: given footage of human motion in different locomotion modes and performing different transitions, a learning algorithm could derive on its own what kind of regrasping operations are necessary, and how they should be performed.

6.2.7 Keyframe Timing Relaxation

A very interesting addition would be some form of keyframe motion optimization as suggested by [LC95]. Our method generates poses for the model to assume in its motion towards the goal, but it does not specify explicitly when these poses are to be assumed, and what the time delay should be between two consecutive configurations. This fits the description of a keyframe motion optimization's problem, in that we know what poses should be struck, but we do not know when, nor the velocities involved. By applying this method on top of our planner's solution, one should be able to obtain improved results.

6.2.8 Smoothing using splines

As illustrated in Figure 4.9, the smoothing process uses straight line segments when smoothing out the original path. When this method is applied to paths representing the movement of our model it has the undesirable effect of introducing G^1 discontinuities at every straight line juncture, which causes the model to appear to be walking like a robot, with parts of its body changing velocities in a discontinuous manner. It would be interesting to see how much better the results would be if we used a more continuous smoothing method, perhaps one that relies on splines. Even with this addition, however, it is important to note that there will still be discontinuities due to limbs grasping the obstacles, but we consider these to be desirable and being part of the "natural" look of human motion.

Bibliography

[BL91] Jérôme Barraquand and Jean-Claude Latombe. Robot motion planning: A distributed representation approach. *The International Journal of Robotics Research*, 10(6):628–649, December 1991.

[BPW93] Norman I. Badler, Cary B. Phillips, and Bonnie L. Webber. *Simulating Humans: Computer Graphics Animation and Control*. Oxford University Press, 1993.

[BW95] Armin Bruderlin and Lance Williams. Motion signal processing. In *Computer Graphics Proceedings*, Annual Conference Series, pages 97–104. SIGGRAPH, 1995.

[Coh92] Michael F. Cohen. Interactive spacetime control for animation. In *Computer Graphics Proceedings*, volume 26, pages 293–302. SIGGRAPH, 1992.

[Fac] Motion Factory. Motivate 3D game development system. http://www.motionfactory.com/.

[Gle98] Michael Gleicher. Retargetting motion to new characters. In *Computer Graphics Proceedings*, Annual Conference Series, pages 33–42. SIGGRAPH, 1998.

[HP97] Jessica K. Hodgins and Nancy S. Pollard. Adapting simulated behaviors for new characters. In *Computer Graphics Proceedings*, Annual Conference Series, pages 153–162. SIGGRAPH, 1997.

[KAB$^+$] Yotto Koga, Geoff Annesley, Craig Becker, Mike Svihura, and David Zhu. On intelligent digital actors. Online at http://www.motionfactory.com/products/whppr_imagina.htm.

[KB92] Hyeongseok Ko and Norman I. Badler. Straight line walking animation. In *Proceedings of Graphics Interface '92*, pages 273–281, 1992.

[KKKL94] Yoshihito Koga, Koichi Kondo, James Kuffner, and Jean-Claude Latombe. Planning motions with intentions. In *Computer Graphics Proceedings*, Annual Conference Series, pages 395–408. SIGGRAPH, 1994.

[Las96] Joseph Laszlo. Controlling bipedal locomotion for computer animation. Master's thesis, University of Toronto, 1996.

[LC95] Zicheng Liu and Michael F. Cohen. Keyframe motion optimization by relaxing speed and timing. In *Computer Animation and Simulation '95 – Proceedings of the 6th Eurographics Workshop on Simulation and Animation*, pages 144–153, Springer Verlag. Maastricht, Netherlands, 1995.

[LGC94] Zicheng Liu, Steven J. Gortler, and Michael F. Cohen. Hierarchial spacetime control. In *Computer Graphics Proceedings*, Annual Conference Series, pages 35–42. SIGGRAPH, 1994.

[LPW91] Jean-Claude Latombe, Cary B. Phillips, and Bonnie L. Webber. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[LvF96] Joseph Laszlo, Michiel van de Panne, and Eugene Fiume. Limit cycle control and its application to the animation of balancing and walking. In *Computer Graphics Proceedings*, Annual Conference Series, pages 155–162. SIGGRAPH, 1996.

[LWZB90] Philip Lee, Susanna Wei, Jianmin Zhao, and Norman I. Badler. Strength guided motion. In *Computer Graphics*, volume 24, pages 253–262. SIGGRAPH, 1990.

[PB91] Cary B. Phillips and Norman I. Badler. Interactive behaviors for bipedal articulated figures. In *Computer Graphics*, volume 25, pages 359–362. SIGGRAPH, July 1991.

[RH91] Marc H. Raibert and Jessica K. Hodgins. Animation of dynamic legged locomotion. In *Computer Graphics*, volume 25, pages 349–358. SIGGRAPH, July 1991.

[Sim94] Karl Sims. Evolving virtual ceratures. In *Computer Graphics Proceedings*, volume 28, pages 15–34, 1994.

[Stu84] David Sturman. Interactive keyframe animation of 3d articulated models. In *Proceedings of Graphics Interface '84*, pages 35–40, 1984.

[Tor97] Nick Torkos. Footprint-based quadruped motion synthesis. Master's thesis, University of Toronto, 1997.

[Tra] Transom Technologies. http://www.transom.com. Ann Arbor, Michigan.

[Tv98] Nick Torkos and Michiel van de Panne. Footprint-based quadruped motion synthesis. In *Proceedings of Graphics Interface '98*, pages 151–160, 1998.

[UAT95] Munetoshi Unuma, Ken Anjyo, and Ryozo Takeuchi. Fourier principles for emotion-based human figure animation. In *Computer Graphics Proceedings*, Annual Conference Series, pages 91–95. SIGGRAPH, 1995.

[van96] Michiel van de Panne. Parametrized gait synthesis. In *IEEE Computer Graphics and Applications*, pages 40–49. IEEE, March 1996.

[van97] Michiel van de Panne. From footprints to animation. In *COMPUTER GRAPHICS forum*, volume 16, pages 211–223, 1997.

[vF93] Michiel van de Panne and Eugene Fiume. Sensor-actuator networks. In *Computer Graphics Proceedings*, Annual Conference Series, pages 335–342. SIGGRAPH, 1993.

[vKF94] Michiel van de Panne, R. Kim, and Eugene Fiume. Virtual wind-up toys for animation. In *Proceedings of Graphics Interface '94*, pages 208–215, 1994.

[WK88] A. Witkin and M. Kass. Spacetime constraints. In *Computer Graphics*, volume 22, pages 159–168. SIGGRAPH, August 1988.

[WP95] Andrew Witkin and Zoran Popović. Motion warping. In *Computer Graphics Proceedings*, Annual Conference Series, pages 105–108. SIGGRAPH, 1995.