NASA TileWorld Manual (System Version 2.0)

Andrew B. Philips
John L. Bresina
Sterling Federal Systems
AI Research Branch, Mail Stop 244-17
NASA Ames Research Center
Moffett Field, CA 94035

NASA Ames Research Center
Artificial Intelligence Research Branch

Technical Report FIA-91-11

May, 1991

NASA TileWorld Manual

(System Version 2.2)

Andrew B. Philips and John L. Bresina

Sterling Federal Systems
NASA Ames Research Center
Mail Stop: 244-17
Moffett Field, CA 94035

September 1991

Contents

Chapter 1

Introduction

1.1 Overview and History

This manual documents the commands of the NASA TileWorld simulator, as well as providing information about how to run it and extend it. The simulator, implemented in Common Lisp with Common Windows, encodes a particular range in a spectrum of domains, for controllable research experiments, collectively called NASA TileWorld. The domains in the spectrum all involve a grid of cells, a set of tiles, and a set of agents which can grasp and move tiles. Points along the spectrum vary in terms of tile characteristics, agent capabilities, single agent vs. multi-agent, grid topology, and the underlying physics of the grid. Alternatives along these dimensions are briefly described in the next section.

NASA TileWorld is historically related to the sliding tile domain developed by N.S. Sridharan, C.F. Schmidt, and J.L. Bresina (reported in [Sridharan and Bresina, 1984]). In the Summer of 1989, Bresina sketched the initial design of the NASA TileWorld domain; this sketch was refined by Bresina, Philips, Mark Drummond, and Mark Boddy to form the simulator specifications. The implementation of the specifications was carried out by Philips. Around this same time period, other related simulated domains were developed independently; e.g., the tileworld at SRI [Pollack and Ringuette, 1990] and Sutton's gridworld [Sutton, 1990]. Though similar in name, these three simulated domains are rather different in nature. For a detailed discussion on the design motivation for TileWorld see [Philips et al., 1991].

In the rest of this manual, whenever "TileWorld" is mentioned, we are referring to the NASA TileWorld simulator.

1.2 TileWorld Simulator

TileWorld is a two-dimensional grid of cells populated with tiles and a single agent. The grid cells in TileWorld are identified by their (x, y) absolute coordinates, where $(0, 0)$ is the lower left cell; above this corner cell is $(0, 1)$ and to the right is $(1, 0)$. The grid is oriented such that North is up, South is down, East is to the right, and West is to the left. A tile is a polygon which has a name and fits in a single cell.

The agent fits in a single cell and has four grippers which extend in the four compass directions. The agent can perform the following effector actions: use one of its grippers to grasp a tile in an adjacent cell (in a compass direction), release a tile that it is grasping, and move in a compass direction to an adjacent cell. The agent can sense its location (in absolute coordinates), can determine whether it is grasping a tile in a given direction, can sense the contents of any cell, regardless of distance to the cell or line-of-sight "obstructions", and can request the current time from the TileWorld clock.

In addition to agent-executable actions, there is an external event over which the agent has no control; this event corresponds to a "gust of wind". Winds operate as vectors of force originating from one of the four grid borders. A tile can be "blown" by a gust of wind only if the following two conditions hold: (i) the path between the tile and the wind's origin is clear; and (ii) the cell into which the tile would be blown is empty.

No two objects can occupy the same place at the same time. All actions and events are discrete. That is, at any moment, an object is entirely within one cell, never between cells. Likewise, a gripper is never partially extended nor partially grasping a tile. When the agent is grasping a tile or the wind is blowing a tile, that tile is said to be un*free*. When not being grasped or blown, the tile is said to be *free*. Only *free* tiles can be grasped or blown.

The simulator has three types of commands: interaction, display, and customization. The first type is for agent control, and the latter two types are for experiment control. Interaction commands allow an agent controller to sense the world state and operate the agent. Display commands allow the researcher to have access to and modify presentation of the output interfaces (graphical and ASCII). Customization commands allow the researcher to create a TileWorld problem instance, to tune simulator parameters (e.g., movement speed), to adjust simulator dynamics (e.g., behavior of the winds), and to introduce probabilistic errors in the behavior of the agent's effector actions (e.g., to make the agent sometimes "veer" off course or to sometimes have a gripper slip while attempting to grasp a tile).

The user can interact with the TileWorld system in three ways: by calling TileWorld functions from Lisp's top level, from inside programs, and manually via mouse clicks in the graphics window. Some TileWorld functions are not accessible through the mouse.

A key feature of the simulator implementation is that it uses the Franz Lisp multiprocessing environment to run in real time. In this way, an agent control program and TileWorld can run in the same Lisp image (without the user having to write a process scheduler). The dynamic,

real-time behavior of the simulator, along with the facility to specify probabilistic effector errors, enables the construction of experiments that are more like "real world" problems in the sense that no two runs will be identical.

1.3 Getting Started

This section decribes how to install TileWorld into your system, how to load the files under Lisp, and how to start the simulator. This version of TileWorld runs on Sun3 or Sun4 (Sparcstation) computers with Allegro Common Lisp[1], Version 3.1.13 and Version 4.0.[2] Note: the TileWorld system requires the Common Lisp loop macro. In Allegro Version 3 you must call `(require :loop)` before you load or compile the system.

To get a copy of TileWorld, send mail to tileworld@ptolemy.arc.nasa.gov. We will arrange to send a single file to you via some convienent route (tape, ftp, Sparc floppy, or Email).

1.3.1 Installation

After you have received the file containing the system, make a directory and place the file there. The directory containing this file will be referred to as, `.../tileworld/`. The name of the file will be '`tw.tar.Z.shar`'. Of course these need not be the actual names of the directory or the file but are used in the following discussion for ease of communication.

Change directory to `.../tileworld/` and type the following at a UNIX[3] prompt:

```
sh tw.tar.Z.shar
```

The system will automatically unpack, uncompress, and untar itself. At this point, `tw.tar.Z.shar` is no longer needed, but keep a copy around until TileWorld has been run successfully. Check Appendix A on page 45 to make sure that all files are present.

In the file '`startup.lisp`', there is a global variable called `*tileworld-path*`. This global contains path information on the location of the TileWorld system. This must be set to "`.../tileworld/`" for the system to run.

1.3.2 Starting TileWorld

For purposes of speed, you may wish to compile all of the files by loading (within Lisp) the file '`compile.lisp`' in the `.../tileworld/` directory. Loading the file '`startup.lisp`', in

[1] Allegro CL is a registered trademark of Franz, Inc.

[2] We attempted to make the simulator as general as possible and, therefore, more likely to run on other platforms. If you modify TileWorld to make it run in other Lisp systems, we welcome your feedback.

[3] UNIX is a trademark of AT&T Information Systems.

.../**tileworld/**, loads all of the files necessary for TileWorld to run correctly.

One command starts the system, **tileworld**. It takes two arguments, **width** and **height**, and six **&key** arguments. The **:display** key for the **tileworld** command is used to determine the type of I/O used, graphics, or ASCII. If your Lisp has common-windows, use the command **(tileworld width height :display :graphics)**, otherwise use **(tileworld width height :display :ascii)**. If you wish to use both the graphics interface and the ASCII interface use the command **(tileworld width height :display :both)**.

The following chapter presents a tutorial on using TileWorld starting with the commands used by the agent controller and then the commands used by the experimenter, presenting increasingly more advanced material on possible uses and customizations of the simulator. To understand the details of using TileWorld, consult the Command Reference on page 23 and the Programmer's Notes on page 35.

Chapter 2

Tutorial

Sections in this chapter describe how to use various commands listed in the command reference in Chapter 3. Each section has a short tutorial followed by at least one *dribble*d output of a Lisp run using the commands in that chapter. Some of the *dribble*d outputs start off by loading a file to setup TileWorld in a particular configuration. These files are located in a directory called .../`tileworld/demo` and should be loaded as directed. Output that is in boldface type, e.g., **(release 'N)**, directs you to type that command into Lisp. A comment such as ";;; See Figure 2.1" in the Lisp code indicates that the configuration depicted in the figure will occur sometime after that command is executed. These figures contain snapshots showing the exact appearance of the TileWorld graphics window.

If your Lisp does not support graphics, there are two important differences between what is shown here and what TileWorld can display on your machine. First, when loading the demo files, load the ASCII setup file instead of the graphics setup file. For instance, at the top of page 10 is the command **(load ".../tileworld/demo/setup-graph.lisp")**. This sets up the TileWorld with graphics. Substitute *ascii* for *graph* in the file name of the load command and the TileWorld session will be configured for ASCII only. Second, although each figure reference contains a description of the current state, you will not see this because in ASCII mode, TileWorld will not display the grid automatically. To examine the contents of TileWorld, use the command **(`ascii-display`)** (see section 2.2.2, page 12).

2.1 Interaction Commands

The interaction commands allow an agent controller to operate the agent and sense the world state. These are intended to be the only commands that the agent controller uses to interact with TileWorld.

2.1.1 Effectors

The effector commands are **grasp**, **release**, and **move-agent**; they allow the agent controller to manipulate the world. Each of these commands requires a *direction* argument.

The command **move-agent** attempts to move the agent one cell in a horizontal or vertical compass direction. The agent and any tile it holds cannot move beyond the borders of TileWorld, nor can they move so that they overlap with other tiles. The agent cannot move a tile it is not grasping. This means that the agent cannot push against a tile to move it, push a tile through a grasped tile (holding a tile in the direction of motion and pushing against a second tile), nor "sweep" a tile with a grasped tile (holding a tile perpendicular to the direction of motion and pushing against a second tile). In each of these examples the command simply fails. Note that return status does not reflect failure.

The command **grasp** attempts to grasp a tile in a horizontal or vertical compass direction. The agent has four grippers and, consequently, can grasp four tiles simultaneously. Because a grasped tile is an extension of the agent, grasped tiles cannot be blown by the wind. Also, the agent cannot grasp a tile that is being blown by the wind. It is not an error to command a gripper to grasp at an empty cell.

The command **release** attempts to release a tile from one of the grippers in a compass direction. It is not an error to issue a release command to a gripper that is not grasping a tile.

Because it seems unreasonable and unrealistic for the agent to "know" the results of its actions without active sensing, function calls to these effector commands always return immediately with a value of T. That is, when the agent is asked to (**grasp** 'S), the function may return before the action is actually taken; consequently, the success of the action and its duration is undetermined.

2.1.2 Sensors

The sensor commands allow the agent controller to gain (discrete) information about TileWorld. The agent has four sensors: (i) **my-location**, which determines the cell coordinates in which the agent is located, (ii) **attached**, which determines if something is being grasped in a given direction, (iii) **in**, which examines the contents of any given cell, and (iv) **world-time**, which returns the time in seconds since TileWorld was created.

The command **my-location** returns the agent's location as a list of length 2 of absolute coordinates on the TileWorld grid. The first member of the list is the X coordinate and the second is the Y coordinate. TileWorld's coordinate system originates in the lower-left corner of the grid and continues in the positive direction to the right and up, exactly like the first quadrant of the Euclidean plane.

The command **in** takes, as arguments, an (x, y) pair describing a cell location within the

TileWorld grid and returns the contents of that cell. If the cell is empty, NIL is returned. If a tile is present, the number of sides and the name of that tile are returned as an association list. Requests for the contents of cells outside the bounds of the grid return 'UNDEFINED.

The command attached takes a direction argument and returns T if the agent is grasping a tile in that direction, and returns NIL otherwise. No other information is returned by this command; it serves only to indicate whether or not the given manipulator is grasping a tile.

The command world-time takes as single argument. If the argument is T, this sensor command returns the time in seconds since TileWorld was created. If the argument is NIL, the sensor returns an integer that is equal to a constant multiplied by the time in seconds since TileWorld was created. In most applications, the argument will be T. Two macros are provided which convert between these different types of seconds, world-time-to-seconds and seconds-to-world-time.

All sensors have perfect accuracy and unlimited range. These commands return information that is always correct at the instant in time they are called. However, there is no guarantee on how these sensor requests are ordered with respect to effector actions or other TileWorld changes like tiles being blown by wind. For instance, suppose the agent is in the cell $(0, 0)$, there is a square tile in the cell to the north $(0, 1)$, you wish the agent to grasp that tile, and to also determine if the grasp was successful. Your Lisp code might look like this:

```
(grasp 'N)
(if (not (attached 'N)) (error "Not Grasping!"))
```

Even though the Lisp command (grasp 'N) returns, that does not mean the effector action it calls for has completed. When the agent is requested to sense whether (attached 'N) is T, it cannot be known for certain if the command failed or it has not finished execution.

Sample Agent Interaction

The following page contains a sample run of the TileWorld system. A demo file is loaded which builds a TileWorld, adds an agent and adds a tile. Agent effector and sensor commands are also demonstrated. The agent moves the tile from the center of the grid to the northwest corner and executes sensor actions during the move.

To follow the sample run, load the TileWorld startup file 'startup.lisp' as explained in section 1.3.2 on page 5. Then load the file 'setup-graph.lisp' located in .../tileworld/demo. If you don't have Allegro Common Windows, use the file 'setup-ascii.lisp' and after the figure references, use the command (ascii-display) to get an ASCII representation of the TileWorld database. <user> is the Lisp command prompt. Looking closely at the text, you'll notice that (in 1 1) does not appear on the same line as the command prompt <user>. One of the mulitprocessing functions prints out the line "Loading body ...done." which appears after the Lisp command interface is ready to accept input. This or any other slight differences between the trace given here and what you obtain should produce no ill effects.

```
<user> (load "../tileworld/demo/setup-graph.lisp")

; Loading .../tileworld/demo/setup-graph.lisp.
Destroying wind
Tileworld created with agent and one 8-sided object named Elvis
T
<user>
Loading body of the agent...  pushing down to server...  done.
(in 1 1)

((SIDES 8) (NAME "Elvis"))
<user> (in 0 1)

NIL
<user> (my-location)

(0 0)
<user> (in 0 0) ;;; See Figure 2.1

AGENT
<user> (move-agent 'E)

T
<user> (grasp 'N)

T
<user> (move-agent 'N)

T
<user> (move-agent 'W) ;;; See Figure 2.2

T
<user> (attached 'N)

T
<user> (release 'N)

T
```

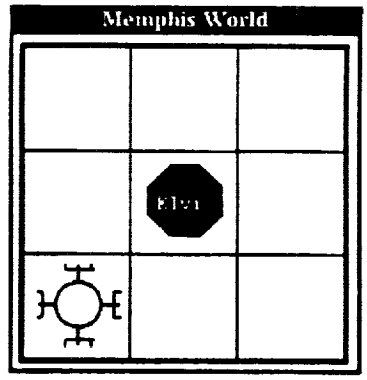


Figure 2.1: Agent and Square

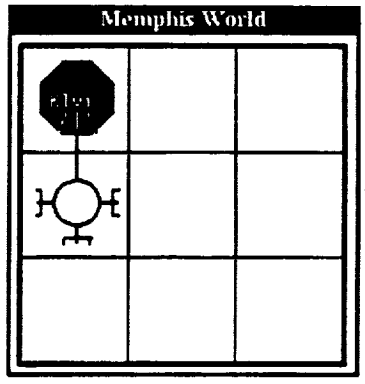


Figure 2.2: Holding Square

2.2 Display Commands

To observe changes occuring in TileWorld while an experiment is being run, either use the graphics display facility or periodically request an ASCII display. The graphics interface provides a facility to get real-time display of events occuring in TileWorld; i.e., the graphics display is automatically updated whenever a change takes place in TileWorld. The ASCII interface produces an ASCII representation of TileWorld only when the command `ascii-display` is called and does not automatically print a new display every time a change takes place in TileWorld.

2.2.1 Graphics Display

The graphics interface runs using common windows under Franz Allegro Common Lisp. It is closely coupled with the TileWorld database and relies heavily on it for update information and repaint requests to maintain screen integrity. There are a number of features that the graphics interface provides.

The graphics facility displays the current state of TileWorld in its own window. This part of the system executes efficiently to prevent I/O from becoming a bottleneck. All of the agent actions, agent movement, and grasping and releasing of tiles are portrayed in the graphics window. Tiles are moved about in the display when winds blow them. Arrows describing the wind scripts[1] are placed along the borders of the world. The color of these arrows is inverted or normal depending on whether or not the the wind scripts are paused. There are no functions to control the graphics display, all control is handled as side effects of TileWorld database manipulations.

Figure 2.3 is an example of a typical TileWorld. There are eight tiles defined, some with names, and an agent grasping one of the tiles. There are two wind scripts defined, each blowing from the east for four cells with periods of [5..20]. If the cell size is sufficiently large, data describing each script arrow is displayed within the arrow graphic.

Note that in the figure, the wind script arrows are on the border of TileWorld. Typically, only winds defined from the border are used, but this policy is not enforced by the code. Should a wind script be located in the interior of the grid, an arrow will appear inside the grid as well. The complete visual effect of this is undefined and unsupported. If you define wind scripts in the interior, it is probably best to set the variable `*display-scripts*` to `NIL`. The display window automatically resizes to a smaller window if there are no wind scripts defined or resizes larger when wind scripts are defined. This resizing behavior can be controlled via the global variable `*auto-resizing*`.

The graphics window is responsive to a number of button press events. If the middle button of the mouse is pressed and held while the mouse cursor is within a cell, the location of that

[1] For information on wind scripts see section 2.3.2 on page 16

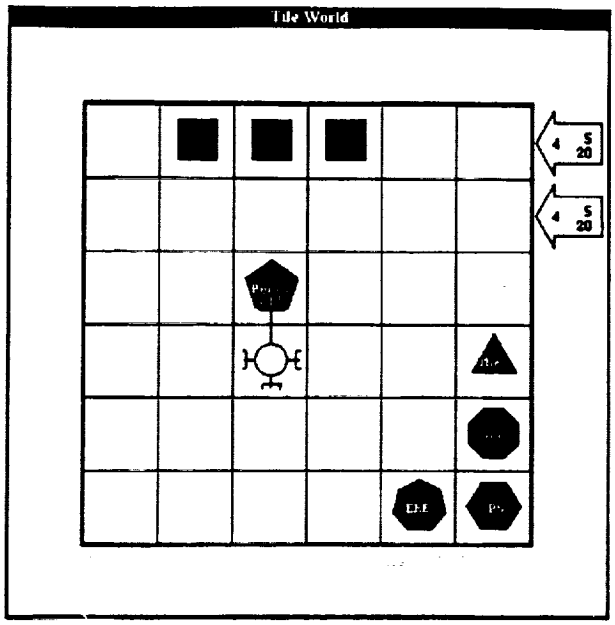


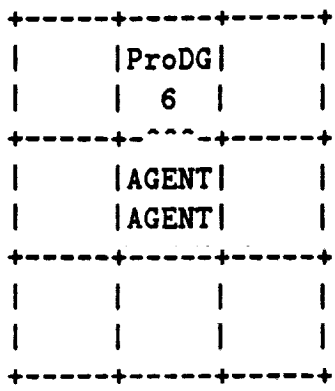
Figure 2.3: Typical TileWorld Grid

cell is displayed on the title bar of the graphics window. Also, if the middle button is pressed and held while the cursor is over a wind script arrow residing on the border, information about that script is displayed in the title bar. Should multiple scripts be defined in the arrow, the wind event that is earliest in the wind queue is selected (i.e. the next event to be blown). If the button was depressed while the cursor was over the top script arrow on the right hand side of Figure 2.3, the title bar would appear as **(5 5) W 4 [5..20]**. Other mouse button interactions are described later.

2.2.2 ASCII Display

If the ASCII display is active (to activate, see `tileworld`, section 3.3.1 on page 29), TileWorld can be examined with the `ascii-display` command. The entire grid can be displayed, or the display can be restricted to some subpart of the grid. The subpart can be specified either as a square centered wherever the agent is located or as a specific rectangular region of the grid.

For tiles, the name and number of sides are displayed. The agent is indicated by `AGENT`; to indicate that the agent is grasping a tile, "arrow heads" are displayed pointing in the direction of the grasped tile. For example, here is how `ascii-display` shows a 3x3 TileWorld with the agent grasping a six sided tile named "ProDG":



Information about the wind scripts is not displayed with this command.

2.3 Customization Commands

This section describes the domain customization commands which allow the researcher to create a TileWorld problem instance, to tune simulator parameters (e.g., movement speed), to adjust simulator dynamics (e.g., behavior of the winds), and to introduce probabilistic errors in the behavior of the agent's effector actions (e.g., to make the agent sometimes "veer" off course).

2.3.1 Static Configuration Creation

The command `tileworld` creates a TileWorld grid of a specified size. The grid can be of any width and height greater than 0 and less than the memory and array limitations of the Lisp environment. Upon creation of the grid, a title is assigned, the agent appearance is chosen, and, most importantly, the display medium is selected. If Lisp contains a graphics facility and the graphics option is selected, TileWorld creates a window for displaying changes to the world in real-time. The title and agent appearance have little relevance if the ASCII option is selected.

Once TileWorld has been created, the commands `add-object` and `add-agent` are used to place tiles and the agent onto the grid. `add-object` takes as arguments the number of sides, a name, and an xy-location. The resulting effect is the placement of a polygonal tile in the specified cell and the value returned is the object structure pointing at that tile. This structure should be saved if the user might later want to remove tiles without destroying the entire grid. This pointer is supplied solely for removal of objects by a designer and is not intended to be available to code controlling the agent (that would be cheating). The function returns `NIL` should the location already be occupied.

`add-agent` places the agent at a given xy-location. Should the location be occupied, the function returns `NIL`. None of the effector operations and only one of the sensor operations

work until the agent is defined. It is still possible to operate TileWorld without an agent, since tiles can be added and winds defined.

`remove-object` takes an object structure pointer and removes the tile from the grid. The pointers to those objects are lost if they are not saved when the object was created. The agent automatically releases a grasped tile when that tile is removed.

`remove-agent` will remove the agent from the TileWorld grid. The agent will automatically release all grasped tiles before it disappears.

`*speed-of-agent*` controls how fast the agent can move about the world. The value of this variable represents the number of grid cells that the agent can move in one second, e.g. 100 means that the agent can move 100 cells per second or equivalently it takes .01 seconds to move one cell. There is an effective upper bound on the speed of the agent related to the speed of the graphics, the speed of the machine, and the number of time slices the scheduler gives to the database process. The default value for the speed is 10 cells per second or about 100 milliseconds per move, which, it turns out, is close to the average minimum limit on human decisions/actions (about 200ms)[Kantowitz, 1974, pp. 1-39].

Sample Configuration Creation

Page 15 contains a sample configuration of the TileWorld system. A TileWorld is created using the graphics display, an agent is added, and an object named Elvis is added.

To follow the sample run, it is assumed that you already have the TileWorld system up and running. Refer to the previous section if you do not. If you are using a system that does not support graphics, replace the "`:graphics`" in the `tileworld` command with "`:ascii`". If you wish to simultaneously test what the graphics and ASCII look like (and if you have graphics) replace the "`:graphics`" with "`:both`".

```
<user> (tileworld 3 3 :display :graphics :title "Memphis World")
;;; See Figure 2.4

T
<user> Destroying wind

Loading body of the agent...  pushing down to server...  done.
(add-agent 0 0)

T
<user> (add-object 1 1 8 "Elvis")  ;;; See Figure 2.5

OBJ1
<user>
```

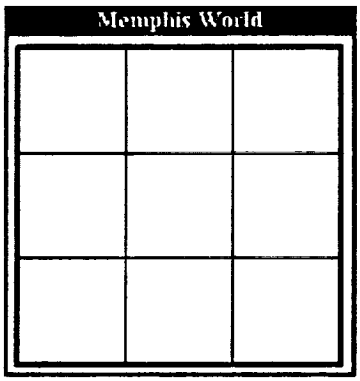


Figure 2.4: Empty Tileworld

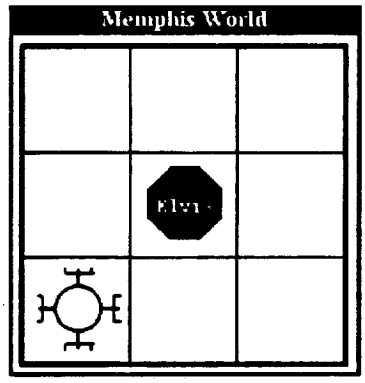


Figure 2.5: Agent and Tile added

2.3.2 User Defined Winds

Gusts and wind scripts can be created in TileWorld. A gust of wind is a single wind event that can blow a tile in a straight line; it has a point of origin, a strength, and a direction. Strength determines the number of cells a tile can be blown. If a *free*[2] tile is at the point of origin when the wind is created, that tile is blown in the direction specified for as many grid cells as the specified strength if the path is clear. If no *free* tile is present, then the gust dies. A tile blown by a gust moves to the limit of that gust, unless it encounters another object. This means that the tile will stop if it hits another tile, the agent, or a TileWorld border. To create a gust of wind, use the command `wind-blows-object`.

Wind scripts can also be created. A wind script is a gust of wind with a strength and a direction which repeats periodically. Wind scripts generally originate on the border of the world.[3] Strength determines the maximum number of cells a wind script can blow a *free* tile. Wind scripts differ from gusts in two important ways.

First, gusts occur only once, whereas, scripts occur repeatedly until paused or stopped. The period of the wind script is an interval described by a lower bound and an upper bound in seconds (real time). When a script is active, a time is randomly generated that falls within the interval (boundaries are included). The wind script then waits that many seconds and blows once. This repeats until the scripts are paused or stopped.

Second, a gust affects a *free* tile only if it exists at the origin of that gust, whereas, a wind script grabs the first *free* tile it encounters along the path described by its strength and direction and blows it to the end of that path. If the first object encountered is not *free*, the wind script stops for that iteration. The agent, a tile being **grasped**, or a moving tile in the path of a wind script stops the script for one cycle. The *strength*, *direction*, and *origin* of a gust describe a path starting at the *origin* and ending *strength* number of cells in *direction*. A path running along the entire bottom of an eight sided TileWorld is defined with its *origin* at $(0, 0)$, its *direction* as east, and its *strength* as seven.

To create a wind script use the command **`register-wind`**. It takes six arguments, two of which are optional. **`register-wind`** must have the column and row location for the origin of the wind, the direction of the wind, and the strength or distance that it blows. The last two arguments describe lower and upper bounds on the period of the wind. Once registered, the wind script processor takes control of the script and schedules it for gusting.

Sample wind creation – See next page.

[2]See definition of *free*, Section 1.2, page 4

[3]The point of origin of a scripted wind can be anywhere in the grid, but typically, the scripted winds originate only on the borders. The "real world" analogy is that the winds appear to be created from outside the bounds of the world. Also, the graphics are not very good at displaying wind scripts that originate from the interior (see section 2.2.1, page 11 for more details).

```
<user> (load " abp/tw/demo/wind-graph") ;;; See Figure 2.6

; Loading .../tileworld/demo/wind-graph.lisp.
Destroying wind
Tileworld created with agent and one 4-sided tile
T
<user>
Loading body of the agent...  pushing down to server...  done.

(register-wind 2 1 'W 2 1 1)

T
<user> (move-agent 'S) ;;; See Figure 2.7

T
<user> (move-agent 'N)

T
<user> (grasp 'W)

T
<user> (move-agent 'E)

T
<user> (release 'W) ;;; See Figure 2.8

T
<user> (move-agent 'S) ;;; See Figure 2.9

T
<user>
```

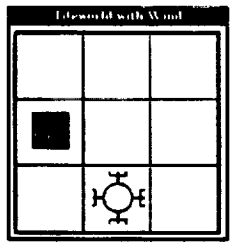


Figure 2.7: Wind blows "Paper"

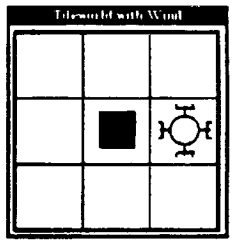


Figure 2.8: Agent blocks wind

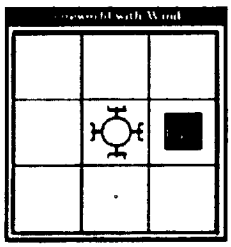


Figure 2.6: Agent supports "Paper"

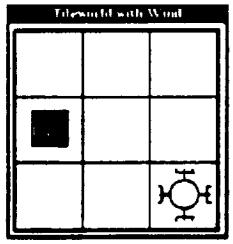


Figure 2.9: Wind blows "Paper"

2.3.3 Creating Winds with the Mouse

Through use of the mouse, the user can create wind gusts within TileWorld. To create a gust, push the left button down and release in a cell. The cursor should turn into a "+" and the cell corresponding to the ending location of the wind will flash once (see figure 2.10 for sample flash). Move the cursor around within the window to select the cell towards which the wind will blow. Push down on the left button again and a wind will be created that blows from the cell of origin in the direction and with force as specified. This is exactly as if the command `wind-blows-object` was called. Note that gusts can only occur along horizontal and vertical axes.

The user can also create wind scripts via the mouse. The process is very similar to wind gust creation. Use the right mouse button to start the wind creation process by clicking it within a border cell. Only border cells can be used to define wind scripts when using the mouse. The cursor will change to an "X" and the row or column will invert, describing the strength and direction of the script (see figure 2.11 for sample highlight). Push down on the right button again, and a wind script will be created with the point of origin, direction, and strength as specified, exactly like the command `register-wind`. The default values for the period interval bounds are used.

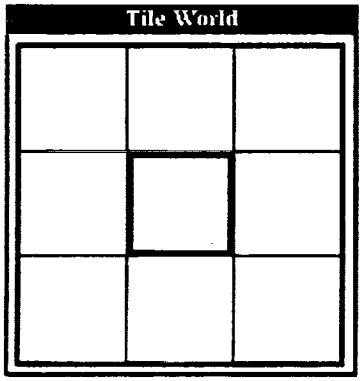


Figure 2.10: Creation of a Gust

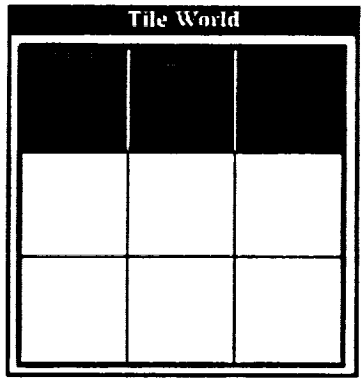


Figure 2.11: Creation of a Script

2.3.4 Specifying Probabilistic Effector Errors

In order to model some of the uncertainty inherent in real world problems, TileWorld includes a facility to specify probabilistic errors in the operation of the effector commands. A variety of options have been provided for simulated errors, but for any serious extensions, including sensor accuracy, some programming is required. See section 4.3 on page 39 for more information or contact us.

Each of the effectors has at least one failure mode known as its *deviation-class* which determines its general behavior. The three deviation classes are: (i) `:grasp`, (ii) `:release`, and (iii) `:move` which refer to the gripper slipping, the release mechanism sticking, and moving in the wrong direction, respectively.

Each effector function has a domain of arguments that it takes and each argument is a list. One *argument-list* for the function `move-agent` is `'(N)`. The *argument-domain* for the function `move-agent` is `'((N) (S) (E) (W))`. Because the exact argument list can be specified, a greater degree of control of the probabilistic errors is possible. For instance, the agent can be made to limp northwest half of the time it intends to move north by executing the following two lisp functions:

1. `(zero-deviation :move '(N))`
2. `(change-deviation :move '(N) :north 0.50)`
3. `(change-deviation :move '(N) :west 0.50)`

Line 1 zeros out the probabilities for the *deviations* of `(move-agent 'N)`. Actually, just after the probabilities are zeroed, the function `(move-agent 'N)` has no effect. Line 2 sets up the probabilities so that half the time `(move-agent 'N)` is called the agent will move north, the other half it will do nothing. Line 3 sets up the probabilities so that now half of the time the agent will move northwest, and the other half it will move just north. The `:west` is a misnomer, it should actually be `:north-west`. At no time are any of the other directions of movement affected by the these changes.

The probabilities of all deviations for an argument-list are stored as a cumulative distribution function (CDF) in a slot in `*probability-of-deviation*`. A CDF is a way of representing the sum total of the probabilities associated with a particular event. These CDFs are compiled when a deviation is added with `add-deviation` or when it is changed with `change-deviation`. For correct operation, these CDFs must never exceed 1.0.

Suppose you wish to change the current `(move-agent 'e)` command so that when the agent moves east, it will move directly east 97% of the time, move east and limp north 2% of the time and move east and limp south 1% of the time. Here, the *deviation-class* is `:move` and the *argument-list* is `'(E)`. The three *deviations* already exist and are called: `:east`, `:north` and `:south`. Type the following to get the `move-agent` command to act this way:

1. `(zero-deviation :move '(E))`
2. `(change-deviation :move '(E) :east 0.97)`
3. `(change-deviation :move '(E) :north 0.02)`
4. `(change-deviation :move '(E) :south 0.01)`

After initialization the system contains predefined deviations for each *deviation-class*. Each class is initialized to work correctly 100% of the time (within the physics of TileWorld). The command **change-deviation** is used to modify the probabilities of actions, while the command **add-deviation** is used to change the actual actions.

For a complete listing of all the effector *deviation-classes* and their associated *deviations* refer to figure 3.1 on page 32.

There are three functions which set each effector deviation back to 100% correct behavior:

```
initialize-gripper-slip
initialize-move-errors
initialize-release-stuck
```

Sample Simulated Error

See next page.

```
<user> (load " abp/tw/demo/errors-graph") ;;; See Figure 2.12

; Loading /home/copernicus/abp/tw/demo/errors-graph.lisp.
Destroying wind
Tileworld created with agent and one 6-sided tile named eel.
T
<user>
Loading body of the agent...  pushing down to server...  done.

(grasp 'E) ;;; See Figure 2.13

T
<user> (release 'e)

T
<user> (change-deviation :grasp '(E)
:normal 0.0)

NIL
<user> (grasp 'E) ;;; See Figure 2.14

T
<user> (move-agent 'S)

T
<user> (move-agent 'E)

T
<user> (grasp 'N) ;;; See Figure 2.15

T
```

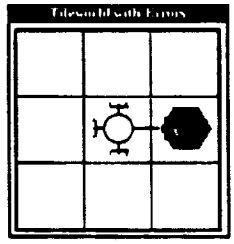


Figure 2.13: Agent grasps "eel"

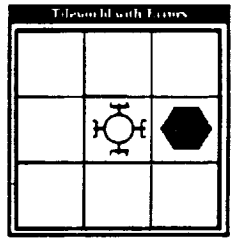


Figure 2.14: East gripper fails

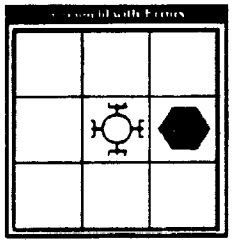


Figure 2.12: Agent west of "eel"

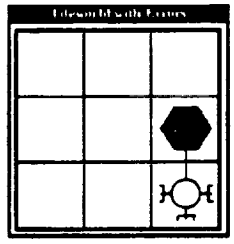


Figure 2.15: North gripper works

Chapter 3

Command Reference

This section contains a categorized and alphabetized listing of all TileWorld functions, variables, and constants. These commands fall into three categories: interaction, display, and customization. Within each category the commands are alphabetized and a short description is given.

3.1 Interaction Commands

The interaction commands allow an agent controller to operate the agent and sense the world state. These are intended to be the only commands that the agent controller uses to interact with TileWorld.

3.1.1 Effectors

These commands can be customized to exhibit probabilistic "errors" (see section 2.3.4, page 19).

grasp *direction* [*Function*]

grasp can grasp *free*[1] tiles in the compass *direction* specified. It is legal to try to **grasp** an empty cell.

[1] See definition of *free*, Section 1.2, page 4

`move-agent` *direction* [*Function*]

`move-agent` moves the agent one cell in the compass *direction* specified if all associated cells are clear and within the bounds of TileWorld. For instance, if the agent is grasping a tile to the west and the command (`move-agent 'N`) is issued, the cells directly north of the agent and northwest of the agent must be clear for the move to work. The speed of the agent's movement is regulated by the global variable, `*speed-of-agent*`. If a `move-agent` command is issued before the last `move-agent` command has completed, the second command is dropped *without* feedback. See section 3.3.1, page 29, for more details on `*speed-of-agent*`.

`release` *direction* [*Function*]

`release` releases a tile being grasped in the specified *direction*. It is legal to try to `release` nothing.

3.1.2 Sensors

`attached` *direction* [*Function*]

`attached` tests if the agent is currently grasping a tile in the *direction* specified. It returns either T or `NIL`.

Examples:

```
(attached 'N)   ⇒ T
(my-location)   ⇒ (1 2)
(in 1 3)        ⇒ ((SIDES 5) (NAME "Pengi"))
```

`world-time` *seconds-p* [*Function*]

`world-time` returns time in two forms, TileWorld seconds and real seconds. Real seconds are the actual number of seconds that have passed since the TileWorld database was started. TileWorld seconds are some multiple of the real world seconds. If *seconds-p* is T, actual elapsed time is returned. If `NIL`, the time is returned in TileWorld seconds. It takes longer to compute real world seconds, so in cases where speed is important you might want to use TileWorld seconds instead. Values returned are guaranteed to never be decreasing. The time is computed using Lisp's internal clock using the function, `get-internal-real-time`. [Steele, 1984, Miscellaneous Features].

Two macros are provided for converting the two different types of times into one another. They are `world-time-to-seconds` and `seconds-to-world-time`. Each will convert one time into the other and can take any value as long as it is a number (float or integer).

Examples:

```
(world-time t)                                          ⇒ 127.1
(world-time nil)                                        ⇒ 131987
(seconds-to-world-time 127.1)                           ⇒ 127100
(world-time-to-seconds (seconds-to-world-time 127.1))   ⇒ 127.1
```

`in` *column row* [*Function*]

`in` returns information about the cell, (*column,row*). If an object is present, explicit information on that object is returned. If no object is present, `NIL` is returned. If (*column,row*) is outside the bounds of TileWorld, `UNDEFINED` is returned.

Examples:

```
(in 3 8)    ⇒ ((SIDES 6) (NAME "Shakey"))
(in 5 2)    ⇒ AGENT
(in 1 0)    ⇒ NIL
(in -1 4)   ⇒ UNDEFINED
```

`my-location` [*Function*]

`my-location` returns the current location of the agent in TileWorld as a list of column and row. If the agent doesn't exist, `NIL` is returned.

Example:

```
(my-location)   ⇒ (3 2)
(in 3 2)        ⇒ AGENT
```

3.2 Display Commands

To observe changes occuring in TileWorld while an experiment is being run, either use the graphics display facility or periodically request an ASCII display. The graphics interface provides a facility to get real-time display of events occuring in TileWorld; i.e., the graphics display is automatically updated whenever a change takes place in TileWorld. The ASCII interface produces an ASCII representation of TileWorld only when the command `ascii-display` is called and does not automatically print a new display every time a change takes place in TileWorld.

3.2.1 Graphics Display

These are the global variables used by the graphics process to determine visual attributes of the display window. Use the variables ***auto-resizing*** and ***display-scripts*** to determine wind script presentation. The other variables along with keyword arguments to the command **tileworld** help determine the display window's appearance.

auto-resizing [*Variable*]

The value of ***auto-resizing*** determines whether or not the graphics display is automatically resized when wind scripts are created or destroyed. If ***auto-resizing*** is T, then the TileWorld window is resized whenever the number of wind scripts changes from none to many or many to none. If NIL, no resizing occurs unless ***display-scripts*** is set to T, in which case the window resizes larger once.

border [*Variable*]

The value of ***border*** determines the width in pixels of the border around the grid.

display-scripts [*Variable*]

The value of ***display-scripts*** determines whether or not the wind scripts are displayed in the graphics window. A value of T indicates that they are displayed, a value of NIL indicates not.

margin-normal [*Variable*]

The value of ***margin-normal*** determines the margin size in pixels when wind scripts are not being displayed.

token-ratio [*Variable*]

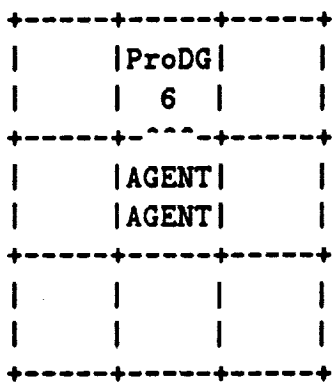
The value of ***token-ratio*** determines the size of the tiles as a fraction of ***cell-size***. It defaults to 0.75 or 75%.

3.2.2 ASCII Display

ascii-display &key :left :bottom :width :height :agent-centered [*Function*]

If the ASCII display is active (to activate, see **tileworld**, this section), TileWorld can be examined with this command. If none of the key arguments, **:left**, **:bottom**, **:width**, **:height** or **:agent-centered**, are specified, then a representation of the entire grid is displayed. By specifying the key arguments, smaller areas of the grid can be viewed. If **:agent-centered** is specified (it takes a number), the other arguments are ignored and the grid displayed will be an agent centered view with a box twice the size of the number given.

Here is how **ascii-display** shows a 3x3 TileWorld with the agent grasping a six sided tile named "ProDG":



This example could be displayed with any of the following:

```
(ascii-display)
(ascii-display :agent-centered 1)
(ascii-display :left 0 :bottom 0 :width 3 :height 3)
```

3.3 Customization Commands

This section describes the domain customization commands which allow the researcher to create a TileWorld problem instance, to tune simulator parameters (e.g., movement speed), to adjust simulator dynamics (e.g., behavior of the winds), and to introduce probabilistic errors in the behavior of the agent's effector actions (e.g., to make the agent sometimes "veer" off course).

3.3.1 Static Configuration Construction

`add-agent` *column row* `&key :stream :time-out` [*Function*]

Adds an agent to TileWorld at the specified location, (*column,row*), if the location is an unoccupied cell within the bounds of TileWorld. The function returns either `T` or `NIL`, denoting success or failure to place the agent. Should `add-agent` fail to place the agent, a message is printed to `:stream`. Due to the multiprocessing environment, a `:time-out` is necessary to ensure proper communication. This defaults to 10 seconds.

`add-object` *column row sides name* `&key :immovable :stream :time-out` [*Function*]

Adds a tile to the database at (*column,row*) if that location is an unoccupied cell within the bounds of TileWorld. *sides* is normally an integer greater than two but can be any Lisp object including lists and structures. If third is an integer greater than two, it will be displayed via the graphics as an object with that number of sides, otherwise, it will appear as a solid block occupying the entire grid cell. *name* can be any Lisp object including lists and structures. If *sides* or *name* is a list or a structure, then data can be stored with the object and is available through the sensors. See section 3.1.2, page 25 for examples on how to access this information. `:immovable` determines whether or not the agent or winds can move this object. The value defaults to `NIL`.

This command returns a structure of type `object` if the tile was successfully added. It returns `NIL` and prints a message to `:stream` if the command fails or if it timed out. The default for `:time-out` is 10 seconds.

`clear-tileworld` [*Function*]

Clears all objects from the TileWorld database including the agent, removes all wind-scripts, resets all counters, and resets the time clock to zero. However, the database is not actually removed, the graphics window and ASCII display buffer are not recreated (although they are updated). This command is useful because the startup time in initializing all of the database and display buffers can be avoided. This command finds its greatest use when running multiple tests on the same problem: restarts costs only the time in repopulating TileWorld.

`remove-agent` [*Function*]

Removes the agent from TileWorld. The agent releases all held tiles before it is removed. If the agent does not exist when `remove-agent` is called, a warning message is printed.

`remove-object` *object* [*Function*]

Removes a tile from TileWorld. *object* must be a structure of type `object` and refer to the tile in question. This structure is returned by the command `add-object`. See `add-object`, this section, for more information.

`reset-world-time` `&optional` *seconds* [*Function*]

This resets the TileWorld clock back to zero or to the value of *seconds* should that be supplied. This command is useful to reset the clock just before an experiment is run.

`*speed-of-agent*` [*Variable*]

The value of `*speed-of-agent*` determines the rate at which the agent can move through TileWorld. The number represents the number of cells per second. The default for this variable is 10. This means that in one second the agent can move 10 cells, or that the agent moves one cell in 1/10 of a second. This variable may be adjusted at any time.

`tileworld` *columns rows* `&key :display :agent-default :title` [*Function*]
`:cell-size :agent-size :grid-lines`

This creates a TileWorld of size *columns* × *rows*. The `:title` of TileWorld defaults to "Tile World". `:display` must be one of the following: `:graphics`, `:ascii` or `:both`. If it is `:graphics`, then output is handled by the graphics process only. If it is `:ascii`, then the TileWorld state can only be accessed with the ASCII interface via `ascii-display`. If it is `:both`, then both graphics and ASCII displays will work. `:display` defaults to `:graphics`.

`:agent-default` can be `T`, `NIL`, or a filename. `:agent-default` defaults to `T` which indicates that the graphics package should draw a circle to represent the agent. `NIL` indicates that the user should be queried for the proper bitmap. If a filename is supplied, it must be a Common Lisp pathname [Steele, 1984, File System Interface] and must point to an X Windows bitmap file describing the agent's body. See Appendix A, page 45 for more details on the agent's body bitmap.

`:title` appears in the title bar of the graphics window when the graphics process is enabled. `:cell-size` determines the pixel size of a TileWorld cell in the graphics window. This defaults to the previous number given or 66 the first time called. `:agent-size` determines the agent's size in the graphics window. This value can either be a number no greater than half that of `:cell-size`, or can be the keyword argument `:half` and it defaults to the previous value given or to 29 the first time called. `:grid-lines` defaults to `T` and determines if the grid lines forming the cell boundaries within TileWorld are displayed.

3.3.2 User Defined Winds

`pause-wind &key :stream` [*Function*]

This function temporarily halts all wind scripts. Winds registered while the scripts are paused will likewise be paused. The global variable, `*wind-script-active*`, holds the current status of the wind scripts. `:stream` is the output stream for information printed by this command. `NIL` prevents the output from being displayed. `resume-wind` resumes the scripts once paused.

`register-wind` *column row direction strength* [*Function*]
`&optional` *lower-bound upper-bound*

`register-wind` creates a wind script originating at the cell, (*column*,*row*), blowing in *direction*. *direction* can be one of the four compass directions: `N`, `S`, `E`, or `W`. *strength* is an integer describing how many cells a tile can be blown. The interval described by [*lower-bound*..*upper-bound*] is the period for how often the winds are blown. This defaults to [5..20]. These winds normally blow from the edges of TileWorld towards the interior, although the system allows a wind to originate anywhere within TileWorld and move in any direction.

Examples:

`(register-wind 1 0 'N 5)`	Origin (1,0), blows north 5 cells, period of [5..20]
`(register-wind 0 8 'E 6 7)`	Origin (0,8), blows east 6 cells, period of [7..20]
`(register-wind 8 5 'W 2 12 31)`	Origin (8,5), blows west 2 cells, period of [12..31]
`(register-wind 3 8 'S 1 17 17)`	Origin (3,8), blows south 1 cell, period of [17..17]

`resume-wind &key :stream` [*Function*]

`resume-wind` restarts paused wind scripts. Because of the way winds are stored, a lot of wind scripts back up during the time they are paused. When the scripts are resumed, these winds will all release at the same time. See `pause-wind` for more details.

`stop-wind &key :stream` [*Function*]

This command destroys all wind scripts. The wind scripts are removed and the wind script status is reset to active, even though there will be no winds to be blown.

`wind-blows-object` *column row direction strength* [*Function*]

This function creates a gust of wind, a single wind event that causes a wind to blow in the cell, (*column,row*), in the *direction* given. Should there be a tile present in the cell and that tile is *free*[2], it is blown for as many cells as the *strength* supplied. If there is no tile in the cell, the wind dies (and does not continue on to possibly blow other tiles). In other words, this commands produces a wind that originates in a cell and exhausts itself immediately.

`*speed-of-tile*` [*Variable*]

The value of `*speed-of-tile*` determines the rate at which tiles are blown by the wind. The number represents the number of cells per second. The default value for this variable is 10. This means that in one second the wind can blow a tile 10 cells, or that a tile is blown through one cell in 1/10 of a second. This variable may be adjusted at any time.

`*wind-script-active*` [*Variable*]

`*wind-script-active*` is a list of length one whose car determines the activity status of the wind scripts. If the car is `T`, the wind scripts are cycling and blowing. If `NIL`, they are paused. Only `pause-wind` and `resume-wind` should be used to change `*wind-script-active*` because these functions perform important side effects.

3.3.3 Probabilistic Effector Errors

`add-deviation` *deviation-class argument-list deviation probability functions* [*Function*]

`add-deviation` takes a *deviation-class*, an *argument-list*, a *deviation*, a *probability* for that *deviation*, and a list of *functions* to execute if the random number generator selects this *deviation*. Each effector function eventually bottoms out in one of the `HELP-` functions, which is then used by the probabilistic failure mechanism to control the effector. These `HELP-` functions are the actual Lisp code that run the effectors. See section 4.3 on page 39 for more information on `HELP-` functions.

Example:

```
(add-deviation :release '(S) :normal 0.9 '((HELP-RELEASE S)))
```

When the effector action, `(release 'S)`, is called, it calls the function, `(HELP-RELEASE)`, 90% of the time. The other 10% of the time it defaults to nothing, assuming that no other *deviations* have been specified.

[2]See definition of *free*, Section 1.2, page 4

`change-deviation` *deviation-class argument-list deviation probability* [*Function*]

`change-deviation` takes a *deviation-class*, an *argument-list*, a *deviation*, and changes its probability to be *probability*. `change-deviation` assumes that the deviation has already been added by `add-deviation`.

Example:

```
(change-deviation :release '(E) :normal 0.5)
```

When the effector action, `(release 'E)`, is called, it will now work 50% of the time, instead of its previously set value. It is an error to make a change to a *deviation* that causes the total probability for the *deviation-class* and *argument-list* to exceed 1.0. Refer to figure 3.1 on page 32 for all the classes and deviations.

:MOVE and its associated deviations		
`(move-agent 'N)`	`:north`	Moves the agent true north
	`:east`	Moves the agent north and east
	`:west`	Moves the agent north and west
`(move-agent 'S)`	`:south`	Moves the agent true south
	`:east`	Moves the agent south and east
	`:west`	Moves the agent south and west
`(move-agent 'E)`	`:east`	Moves the agent true east
	`:north`	Moves the agent east and north
	`:south`	Moves the agent east and south
`(move-agent 'W)`	`:west`	Moves the agent true west
	`:north`	Moves the agent west and north
	`:south`	Moves the agent west and south

:GRASP and its associated deviations		
`(grasp 'N)`	`:normal`	Grasps an object to the north
`(grasp 'S)`	`:normal`	Grasps an object to the south
`(grasp 'E)`	`:normal`	Grasps an object to the east
`(grasp 'W)`	`:normal`	Grasps an object to the west

:RELEASE and its associated deviations		
`(release 'N)`	`:normal`	Releases the gripper to the north
`(release 'S)`	`:normal`	Releases the gripper to the south
`(release 'E)`	`:normal`	Releases the gripper to the east
`(release 'W)`	`:normal`	Releases the gripper to the west

Figure 3.1: Deviation Classes and their deviations

initialize-deviation *deviation-class argument-domain* [*Function*]

initialize-deviation initializes the discrete cumulative distribution functions (CDFs) for the particular effector *deviation-class* and its *argument-domain*. It enters the *deviation-class* into the global, ***probability-of-deviation***, or zeros all probabilities if the *deviation-class* is already present. The *argument-domain* is used to determine an effector's actions depending upon the argument-list given.

Example:

```
(initialize-deviation :grasp '((N) (S) (E) (W)))
```

This initializes the **:grasp** deviation and sets up the domain of symbols **grasp** can receive. That is, it can receive four sets of arguments, each set is of length one and refers to one of the compass directions.

initialize-gripper-slip [*Function*]

initialize-gripper-slip resets the effector action **grasp**. This function is called when the system is first started. It only needs to be called when the probabilities have been modified.

initialize-move-errors [*Function*]

initialize-move-errors resets the effector action **move-agent**. This function is called when the system is first started. It only needs to be called when the probabilities have been modified.

initialize-release-stuck [*Function*]

initialize-release-stuck resets the effector action **release**. This function is called when the system is first started. It only needs to be called when the probabilities have been modified.

zero-deviation *deviation-class argument-list* [*Function*]

zero-deviation zeros the particular *argument-list* of a *deviation-class*. Once a deviation has been set, it is best to zero it before changing its probability, so that there is no chance of causing the CDF to exceed 1.0.

Example: **(zero-deviation :grasp '(N))**

This zeros the CDF for the effector action, **(grasp 'N)**. Executing **(grasp 'N)** will have no effect, until **change-deviation** is used to adjust the probabilities.

Chapter 4

Programmer's Notes

We created TileWorld with a certain research agenda in mind which may or may not suit your needs. If it does, that's excellent! If not, you should be able to easily modify TileWorld or use it as a base for other simulated environments. This chapter provides information, direction, and pointers to the code to ease modifications. If most of the system suits your needs, but there are some tweaks that would make it more useful, by all means, make them. For help, send email to Andy Philips at address tileworld@ptolemy.arc.nasa.gov.

4.1 Multiprocessing

Every reference to multiprocessing commands uses one of the macros defined in the file `macros.lisp`. Therefore, switching to other Lisp dialects with multiprocessing is simplified. In addition we have tried to follow well-founded concepts for concurrent programming, like the macros '`critical-section`' and '`cobegin`' [Ben-Ari, 1982]. Queues are used to transport data between processes.

4.1.1 Queues

Queues and Priority Queues are sound techniques for two or more concurrent processes to communicate. They are used by the concurrent processes in the TileWorld system to ensure database and screen integrity. The MultiProcessing Queue structure in `queues.lisp` is designed to allow fast queue updates, priority queue ordering, and prevention of data corruption.

Because queues are the only way that two concurrent TileWorld processes can communicate, each queue has a lock which is used to maintain queue integrity. These locks provide the necessary security.

There is a head pointer and a tail pointer to each queue. The head points to the next item to be dequeued and the tail to the item most recently entered. Should the queue be a priority queue, the tail pointer is unused.

If the queue is a priority queue, it has an accessor function which takes as an argument a member of the queue and returns a number representing that item's priority. The lower the number for the priority, the "earlier" it will be ordered in the queue. This type of ordering makes sense for priority queues that are ordered by time. The accessor function is compiled when the queue is initialized.

Here is a list of globals containing queues used by the system:

Global	Type	Description
`*database-queue*`	FIFO	For sending commands to the database process.
`*graphics-queue*`	FIFO	For sending commands to the graphics display process.
`*ascii-queue*`	FIFO	For sending commands to the ASCII display process.
`*display-queue*`	- -	Set to either `*ascii-queue*` or `*graphics-queue*`.
`*script-queue*`	Priority	For maintaining which wind script executes next.
`*wind-queue*`	Priority	For maintaining which tile blows next.

4.1.2 Processes

The system is divided into a number of concurrent processes, each of which is responsible for certain behaviors. A database process keeps track of the agent, tiles, the grid, time, effectors, and sensors and upholds the physics. Two display processes, ASCII and graphics, keep track of data output. A gust process blows all tiles. A script process maintains all wind scripts.

Any process may enter data into any queue, but only one process is allowed to remove data from a particular queue. This restriction is enforced by the programmer, not by the system. There are five queues, so there are five processes that read those queues and five global variables to contain them.

Global	Actual Lisp Function	Associated Queue
`*database-process*`	`database-process`	`*database-queue*`
`*graphics-process*`	`graphics-process`	`*graphics-queue*`
`*ascii-process*`	`ascii-process`	`*ascii-queue*`
`*script-process*`	`script-process`	`*script-queue*`
`*wind-process*`	`wind-process`	`*wind-queue*`

The database process constructs and maintains the TileWorld database. These functions manage TileWorld in terms of creating, adding, removing, and moving tiles and the agent, but do not direct the autonomous movement of tiles via winds (handled by the wind process). Although it is not necessary to use the graphics facility with the database, they were written to be used together.

The two display processes, ASCII and graphics, maintain a correct world map and rely upon receipt of update information from other processes, mostly the database. If the graphics window is being used, then **graphics-process** will be running. If ASCII is being used, then **ascii-process** will be running. Both processes run when both display formats are being used, with the **graphics-process** passing information to the **ascii-process** when the global variable, ***graphics-pass***, is set to T.

The script process handles the wind scripts. The process will sleep for as many seconds between *now* and the time of the highest priority script (top of the priority queue). If a higher priority script is inserted onto the priority queue (sooner in time), the script process restarts its sleep cycle based on the new script. If it finishes its sleep cycle without any higher priority scripts being inserted, the top script will be considered active and be made to blow. Once a script is activated, its next time of activation is calculated, and it is reinserted into the queue. Any scripts on the queue that are overdue, i.e. scripts whose times are in the past, will be taken care of immediately.

The wind process is very similar to the script process. It handles its priority queue exactly the same except that it does not reinsert tile movement events until there is confirmation from the database that the tile moved succesfully. Note, it *is* possible to determine the effects of database changes even though the agent's effectors do not return success or failure. The wind process is considered part of the omniscient universe and can "know" if actions succeed, whereas the agent may not have access to such "global" knowledge.

4.1.3 Effectors in Multiprocessing

When an effector action is called, three things happen. First, the effector action is queued onto the database input queue as an external function. The database calls this function when it is dequeued, allowing that function to manipulate the TileWorld database alone; that is, the effector function will be acting as if it was a database process function. Second, the simulated error code is called, to determine the exact outcome of the effector action. Finally, the actual function which executes the effector action is run.

The effectors may be as simple or as complex as your needs require. It is best if the functions operate quickly. They are called as extensions of the TileWorld database, and if they take a long time to execute, they will tie up the database and keep other database events from executing in a timely fashion (e.g. wind blown tiles, redraws).

4.1.4 Motion, Asynchronous Behavior, and Time

If two tiles or the agent and a tile are moving towards the same cell, the first one into that cell will occupy it and, the other will be stopped. If a tile is moving past the agent, the agent will not be able to **grasp** that tile. Likewise, if the agent attempts to **grasp** a tile that is suddenly blown by a wind, the **grasp** action will fail.

To synchronize system operations like winds, gusts, agent speed, and tile speed, the Lisp internal timer is used (See section 3.1.2, page 24, `world-time`). To control speed the agent and tiles are marked with the time that they were last moved and are prevented from moving until the clock progresses past their time marks. The TileWorld clock keeps time by lazy evaluation. That is, the function doesn't actually keep track of clock ticks, rather it calculates the passage of time since the last time request and returns the new time. This is completely reasonable, except that because of the functions that it uses in Lisp this (very rarely) may cause a slight weirdness in time keeping. The timer uses the Lisp function `get-internal-real-time`. The number that this function returns cycles, so that at some point it goes from a big positive number to a little positive number. I imagine the math isn't that complex to compute time elapsed, but rather than make the calculations general, when between time requests the Lisp clock cycles in this manner, the TileWorld clock assumes ZERO TIME has passed. If the TileWorld clock is sampled often, this glitch will hardly be noticed. And with winds blowing, the clock is sampled enough and, therefore, there is no problem.

4.2 Allegro CL Graphics Commands

Here is a list of the Common Windows functions used in the graphics system:

```
bitblt
clear-rectangle-xy
clear-window-stream
clear
control-mouse-cursor-move-events
draw-rectangle-xy
draw-polygon-xy
draw-filled-polygon-xy
draw-image-xy
draw-lines
draw-line-xy
font-baseline
font-character-height
font-string-width
get-mouse-state
make-bitmap-stream
make-position
make-window-stream
modify-window-stream-method
read-bitmap
window-stream-operation
window-stream-font
window-stream-mouse-cursor
window-stream-mouse-cursor-move
window-stream-title
window-stream-x-position
window-stream-y-position
```

From the Allegro CL Common Windows Manual [Franz, Inc., 1989, page 3-1]:[1]

> The *window-stream* is the basic data structure of Common Windows. The window you see on the screen is the physical display of a window-stream. Window-streams

[1] reprinted with the permission of Franz Lisp, Inc.

are Common Lisp structures, and, as such, have *slots* which hold information and, in many cases, can be used to change it. [sic]

An active-region is an area of a window separately sensitive to the mouse. A mouse event when the cursor is in an active-region is caught by the active-region, not the parent window. Also when the cursor enters the active-region, a cursor in event is generated for the active-region.... The parent of an active-region must be in a window-stream.... An active-region with more than one cell is also called an active-grid.... Active-regions can be activated and deactivated just like window streams.

[`make-position`] returns a new position, whose coordinates are specified by the values of the `:x` and `:y` arguments, which are integers are defaulting to 0. In Common Windows, positions are used to specify the locations of objects, for example the mouse cursor, windows, and the endpoints of lines.

4.3 Adding New Probabilistic Effector Errors

When an effector is called, it has a helping function which actually performs the work of modifying the database. Neither you nor the agent should call these helping functions directly; they are only to be called via the simlated error facility. Each effector has at least one helping function, and all helping functions begin with 'HELP-' by convention.

For simulated error purposes, the helping function for **grasp** is `HELP-GRASP`. The helping functions for **move-agent** are `HELP-MOVE-AGENT` and `HELP-MOVE-AGENT-DIAGONALLY`. The helping function for **release** is `HELP-RELEASE`.

Let's say you want to initialize the **grasp** command to work as discussed earlier in this manual. The *deviation-class* is **:grasp**. The *argument-domain* is `'((N) (S) (E) (W))`. One particular *argument-list* is `'(N)`. The *probability* for *deviation* `:normal` for this *argument-list* might be `1.0`. The set of *functions* to execute for this *deviation* might be `'((HELP-GRASP N))`. Thus, to set up the **grasp** command, you might do the following:

```
(initialize-deviation :grasp '((N) (S) (E) (W)))
(add-deviation :grasp '(N) :normal 1.0 '((help-grasp N)))
(add-deviation :grasp '(S) :normal 1.0 '((help-grasp S)))
(add-deviation :grasp '(E) :normal 1.0 '((help-grasp E)))
(add-deviation :grasp '(W) :normal 1.0 '((help-grasp W)))
```

This establishes an error-free **grasping** action in all compass directions.

For a second example, suppose you wish to modify the current (**move-agent** `'e`) command so that when the agent moves east, it will move directly east 97% of the time, move east and limp north 2% of the time and move east and limp south 1% of the time. Here, the

deviation-class is `:move` and the *argument-list* is `'(E)`. Create three new *deviations* called: `:east`, `:north`, and `:south` (these are arbitrary names). Then type the following to get the `move-agent` command to act this way:

```
(zero-deviation :move '(E))
(add-deviation :move '(E) :east 0.97 '((HELP-MOVE-AGENT E)))
(add-deviation :move '(E) :north 0.02 '((HELP-MOVE-AGENT-DIAGONALLY E N)))
(add-deviation :move '(E) :south 0.01 '((HELP-MOVE-AGENT-DIAGONALLY E S)))
```

This example can be done more easily with `change-deviation`, *see Section 2.3.4, page 19.*

4.4 How the Wind Gusts and Scripts Are Done

Functional control for definition of wind gusts is handled via the window stream with a left-button-down handler. The first left-button-down event in the window stream has a number of effects: (i) starts the wind gust definition process, (ii) sets the point of origin for the wind gust, (iii) changes the mouse cursor to a plus sign, and (iv) activates the mouse-cursor-in-handler method. This method tracks the cursor as it moves throughout the grid cells (equivalent to the TileWorld cells) and keeps track of proper direction and force for the wind gust. When the second left button down event occurs, the database, if given a legal wind direction and force, will cause a wind to blow, and then reset the window properties, preparing it to accept a new wind definition from the user.

A similar procedure and right-button-down handler are used for wind scripts.

Acknowledgements

Thanks to Mark Drummond, Rich Levinson, Smadar Kedar, and Keith Swanson for testing the simulator code. Thanks to Martha DelAlto, Mark Drummond, Kate McKusick, Andrew Philpot, and Keith Swanson for comments on previous versions of this manual.

This software has been developed with joint funding from NASA and DARPA. NASA funding for the contract is in the AI Research Program under RTOP 590-12-33, and DARPA co-funding is provided by the Information Sciences Technology Office under DARPA Order 7382. Software development has been carried out by Sterling Software under contract to the NASA Ames Research Center.

Bibliography

[Ben-Ari, 1982] Ben-Ari, M. 1982. *Principles of Concurrent Programming.* Englewood Cliffs, NJ: Prentice-Hall International.

[Franz, Inc., 1989] Franz, Incorporated. 1989. *The Allegro Common Windows Manual.* Berkeley, CA: Franz, Incorporated.

[Kantowitz, 1974] Kantowitz, B.H. (Ed.). 1974. *Human Information Processing: Tutorials in Performance and Cognition.* New York, NY: Lawrence Erlbaum.

[Philips et al., 1991] Philips, Andrew B., Swanson, Keith J., Drummond, Mark E., and Bresina, John L. 1991. *The NASA TileWorld Simulator: Instantiating key domain attributes while discarding irrelevant semantic baggage* (NASA Ames Technical Report TR-FIA-91-04). Moffett Field, CA: NASA Ames Research Center, Code FIA.

[Pollack and Ringuette, 1990] Pollack, M.E., and Ringuette, M. 1990. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 183–189), Menlo Park, CA: AAAI Press.

[Sridharan and Bresina, 1984] Sridharan, N.S., and Bresina, J.L. 1984. *Exploration of Problem Reformulation and Strategy Acquisition - A Proposal* (Rutgers Technical Report RU-LCSR-TR-53; RU-CBM-TR-137). New Brunswick, NJ: Rutgers University, Department of Computer Science.

[Steele, 1984] Steele, Guy. 1984. *Common Lisp: The Language.* America: Digital Press.

[Sutton, 1990] Sutton, Richard S. 1990. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynmaic Programming. *Proceedings of the Seventh International Conference on Machine Learning* (pp. 216-224), San Mateo, CA: Morgan Kaufmann Publishers.

Appendix A

Files in the System

.../tileworld/

File	Functions...
ascii	to display TileWorld in ASCII text format.
compile	to compile the files in the TileWorld system.
database	to construct and maintain the TileWorld database.
effectors	for the agent to take actions in TileWorld.
graphics-agent	to maintain the display states of the agent.
graphics-wind	to create winds via the mouse and display them.
graphics	that display TileWorld from Allegro using common windows.
macros	to abstract the multiprocessing code.
processes	to initialize all TileWorld processes.
queues	to control multiprocess communication via queues.
sensors	for the agent to gain information about TileWorld.
simulated-errors	to introduce alternate behavior in the agent's effectors.
startup	to load the appropriate files in the TileWorld system.
tileworld-database	to set up TileWorld globals and load files.
time	to run the TileWorld timer.
wind-blow	to blow a tile about TileWorld.
wind-script	to create wind scripts for blowing tiles.

.../tileworld/image/

Here reside the files that are used to represent the body of the agent in the TileWorld domain. To create an agent body use the X11 facility 'bitmap'. Read the UNIX man pages for a full explanation of this command. Bitmaps of the agent's body should be 29x29 pixels unless the graphics constant `*agent-size*` is changed. Bitmaps larger than `*agent-size*`

will only be partially used and smaller bitmaps may not be centered correctly. TileWorld only looks for agent bitmap filenames that end in ".bit". All other files will be ignored.

File	Description
README	Description of directory.
eye.bit	Agent with an eye peering out from the interior.
nasa.bit	Agent with a NASA logo in the interior.
standard.bit	Standard agent bitmap for the graphics display.

.../tileworld/Doc/

File	Description
Manual.tex	User's Guide in the latex format.
Manual.dvi	User's Guide in the DVI format (TileWorld images not included).
Manual.ps	User's Guide in PostScript format (images included).
PS/fig?.ps	PostScript figures for the Manual.
script?.tex	Tutorial Scripts derived from dribbled Lisp output.

Appendix B

Known Bugs

Despite our best efforts to prevent them, there are still some bugs in the system. Please report any new bugs to us. We cannot promise any support, but we will make an effort.

- The current package is switched to :cw when TileWorld is loaded. This may cause some symbol conflicts with the :user package. Solution: clobber those symbols (unintern option) if an error occurs.
- For Allegro Version 3 Lisp users, check to see if the LOOP macro is properly loaded. Contact me (Andy Philips) if you need a copy of the macro. It is also available from a number of FTP sites around the country.

Index

Errata Sheet for NASA TileWorld (Version 2)

The following sections describe changes made to the TileWorld program and errors found in the original manual. These reflect updates to the TileWorld program for Version 2.1.

Errors in TileWorld 2.0 Manual

- On page 7, paragraph 2 the text reading "... the top of page 9 is the ..." should read "... the top of page 10 is the ...".
- On page 43 in the bibliography on the fourth entry, [Philips et al., 1991], the technical report number is incorrect, it should be TR-FIA-91-04.

Changes to existing features in TileWorld 2.1

- There was a problem with the display of wind scripts in the graphics window. The scripts would not always be displayed when first created by the mouse and were never first displayed when created with the command **`register-wind`**. Both of these problems have been fixed.
- On page 25 under the Section 3.2.1 **Graphics Display**, the text should now read:

  > These are the global variables used by the graphics process to determine visual attributes of the display window. Use the variables **`*auto-resizing*`** and **`*display-scripts*`** to determine wind script presentation. The other variables along with keyword arguments to the command **`tileworld`** help determine the display window's appearance.

- Under the same section, the following constants should be removed from the documentation because they should now only be changed by calls to the **`tileworld`** command:

  `*agent-size*`, **`*cell-size*`**, and **`*script-arrow*`**

  The agent's size and the grid cell size can now be controlled through keyword argument's to the **`tileworld`** command. The script arrow's size is linked to the cell size (a few pixels smaller) and is computed when the graphics window is constructed. Modifying these variables during a session has undetermined effects. It should be noted that under a certain size, script arrows will no longer display strength and period numbers due to lack of room. Information can still be obtained through the middle button facility.

- Also, the following constants have been changed to variables, but should only be changed before a new TileWorld grid is created:

  `*border*`, `*margin-normal*`, and `*token-ratio*`.

- The command `add-object` on page 28 should now read:

  `add-object` *column row sides name* `&key :immovable :stream :time-out` [*Function*]

  Adds a tile to the database at (*column*,*row*) if that location is an unoccupied cell within the bounds of TileWorld. *sides* is normally an integer greater than two but can be any Lisp object including lists and structures. If third is an integer greater than two, it will be displayed via the graphics as an object with that number of sides, otherwise, it will appear as a solid block occupying the entire grid cell. *name* can be any Lisp object including lists and structures. If *sides* or *name* is a list or a structure, then data can be stored with the object and is available through the sensors. See section 3.1.2, page 25 for examples on how to access this information. `:immovable` determines whether or not the agent and wind can move this object. The value defaults to `NIL`.

  This command returns a structure of type `object` if the tile was successfully added. It returns `NIL` and prints a message to `:stream` if the command fails or if it timed out. The default for `:time-out` is 10 seconds.

- The command `tileworld` on page 29 should now read:

  `tileworld` *columns rows* `&key :display :agent-default :title :cell-size :agent-size :grid-lines` [*Function*]

  This creates a TileWorld of size *columns* × *rows*. The `:title` of TileWorld defaults to "Tile World". `:display` must be one of the following: `:graphics`, `:ascii` or `:both`. If it is `:graphics`, then output is handled by the graphics process only. If it is `:ascii`, then the TileWorld state can only be accessed with the ASCII interface via `ascii-display`. If it is `:both`, then both graphics and ASCII displays will work. `:display` defaults to `:graphics`.

  `:agent-default` can be `T`, `NIL`, or a filename. `:agent-default` defaults to `T` which indicates that the graphics package should draw a circle to represent the agent. `NIL` indicates that the user should be queried for the proper bitmap. If a filename is supplied, it must be a Common Lisp pathname [Steele, 1984, File System Interface] and must point to an X Windows bitmap file describing the agent's body. See Appendix A, page 45 for more details on the agent's body bitmap.

  `:title` appears in the title bar of the graphics window when the graphics process is enabled. `:cell-size` determines the pixel size of a TileWorld cell in the graphics window.

This defaults to the previous number given or 66 the first time called. `:agent-size` determines the agent's size in the graphics window. This value can either be a number no greater than half that of `:cell-size`, or can be the keyword argument `:half` and it defaults to the previous value given or to 29 the first time called. `:grid-lines` defaults to T and determines if the grid lines forming the cell boundaries within TileWorld are displayed.

- In the chapter on Programmer's Notes, under section 4.4, **How the Wind Gusts and Scripts Are Done**, the second paragraph mentions "...mouse-cursor-left-in-handler...". It should now read "...mouse-cursor-in-handler...". A modification was made to the way active regions are handled. Instead of separate active grid for the wind script creation process and the wind gust creation process, a single active grid is used.

New features in TileWorld 2.1

- Under Section 2.2.1, **Graphics Display**, the following text should be included:

  > The graphics window is responsive to a number of button press events. If the middle button of the mouse is pressed and held while the mouse cursor is within a cell, the location of that cell is displayed on the title bar of the graphics window. Also, if the middle button is pressed and held while the cursor is over a wind script arrow residing on the border, information about that script is displayed in the title bar. Should multiple scripts be defined in the same arrow, the wind event that is earliest in the wind queue is selected (i.e. the next event to be blown). Other mouse button interactions are described later.

- Under Section 3.1.2, page 24, two macros are described: `world-time-to-seconds` and `seconds-to-world-time`. The associated text describing the `world-time` command has been changed to the following:

  `world-time` *seconds-p* [*Function*]

  `world-time` returns time in two forms, TileWorld seconds and real seconds. Real seconds are the actual number of seconds that have passed since the TileWorld database was started. TileWorld seconds are some multiple of the real world seconds. If *seconds-p* is T, actual elapsed time is returned. If `NIL`, the time is returned in TileWorld seconds. It takes longer to compute real world seconds, so in cases where speed is important you might want to use TileWorld seconds instead. Values returned are guaranteed to never

be decreasing. The time is computed using Lisp's internal clock using the function, `get-internal-real-time`. [Steele, 1984, Miscellaneous Features].

Two macros are provided for converting the two different types of times into one another. They are `world-time-to-seconds` and `seconds-to-world-time`. Each will convert one time into the other and can take any value as long as it is a number (float or integer).

Examples:

```
(world-time t)                                          ⇒ 127.1
(world-time nil)                                        ⇒ 131987
(seconds-to-world-time 127.1)                           ⇒ 127100
(world-time-to-seconds (seconds-to-world-time 127.1))   ⇒ 127.1
```

- Under Section 3.3.1, **Static Configuration Construction**, the following two commands should be included:

`clear-tileworld` [*Function*]

Clears all objects from the TileWorld database including the agent, removes all windscripts, resets all counters, and resets the time clock to zero. However, the database is not actually removed, the graphics window and ASCII display buffer are not recreated (although they are updated). This command is useful because the startup time in initializing all of the database and display buffers can be avoided. This command finds its greatest use when running multiple tests on the same problem: restarts costs only the time in repopulating TileWorld.

`reset-world-time &optional` *seconds* [*Function*]

This resets the TileWorld clock back to zero or to the value of *seconds* should that be supplied. This command is useful to reset the clock just before an experiment is run.

- Under Section 4.2, page 38, two Allegro CL graphics commands should be added: `active-region-mouse-cursor` and `window-stream-title`.

REPORT DOCUMENTATION PAGE

OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Dates attached	

4. TITLE AND SUBTITLE

Titles/Authors - Attached

5. FUNDING NUMBERS

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Code FIA - Artificial Intelligence Research Branch

Information Sciences Division

8. PERFORMING ORGANIZATION REPORT NUMBER

Attached

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Nasa/Ames Research Center

Moffett Field, CA. 94035-1000

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Available for Public Distribution

Peter Friedland 5/14/92 BRANCH CHIEF

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Abstracts ATTACHED

14. SUBJECT TERMS			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT