Chapter 4

Sensing the Environment

4.1 Introduction

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

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

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

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

4.2 Simulated Visual Perception

Overview

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

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

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

Approximate Synthetic Vision

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

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

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

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

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

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

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

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

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

4.3 Internal Representation and Memory

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

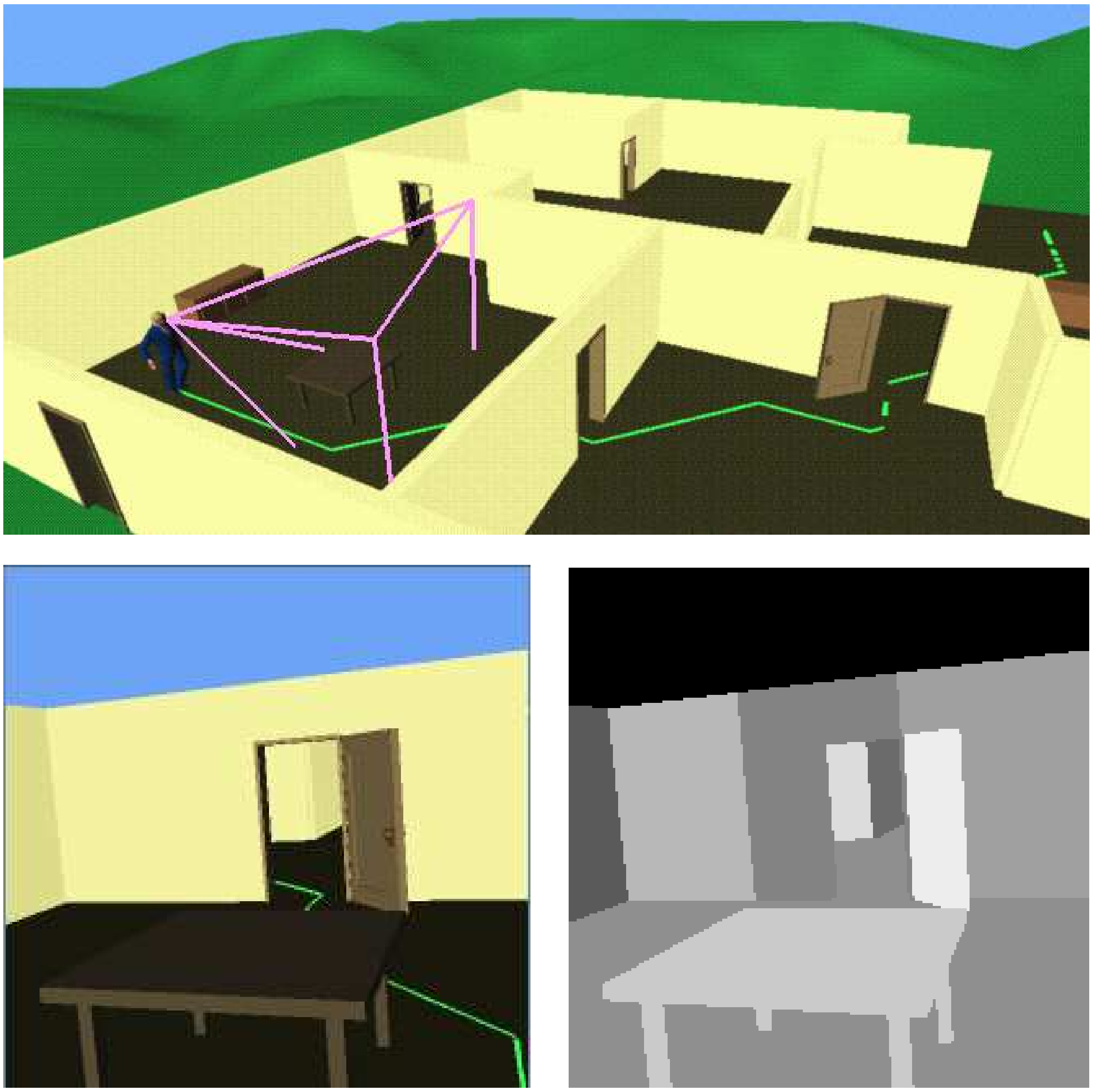

Figure 4.1: The top image shows a character navigating in a virtual office. An outline of a representative portion of the character's viewing frustum is shown. The bottom two images show the scene rendered from the character's point of view. The image on the right is the true-color, lighted model, while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

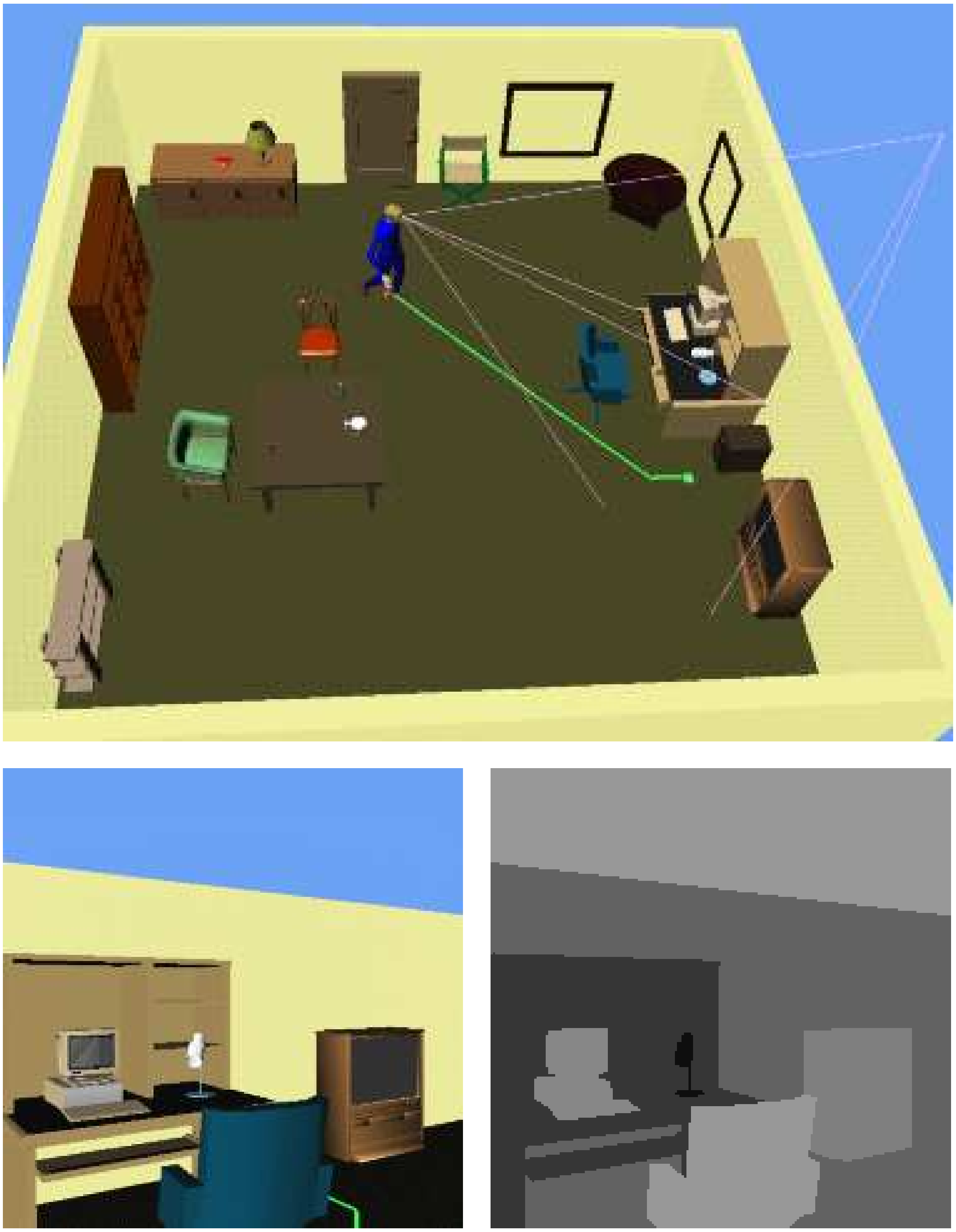

Figure 4.2: Another example view in a virtual office scene.

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

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

4.4 Perception-Based Navigation

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

Updating $\mathcal{M}$ in Static Environments

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

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

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

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

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

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

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

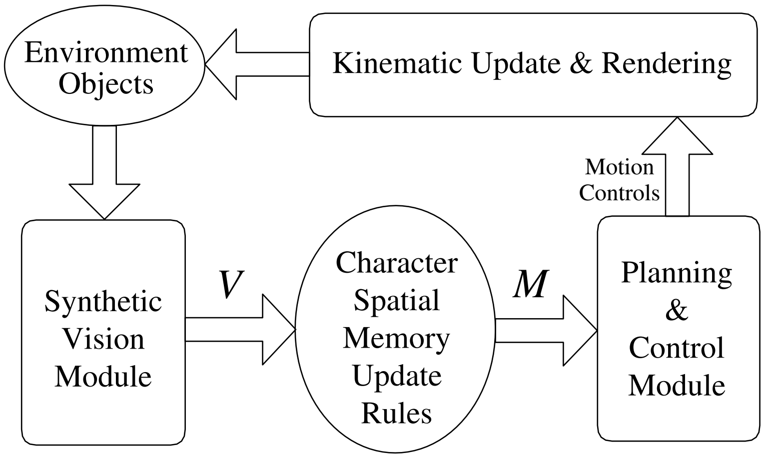

Figure 4.3: The basic sense-plan-control loop for static environments.

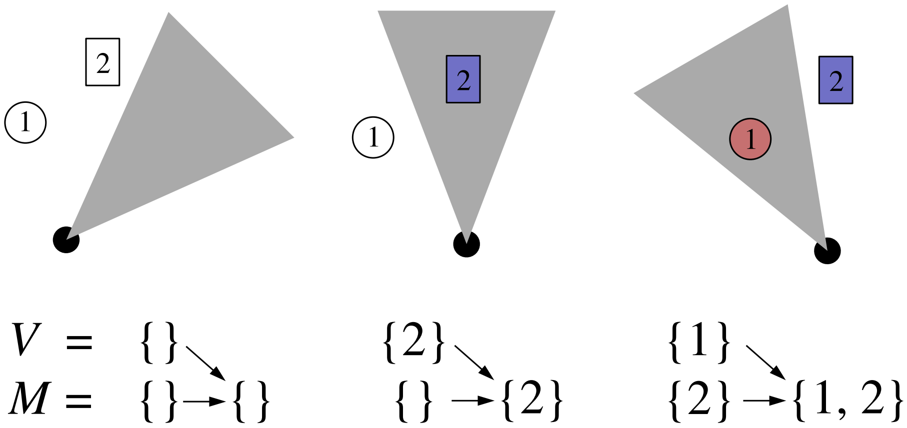

Figure 4.4: Updating $\mathcal{M}$ in static environments

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

Updating $\mathcal{M}$ in Dynamic Environments

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

- A character observes an object o_2 at a location T_2. Later, o_2 moves to a new location T'_2, and is again observed. In this case, the state information for o_2 in $\mathcal{M}$ should be simply updated to reflect the new location.

- A character observes an object o_1 at a location T_1. Later, o_1 moves to a new location T'_1. However, before the new location is observed, the character observes its former location (o_1 is now missing). In this case, the previous observation of o_1 in $\mathcal{M}$ should be deleted or invalidated.

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

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

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

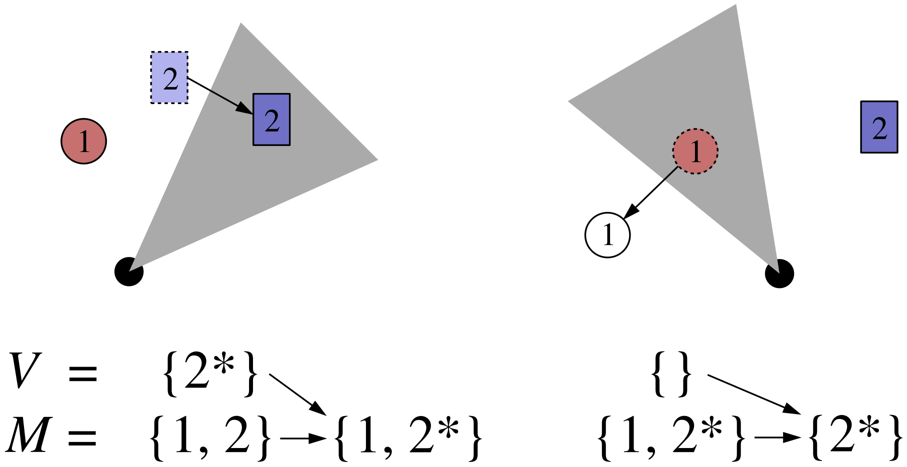

Figure 4.5: Updating $\mathcal{M}$ in dynamic environments

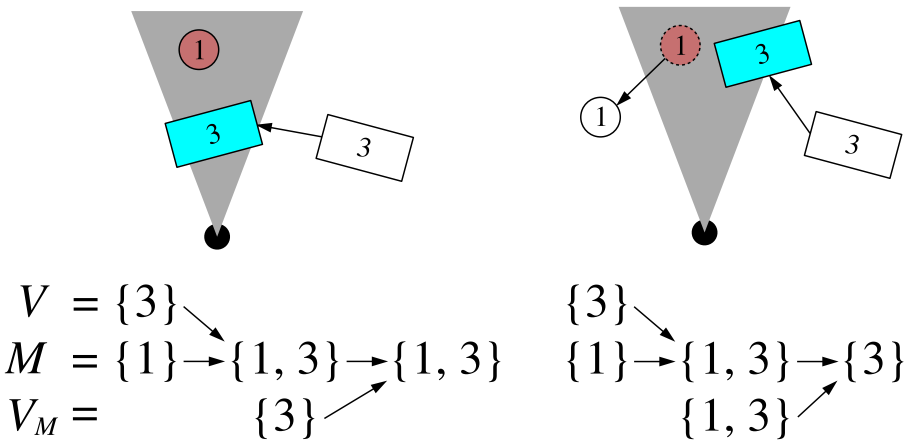

Figure 4.6: "Obscured" vs. "Missing" scenarios requiring $\mathcal{V}_M$ in order to distinguish between them and update $\mathcal{M}$ properly.

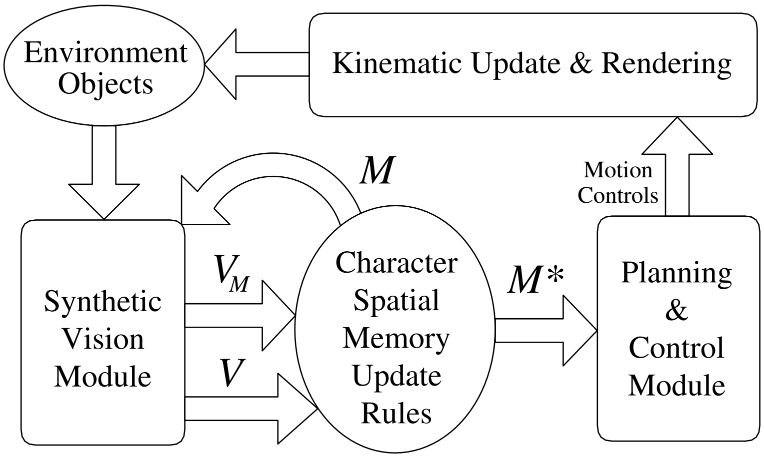

Figure 4.7: The revised sense-plan-control loop for dynamic environments.

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

4.5 Learning and Forgetting

Memory Types

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

Rule-Based Models

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

Probabilistic Models

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

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

4.6 Perception-Based Grasping and Manipulation

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

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

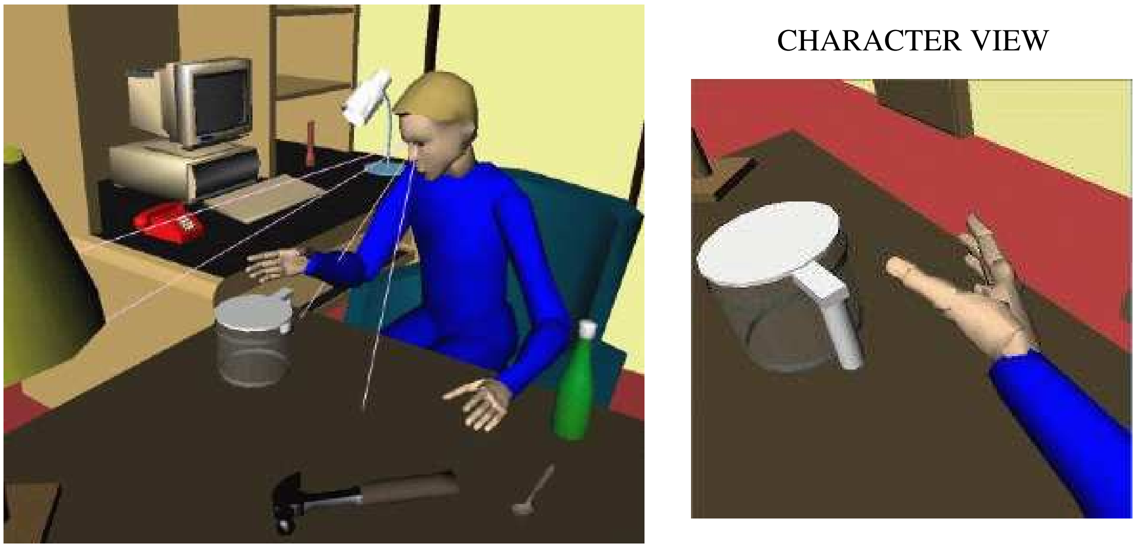

Figure 4.8: Grasping a coffee pot

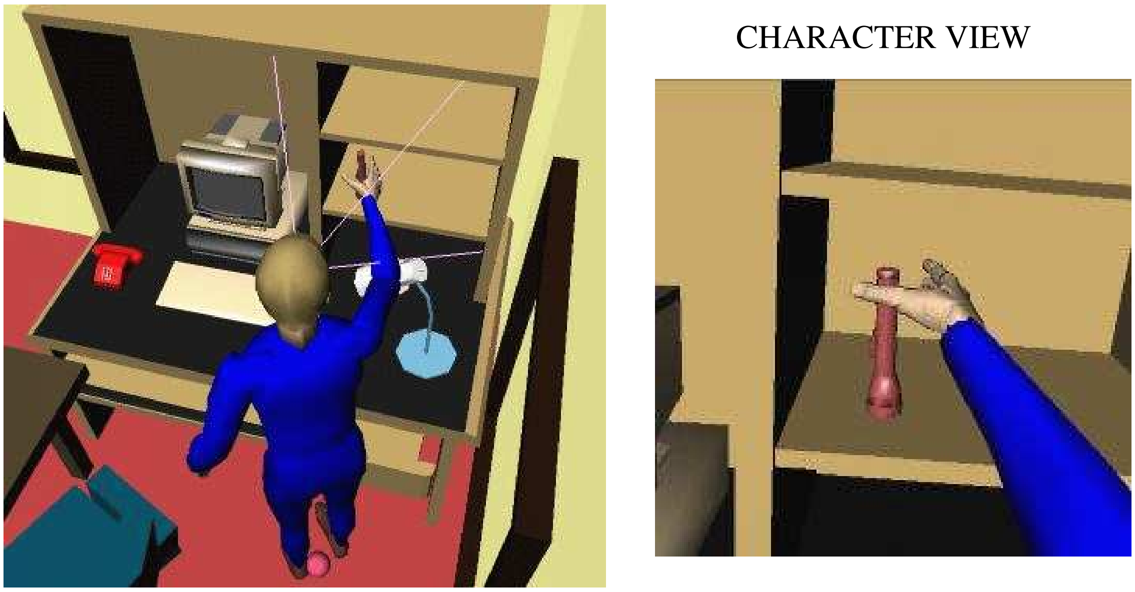

Figure 4.9: Grasping a flashlight

4.7 Other Forms of Perception

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

Simulated Aural Sensing

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

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

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

Simulated Tactile Sensing

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

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

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

4.8 Results and Discussion

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

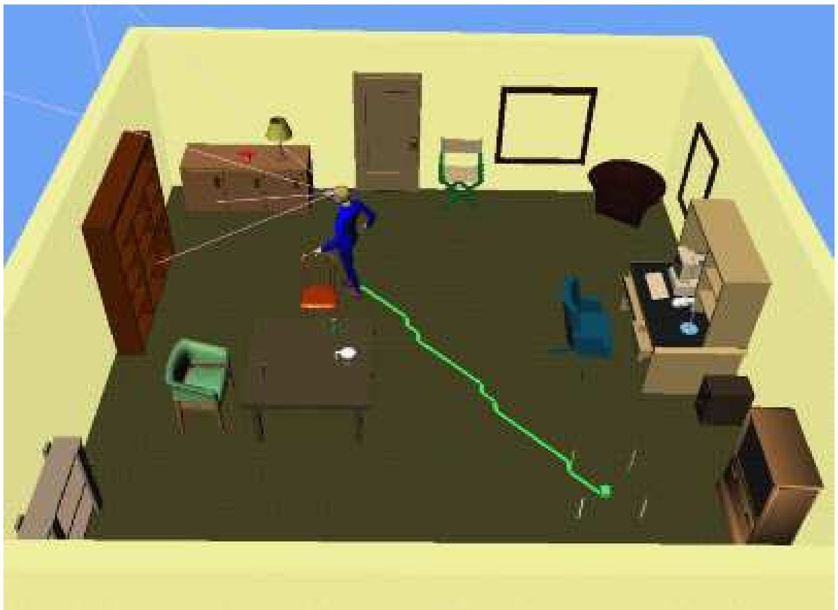

Figure 4.10: After viewing all objects in a room, a character is commanded to walk to one corner.

simulations of many more characters.

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

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

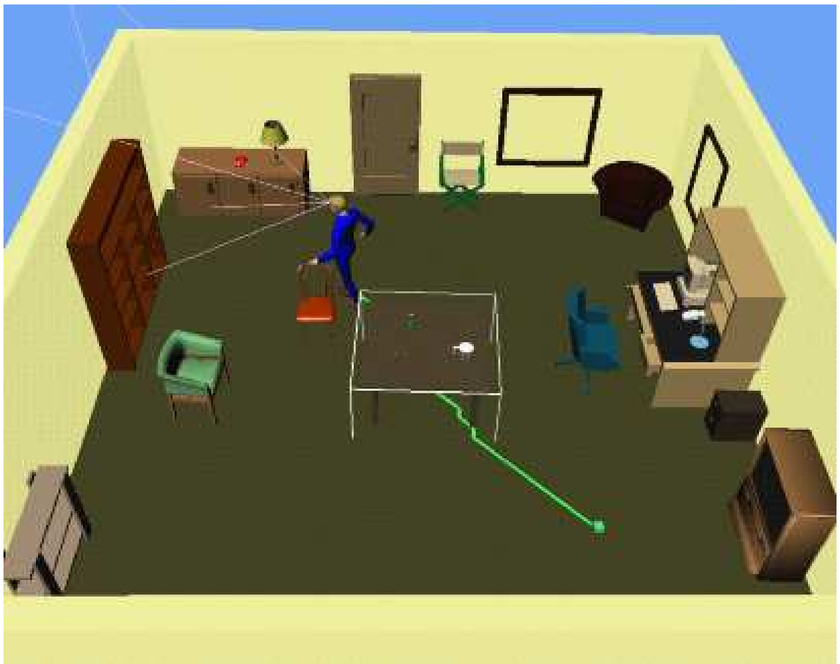

Figure 4.11: A table is repositioned behind the character's back.

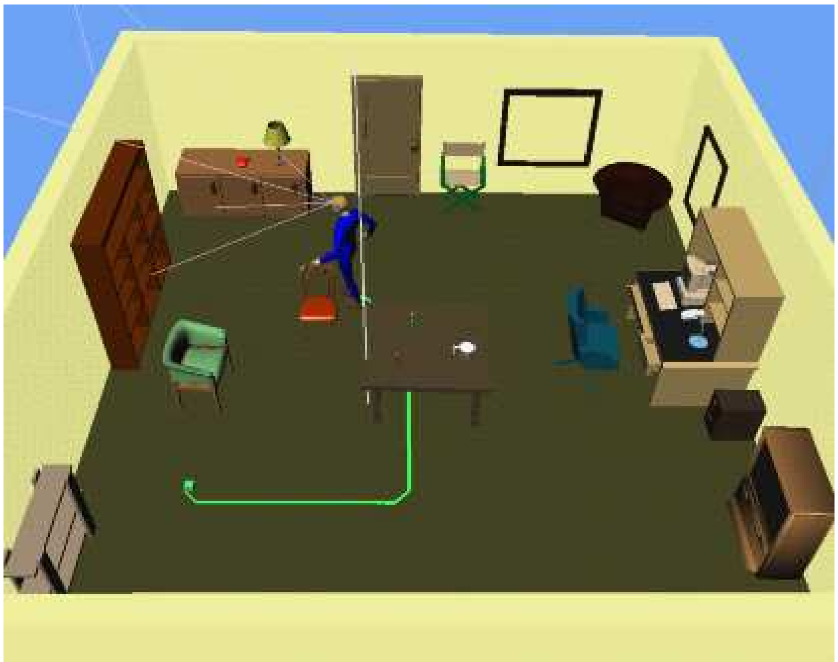

Figure 4.12: As the goal location is moved, the character continues to plan using the table's formerly observed location.

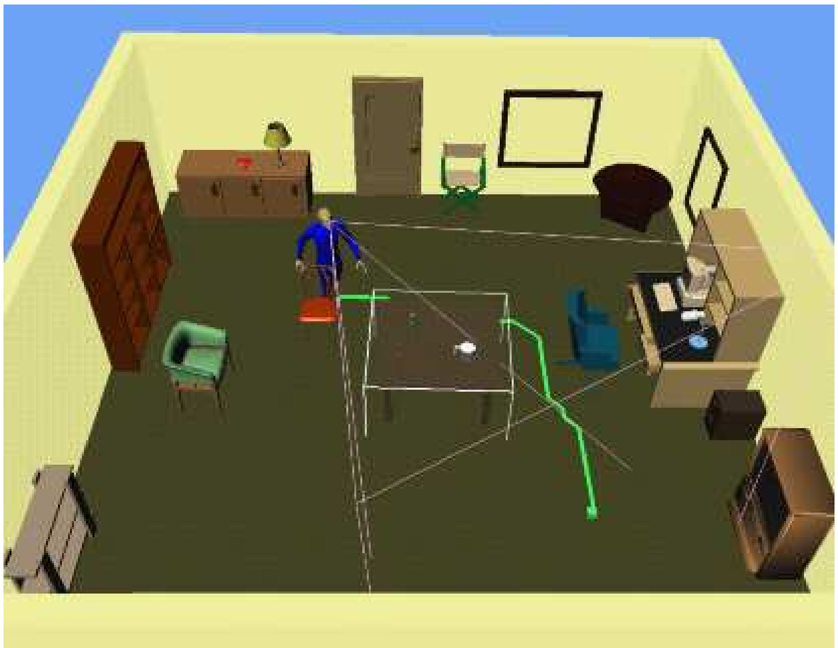

Figure 4.13: After the character turns around and sees the table in its new location, it updates its memory model and replans.

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

Discussion

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

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

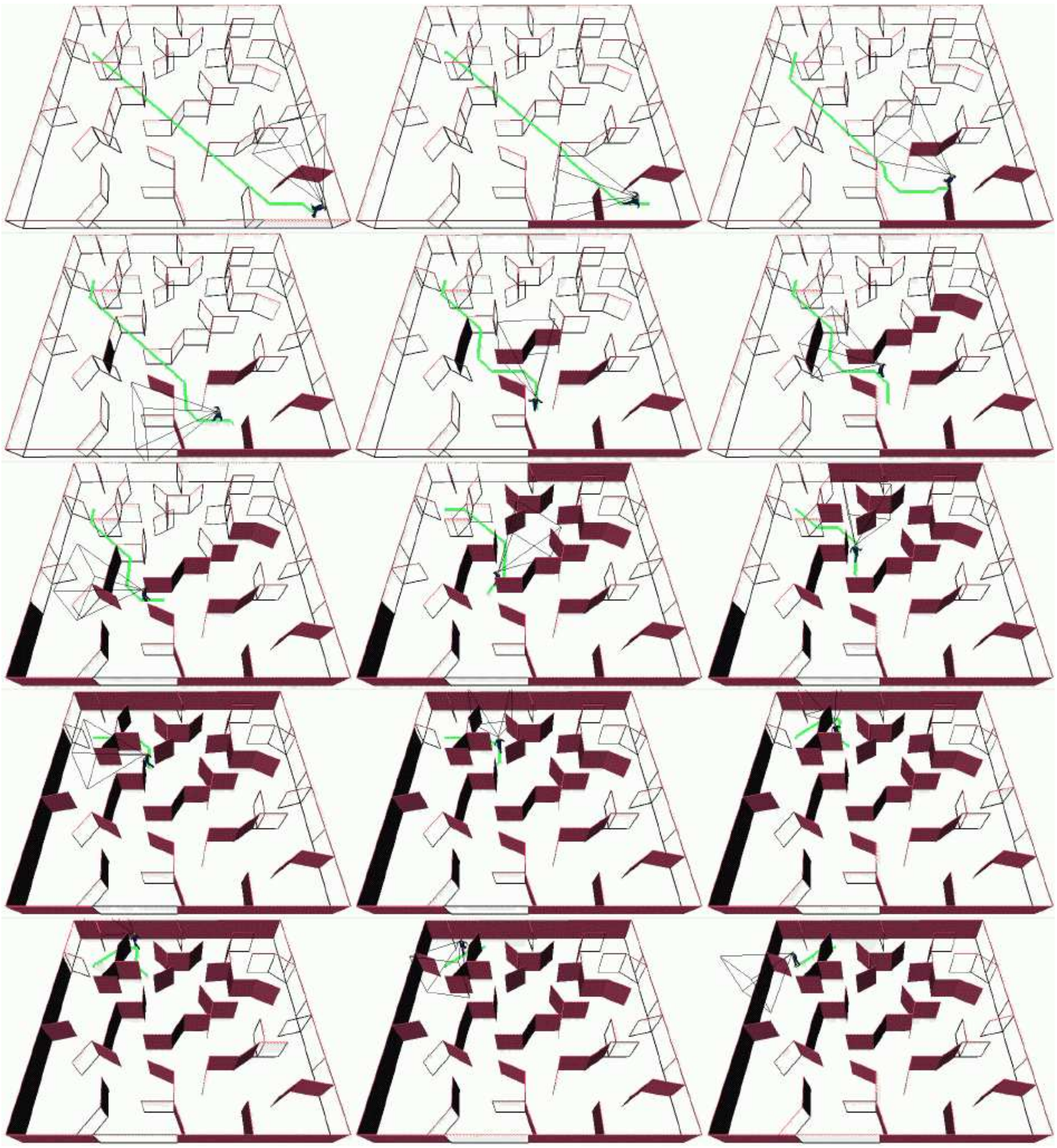

Figure 4.14: Snapshots of a character exploring an unknown maze environment. Objects rendered solid are those contained in $\mathcal{M}$, while those rendered wireframe are unknown. The top-left image shows the initial frame with the character given the task of navigating from the bottom-right corner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

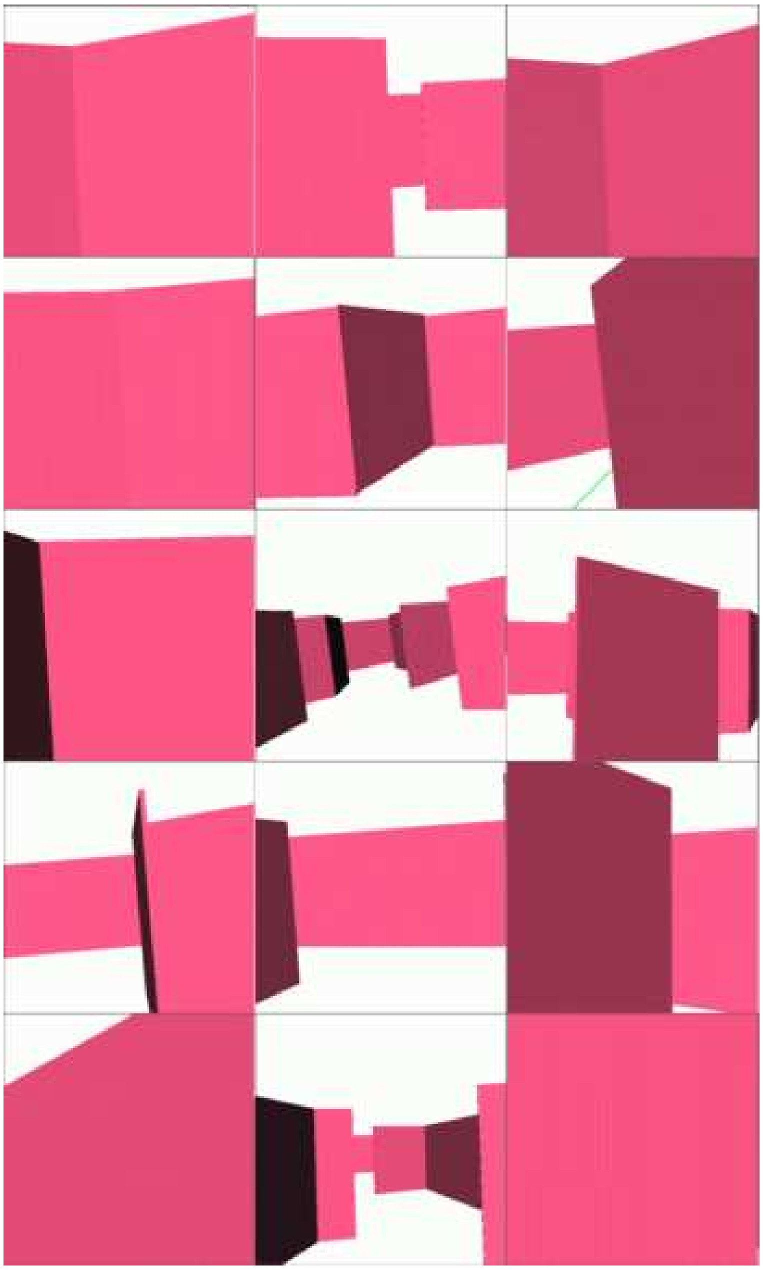

Figure 4.15: Images of the scene rendered from the character's viewpoint during each of the snapshots shown in Figure 4.14.

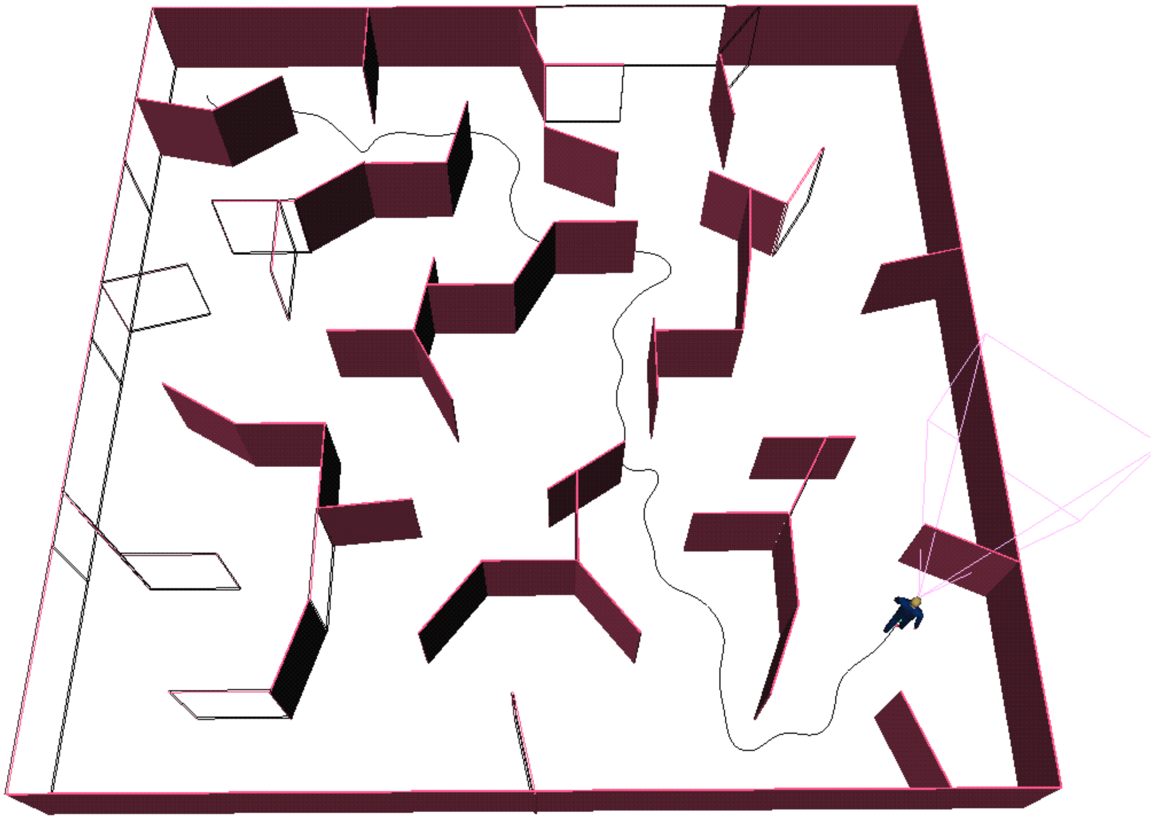

Figure 4.16: Detail of the trajectory followed by the character during the exploration of the unknown maze environment of Figure 4.14.

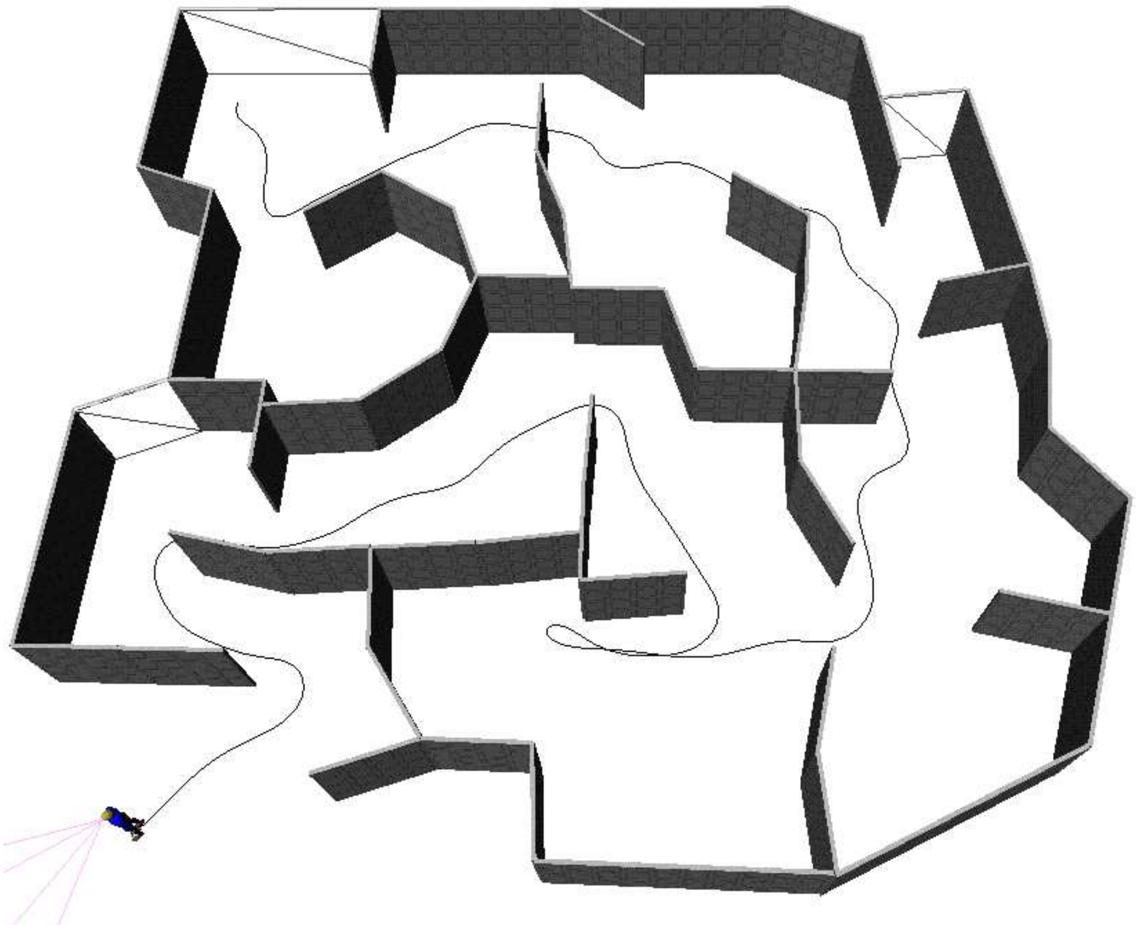

Figure 4.17: The trajectory followed during exploration of another unknown maze environment.

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

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

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

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

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

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

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

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

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