CDC™
Workplace Safety and Health

IC 9457
INFORMATION CIRCULAR/2001

Miner Training Simulator: User's Guide and Scripting Language Documentation

U.S. DEPARTMENT OF HEALTH AND HUMAN SERVICES
Public Health Service
Centers for Disease Control and Prevention
National Institute for Occupational Safety and Health

NIOSH

Information Circular 9457

Miner Training Simulator: User's Guide and Scripting Language Documentation

By Todd M. Ruff

U.S. DEPARTMENT OF HEALTH AND HUMAN SERVICES
Public Health Service
Centers for Disease Control and Prevention
National Institute for Occupational Safety and Health
Pittsburgh Research Laboratory
Pittsburgh, PA

June 2001

ORDERING INFORMATION

Copies of National Institute for Occupational Safety and Health (NIOSH) documents and information about occupational safety and health are available from

NIOSH–Publications Dissemination
4676 Columbia Parkway
Cincinnati, OH 45226-1998

FAX:	513-533-8573
Telephone:	1-800-35-NIOSH (1-800-356-4674)
E-mail:	pubstaft@cdc.gov
Web site:	www.cdc.gov/niosh

DHHS (NIOSH) Publication No. 2001-136

CONTENTS

MINER TRAINING SIMULATOR:
USER'S GUIDE AND SCRIPTING LANGUAGE DOCUMENTATION

By Todd M. Ruff[1]

ABSTRACT

A training software package for new mine employees, called Miner Training Simulator (MTS), has been developed by researchers at the National Institute for Occupational Safety and Health. MTS is a computer-based tool that allows a trainee to enter a simulated mine and interact with his/her surroundings in order to learn basic mining concepts, safety procedures, mine layouts, and escape routes. The training simulator software and instructions for its use are described in this report. Also, each mine using the software will have different requirements with regard to safety training. To customize the simulator for these differences, an interpreted scripting language is used to define interactions between the trainee and the virtual mine and objects in it. The scripting language, called Tool Command Language, uses simple commands to control various actions in the simulation, such as sounds, safety messages, hazards, and movement of objects. The basics of the scripting language are described here, along with many examples and instructions for building a script for MTS.

[1] Electrical engineer, Spokane Research Laboratory, National Institute for Occupational Safety and Health, Spokane, WA.

INTRODUCTION

Miner Training Simulator (MTS) is a computer-based tool that allows a trainee to enter a simulated mine and interact with his/her surroundings in order to learn basic mining concepts, safety procedures, mine layouts, and escape routes (Filigenzi 2000). This is accomplished by using a program that integrates a three-dimensional graphics engine developed by Twilight Corp., Finland,[2] and custom simulator code written by researchers at the National Institute for Occupational Safety and Health (NIOSH). The graphics engine handles rendering of the current scene or view. The simulator code handles the graphical user interface (GUI), the calculations for trainee movement, interactions within the simulated mine, and other details that make the simulator specific to mine safety training. All code was developed using C++ computer language.

MTS can import a computer (virtual reality) model of a specific mine that has been developed using 3D Studio Max,[3] a commercially available modeling and animation program. A sample model is shown in figure 1. An interpreted scripting language called Tool Command Language (TCL) is used to define the way a trainee interacts with the objects within this virtual mine. Without a scripting language, the MTS source code would have to be modified and recompiled for every different mine or training scenario. TCL interfaces with MTS and provides the basic commands needed to define many different actions and effects.

[2]Mention of specific products or brand names does not imply endorsement by the National Institute for Occupational Safety and Health.

[3]3D Studio Max, Autodesk, Inc., San Rafael, CA.

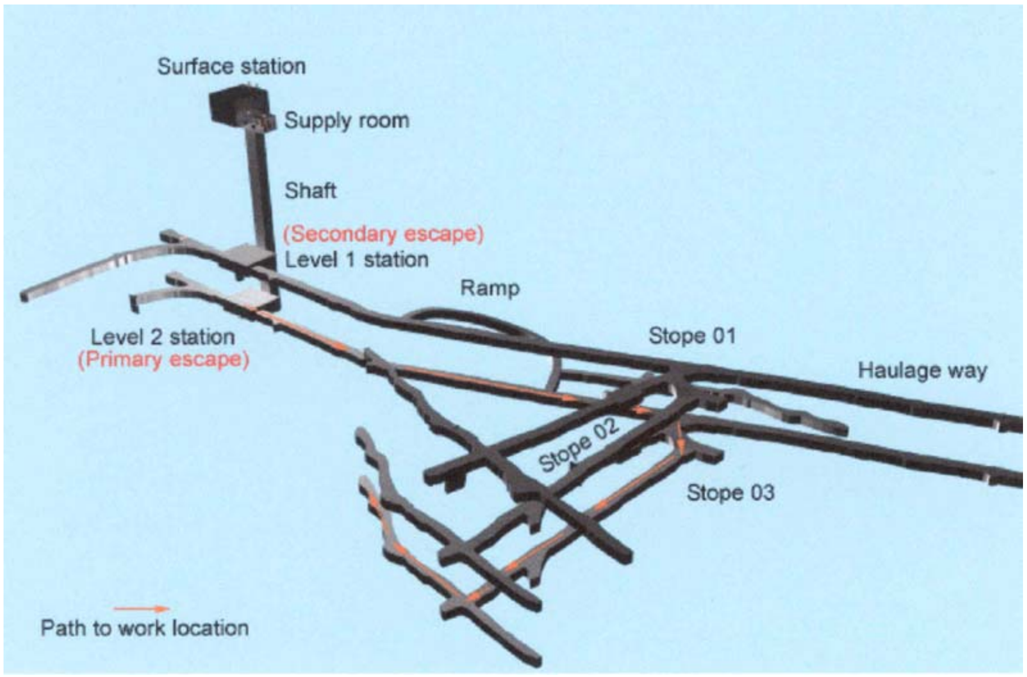

Figure 1.–A mine model used in the *evac1* scenario.

USER'S GUIDE

To install and run MTS, the following minimum computer system specifications are required: Pentium 120 or faster processor, 32 Mbytes RAM, 17 Mbytes hard disk space, Microsoft Windows versions 95 or 98, a video card with 8 Mbytes of video memory and three-dimensional graphics acceleration (3dfx Voodoo 3 graphics card is recommended for best results), sound card, and Microsoft's DirectX Version 5 or newer.

To install MTS, insert the CD[4] and run the *setup.exe* program (this may happen automatically if the computer's CD drive is configured to do this). Follow the instructions of the installation program. On the installation screen, choose **Typical** to accept all defaults or choose **Custom** to install the software in a chosen directory. On the custom installation screen, choose **High Resolution** unless there are problems with the video card and click **Change** to enter a new installation directory. Near the end of the installation, a video setup screen will appear. A list of video drivers and resolutions will be shown, but one will be highlighted. This is the recommended driver and resolution for the computer's system, so it is best to click on that driver and then click **Next.** (Video settings can be changed any time by running *VideoSetup.exe* in the installation directory.) Finally, the *MTSreadme.txt* file will appear. After this file has been reviewed, click **Finish**, and the installation will be complete. An MTS shortcut should appear on the desktop automatically.

GETTING STARTED

To run MTS, click on the shortcut installed on the desktop or go to the Windows **Start** button. Select **Programs** and select **NIOSH MTS**. The main menu will appear as below:

Miner Training Simulator

Instructions

Options

Go To Training Session

Restart Training Session

Quit

- Using the up and down arrow keys on the keyboard, go to **Options** and press the Enter key.
- Select **Choose Scenario**. This screen will list all available training scenarios. Each scenario will typically have its own mine model, training objectives, map, and script file. Several demonstration training scenarios are included in the training software.
- Select a scenario such as *evac1* (which will be explained below) using the arrow keys and the Enter key. The *evac1* scenario will load and place you at the starting point of the virtual mine, which in this case is the supply room.
- Press the **Escape** key to return to the previous menu.
- Select **Instructions**.
- Select **Training Objectives** to view the objectives of this scenario.
- Select **Keyboard and Mouse Commands** to learn how to navigate in the virtual mine.
- Select **Map** to view the entire virtual mine layout.
- Press the **Escape** key to reenter the simulated mine. (Selecting **Go To Training Session** will also accomplish this.)

In the simulation, either the mouse or the arrow keys can be used to navigate. Moving the mouse changes the view and the direction. The left mouse button moves you forward, and the right mouse button moves you backward. Other keys, such as the **Tab** key, will also be used for other actions such as opening doors. A message will appear at the top of the screen to give further instructions when necessary.

The main menu can be reentered at any time by pressing the **Escape** key. The simulation can be reset and restarted by selecting **Restart Training Session**. MTS can be exited by selecting **Quit**.

EVACUATION SCENARIO

Several demonstration scenarios are available with this software. To become familiar with the capabilities of the software and the methods used to navigate through the mine, select the *evac1* training scenario from the **Options, Load Scenario** menu.

This scenario assumes the user is new to this mine. It involves finding the location of the work assignment and practicing primary and secondary escape routes. This is a simplified scenario in that the primary escape route involves getting on the cage at the same level as the work location. The secondary route involves only getting on the cage at the level directly above the work location. The user will need to become familiar with the layout of the stopes and the ramp to find his or her way out of the mine.

After the scenario is loaded, instructions on the menu screen should be reviewed. The work location is specified in the objectives listed under the **Training Objectives** menu. (See **Map** for directions to the work location [figure 1].) When the user is familiar with the mine layout, the **Escape** key should be pressed until the simulation is reentered.

The simulation is begun in the supply room of the mine. The safety items needed for going underground should be obtained by walking next to and looking directly at them. An

[4]The CD is available from the author upon request. Contact Todd Ruff, Spokane Research Laboratory, NIOSH, 315 E. Montgomery Avenue, Spokane, WA, 99207, or e-mail ter5@cdc.gov.

icon will appear at the bottom of the screen for each piece of personal protective equipment as it is acquired. A health score also appears here (bottom of figure 2).

The trainee exits the supply room and enters the cage. The **Tab** key is typically used to open doors and gates. The trainee should follow the directions at the top of the screen to go to the correct mine level. He or she should then check the messages at the top of the screen to receive more instructions. For instance, pressing the **F** key will turn the employee's cap lamp on.

The trainee should proceed to the assigned stope. If the trainee is having trouble finding it, the mine map can be viewed on the menu screen at any time. When the trainee arrives at the work location, he or she will see a miner and a jackleg. The trainee should approach the miner and assist him. Soon after, an evacuation alarm will sound. The primary escape is a cage at the same level as the work location. If the cage is unavailable at this point, a secondary escape is the cage at the level above. The trainee should enter the cage and return to the surface.

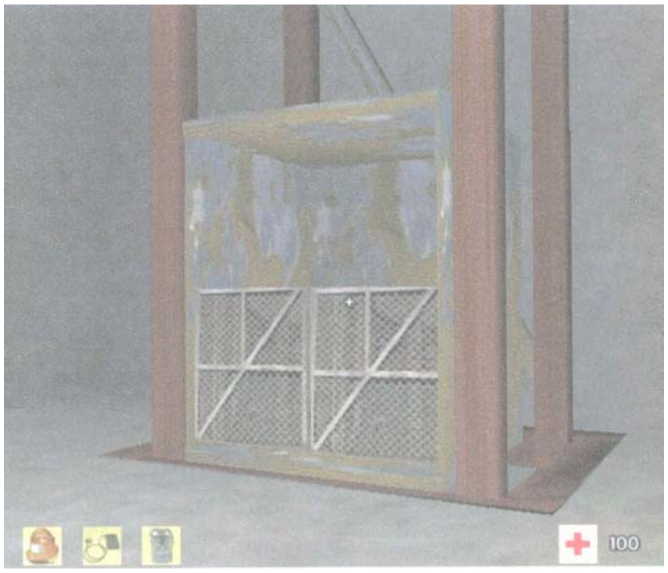

Figure 2.–A view of the cage *(elev01)* from MTS.

MODIFYING MTS

MTS can be customized to meet a particular mine's training needs. For instance, the virtual mine can be constructed from the mine's own maps so the mine layout matches what the employee will actually see. The safety hazards and messages presented to the trainee can be modified to represent actual hazards in the mine. To create a custom mine model and objects in the mine, a three-dimensional modeling package such as 3D Studio Max is required. To customize the way the trainee interacts with objects and hazards and to change safety messages and evacuation scenarios, a scripting language that interfaces to MTS is required. The scripting language, called Tool Command Language (TCL), is included with MTS, and a detailed description of its use is discussed in following sections. A three-dimensional modeling package is not included with MTS, and detailed instructions in the use of the modeling software will need to be reviewed by referring to the modeling software's documentation.

To create a custom training scenario, several files must be added to the directory structure of MTS (typically C:\Program Files\Niosh\Miner Training Simulator\). The mine model must be saved as a *.3ds* file (*evac1.3ds*, for example) in the data directory along with any textures used in the model. Creation of the mine model in 3D Studio Max requires basic skills that can
be acquired by going through any available tutorials. Some details about object creation are discussed in following sections. The trainee's starting position in the training scenario is defined by the placement of a single camera in the mine model.

A text file containing the training scenario objectives can also be saved in the data directory. This file can be created using Notepad[5] or any text editor, but must be saved with the *.txt* extension and must have the same name as the mine model (*evac1.txt*, for example). There is a size restriction on this file because MTS can only show the contents of the file that will fit on one screen (about 14 lines). Anymore lines may not appear on the screen.

A third file that shows the map of the mine can be saved to the data directory. This file is a graphics file with the same name as the mine model and an extension of *.jpg* (*evac1.jpg*, for example). The resolution of this graphic representation of the mine should not exceed 640 by 480 pixels; however, if greater detail is needed, the graphic can be split up into six sections of 256 by 256 pixels each. MTS will automatically reconstruct the picture when the file name matches the mine model name followed by the numbers 1 through 6 (*evac11.jpg, evac12.jpg*, etc.).

The final file that is needed in the data directory contains the scripting commands that define interactions, messages, and object movements in the mine model. This file also has the same name as the mine model, but contains the *.tcl* extension (*evac1.tcl*). Detailed instructions on creating this scripting file are discussed in the next section.

TOOL COMMAND LANGUAGE (TCL)

TCL was developed by John Ousterhout at the University of California at Berkley in 1988 and is freely available. Its purpose was to provide a cross-platform programming interface for the development and testing of software applications. For a comprehensive explanation and tutorial on the use of TCL, several books are available (for example, Sastry and Sastry 2000; Nelson 2000).

[5]Notepad, Microsoft Corp., Redmond, WA.

The advantage of using a scripting language when developing a training simulator such as MTS is that it provides a seamless, but separate, programming environment to customize the simulator. While the TCL file is just a standard text file created in a text editor such as Notepad, it behaves as if it were a part of the C++ code of MTS. However, the syntax that TCL uses is simpler than C++, and nonprogrammers can generate customized interactions between mine objects and the trainee. These interactions can be quickly programmed, tested, and modified without modifying or recompiling MTS.

A SHORT EXAMPLE

Before presenting detailed information on TCL, it may be beneficial to go through an example of a TCL command. This will make the purpose of TCL clearer and will provide some needed background on MTS. It is assumed that the reader has a copy of MTS running on his/her computer with the *evac1* training scenario installed.

A message written to the screen is a simple example of a TCL command. Several safety messages are presented in *evac1.* The mechanism for writing messages to the screen will be displayed shortly. First, however, it is necessary to go back to the creation of the *evac1* mine model.

In 3D Studio Max, a simplified mine model was created (figure 1) from mine maps. After the mine tunnels and shaft were modeled, various other items were inserted to "populate" the mine with familiar items, such as doors, a mancage (elevator), shovels, etc. For instance, the cage was modeled as an object named *elev01* and placed in the shaft (figure 2). The mine model was then exported as a file with the *.3ds* extension (MTS does not support 3D Studio Max files with the *.max* extension).

When MTS is started and the desired training scenario has been selected, the mine model and all of its objects are loaded into the simulation. But, without defining how a trainee interacts with these objects, they would just sit there, nice to look at, but stagnant. The TCL file contains commands that define what happens when the trainee interacts with an object, such as touching it or standing on it.

For this example, the presentation of an on-screen message is triggered when the trainee steps into the cage. This message reads, "*Always wear safety glasses while riding the cage*." The TCL command to present this message looks like this:

```
rule
trigger on elev01
action message 1 "Always wear safety glasses while
riding the cage."
```

The command *rule* defines this section as a rule. *Trigger* is another command that supplies the conditional statement, "*If the trainee is standing on elev01, then execute the action. Otherwise, skip the action."* There are several types of triggers, which will be discussed later. *Trigger on* requires that the trainee be standing on a specified object (in this case *elev01).* If this condition is true, then a message is written on the screen as long as the trainee is standing in the cage. After exiting the cage, the message turns off after 1 second. All the conditions following a rule must be true for the action(s) to be executed.

Note that *trigger* is not a core TCL command. TCL is versatile in that a custom command can be written in C++ and registered as a TCL command for the scripting language. Many custom commands have already been written for MTS, and no additional commands should be needed.

Almost all the objects in the sample mine have a TCL command associated with them. These commands allow doors to open, the cage to move, items to be picked up, and sounds to be played. TCL commands are also used to do complex operations, such as causing a groundfall that will injure a trainee if a hard hat is not being worn.

CREATING CUSTOM SCRIPTS

The first step in creating a TCL script for MTS is to identify those objects in the 3D Studio Max model that need to have an action associated with them. If no TCL file is created, no interaction will take place between the trainee and the objects in the mine. Doors will not open, items cannot be picked up, and so on. Once the object is identified, then an action(s) and a trigger(s) must be selected for that object as seen in the above example. The exact name of the object in the 3D Studio Max file must be used in the TCL script.

A TCL script is created in a standard text editor such as Notepad, but is saved with the extension *.tcl* in the "data" directory. The data directory can be found under the main directory called "Miner Training Simulator." (In a typical installation, the directory structure looks like C:\Program Files\Niosh\Miner Training Simulator\Data.) The name of the TCL file must match the scenario name. For example, if the scenario name is *evac1.3ds*, then the TCL file would be *evac1.tcl.*

All custom TCL commands developed for MTS are listed in the section labeled "Commands" along with some of the commonly used TCL commands that are standard to the language. Many more commands are available and are described in the programmer's reference (Nelson 2000). The following sections will describe the construction of TCL scripts for MTS in detail.

Summary of Syntax

; or *newline*	command separator.
xyz	example of a simple variable called xyz.
$xyz	specific value of the variable xyz.
[*command*]	brackets used for evaluating TCL commands or user-defined procedures.
"*hello $xyz*"	immediate substitution of an embedded variable.
{*hello $xyz*}	prevents substitution of an embedded variable or used to mark the beginning and end of a group of arguments/commands.
\	command continuation.
#	beginning of a comment line.

A TCL script is a text file with an extension of *.tcl,* which contains TCL commands. A TCL command is made up of a command name followed by arguments separated by a white space. For example,

```
trigger near "*USER*" $i 15
```

trigger near is the command name and *"*USER*", $i,* and *15* are its arguments.

A TCL script is a sequence of commands separated by new lines or semicolons. Most of the commands in MTS scripts are grouped into rules. For example,

```
rule
trigger on elev01
action message 1 "Always wear safety glasses while
riding the cage."

rule
trigger on elev01
trigger eval {expr {[string compare $elev01 "down1"]
== 0}}
trigger keydown KB_0
action slide elev01 0 144 0 15
action slide hgate01 0 144 0 15
action slide hgate02 0 144 0 15
action eval {set elev01 "moving"}
action wait 15 eval {set elev01 "up"}
```

This is a valid TCL script.

Variables

Variables are important to TCL as in any other programming language. A variable simply holds a value. Variable names can contain letters, digits, and underscores. Examples of possible variable names are *Doors, medkit2,* and *Gate_1.* Variables cannot contain characters reserved for use by TCL, as shown in the summary of syntax above.

Variables are defined and assigned a value using the command *set* and cleared using *unset.* The *set* command takes a variable name (in the following examples, the variable name is *Doors*) from its first argument and sets it to be the equivalent of the second argument.

Example	Value contained in *Doors*
set Doors 5	5
set Doors "off"	“off”
set Doors Gates	Gates

To set a variable to be equivalent to the value of another variable, variable substitution is used. Variable substitution consists of putting a $ before the name of the variable so that the command name knows that the value of the variable should be used and not the variable name. For example,

```
set xyz 5
set Doors $xyz
```

First, *xyz* is set to 5. Then *Doors* is set to the value of *xyz*, which is 5.

Command Evaluation

Commands can be simple, like those shown above, or they can be nested. Square brackets are used to invoke a nested command. For example,

```
set x 2
set y 3
set z "The result of x+y is [expr $x + $y]."
```

So *z* would contain the expression “*The result of x+y is 5*.” Note that *expr* invokes arithmetic computation. Arithmetic operators include + (plus), - (minus), * (multiply), / (divide), && (logical AND), and || (logical OR). A more comprehensive list can be found in Nelson (2000).

Lists

A list is a series of elements separated by spaces. Below is an example list used in a TCL file.

```
set inventory "selfresc lamp glasses"
```

In this example, the variable *inventory* is set to equal *selfresc lamp glasses*, which is a list of elements. It can also be thought of as a string whose words are elements that can be searched for, appended, or deleted. For example, the command *lappend* appends an item to a list, and *lsearch* searches through a list to see if an item is present. *lsearch* returns -1 if the item is not in the list or 0 if the item is in the list. More details on these commands are provided below.

Commands

This section lists TLC commands and usage. Many other commands are available and can be found in the references (Sastry and Sastry 2000; Nelson 2000). The *evac1.tcl* file, along with detailed explanations of each command, are included in appendix A. This file contains many examples of simple and complex commands. In fact, a TCL script for a new training scenario could be built by simply copying most sections of the example code and changing object names and such to match the appropriate mine model. The following describes commands most commonly used in building TCL scripts for MTS.

action eval — Evaluates a TCL script and returns the result.

Usage: action eval **{script}**

Examples: action eval **{set evac "off" }**

Description: *action eval* interprets a TCL script and performs the command or commands in the script.

action message — Displays a message on the screen.

Usage: action message **time "string"**

Example: action message **5 "You picked up a hard hat."**

Description: *action message* displays the string "*You picked up a hard hat*." on the screen for a given amount of time (5 seconds). A time of 0 second can be used to display the message only while the associated trigger is true.

action rotate — Rotates an object.

Usage: action rotate **object x y z time**

Example: action rotate **Door1 0 -5 0 4**

Description: *action rotate* rotates the object *Door1* around a given axis *x, y, z* for an amount of time (in seconds). The axis is chosen by assigning a nonzero number to one of axes *x, y*, or *z*. The object can rotate around one axis only. Direction of rotation is controlled by the sign of the number, and speed of rotation is determined by the magnitude of the number. For instance, *Door1* will rotate for 4 seconds at a fairly fast rate (five times faster than if the numbers were *0 –1 0 4)*. Unfortunately, the appropriate amount of rotation is determined through trial and error, but with practice, fairly accurate rotation can be accomplished. Note that the origin of the axis of the object is determined in 3D Studio Max when the object is created. If the object must rotate around one of its edges (like a door on a hinge), the object's origin must be set to one edge.

action slide — Moves an object at a constant speed and direction.

Usage: action slide **object x y z time**

Example: action slide **elev01 0 144 0   15**

Description: *action slide* moves the object *elev01* (the exact name of the 3D Studio Max object) from its starting position to the position with coordinates 0,144,0. It takes 15 seconds to get to the final position.

action sound — Plays a *.wav* sound file when the trainee is within a certain distance of the sound's origin.

Usage: action sound **-loop file object {x y z} distance**

Example: action sound **-loop "sounds/water.wav" Box97 {0 0 0} 20**

Description: *action sound* plays the *water.wav* file found in the "sounds" directory. The origin of the sound is at the 3D Studio Max object *Box97*. *x y z* entries allow the sound to be offset if needed, but in this example, there is no offset. The sound is heard only if the trainee is within a distance of 20 feet. (The units of distance here depend on the units settings of the 3D Studio Max model.) The *-loop* option plays the sound file in a continuous loop. Eliminating the *–loop* entry allows the sound to be played just once.

action spline Moves an object with acceleration and deceleration.

Usage:

```
action spline
object {{ {LX1  LY1  LZ1} {VX1  VY1  VZ1} T1}
{ {LX2  LY2  LZ2} {VX2  VY2  VZ2} T2} }
.
.
.

}
```

Example:

```
action spline elev01 {
{{0 -5 0} {0 -10 0} 1}
{{0 -139 0} {0 -10 0} 13.4}
{{0 -144 0} {0 0 0} 1}
}
```

Description: *object* is the object to be moved. *LXn, LYn*, and *LZn* specify the coordinates of the new location of the object relative to the object's original position. *VXn, VYn*, and *VZn* specify the velocity of the object in units of measure designated by the model (feet per second, meters per second). *Tn* indicates the time in seconds it will take for the object to reach the specified position and velocity.

As many segments as needed can be used to specify different velocities along a path. For instance, an elevator that accelerates and decelerates is more realistic than simply using the slide command.

In the above example, the elevator is at the top of the shaft, which is considered its starting position (0,0,0) as defined in 3D Studio Max. The first line moves the elevator down 5 feet in the y direction with a final velocity of 10 feet/second. It accomplishes this movement in 1second. The second line moves the elevator 139 feet below its starting position with a final velocity of 10 feet/second No acceleration takes place because the final velocity matches the starting velocity. It takes 13.4 seconds to move the elevator this far, which was determined by trial and error so that the motion appeared smooth. Finally, the elevator is moved to the bottom of the shaft, 144 feet below its starting position, with a final velocity of 0. This is done in 1 second, giving the appearance of deceleration as the elevator approaches its stopping point.

action wait Waits a specified amount of time before a command is executed.

Usage:

```
action wait time command
```

Examples:

```
action wait 7 eval {set doors "off"}
action wait 5 rotate box01  0  2  0  1
```

Description: *action wait* waits an amount of time in seconds before executing a command. In the first example, there is a 7-second delay before the *eval* command is executed, setting the variable *doors* to *off*. In the second example, there is a 5-second delay before the *rotate* command is executed.

expr Evaluates an arithmetic expression.

Usage:

```
expr argument argument argument …
```

Examples:	expr **20 + 2**
	expr **{[string compare $switch "on"] == 0 }**
Description:	*expr* takes all the arguments following it, evaluates them as an expression, and returns the result. The first example returns 22. The second example checks to see if the variable *switch* is on. The *string compare* returns 0 (true) if the value of *switch* is on, or –1 (false) if it is not on. Then *expr* returns a 0 if *switch* is on and allows execution of the next command. This is useful in conditional statements.
foreach	Iterates through all the elements in a list.
Usage:	foreach **variable {list} {script}**
Example:	foreach **x {boots lamps glasses} {action message 5 "You have $x."}**
Description:	*foreach* is followed by a variable name, a list of values to be assigned to the variable name, and a TCL command or script. *foreach* goes through the list of values and, for each one, assigns the specified value to the variable name and interprets the TCL script. First, *"You have boots*." would be written on the screen for 5 seconds, then *"You have lamps*." and so on.
getnames	Obtains the names of the objects in the 3D Studio Max file of the current scenario.
Usage:	[getnames]
Example:	See appendix A for a full example that uses *getnames*.
Description:	*getnames* is a custom command and is executed by inserting it in brackets.
if	Executes scripts conditionally.
Usage:	if **{expression} {script}**
Example:	if **{ x == 5 } { set x 7 }**
Description:	*if* evaluates the expression following it and returns a Boolean value of 0 if true or -1 if false. If the expression is true, *script* is interpreted. If the expression is false, *script* is skipped.
lappend Adds or appends elements to a list.	
Usage:	lappend **listname value value**
Example:	lappend **inventory "selfresc" "lamp"**
Description:	*lappend* adds the items *selfresc* and *lamp* to the *inventory* list.
lsearch	Searches for an item in a list.
Usage:	lsearch **$listname value**
Example:	lsearch **$inventory "lamp"**
Description:	*lsearch* searches through the list *inventory* to see if *lamp* exists in the list. Note that $ precedes the list name so that the actual list is searched and not just the list name. If *lamp* is found in the list, then *lsearch* returns 0; if not, it returns –1.
modstat	Allows access to certain variables in the MTS code.
Usage:	modstat **variable value**
Example:	modstat **lamp 1**
Description:	*modstat* allows the TCL script to change the variable values used in the MTS code. Only certain variables are accessible (see appendix A for other examples). Here, *modstat* allows the variable *lamp* to be set to 1, which then allows the small cap lamp icon to be shown on the lower portion of the screen. This occurs when the cap lamp is acquired in the supply room. (See appendix A.)
regexp	Compares an expression or variable with a string.
Usage:	regexp **expression string**
Examples:	regexp **door.* $i**
	regexp **$var "lamp"**

Description: *regexp* compares an expression and a string. If they match, *regexp* returns a 0; if not, it returns –1. In the first example, the wildcard *door.** is compared to the value of the variable *i*. In the second example, the value of *var* is compared to the string *lamp.*

rule Specifies a set of commands that make up a rule.

Usage: rule
command

Examples: rule
trigger near **"*USER*" orepass 7**
action message **0 "You are near the orepass. Tie off before getting closer."**

Description: *rule* is used to specify that a set of following commands should be grouped to form a rule. A rule consists of conditions and actions. All of the conditions (usually triggers) must be true before the actions are executed. If one of the conditions is not true, all lines following are skipped until the next rule. In the above example, the *trigger near* condition must be true before the *action message* is executed. Multiple triggers and actions can be contained in a rule as in appendix A.

set Writes a value to a variable.

Usage: set **variable value**

Example: set **x 5**
set **medkit "on"**

Description: *set* assigns a *value* to a *variable* and returns the *value* as a result. If the variable name does not already exist, it will create a new variable with that name. In the first example, *x* is assigned the value *5*. In the second example, the variable *medkit* is assigned the string *on.*

string compare Compares strings.

Usage: string compare **string1 string2**

Examples: string compare **$medkit "on"**
string compare **"one" "on"**

Description: *string compare* compares *string1* and *string2* and returns -1 if *string1* is less than *string2*, 0 if *string1* and *string2* are equal, or 1 if *string1* is greater than *string2*.

trigger eval Evaluates a TCL script.

Usage: trigger eval **{script}**

Example: trigger eval **{expr { $x == 5 } }**

Description: *trigger eval* interprets *script*, which must return a Boolean value. If *script* returns as true, then TCL will continue to interpret the current rule; otherwise, TCL will stop interpreting the current rule and move to the next one. In the example, if the value of *x* is equal to 5, then the next command in the rule is interpreted.

trigger far Checks to see if an object is a certain distance away from another object.

Usage: trigger far **obj1 obj2 dist**

Example: trigger far **"*USER*" supplyroom 20**

Description: *trigger far* checks the distance between two objects, *obj1* and *obj2*. If the distance between the objects is greater than *dist*, then TCL will continue to interpret the current rule. If the distance is less than or equal to *dist,* then TCL will stop interpreting the current rule and move on to interpreting the next rule. If the trainee's, or user's, position is to be checked, **USER** is entered as *obj1*. Other objects must be valid names from the 3D Studio Max model. In the example, the distance between the trainee and the mine's supply room is checked.

trigger keydown	Checks the state of a key.
Usage:	trigger keydown **key**
Example:	trigger keydown **KB_TAB**
Description:	*trigger keydown* determines whether a certain key on the keyboard is being pressed. If *key* is pressed, TCL will continue to interpret the current rule. If *key* is not pressed, TCL will stop interpreting the current rule and skip to the next rule. The example shows how to specify a key by using *KB_*, which is followed by the desired key, for example, *KB_2, KB_G.*
trigger near	Checks to see if an object is near another object.
Usage:	trigger near **obj1 obj2 dist**
Examples:	trigger near **"*USER*" truck 5**
	trigger near **elev01 forklift 30**
Description:	*trigger near* checks the distance between two objects, *obj1* and *obj2*. If the distance between the objects is less than *dist*, then TCL will continue to interpret the current rule. If the distance is greater than or equal to *dist,* then TCL will stop interpreting the current rule and move on to interpreting the next rule. If the trainee's or user's position is to be checked, **USER** is entered as *obj1*. Other objects must be valid names from the 3D Studio Max model. In the first example, the distance between the trainee and the object *truck* is checked to see if it is less than 5. In the second example, the distance between two objects, *elev01* and *forklift*, is checked to see if it is less than 30.
trigger on	Checks if the trainee is standing on an object.
Usage:	trigger on **object**
Example:	trigger on **elev01**
Description:	*trigger on* checks to see if the user (or current camera position) is on *object*. If the user is on *object*, then TCL continues to interpret the current rule. If the user is not on *object,* TCL stops interpreting the current rule and skips to the next rule.
trigger touch	Checks to see if an object is being "touched" by the trainee. For an object to be touched, the trainee must be very close to an object and looking at it.
Usage:	trigger touch **object**
Example:	trigger touch **lamp**
Description:	*trigger touch* checks to see if *object* is being touched by the user (current camera position). For an object to be touched, it must be in the camera's view plane (visible on the screen), and the trainee must be very close to it. If *object* is being touched by the trainee, TCL will continue to interpret the current rule; otherwise, TCL will move to the next rule.
updatecvars	Executes a variable update.
Usage:	updatecvars **1**
Examples:	modstat lamp **1**
	updatecvars **1**
Description:	*updatecvars* is used after a *modstat* command to signal to MTS code that a variable value has been changed.

REFERENCES

Filigenzi, Marc T., Timothy J. Orr, and Todd M. Ruff. 2000. Virtual Reality for Mine Safety Training. *Journal of Applied Occupational and Environmental Hygiene*, vol. 5, no. 6, pp. 465-496.

Nelson, Christopher. Tcl/Tk Programmer's Reference. 2000. McGraw-Hill, 539 pp.

Sastry, Venkat V.S.S., and Lakshmi Sastry. 2000. Teach Yourself Tcl/Tk in 24 Hours. Sams Pub., 494 pp.

APPENDIX A

The following is an example of a tcl file for the evacuation route scenario – *evac1*. You can find this file (*evac1.tcl*) in the data directory. Comments and explanations added here will be in **bold text.** Comments or unused commands in the actual code will be preceded by a comment indicator (#).

This TCL file must begin by initializing variables. This is done within the *action eval* command. Variables can be declared on the fly and do not need to be declared at the beginning, but it is important to initialize the variables. Additional explanations for each variable can be found where that variable is used.

```
action eval {
```

source utils.tcl	**The location of the tcl utilities must be specified (*utils.tcl* must reside in the same directory as *MTS.exe*).**
set elev01 "up"	**This variable keeps track of elevator position.**
set inventory""	**This variable keeps track of the inventory for safety items and is initially set to be empty.**
set rockfall01 "up"	**This variable keeps track of loose rock position**.
set Gates "off"	**This variable keeps track of gate status.**
set Doors "off"	**This variable keeps track of door status.**
set evac "off"	**This variable keeps track of whether or not the evacuation procedure has been initiated**.
#set medkit1 "on"	**This is variable is optional and is needed only if the user has medical kits (medkits) that need to be collected in the scenario. See the** *demo* **scenario for an example. This command must be repeated to match the number of medkits in the scenario.**

```
    }
```

This section of code checks to see if the object the trainee or user (USER**) is next to is a door. First, the object name is acquired using a custom routine called *getnames*. Each object is compared to the expression *door.** using *regexp.* If the object matches, then the door functions are executed. If the user is near a door, a message appears that gives instructions for opening it. The Tab key (*KB_TAB*) is used to open the door. Remember that 0 equals true and –1 equals false in TCL code**.

```
foreach i [getnames]
    { if {[regexp door.* $i]} {
        rule
        #Check to see if the door is already open.
        trigger eval {expr {[string compare $Doors "off"] == 0}}
        #If the door is closed (off), then display a message when the user is near the door.
        trigger near "*USER*" $i 2
        action message 0 "Press tab to open door."
```

Note that in a rule section, all the condition statements, such as *trigger*, must be true in order for the final actions to execute at the end of the rule. This rule section would sound like this in English: For this rule, if the user is near the door, and the tab key is pressed, and the door is not already open, set the status of the door to open and rotate the door to the open position. Then wait 5 seconds and rotate the door back to the closed position and set the status of the door back to closed.

```
        rule
        #Open the door using the tab key when near the door.
        trigger near "*USER*" $i 2
        trigger keydown KB_TAB
        #Also check to see if the door is already open so the door does not
        #swing too far.
        trigger eval {expr {[string compare $Doors "off"] == 0}}
        action eval {set Doors "on"}
        #Rotate the door open. The direction is controlled by the sign in front of
        #the rotation axis offset; in this case, along the y axis at the edge of door.
        action rotate $i 0 -2 0 1
        #Wait 5 seconds and rotate the door closed.
        action wait 5 rotate $i 0 2 0 1
        action wait 7 eval {set Doors "off"}
        }
```

This section of code handles the acquisition of personal safety equipment in the supply room.

```
    #Look for objects named hardhat.
    if {[regexp hardhat.* $i]} {
        rule
        #Check to see if the user already has the hard hat.
        trigger eval {expr {[lsearch $inventory "hard hat"] == -1}}
        #Make sure the user is very close to or touching the hard hat.
        trigger touch $i
        #Then add the hard hat to the inventory.
        action eval {lappend inventory "hard hat"
        #Update this variable so that the hard hat icon appears on screen.
        modstat cap 1
        updatecvars 1}
        #Display a message.
        action message 5 "You picked up a hard hat."
        }
```

Do the same for the cap lamp and battery.

```
    if {[regexp battery.* $i]} {
        rule
        trigger eval {expr {[lsearch $inventory "lamp"] == -1}}
        trigger touch $i
        action eval {lappend inventory "lamp"
        modstat lamp 1
        updatecvars 1}
        action message 5 "You picked up a cap lamp."
        }
```

Do the same for the self-rescuer.

```
    if {[regexp selfresc.* $i]} {
        rule
        trigger eval {expr {[lsearch $inventory "selfresc"] == -1}}
        trigger touch $i
        action eval {lappend inventory "selfresc"
        modstat rescuer 1
        updatecvars 1}
        action message 5 "You picked up a self rescuer."
        }
```

Do the same for the safety glasses.

```
    if {[regexp glass.* $i]} {
    rule
        trigger eval {expr {[lsearch $inventory "glasses"] == -1}}
        trigger touch $I
        action eval {lappend inventory "glasses"
        modstat glasses 1
        updatecvars 1}
        action message 5 "You picked up safety glasses."
        }
```

Do the same for the earplugs.

```
if {[regexp earplug.* $i]} {
        rule
        trigger eval {expr {[lsearch $inventory "earplug"] == -1}}
        trigger touch $i
        action eval {lappend inventory "earplug"
        modstat earplug 1
        updatecvars 1}
        action message 5 "You picked up ear plugs."
        }
```

This section of code checks to see if the gates to the elevator (cage) are closed. If so, and the user is near the gates, instructions are given to open them.

```
rule
trigger eval {expr {[string compare $Gates "off"] == 0}}
trigger near "*USER*" hgate01 2
trigger near "*USER*" hgate02 2
action message 0 "Press tab to open gates."

rule
trigger near "*USER*" hgate01 2
trigger near "*USER*" hgate02 2
trigger keydown KB_TAB
trigger eval {expr {[string compare $Gates "off"] == 0}}
action eval {set Gates "on"}
#Two rotate actions are required here because the gate is actually two swinging doors.
action rotate hgate01 0 1.5 0 1
action rotate hgate02 0 -1.5 0 1
#Wait 4 seconds, then close the gates.
action wait 4 rotate hgate01 0 -1.5 0 1
action wait 4 rotate hgate02 0 1.5 0 1
action wait 7 eval {set Gates "off"}
```

This section of code controls the elevator or cage. It handles the complex problem of having multiple levels in the mine and designating to which level to go.

```
################elevator code ##############
rule
#Check to see if the user is standing on the elevator.
trigger on elev01
#Check to see whether the elevator is at the surface or top level.
trigger eval {expr {[string compare $elev01 "up"] == 0}}
#Give instructions for moving the elevator.
action message 0 "Press 1 for level 1, 2 for level 2, etc."

#If the elevator is at the surface and 1 is pressed, then go to level 1.
```

```
rule
trigger on elev01
trigger eval {expr {[string compare $elev01 "up"] == 0}}
trigger keydown KB_1
#This is a spline movement action for the elevator.  It allows acceleration and deceleration. The coordinates to move to can
be estimated in 3D Studio Max and refined by adjusting these values.
action spline elev01 {  {{0 -5 0} {0 -10 0} 1}
{{0 -139 0} {0 -10 0} 13.4}
{{0 -144 0} {0 0 0} 1}
}
```

Both gates must follow the elevator using the same movements since they are attached to it. Parent/child relationships like this must be done here because this information does not import correctly from 3D Studio Max.

```
action spline hgate01 {
{{0 -5 0} {0 -10 0} 1}
{{0 -139 0} {0 -10 0} 13.4}
{{0 -144 0} {0 0 0} 1}
}

action spline hgate02 {
{{0 -5 0} {0 -10 0} 1}
{{0 -139 0} {0 -10 0} 13.4}
{{0 -144 0} {0 0 0} 1}
}

action eval {set elev01 "moving"}
# Keep track of where the elevator is.
action wait 20 eval {set elev01 "down1"}

# If the elevator is at the surface and 2 is pressed, then go to level 2.
rule
trigger on elev01
trigger eval {expr {[string compare $elev01 "up"] == 0}}
trigger keydown KB_2
action spline elev01 {
{{0 -5 0} {0 -10 0} 1}
{{0 -185 0} {0 -10 0} 20}
{{0 -191 0} {0 0 0} 1}
}
action spline hgate01 {
{{0 -5 0} {0 -10 0} 1}
{{0 -185 0} {0 -10 0} 20}
{{0 -191 0} {0 0 0} 1}
}
action spline hgate02 {
{{0 -5 0} {0 -10 0} 1}
{{0 -185 0} {0 -10 0} 20}
{{0 -191 0} {0 0 0} 1}
}
action eval {set elev01 "moving"}
action wait 25 eval {set elev01 "down2"}
```

If the user is on either of the lower levels and reenters the elevator, then the user can only return to the surface. This simplifies things.

```
#Check to see what level the user is on and give instructions.
rule
trigger on elev01
trigger eval {expr {[string compare $elev01 "down1"] == 0}}
action message 0 "Press 0 to return to surface."

#Check to see what level the user is on and give instructions.
rule
trigger on elev01
trigger eval {expr {[string compare $elev01 "down2"] == 0}}
action message 0 "Press 0 to return to surface."
```

Return to the surface from level 1. This movement uses slide instead of spline just to show the difference in movement. Either can be used, but spline allows acceleration and deceleration, making the motion more realistic.

```
#Go to surface from level 1.
rule
trigger on elev01
trigger eval {expr {[string compare $elev01 "down1"] == 0}}
trigger keydown KB_0
action slide elev01 0 144 0 15
action slide hgate01 0 144 0 15
action slide hgate02 0 144 0 15
action eval {set elev01 "moving"}
action wait 15 eval {set elev01 "up"}

#Go to surface from level 2.
rule
trigger on elev01
trigger eval {expr {[string compare $elev01 "down2"] == 0}}
trigger keydown KB_0
action slide elev01 0 191 0 15
action slide hgate01 0 191 0 15
action slide hgate02 0 191 0 15
action eval {set elev01 "moving"}
action wait 20 eval {set elev01 "up"}
```

```
##########################
```

This section of code controls the evacuation scenario. When the user arrives at the designated work location, stench gas is released, signaling that the miners must evacuate. The user is to follow the primary escape route from the work location. When the user arrives at the elevator, something goes wrong, and the elevator moves to an upper level. Now the user must use the secondary escape route. When the user finally reaches the surface, the evacuation simulation ends.

```
########## Evacuation code ###################

#Start the evacuation training when the user arrives at the correct work location.
rule
#The user's designated work assignment is to help a miner with drilling. Arrival at the
#work location is triggered when the user is near the other miner's hard hat.
trigger near "*USER*" hardhat04 10
#Start the evacuation with this message:
action message 0 "Mine emergency - You smell stench gas! Proceed to primary escape."
#Play a warning sound, too.
```

```
action sound "Sounds/spark7.wav" hardhat04 {0 0 0} 10
action eval {set evac "on"}

#Move the elevator so that the secondary escape route must be used.
rule
#Check to see if the simulation has started.
trigger eval {expr {[string compare $evac "on"] == 0}}
#Check the position of the elevator.
trigger eval {expr {[string compare $elev01 "down2"] == 0}}
#Use the muck pile on level 2 to trigger the movement of the elevator.
trigger near "*USER*" Cone01 10
#Move the elevator up to level 1 to throw in a glitch.
action slide elev01 0 47 0 15
action slide hgate01 0 47 0 15
action slide hgate02 0 47 0 15
action eval {set elev01 "moving"}
action wait 5 eval {set elev01 "down1"}

#Now let the user know that the secondary escape route must be used.
rule
trigger near "*USER*" ext02 20
trigger eval {expr {[string compare $evac "on"] == 0}}
trigger eval {expr {[string compare $elev01 "down1"] == 0}}
action message 0 "Cage is not available here - use secondary escape!"

#If the user makes it out of the mine, end the evacuation portion of the simulation.
rule
trigger eval {expr {[string compare $evac "on"] == 0}}
trigger eval {expr {[string compare $elev01 "up"] == 0}}
trigger near "*USER*" hgate02 10
action message 1 "Congratulations - you escaped."
action message 1 "Press Esc to end the simulation."
action wait 15 eval {set evac "off"}
```

```
####################
```

This section of the code introduces several hazards that will harm the user if the appropriate personal protective equipment was not acquired.

```
##########Code for checking whether all safety equipment was acquired########
```

If no hard hat was acquired and the user walks under loose rocks on level 1, the rocks fall.

```
rule
#Check to see if the user is standing on the loose rock trigger rockfall01.
trigger on rockfall01
#Check to see if the hard hat has been acquired.
trigger eval {expr {[lsearch $inventory "hard hat"] == -1}}
#Check to see if the rock fall has already been triggered.
trigger eval {expr {[string compare $rockfall01 "up"] ==0}}
#Then set the rock fall to the down position.
action eval {set rockfall01 "down"}
action eval {
    updatecvars 0
    #Set the user's health to 0 (which causes death).
```

```
        modstat health 0
        #Update the health icon on the screen.
        updatecvars 1
    }
    #Move the rocks down.
    action slide rock01 0 -10 0 .5
    #Let the user know what happened.
    action message 10 "Rocks fell on your head."
```

Remind the user to wear a hard hat outside of the supply room.

```
    rule
    trigger far "*USER*" supplyroom 20
    trigger eval {expr {[lsearch $inventory "hard hat"] == -1}}
    action message .1 "You forgot your hardhat."
```

Remind the user never to go underground without a self-rescuer.

```
    rule
    trigger far "*USER*" supplyroom 50
    trigger eval {expr {[lsearch $inventory "selfresc"] == -1}}
    action message .1 "Do not go underground without a self-rescuer."
```

If the user goes underground without a self-rescuer and the evacuation simulation starts, the user's health is decremented until death occurs.

```
    rule
    trigger eval {expr {[string compare $evac "on"] == 0}}
    trigger eval {expr {[lsearch $inventory "selfresc"] == -1}}
    trigger far "*USER*" hardhat04 2
    action eval {
        modstat health 0
        updatecvars 0
        modstat health [expr {[statval health] - 1}]
        updatecvars 1
    }

    action message .1 "You are dying from carbon monoxide poisoning."

################
```

Here is an example of a falling hazard. If the user falls into the ore pass on level 1, then health is decremented until death occurs.

```
    rule
    trigger on orepass
    action eval {
        modstat health 0
        updatecvars 0
        modstat health [expr {[statval health] - 1}]
        updatecvars 1
    }
```

This section of the code provides safety reminders.

```
#########Safety reminders###########

        rule
        trigger on elev01
        action message 1 "Always wear safety glasses while riding the cage."

        rule
        trigger near "*USER*" orepass 7
        action message 0 "You are near the ore pass. Tie off before getting closer."

        rule
        trigger near "*USER*" Box10 5
        action message 0 "Be cautious around electrical equipment."

        rule
        trigger near "*USER*" door02 6
        action message 0 "Danger - explosives storage."

        rule
        trigger near "*USER*" Box50 10
        action message 0 "Keep hands inside the cage at all times."

        rule
        trigger near "*USER*" Box58 10
        action message 0 "Press F to turn your cap lamp on."

        rule
        trigger near "*USER*" topbucket2 35
        action message 0 "LHD ahead. Does the operator see you?"

        rule
        trigger near "*USER*" fan03 10
        action message 0 "This ventilation fan is loud. Are you wearing ear protection?"

        rule
        trigger near "*USER*" fan01 10
        action message 0 "This ventilation fan is loud. Are you wearing ear protection?"

        rule
        trigger near "*USER*" ebox03 5
        action message 0 "Be cautious around electrical equipment."

        rule
        trigger near "*USER*" Box98 5
        action message 0 "Be cautious around electrical equipment."

        rule
        trigger near "*USER*" rock11 20
        action message 0 "Do not enter unstable areas. Report damage to shift boss."

        rule
        trigger near "*USER*" water01 10
        action message 0 "Pumping station. Water depth can be deceiving."
```

```
    rule
    trigger on sr-floor01
    action message 0 "Press escape to review your instructions."

    rule
    trigger eval {expr {[string compare $Gates "on"] == 0}}
    trigger eval {expr {[string compare $elev01 "moving"] == 0}}
    trigger on elev01
    action message 0 "Always close cage doors first."

################
```

This section is an example of how to decrement health if hit by moving equipment. Run the "demo" scenario to see this executed and open *demo.tcl* to see an example of moving equipment.

```
    # Decrement health if hit by a moving loader (LHDGroup02).
    rule
    trigger on LHDGroup02
    action eval {
        modstat health 0
        updatecvars 0
        modstat health [expr {[statval health] - 10}]
        updatecvars 1
    }
```

This section of code controls the sounds. The user needs to be near an object to trigger a sound.

```
##########Sounds##########

    rule
    trigger near "*USER*" Box10 20
    #Trigger an electrical humming noise when near an electrical box.
    action sound -loop "Sounds/hum.wav" Box10 {0 0 0} 20

    rule
    trigger near "*USER*" Box97 50
    action sound -loop "Sounds/Water.wav" Box97 {0 0 0} 20

    rule
    trigger near "*USER*" epump00 50
    action sound -loop "Sounds/pump-02.wav" epump00 {0 0 0} 20

    rule
    trigger near "*USER*" elev01 20
    action sound -loop "Sounds/Wind1.wav" elev01 {0 0 5} 15

    rule
    trigger near "*USER*" fan03 70
    action sound -loop "Sounds/fan.wav" fan03 {0 0 0} 40

    rule
    trigger near "*USER*" fan01 70
    action sound -loop "Sounds/fan.wav" fan01 {0 0 0} 40
    rule
    trigger near "*USER*" orepass 50
    action sound -loop "Sounds/drip.wav" orepass {0 0 0} 20

    rule
    trigger near "*USER*" LHDGroup02 60
    action sound -loop "Sounds/diesel.wav" LHDGroup02 {0 0 0} 50

##############
```

This section of code is not used in *evac1*, but is used in *demo*. It allows the user to pick up medical kits that are strewn about the mine. Each section must be duplicated according to the number of medical kits you have. The maximum number of medical kits that can be used is 11.

```
    #rule
    #trigger near "*USER*" kit01 5
    #trigger eval {expr {[string compare $medkit1 "on"] == 0}}
    #action eval {
    #modstat medkit1 0
    #updatecvars 1
    #set medkit1 "off"}
    #action sound "Sounds/spark7.wav" kit01 {0 0 0} 5
    #action message 5 "You picked up a first-aid kit."

    #rule
    #trigger near "*USER*" kit02 5
    #trigger eval {expr {[string compare $medkit2 "on"] == 0}}
    #action eval {
    #modstat medkit2 0
    #updatecvars 1
    #set medkit2 "off"}
    #action sound "Sounds/spark7.wav" kit02 {0 0 0} 5
    #action message 5 "You picked up a first-aid kit."
```

NIOSH

Delivering on the Nation's Promise:
Safety and health at work for all people
Through research and prevention

For information about occupational safety and health topics contact NIOSH at:

1-800-35-NIOSH (1-800-356-4674)
Fax: 513-533-8573
E-mail: pubstaft@cdc.gov
www.cdc.gov/niosh

DHHS-(NIOSH) Publication No. 2001-136