Chapter 3

Goal-Directed Navigation

3.1 Introduction

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

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

Related Work

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

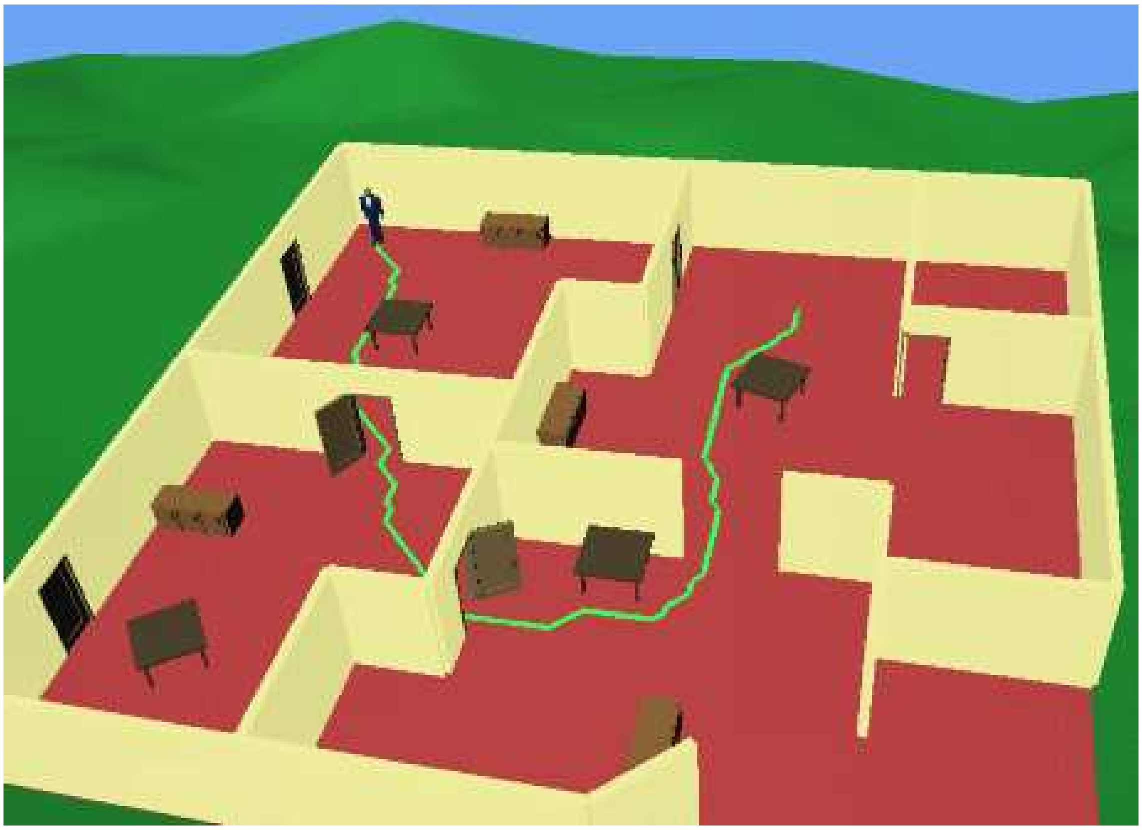

Figure 3.1: A human character navigating in a virtual office.

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

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

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

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

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

3.2 Navigation using Planning and Control

Overview

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

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

Limiting the Degrees of Freedom

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

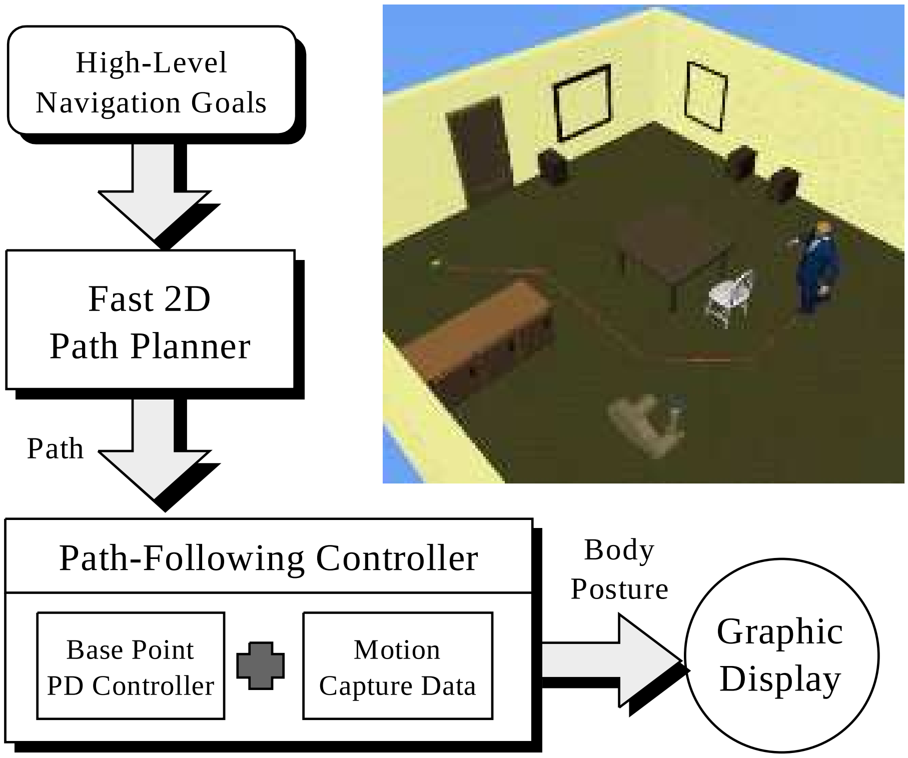

Figure 3.2: Data flow for the navigation strategy

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

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

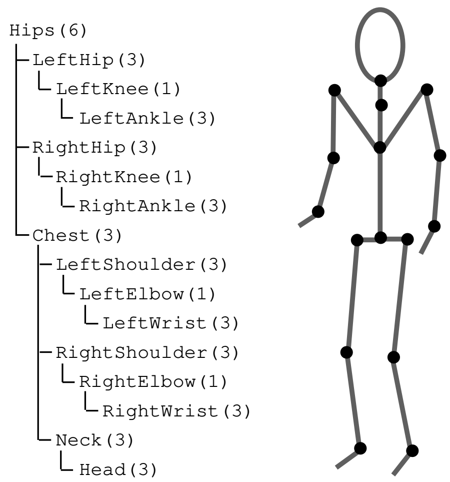

Figure 3.3: The major joints in the kinematic hierarchy used for locomotion are shown, along with the number of DOF for each joint.

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

Using a Bounding Volume

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

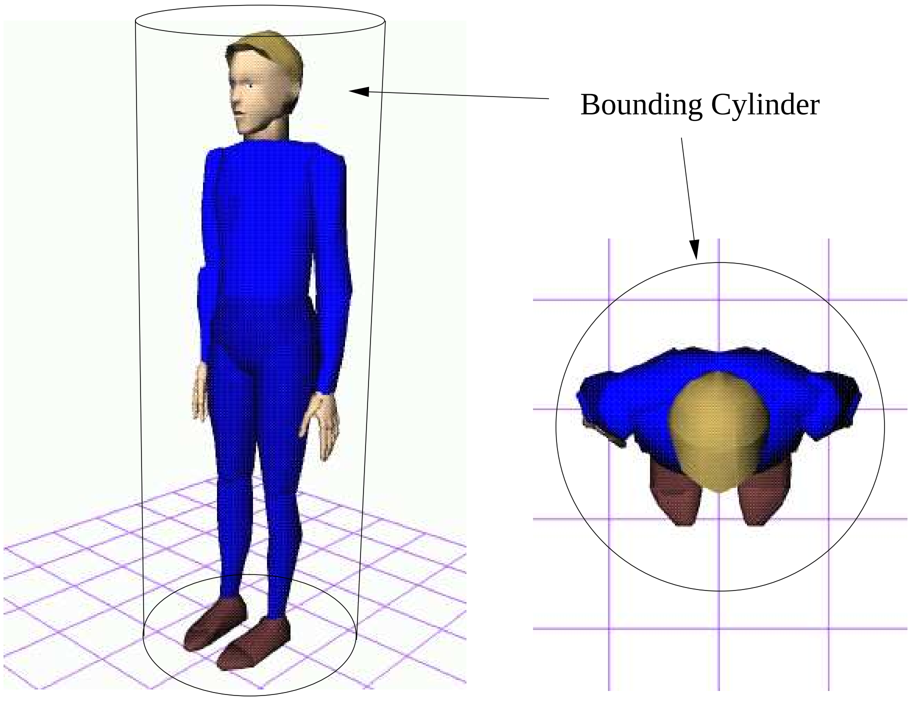

Figure 3.4: The character's geometry is bounded by an appropriate cylinder.

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

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

3.3 Algorithm Outline

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

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

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

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

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

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

3.4 Initialization

Environment Representation

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

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

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

The Goal Location

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

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

Preparing the Motion Capture Data

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

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

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

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

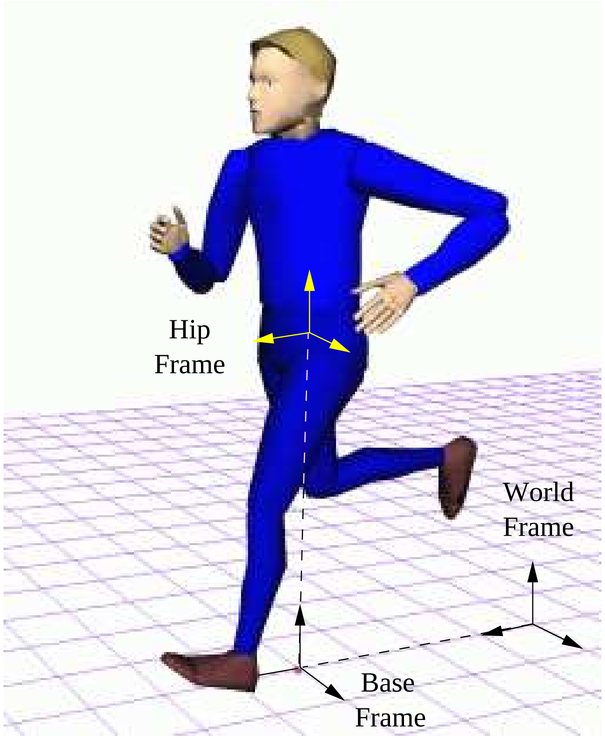

Figure 3.5: Computing the "treadmill" frame of reference.

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

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

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

3.5 Planning

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

Obstacle Projection

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

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

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

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

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

Path Search

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

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

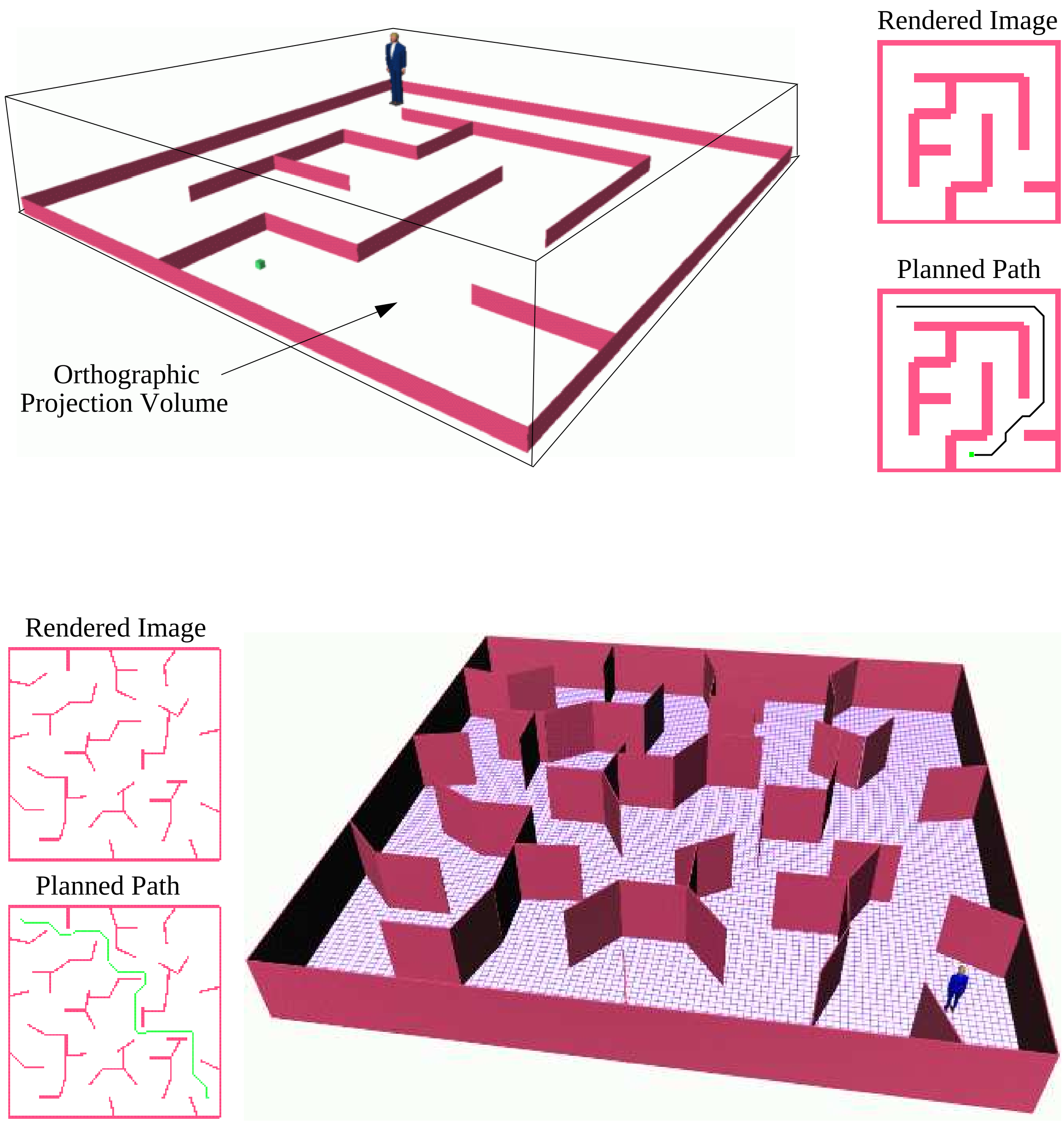

Figure 3.6: The view volumes resulting from an overhead orthographic camera. The near and far clipping planes of the camera are defined according to the vertical extents of the character geometry. All obstacle geometry contained within the view volume is rasterized on a 2D grid for planning. The resulting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

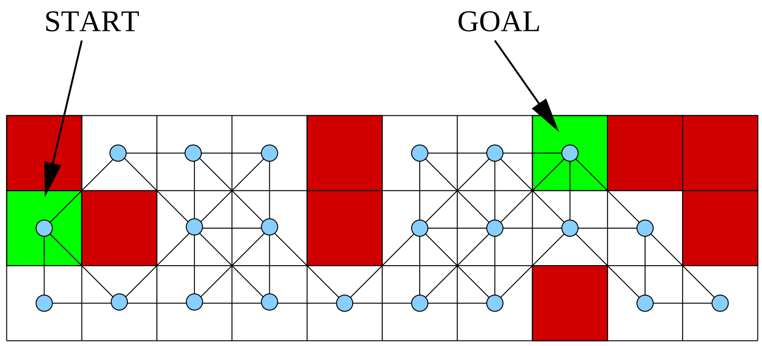

Figure 3.7: Searching an embedded graph defined by a grid

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

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

Searching a Weighted Graph

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

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

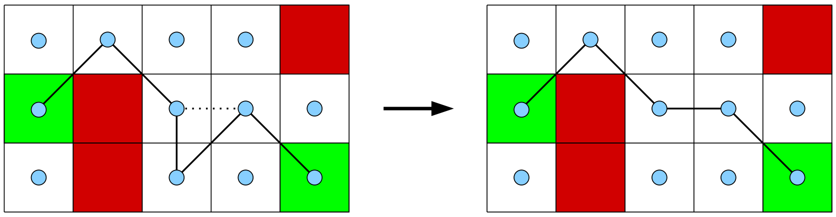

Figure 3.8: Paths which form angles less than 90 degrees can always be shortened.

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

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

Fast Planning on a Square Grid

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

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

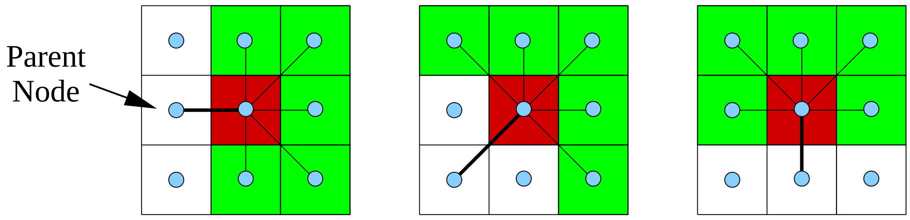

Figure 3.9: Reducing the number of neighbor node expansions.

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

Properties of Computed Paths

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

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

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

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

Complexity Analysis

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

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

3.6 Path Following

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

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

Mathematical Model

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

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

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

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

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

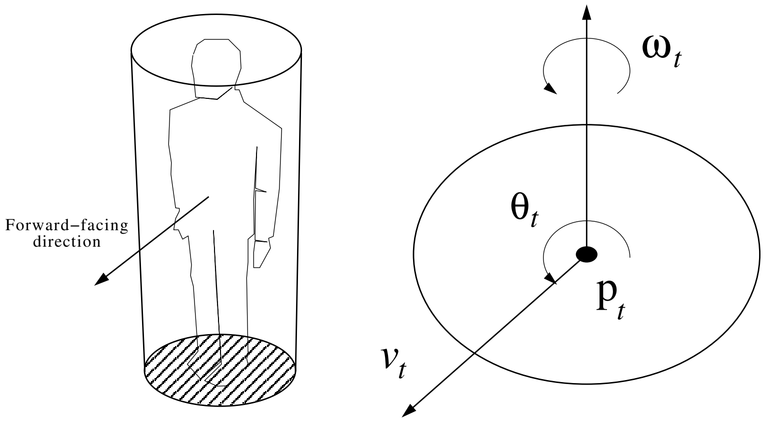

Figure 3.10: The controller model considers the character's motion as that of an oriented disc in the plane. The center of the disc corresponds to the projection of the origin of the root joint of the figure onto the walking surface.

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

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

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

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

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

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

Calculating the Controls

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

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

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

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

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

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

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

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

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

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

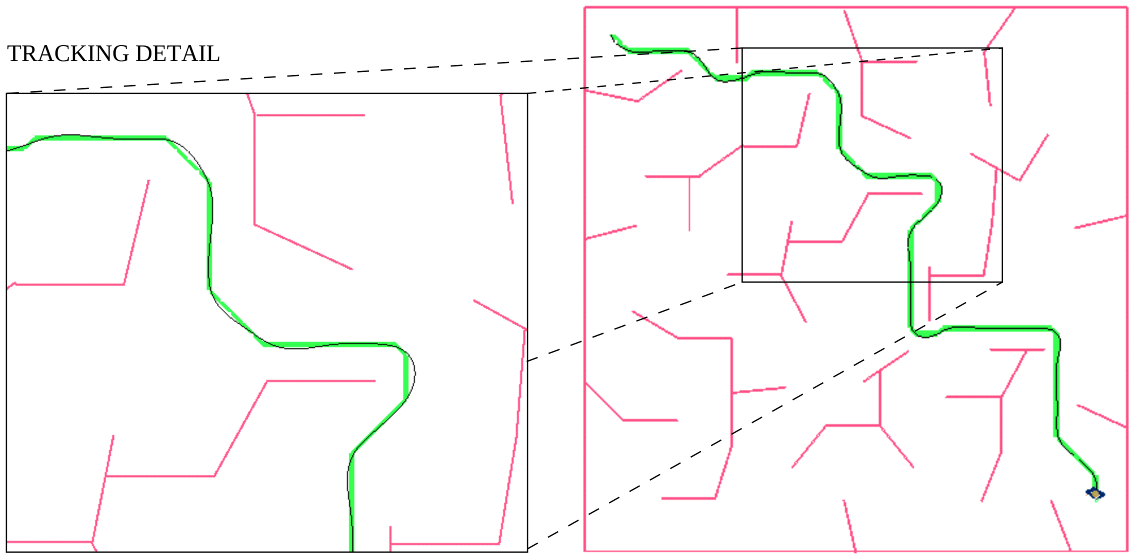

Figure 3.11: Base point tracking performance for path following.

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

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

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

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

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

3.7 Multiple Characters

Sharing the Initial Bitmap

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

Coordinating Multiple Individuals

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

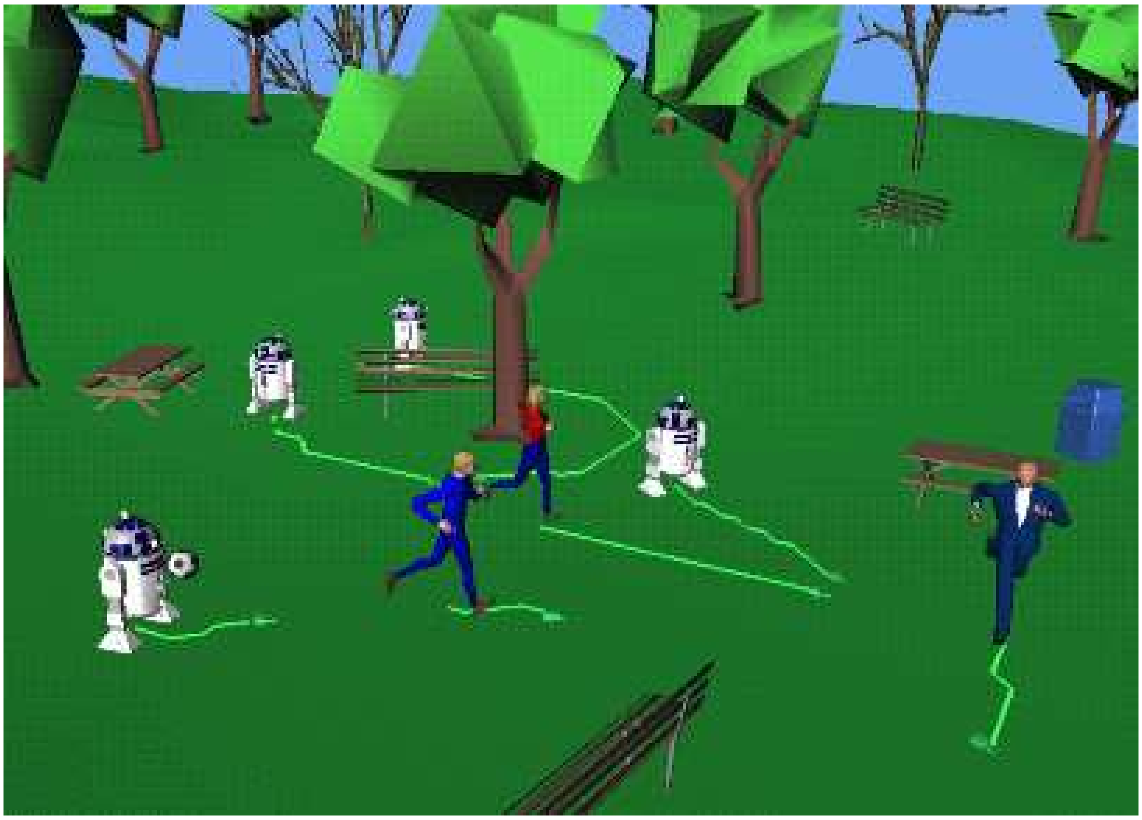

Figure 3.12: Wandering and following behaviors using a velocity-prediction scheme to avoid inter-character collisions.

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

Customized Controllers

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

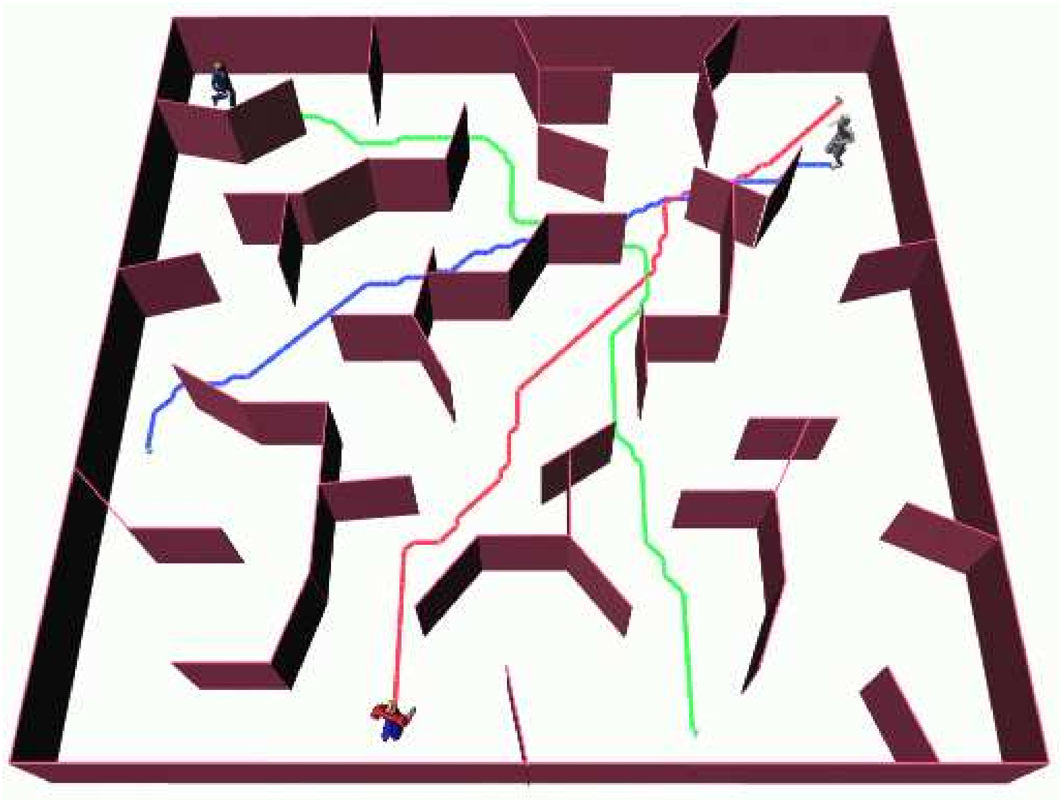

Figure 3.13: Multiple Characters navigating in a maze.

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

3.8 Results and Discussion

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

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

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

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

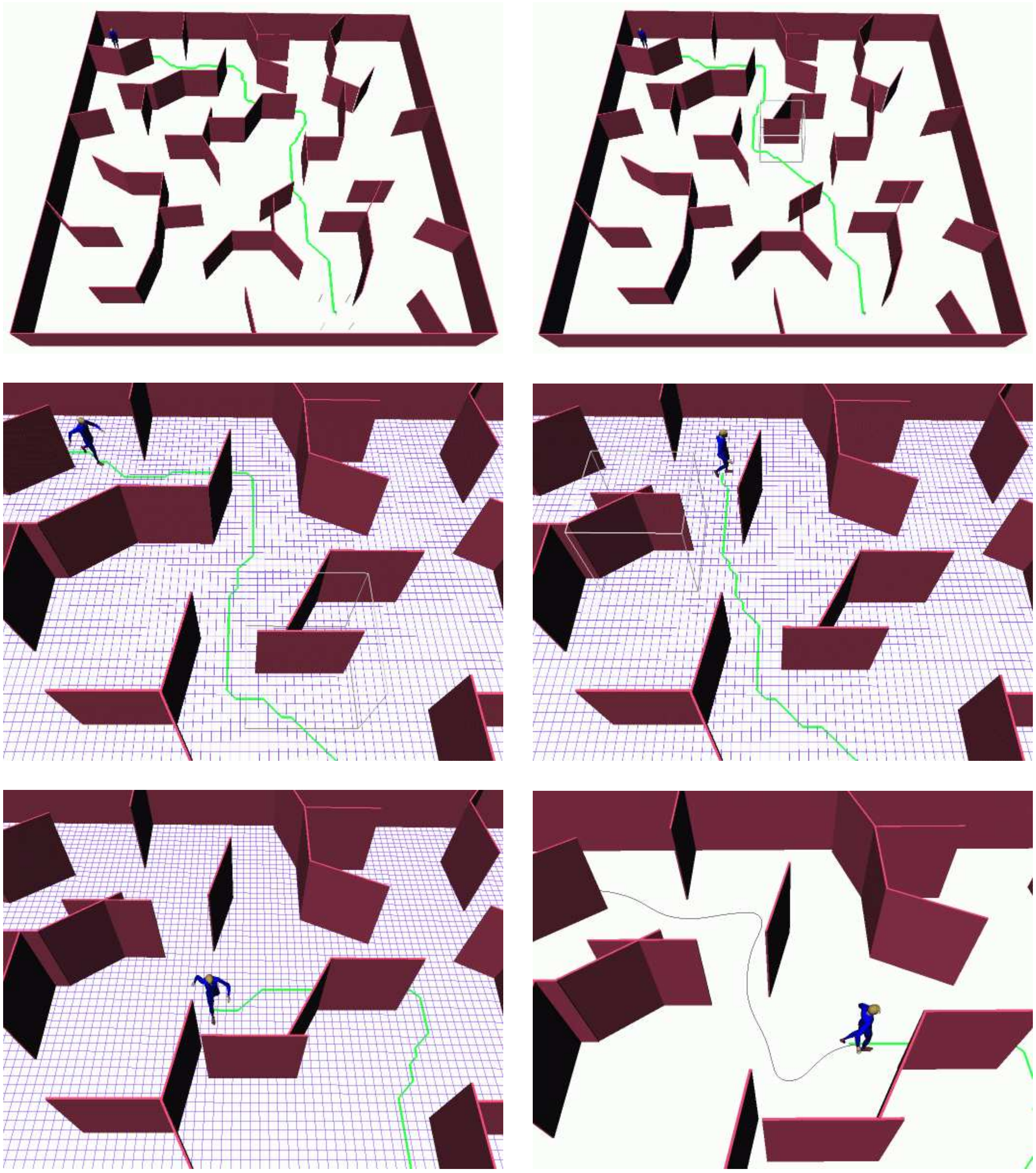

Figure 3.14: A human figure navigating in a maze environment. The path is dynamically recomputed as the goal or obstacle locations change. The detail images illustrate walls being repositioned interactively by the user and the resulting effect on the character's motion. The last image shows the trace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

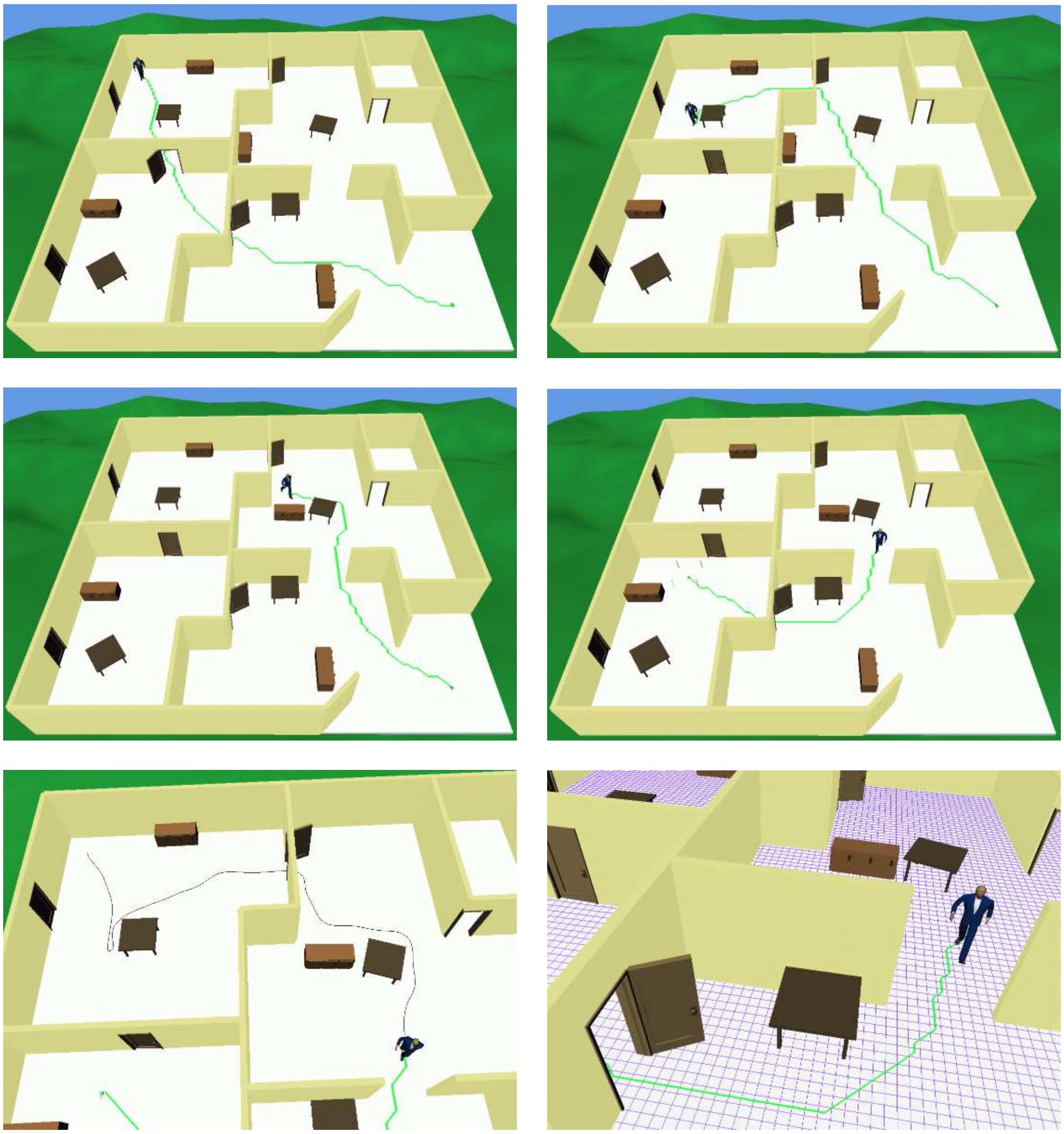

Figure 3.15: A human figure navigating in an office environment. Initially, the character follows a path towards the goal located in the lower-right corner. Later, a door closes forcing the character to select an alternate route. Later still, the character avoids an obstacle placed in its path, and the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

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

Table 3.1: Average total execution time for planning.

Performance Statistics

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

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

Discussion

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

Uneven Terrain

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

Large Environments

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

Path Following

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

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

Planner Completeness

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

Path Searching Alternatives

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

Varied Walking Surfaces

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

Additional Obstacle Avoidance Maneuvers

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

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

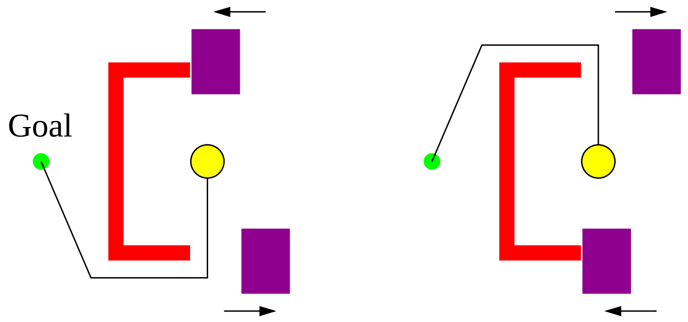

Figure 3.16: Behavior oscillation due to dynamic obstacles. A character attempting to navigate around the wall towards the goal will repeatedly oscillate between following the two paths shown above if the large obstacles at the edges of the wall alternately move back and forth.

Preventing Behavior Loops

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

Obstacle Semantics

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