Advanced features

Mesh can be enjoyed purely as a game, but it also includes a number of advanced features for those who enjoy creating their own puzzles. Under the *File* menu you will find:

Create new puzzle set...: this brings up a dialog which allows you to create a whole new puzzle set. Besides specifying the name for the new puzzle set, you also have two options whether or not to retain the class definitions from the current puzzle set. If you're only interested in creating new puzzle levels, using the classes of objects already defined in the current puzzle set (or if you want to retain those so that you can add more classes of your own devising), then you should leave it at the default setting of "Retain class definitions."

If, however, you intend to create an entirely ***new*** game (that may or may not even have any resemblance to the Hero's Hearts game), and intend to create ALL of your own classes of objects, then you would change this setting to "Delete class definitions". This will create a puzzle set containing only one class of object, Hero, with only very simple class code for that Hero class. You can modify that Hero class, rename it, etc, and add as many (or few) additional classes as you wish, creating a whole new game. Needless to say, this option is only intended for the most advanced users, and will require a great deal of knowledge and programming experience. ***REMEMBER: Class editing is NOT a supported feature. When attempting this, you're on your own!***

Edit levels: this takes the program into its level editing mode where you can add, modify, and delete the levels in the puzzle set. For help on editing levels, see Level Editing.

Edit classes: this takes the program into its class editing mode where you can add, modify, and delete classes of objects in the puzzle set. For help on editing classes, see Class Editing.

Message trace window: this brings up the "message tracing window". The Mesh game engine works by sending a series of *messages* to the objects on the level being played. These messages (or function calls) are made when certain critical events occur, such as the starting of a level, when an object is created or destroyed, when the player makes a move, etc. When creating new classes of your own devising, it can sometimes be difficult to determine exactly why something is (or is not) happening. Because there is no "single stepping" of the class codes, this is even more difficult. However, the Message Trace window can be used to show you exactly which messages are being sent to which objects, by whom the messages are sent, what the optional arguments are, and the exact sequencing of those messages. This can be a great help in determining why things are working the way they are (and why they're not working the way you want them to!) For help on message tracing, see Message Tracing.

Class Editing

This is where you can REALLY mess things up.

WARNING: Class editing is a *VERY* complex subject, and is *NOT* supported. In other words, I reserve the right to not answer your questions on this subject. That's not to say that I *won't*. But answering questions and helping people develop new classes of objects could very easily consume *all* of my time, in which case I would go bankrupt and you wouldn't see any new games from me. So, if (after carefully reading all of the following material) you still have unanswered questions, please let me know. But don't be surprised or hurt if my answer is, "I'm sorry, I can't help with that." Class editing is not intended to be done by "computer naive" people, nor is it my intent to teach you how ***not*** to be "computer naive." However, if there are "holes" in my documentation of this subject, I'd like to plug them.

Class Editing: A General Overview

Class Editing: Class Attributes
Class Editing: Class Codes
Class Editing: Class Codes - Constants
Class Editing: Class Codes - Standard Variables
Class Editing: Class Codes - Statements
Class Editing: Class Images

Class Editing: Messages
Class Editing: Messages - User Defined
Class Editing: Messages and the internal code structure
Class Editing: Message Tracing

Class Editing: Problems (Q & A)

Keywords:

- Animate
- Broadcast
- CallSub
- Create
- DelInventory
- Delta
- Destroy
- FlushClass
- FlushObj
- ForEachObjAt
- Goto
- GotoLevel
- HeightAt
- if-else
- IgnoreKey

JumpTo
Key
Level
LevelCount
Link
LocateMe
Move
NewX
NewY
ObjAbove
ObjBelow
ObjBottomAt
ObjClassAt
ObjDir
ObjTopAt
Popup
PopupColor
PopupLoc
PopupMisc
return
Self
SendMessage
SetInventory
Sound
Trace
VolumeAt
WinLevel
XDir
YDir

Class Editing: A General Overview

The Class Editor is entered via the *File-Edit classes* menu item. Once there, you select the class you wish to edit from the combo box on the toolbar.

A class is defined by three things: its *images* (how it appears on the screen), its *attributes* (general or common traits that control the way the game engine causes it to behave and interact with other objects), and its *class codes* which modify the behavior caused by the *attributes* and which add behaviors not available through the attributes alone.

Every class *must* have at least one image, although a class may have many as 128 images. The entire puzzle set is limited to a total of 512 images, so don't get *too* carried away with adding images to a single class. These images can be used to indicate various *states* of the object, or to provide simple animation of the object (using the *Animate* function). The images are edited in the **Image Editor**, which is accessed via the pushbutton with the profile of a face on the main Class Editor toolbar. See Class Editing: Images.

The *attributes* are set in the **Class Attributes dialog**, which is accessible in the Class Editor by either clicking on the pushbutton on the toolbar that looks like a Windows checkbox control (a rectangle with an X in it), or by pulling down the Edit menu and selecting the *Attributes...* item. The attributes are general properties of objects that most objects share, and which are used by the Mesh game engine to provide basic default behavior when objects move on the playfield and/or collide with each other. See Class Editing: Class Attributes.

The Class Attributes can only provide limited flexibility in creating new classes of objects, so they may be overridden and/or supplemented by actual program code. Each class has Class Codes associated with it (which default to nothing), and are accessed via the pushbutton with the {} symbols on the main Class Editor toolbar. These Class Codes are written in a language peculiar to Mesh (and a peculiar language it is). It's like a cross between Basic and C, as seen through a kaleidascope, including many of the worst features of both languages and a few that I thought up myself. These class codes are partially compiled into a "tokenized" or "p-code" stream of 16-bit words, which are then interpreted by the Mesh game engine when the levels are played.

When a level is played, the *class codes* for each object are given control at critical moments known as ***events***, giving the *class codes* an opportunity to modify the behavior of objects of that *class*. The *class codes* are given control by a series of ***messages***, which are sent to the class codes by the game. For example, when you start a level, all of the initial objects for the level are created on the playfield, and then a MSG_INIT message is sent to each and every object that has a MSG_INIT code block in its class codes. When you're creating a new class of objects, you can have the class respond to, or ignore, whichever messages you want. See Class Editing: Class Codes and Class Editing: Messages.

Sometimes you want to create a class that is very similar to an already existing class. This is a job for ***Edit-Clone class***. This menu item will create a new class just like the CURRENT class (images, class codes, and attributes), and then bring up the Attributes dialog for you to enter the new name for the cloned class.

Exporting and Importing classes

Classes, including their attributes, images, and class codes, can be exported to ASCII text files (with a file extension of .MC for Mesh Class), and can be imported from those text files. To export, access the *Edit-Export class...* menu item which will bring up the Export class dialog. In this dialog, you can choose to export the current class, all classes in the current puzzle set, or a set of selected classes. Once you've chosen the class or classes to export, enter the name of the .MC file or select a previously existing one from the list box. The format of the .MC file is undocumented and subject to possible change, but is fairly self-explanatory should you have any need to look at one.

To import classes from a .MC file select the *Edit-Import class...* menu item which brings up the Import class dialog (amazing, hmm?) Select the file you wish from the list box and click on OK, or just double-click on the item in the list box. If none the classes in the file currently exist in your puzzle set, they will be read in without any further prompting. However, if some do duplicate the names of already existing classes, then a Duplicate Class Name dialog will appear, asking whether you wish to Rename the imported class or Replace the previous class with the imported class. Normally this dialog will appear for each class in the .MC file that duplicates an already existing class. However, if you click on the Replace All pushbutton, then no further prompts will appear and all subsequent duplicate classes will automatically be replaced by the imported class.

If you have more than one puzzle set that uses the same classes, and you make changes to the classes in one puzzle set, then Exporting/Importing classes is a handy way of updating the other puzzle sets. The general steps are:

1) Run Mesh.
2) Select the puzzle set with the updated classes (using *File-Select puzzle set)*.
3) Select *File-Edit Classes*.
4) Select *Edit-Export classes*.
5) Select the ***All*** radio button, give it ALL (or a name of your choice) as the file name.
6) Select *File-Return to game,* or click the Return-to-game pushbutton.
7) Select your other puzzle set which you wish to update.
8) Re-enter the Class editor (*File-Edit classes*).
9) Select *Edit-Import classes*.
10) Select ALL.MC (or the file name you used) as the file to import.
11) You will be prompted whether to rename or replace duplicate classes. Click on the ***Replace All*** pushbutton. The mouse cursor will remain an "hourglass" until the import has finished.
12) When all classes have been imported, return to game play or quit the program to

the save changes to your .MB file.

Steps 7 through 12 may be repeated for any other puzzle sets you have that use the same set of classes.

Class Editing: Class Attributes

The *attributes* are set in the **Class Attributes dialog**, which is accessible in the Class Editor by either clicking on the pushbutton on the toolbar that looks like a Windows checkbox control (a rectangle with an X in it), or by pulling down the Edit menu and selecting the *Attributes...* item.

Many of the numeric values that you assign in the Class Attributes dialog are arbitrary. However, they must be consistent with the values assigned to all other classes in the puzzle set, and values should be chosen that are "spread out" over a range, leaving unused "holes" where new objects that you might create could be fitted in if they needed to have values in between those of other classes. For example, if you create a new puzzle set with all new classes, don't give the first five classes Heights of 1,2,3,4,5 (assuming they all have different heights). A better choice would be 1000, 2000, 3000, 4000, and 5000 (or even further apart), as this leaves room for future additions and modifications.

NOTE: In the "Hero" class of objects, all "pushable" objects are given a Climb value of 25 (unless you want it to be greater). This allows for varying "floor" values that still allow you to push objects over them. So, if you create new classes of objects (that are pushable) for use with Hero, it is STRONGLY recommended that you set their Climb to 25 (or greater).

All of these values (except for the Class Name) can be changed at "run time" by the class codes. When this is done, ONLY the attributes for that specific object are changed, NOT for all objects of that class. Hence, although all objects of the same class start out (when they're created at the beginning of a level) with the same class attributes, they may all actually have different class attributes due to changes made by the class codes during the playing of the level. MSG_INIT is a popular time to modify class (and object) attributes, such as CurImage, LastDir, Shovable, Shape, Arrivals, Departures, etc.

Class name: this is the name by which the class is referenced in class codes and elsewhere. To rename a class after it's been created, simply change the name in the Class Attributes dialog. Each class must have a unique class name.

Hardness and **Sharpness**: These two attributes work together to determine whether either of two objects are destroyed when one object pushes against (hits) another object. When object A pushes against object B, object A will be destroyed if object A's Hardness (in the direction it is pushing) is ***less than*** the Sharpness of object B (in the direction from which object A is pushing). The four values assigned to Hardness and Sharpness are arranged by the compass to match the directions in which an object can push on the screen.

Is the Player: This attribute is used internally by the game engine, and it's important that ONLY the object which is controlled by the player have this check box set. Normally, if this checkbox is set, *Receives keys* will also be set. In Hero's Hearts, only

the Hero class has this bit set.

Receives keys: This attribute determines whether the game engine sends MSG_KEY messages to objects of this class when the player presses a key or clicks the playfield with the mouse. Clicking on the playfield with the mouse causes the game engine to send a series of "cursor keys" to the objects which receive keys, attempting to move the "Is the Player" object(s) to the location at which the mouse was clicked. If there is more than one "Is the Player" object on the playfield, then the game engine selects only one of them to try to move to the clicked location, but ALL of the "Is the Player" objects will receive the same sequence of keystrokes trying to move that one object. It's possible that you might want to create a class of objects (other than the "Is the Player" class) which receives keys for doing special functions. This is why these two checkboxes are separate items.

Density: This field determines how objects are "stacked" on the display when multiple objects occupy the same location on the playfield. The "heaviest" objects (those with the largest Density value) are drawn first, followed in decreasing Density order by the "lighter" objects. The values returned by the ObjTopAt(), ObjBottomAt(), ObjAbove(), ObjBelow(), ObjDir(), and ObjClassAt() functions are affected by these Density settings. If two objects occupying the same location have the same Density, then the one that arrived there most recently will be on top.

Volume: The volume is the controlling factor that determine whether object A can move diagonally between objects B and C:

BD
AC

If A if trying to move diagonally to D, then it must be able to squeeze *between* objects B and C. The game engine checks whether A can squeeze between B and C by adding the volumes of A, B and C. The sum must be LESS THAN OR EQUAL TO 10,000 in order for A to be able to squeeze between them.

Weight: The Weight is how heavy an object is.
Height: The Height of an object affects which other objects can or can't climb over it.
Climb: Climb determines how high an object of this class can climb.
Strength: Strength determines whether objects of this class can push other objects, how many at once, and how far.

These four attributes gets us into one of the more complex areas of creating classes, as the game engine (at its core) is all about MOVING objects around, and how they interact with each other. When object A is next to object B and tries to move to where object B is, then A's Strength, Weight, and Climb Height combine with B's Weight, and Height to determine what, if anything, occurs. Object A first subtracts its own Weight from its Strength, and if less than 0 no move occurs. Then, if B is Shovable and has a Weight less than or equal to A's remaining Strength, then B will be shoved, and then A will move to where B was. If B is NOT shoved, then if A has Climb GREATER THAN OR EQUAL TO the Height of B, then A will move on top of B. A whole line of objects

can be pushed at once, but the pushing object must have Strength greater than or equal to the sum of the Weights of ALL of the objects being pushed. A class might need to have Strength even though it never moves itself, if it moves other objects. Examples of this are Ice, Teleports and Rollers.

Temperature: The Mesh game engine does NOTHING with the Temperature attribute. However, it is such a common attribute (Fire and Ice, burning things), that it is provided in the Attributes so that it is available for all Classes and so that new objects can easily interact with old objects with no modifications of the old objects (assuming that they're class codes are properly written to check Temperature rather than checking for Fire, for example). For Hero's Hearts, these temperature ranges have been arbitrarily assigned:

0	Absolute 0, cold as cold can get.
100	Water becomes Ice (equivalent to 0 degrees C, 32 degrees F)
125	"room temperature" (equivalent to 25 degrees C, 77 degrees F)
137	Hero's "body temperature" (equivalent to 37 degrees C, 98.6 degrees F)
200	Water becomes Steam (equivalent to 100 degrees C, 212 degrees F)
400	Paper burns
500	Wood burns
2000	soft metals melt
5000	hard metals melt
10000	ceramic/brick melts
15000	Stone melts

The Hero class code checks the Temperature of objects beneath it, and dies if the Temperature is less than 60 or greater than 154. Most objects have a Temperature of 125 ("room temperature"). Fire has a Temperature of 1000, Balls burn when on an object that is 750 or above, Raft burns at 500, and Balloons burn at 250. Water has a Temperature of 115. Everything else (in the current class "set") has a Temperature of 125.

Misc4:
Misc5:
Misc6:
Misc7: These four variables are also not defined (or used) by the Mesh game engine, but are provided as a means of extending the class attributes. Values may be entered as a decimal number, or using the boxes to the right, individual bits may be toggled on and off.

Misc4 is used by Hero's Hearts on a "bit-by-bit" basis, with the bits having these meanings:

15	This is a Heart (of any type)
14	Rafts (wooden objects) may float across this
13	Creeper may step on this
12	Creeper ignores this object for most tests

11 Worm may step on this
10 Worm ignores this object for most tests
9 Rollers can NOT move this object
8 Teleports can NOT teleport this object
7 Balls STOP when hitting these rounded/slanted objects
6 Hero WON'T step on this object
5 Hero ignores this object for most tests
4 This is a Hero "type" of object
3
2
1
0

If an object uses Misc4[9] to keep Rollers from moving it sometimes, then ANYTIME that object CLEARS Misc4[9] that object MUST immediately check to see if there is a Roller below it and if so immediately do a SendMessage(ObjBelow(self), USR_ARRIVING, 0, 0) to give the Roller a chance to move the object. Otherwise, a tiny window of time happens between the object becoming movable by the Roller and when the Roller actually realizes it CAN (and does) move it, and in that tiny window of time, the Player can move when they shouldn't be able to. This allows "quick reflex" type movements that should be disallowed, because this can also break the "replay" feature. Currently, this only applies to Arrows and Balloons, but any new objects you create which use Misc4[9] need to obey this rule. Also, if you create another type of object which can move other objects, you need to think VERY carefully about this issue, and make sure that the Hero can never move (because of a key press) while your new object is moving other objects.

Many of these bits are intended to be used ONLY as overrides to the normal behavior caused by Height, Climb, Weight, etc. Whenever possible, use those variables to control interractions between objects, and only use these overrides in special circumstances. Bit 4 (Hero "type" of object) is not used by the game engine (which uses the IsPlayer bit in the attributes dialog), but is rather intended for testing by many other classes. The reason for this duplication is that there are places in the internal game engine code where it NEEDS to know which object is THE "player representational" object (which is denoted by the "Is the Player" bit in the Attributes dialog), but it's quite possible to want to create classes of objects which are TREATED as Hero objects by the other object classes, while at the same time they're NOT the "player representational" object (ie, the "control" object). For example, you might want to create a puzzle set containing a whole host of "characters" (people, etc) which are all treated the same by Balls and Arrows and such, but are NOT the "player representational" object (the Hero) so far as the game engine is concerned. They would ALL have the Misc4:Bit4 bit set (and Balls and Arrows and such check THAT bit), but only the True Hero class would have the "Is the Player" box checked in the Attributes dialog.

Misc5, Misc6, and Misc7 are unused by Hero's Hearts.

Arrivals: Some classes of objects need to know when another object has arrived in its neighborhood, so that appropriate "response" can occur. This control lets you specify which nearby squares on the playfield (up to two away from the object itself) this class of object cares about, so far as "arrivals" are concerned. The central square (dark gray colored) represents the location of the object of this class. Anytime an object moves onto a playfield location marked by this classes *Arrivals* control, then this object will receive a MSG_ARRIVE message. See Class Editing: Messages. A black square means this class will receive MSG_ARRIVE messages for that location, a white square means it will not.

Departures: This is very similar to *Arrivals* except that it flags which neighboring playfield locations cause a MSG_DEPART message to be sent when an object moves off of that location. See Class Editing: Messages. A black square means this class will receive MSG_ARRIVE messages for that location, a white square means it will not.

Shovable: Objects can be shoved in four directions: north, south, east, and west. This control determines in which of those four directions objects of this class can be shoved. A black square means they can be shoved in that direction, a white square means they can not.

Shape: Objects can be square, rounded, or slanted. When one object tries to move against another object, the Mesh game engine checks the relative shapes of the two objects, and if their roundness/slantedness matches, the moving object will "slide" to the side around the other object (assuming no pushing or climbing occurs). The shape of an object can be specified individually for each of the four sides. Just left-click on the white square of your choice, and the current settings are shown on the interior of the control.

Comments: You can enter whatever comments you wish into this edit box. However, you are also expected to enter three special fields, marked by [CLASS], [LEVEL], and [GAME] *headings*. The [CLASS] comments are intended only for the use of class designers, and are not used anywhere else. The [LEVEL] comments are displayed in the Level Editor whenever someone asks for "How to use" help on that class of object, and should contain any special requirements that a level designer might need to know about this class, particularly if the class uses Misc1, Misc2, and Misc3 variables for any purpose. The [GAME] comments are displayed during the playing of the game when the player right-clicks on an object on the playfield. This field should contain the name of the class and any helpful hints you care to give the game player about the behavior of the object.

Everything following a *heading* on the same line is ignored. The displayed comment starts at the beginning of the first line AFTER the *heading*.

Class Editing: Class Codes

See also:

Class Editing: A General Overview

Class Editing: Class Attributes
Class Editing: Class Codes - Constants
Class Editing: Class Codes - Standard Variables
Class Editing: Class Codes - Statements
Class Editing: Class Images

Class Editing: Messages
Class Editing: Messages - User Defined
Class Editing: Messages and the internal code structure
Class Editing: Message Tracing

Class Editing: Problems (Q & A)

The class codes are compiled into *tokens*, *pseudo-codes*, or *p-codes*. These p-codes are then interpreted by the Mesh game engine as the level is played. When you enter the Class Code Editor, Mesh "decompiles" the p-codes back into some semblance of the source code that you entered, providing automatic formatting. You have no control over that formatting, and unfortunately for you, it probably uses the style of {} matching that you don't like. Sorry about that. However, what this does mean is that you don't have to be too finicky about YOUR formatting of the source code, as the Mesh class code "parser" doesn't care much about tabs, spaces, or formatting, so long as all of the braces match up and the syntax is correct. When it decompiles, it will be "cleaned up." However, the parser DOES care that a "statement" fits entirely on a single line, unlike C compilers (where you can split a statement over as many lines as you like).

Basic structure of Class Codes

The class codes for a particular class have this structure:

```
VARS {
}
SUBS {
}
MSG_INIT {
}
MSG_DESTROY {
}
```

The VARS and SUBS sections are REQUIRED, and must appear in that order. Everything else is optional. (MSG_INIT and MSG_DESTROY are shown above only for example.) You can have as many or as few MSG blocks as you wish. There are about 20 predefined "system" messages (all starting with MSG_), and you can add

your own user defined messages by clicking on the "envelope" button while in the Class Code Editor. You enter the Class Code Editor by clicking on the "{}" button on the main Class Editor toolbar.

So, if you want a class to do something when a particular message is sent (ie, when a particular event occurs), include a MSG block for it. If not, you don't need to include an empty MSG block, just leave it out entirely.

User variables (VARS)

Each class can have up to 256 user defined variables. These variables are listed in the VARS block, comma separated, and their names are limited to 7 alphanumeric or underscore characters (yes, I know, just like in the Dark Ages). These variables are "double-words", 32 bit unsigned integer values. You can usually treat them as signed values when adding and subtracting, and the class code compiler supports entering negative values, so don't worry about it too much, however when doing "less than" and "greater than" comparisons, remember that there are no negative values, zero is the smallest value there is.

All user variables (VARS) are initialized to zero at the beginning of each level (when the level is first started and whenever it is re-started).

User variables are "global" within the class. All user variables within a class are accessible by all MSG blocks and subroutines within that class. However, they are ONLY accessible by the object itself (each instance of a class gets its own copy of the user variables defined in the class). Object A cannot access the user defined variables in Object B except by sending a user defined message to Object B requesting the value of some variable, and the class codes for Object B must implement a message handler for that user defined message that returns the value of the requested variable. See the SendMessage() function for information on sending user-defined messages.

Subroutines (SUBS)

The SUBS block contains subroutines that are callable (using the CallSub statement) from the MSG blocks, or from other subroutines. SUBS do not take any arguments, nor return any values. Both of these can be accomplished by passing values back and forth through the user defined variables (VARS), which is clumsy but works when needed.

The SUBS code block is automatically terminated with an internal, "invisible" RETURN instruction, so if execution reaches the end of the SUBS block and you don't provide an explicit RETURN, execution will automatically return to the calling code. However, it's good practice to always specify an explicit return in order to avoid bugs caused by later adding another subroutine and forgetting to add a Return for the previous one.

Labels in subroutines and MSG blocks

Entry points to subroutines are marked with a label which consists of up to 7 alphanumeric or underscore characters followed by a colon. This label must appear at

the beginning of a line. If you put two labels together with no code between (ie, the labels are at the same "address" in the code) then all references to the second label will get decompiled as a reference to the first label (ie, the second label is useless, so don't do it). Labels can be used within MSG blocks also, as the target of infamous Goto statements. See the Goto statement for more information on using labels in MSG blocks. All labels within a class are global within that class and must be unique within that class.

MSG blocks

A MSG block starts with the message identifier, such as MSG_INIT or USR_KLUDGE, followed by an open brace { on the same line. That line must then be followed by zero or more lines of valid code statements, followed by a close brace } on a line by itself. Each MSG code block is automatically terminated with an "invisible" RETURN instruction that returns 0 if you don't provide an explicit RETURN. However, it's good practice to always return an explicit value when a MSG is expected to return a value. See Class Editing: Messages and Class Editing: Messages - User Defined for information on which messages do and don't expect return values. See Class Editing: Class Codes - Statements for information on valid statements.

Standard class variables, and pointers to objects

EVERY object has a set of *standard* variables for holding the standard values associated with all objects. These include such things as Class, Xloc, Yloc, Height, ClimbHeight, Weight, LastDir, etc. Standard variables from one object are accessible from another object (regardless of the class of either object), since standard variables are just that: standard across all classes. All you need is a *pointer* to another object, and then you can access its standard variables.

A *pointer* to another object is usually obtained by calling a function that returns a pointer to an object, such as ObjAbove(), ObjBelow(), ObjDir(), etc. When Object A uses the SendMessage() function to send a message to Object B, then Object B's standard variable MsgFrom will be a pointer to Object A. Sometimes MsgArg1 and/or MsgArg2 are used to pass pointers to objects along with the message. Pointers can be assigned to (stored in) user defined variables (VARS). For example, this statement sets the user defined variable *obj* equal to a pointer to the object *below* (looking *into* the display, NOT *down* the display) the object whose class code is executing:

```
obj = ObjBelow(Self)
```

Self is a function with no arguments that always returns a pointer to the current object (the one whose class code is currently executing). *ObjBelow()* is a function which returns a pointer to the object which is below the object whose pointer is passed as the argument to the ObjBelow() function. You could then find out the class of the object below this object by doing this:

```
c = obj.Class
```

You could also have just executed this:

```
c = ObjBelow(Self).Class
```

Generally, any pointer can be used to access the standard variables for the object pointed to by that pointer. Pointers to objects are also used as arguments to many statements and functions, such as SendMessage(), ObjBelow() and all of the other Obj*This* and Obj*That* kinds of functions, Destroy(), etc.

NOTE: Great care should be taken when using pointers, as it's possible to cause a General Protection Fault by passing a NULL or invalid object pointer to some functions, or in referencing some variables. It is your responsibility, as the class code writer, to make sure that this doesn't happen. Generally, most functions that return a pointer to an object can also return a NULL (zero) pointer value if the requested object doesn't exist. Examples of this are ObjBelow(Self) when the "Self" object is the "bottom-most" object at that location, or ObjDir(Self, DirE) when "Self" is at the Eastern-most (right-most) edge of the playfield so that there is NO object to the East of it. The Mesh game engine checks for NULL pointers and generally returns 0 when one is used to reference standard variables. However, there is no efficient way for Mesh to check for invalid, non-zero "pointers". So, for example, the following code (and anything like it that would result in an invalid pointer to a non-existant object) will usually cause a GPF:

```
obj = 1
c = obj.Class
```

because 1 is an invalid pointer to an object (pointers are actually memory addresses of the location of the data storage for the object being referenced). This code will NOT cause a GPF:

```
obj = 0
c = obj.Class
```

Since 0 (NULL) is a special value that is checked for (and often returned) by the Mesh game engine. In this example, "c" would be set to 0 also. ("0" is an invalid Class "number", because Classes are given numbers starting at 1.)

See Class Editing: Class Codes - Standard Variables for a list of the available standard variables, what they represent, and how they're used.

Predefined constants: Keys, Directions, Bits, Animation methods, and Sounds

For ease of programming and readability of the class codes, a number of constants are predefined as keywords. Most possible keys are defined with KEY_xxx constants, where the "xxx" is the name of the key. There are 8 absolute directions and 8 relative directions, 32 BITxx dinitions (with "xx" being a number from 0 to 31), a handful of constants for use with the Animate() function to control how the animation is done, and

a whole passle of SND_xxxx constants where "xxxx" is the name of the predefined (built into the program) sound.

For listings of all of these constants, see Class Editing: Class Codes - Constants.

Comments

Comments may be included within MSG blocks and the SUBS block, but not outside of those blocks. The comment must start with a /* at the beginning of a line and end with a */ at the end of a line. Comments may span multiple lines, but may not be "in line" nor at the end of a line (ie, no partial line comments).

Operators

The available operators are most of the "standard" ones, although there might be slight "precedence" changes from what you're used to. When in doubt, use extra parentheses and the parser/decompiler will remove them if not needed.

The operators, grouped by precedence, are:

- unary minus
~ bitwise NOT
! logical NOT

* multiply
/ divide
% modulo

+ addition
- subtraction
& bitwise AND
| bitwise OR
^ bitwise XOR

<< shift left
>> shift right

== test for equality
!= test for inequality
< test for less than
> test for greater than
<= test for less than or equal to
>= test for greater than or equal to

&& logical AND
|| logical OR
^^ logical XOR

Class Editing: Class Codes - Constants

Numeric constants

Numbers may be specified either in decimal or hex. Hex numbers are specified the same as in the C language, with a preceding "0x" before the hex number. So, decimal 25 would be specified as 0x19. Numbers may be preceded by a minus sign, but remember that most values in Mesh are treated as unsigned quantities, so the minus operator just does a two's complement on the value.

Directional constants

Directions can be specified with these predefined constants:

DirE	0
DirNE	1
DirN	2
DirNW	3
DirW	4
DirSW	5
DirS	6
DirSE	7
DirF	8
DirLF	9
DirL	10
DirLB	11
DirB	12
DirRB	13
DirR	14
DirRF	15

Absolute directions (with North being the top of your display and South being the bottom) are specified according to the compass. Relative directions are specified by Left/Right and Forward/Back, and are relative to the objects current LastDir value. The numbers after the directional constants in the list above are the actual internal numeric values associated with those directions. The absolute direction is computed from the relative by this formula:

absdir = (LastDir + reldir) modulo 8

Key constants

Almost every key on the keyboard has a unique key code, a few of which are usurped by the Mesh game engine, but most of which are received by the object(s) having the "Receives keys" flag set in the Class Attributes dialog. The Key constants available are:

KEY_BACK
KEY_TAB
KEY_ENTER
KEY_SPACE
KEY_PGUP

KEY_PGDN
KEY_END
KEY_HOME
KEY_LEFT
KEY_UP
KEY_RIGHT
KEY_DOWN
KEY_CENTER (5 on keypad when in cursor-pad mode))
KEY_SHIFT
KEY_CTRL
KEY_BREAK
KEY_CAPSLOCK
KEY_NUMLOCK
KEY_SCRLOCK
KEY_SEMICOLON
KEY_EQUALS
KEY_COMMA
KEY_MINUS
KEY_PERIOD
KEY_SLASH
KEY_TILDE
KEY_OBRACKET
KEY_BACKSLASH
KEY_CBRACKET
KEY_QUOTE
KEY_0
KEY_1
KEY_2
KEY_3
KEY_4
KEY_5
KEY_6
KEY_7
KEY_8
KEY_9
KEY_A
KEY_B
KEY_C
KEY_D
KEY_E
KEY_F
KEY_G
KEY_H
KEY_I
KEY_J
KEY_K

KEY_L
KEY_M
KEY_N
KEY_O
KEY_P
KEY_Q
KEY_R
KEY_S
KEY_T
KEY_U
KEY_V
KEY_W
KEY_X
KEY_Y
KEY_Z
KEY_NUMPAD0
KEY_NUMPAD1
KEY_NUMPAD2
KEY_NUMPAD3
KEY_NUMPAD4
KEY_NUMPAD5
KEY_NUMPAD6
KEY_NUMPAD7
KEY_NUMPAD8
KEY_NUMPAD9
KEY_MULTIPLY
KEY_DECIMAL
KEY_DIVIDE
KEY_F9
KEY_F10
KEY_F11
KEY_F12

These keys are NOT available, as they are intercepted by the game engine:

KEY_ESCAPE
KEY_F1
KEY_F2
KEY_F3
KEY_F4
KEY_F5
KEY_F6
KEY_F7
KEY_F8
KEY_ADD
KEY_SUBTRACT

KEY_INSERT
KEY_DELETE

Bitwise constants

It is frequently useful and/or necessary to test, set, or return specific bitwise values. To make this easier, there are 32 predefined keywords that return a value with only one bit set. They are, redundantly:

BIT0
BIT1
BIT2
BIT3
BIT4
BIT5
BIT6
BIT7
BIT8
BIT9
BIT10
BIT11
BIT12
BIT13
BIT14
BIT15
BIT16
BIT17
BIT18
BIT19
BIT20
BIT21
BIT22
BIT23
BIT24
BIT25
BIT26
BIT27
BIT28
BIT29
BIT30
BIT31

Animation constants

There are a handful of "animation constants" that are used to tell the Animate() function what to do. See the Animate() function for more details. They are:

ANI_ONCE
ANI_LOOP
ANI_STOP

ANI_OSC

Sound constants

There are a lot of sounds (digitized audio) built into the Mesh program, and they are played by using the Sound() function. See the Sound() function for additional details. The SND_xxx constants have been given names that poorly try to give an idea as to the nature of the sound. When in doubt, use the Sound dialog in the Class Editor to actually play the sounds, by double clicking on the desired sound in the list box. The constants available are:

SND_SPLASH
SND_POUR
SND_DOOR
SND_GLASS
SND_BANG
SND_UNHH
SND_UH_OH
SND_FROG
SND_THWIT
SND_KLINKK
SND_POWER
SND_KLECK
SND_CLICK
SND_SMALL_POP
SND_DINK
SND_TICK
SND_CHYEW
SND_CHEEP
SND_ANHH
SND_BRRRT
SND_BRRREEET
SND_BWEEP
SND_DRLRLRINK
SND_FFFFTT
SND_WAHOO
SND_YEEHAW
SND_OLDPHONE
SND_RATTLE
SND_BEEDEEP
SND_THMP_thmp
SND_BEDOINGNG
SND_HEARTBEAT
SND_LOCK
SND_TAHTASHH
SND_BOOOM
SND_VACUUM
SND_RATCHET2

SND_DYUPE
SND_UNCORK
SND_BOUNCE
SND_JAYAYAYNG
SND_DEEP_POP
SND_RATCHET1
SND_GLISSANT
SND_BUZZER

Class Editing: Standard Variables

Unless specified otherwise, all standard variables may be written to "at runtime" by the class codes as well as read from. This means that different objects of the same class can actually have different *attributes* depending upon their circumstances and what the class codes do in response to those circumstances. An example of this is Rafts, which set their Height and Shovable flags to zero when pushed into water.

An objects own standard variables are accessed simply by referencing the name of the variable. To access another objects standard variables, an object must have a pointer to that other object, and then follow that pointer reference with a period character and then the name of the standard variable. Examples are:

Class	this objects class
obj.Class	the class of the object pointed to by *obj*
ObjDir(Self, DirL).Class	The class of the topmost object to the left of this object relative to the direction this object is pointing (LastDir)

16-bit unsigned variables:

Class: This is a number representing the class of which this object is an instance. Class numbers start at 1, and are roughly assigned in ascending sequence as you create new classes, but if you delete a class that number will not "be used" until you create another new class. You should never do any "arithmetic" upon the numbers, nor even worry about their actual value (as they could easily change if the classes are exported and then imported into another puzzle set). This standard variable is Read-Only, it can not be written (stored) into. A class of 0 means it's an invalid object (what's returned when you fetch the class from a NULL object pointer).

Xloc and *Yloc*: These two variable always contain the current (x,y) location of the object on the playfield. Locations start with (1,1) in the upper-left of the playfield, and extend to a maximum of (29,21) in the lower-right. These are Read-Only variables. The Move() and/or JumpTo() statements must be used to change the location of an object.

LastDir: Some objects "point" in a particular direction, and some objects move. For those objects, this variable contains the last direction in which the object moved, or the direction in which the object is pointing. When an object moves (with the Move() or JumpTo() statements) the Mesh game engine may use this variable in conjunction with the direction in which the object is being moved (if the new direction was specified as a relative direction rather than an absolute direction). The Mesh game engine ALWAYS sets this variable when an object is moved. If an object doesn't move, then you can use this variable to mean whatever you want (for example, what direction a "rotating"

object is pointing). If an object calls Move() to try to move in a given direction, and the move fails (for whatever reason), the LastDir variable will still be set to that new direction in which the move attempt was made.

Distance: Each the player moves (presses a key) the Distance variables for all objects on the playfield are set to 0. Any time an object moves, the *larger* of the x-displacement *or* the y-displacement is added to that objects Distance variable. So, if an object moves normally (one space in any of eight possible directions), then Distance will be incremented by 1 each time it moves. However, if an object is moved using the JumpTo() function, then the difference between the old and new x-coordinates is compared to the difference between the old and new y-coordinates, and the larger absolute value of those two differences is added to Distance. Objects can use this as a simple mechanism for knowing if an object has moved this turn or not, and if so, how far (relatively).

CurImage: Contains the current image number that is being displayed for this object. Image numbers start at 0 and go up sequentially through however many images there are for this class. CurImage may be set directly (through an "=" assignment), and can also get set by the Mesh game engine if an Animate() is active for this object. If an Animate() is active, you can still assign a value to CurImage, causing the current image to be changed, but it will not stop the animation (although it might mess it up in an unintentional fashion). Use Animate(ANI_STOP...) or Animate(ANI_ONCE...) to stop an animation. A frequent technique is (for classes that have a concept of pointing in different directions and a different image for each direction) to have a MSG_INIT code block and use the CurImage value to initialize the LastDir variable within the MSG_INIT code block. That way the Level Designer doesn't have to worry about setting the LastDir value everytime they place another object of this class into the level, they can just select the image they want.

Inertia: a not very important variable, this is used *during* a Move() statement call to keep track of the amount of remaining Strength this object has while moving itself and pushing other objects. When the Move() completes, this variable contains any remaining left-over Strength, which value is never used by the game engine, but might be of use to you, the Class Designer, somehow.

Density: This determines how objects are "stacked" on the display when multiple objects occupy the same location on the playfield. The "heaviest" objects (those with the largest Density value) are drawn first, followed in decreasing Density order by the "lighter" objects. The values returned by the ObjTopAt(), ObjBottomAt(), ObjAbove(), ObjBelow(), ObjDir(), and ObjClassAt() functions are affected by these Density settings. If two objects occupying the same location have the same Density, then the one that arrived there most recently will be on

top.

Volume: The volume is one of the controlling factors that determine whether object A can move diagonally between objects B and C. If (in the following diagram) A if trying to move diagonally to D, then it must either be able to climb OVER object B or object C (see *Height* and *Climb Height* below) OR it must be able to squeeze *between* them. The game engine checks whether A can squeeze between B and C by adding the volumes of A, B and C. The sum must be LESS THAN OR EQUAL TO 10,000 in order for A to be able to squeeze between them.

B D
A C

Weight: The Weight is how heavy an object is.

Height: The Height of an object affects which other objects can or can't climb over it.

Climb: Climb Height determines how high an object of this class can climb.

Strength: Strength determines whether objects of this class can push other objects, how many at once, and how far.

When object A is next to object B and tries to move to where object B is, then A's Strength, Weight, and Climb Height combine with B's Weight, and Height to determine what, if anything, occurs. Object A first subtracts its own Weight from its Strength, and if less than 0 no move occurs. Then, if B is Shovable and has a Weight less than or equal to A's remaining Strength, then B will be shoved, and then A will move to where B was. If B is NOT shoved, then if A has a Climb Height GREATER THAN OR EQUAL TO the Height of B, then A will move on top of B. A whole line of objects can be pushed at once, but the pushing object must have Strength greater than or equal to the sum of the Weights of ALL of the objects being pushed. A class might need to have Strength even though it never moves itself, if it moves other objects. Examples of this are Ice, Teleports and Rollers.

HardE:
HardN:
HardW:
HardS: These four variables contain the hardness values for this object in the four compass directions (east, north, west, and south). They work together with the Sharpness of another object to determine whether this object is destroyed or not when hit by another object. If this objects Hardness (in the direction in which it "contacts" another object) is less than the sharpness of the other object in the direction in which it contacts this object.

SharpE:
SharpN:
SharpW:
SharpS: These four variables contain the sharpness values for this object in the

four compass directions (east, north, west, and south). See the Hard descriptions above for details.

ShapeE:

ShapeN:

ShapeW:

ShapeS: Objects can be square, rounded, or slanted. When one object tries to move against another object, the Mesh game engine checks the relative shapes of the two objects, and if their roundness/slantedness matches, the moving object will "slide" to the side around the other object (assuming no pushing or climbing occurs). The shape of an object can be specified individually for each of the four sides. These variables have values from 0 to 3:

- 0 flat
- 1 slanted to the left (when "facing" that side)
- 2 slanted to the right (when "facing" that side)
- 3 rounded (or slanted both left and right, or pointed)

Shape: This is a single variable that is sets and returns all four of the individual Shape values at once. The four shapes are packed into the lower byte (hence, Shape always has a value of 0 to 255), with the individuals bits meaning:

- 0-1 ShapeE
- 2-3 ShapeN
- 4-5 ShapeW
- 6-7 ShapeS

Shovable: Objects can be shoved in four directions: north, south, east, and west. This variable determines in which of those four directions objects of this class can be shoved. Only four of the bits are used:

- 0 Shovable to the East (from the West)
- 1 reserved
- 2 Shovable to the North (from the South)
- 3 reserved
- 4 Shovable to the West (from the East)
- 5 reserved
- 6 Shovable to the South (from the North)
- 7 reserved

Misc1:

Misc2:

Misc3: These three variables are not defined (or used) by the Mesh game engine itself, but are provided as a means of extending the functionality for Class Designers and Level Designers. You (the Class Designer) may want to create a class which has different attributes for different objects on the level, specified at level creation time by the Level Designer (such as UserNotes and what message is displayed, or Rollers and Teleports and whether they're active or not at the start of a level). These variables always return the values that were set by the Level Designer when the object was added to the level.

Misc4:

Misc5:

Misc6:

Misc7: These four variables are not defined (or used) by the Mesh game engine, but are provided as a means of extending the class attributes. These may be used by your new classes however you wish, and the default values for them (for any given class) are set in the *Class Attributes* dialog. Hero's Hearts uses most of the bits in Misc4, but Misc5, Misc6, and Misc7 are available for use by your classes. If you create a whole new set of classes that doesn't include the Hero's Hearts classes, then Misc4 is also available for your use. See Class Editing: Class Attributes for information on how Hero's Hearts uses Misc4.

Temperature: The Mesh game engine does NOTHING with the Temperature attribute. However, it is such a common attribute (Fire and Ice, burning things, etc), that it is provided in the Attributes so that it is available for all Classes and so that new objects can easily interact with old objects with no modifications of the old objects (assuming that they're class codes are properly written to check Temperature rather than checking for Fire, for example). For information on how Hero's Hearts uses these temperature ranges see Class Editing: Class Attributes.

Msg: (READ-ONLY) the current message that's being processed. Usually not of much use, but possibly useful if a subroutine is called by multiple MSG blocks, and the subroutine cares which message it was called by, perhaps.

32-bit unsigned variables:

Arrived: The lower 25 bits are individual flags indicating the location(s) where an object has arrived since the last MSG_ARRIVED message was sent to this object. Anytime a MSG_ARRIVED message is sent, *at least* one bit will be set in this variable. Multiple bits may be set if more than one object has arrived since the last MSG_ARRIVED message was sent. The MSG_ARRIVED message is not guaranteed to be sent while the object that arrived is still there (it may "move on" or get destroyed before that happens). The individual bits refer to these locations, relative to the location of *this* object, where *this* object is located in the center (bit position 12):

4	3	2	1	0
9	8	7	6	5
14	13	**12**	11	10
19	18	17	16	15
24	23	22	21	20

Departed: The lower 25 bits are individual flags indicating the location(s) from which an object has departed since the last MSG_DEPARTED message was sent to this object. Anytime a MSG_DEPARTED message is sent, *at least* one bit will be set in this variable. Multiple bits may be set if more than one object has departed since the last MSG_DEPARTED message was sent. The individual bits refer to the same locations as shown above for *Arrived.*

Arrivals: Same bit settings as with *Arrived*, except that these are the locations where this object "cares" about arrivals. Only those locations with bits set will cause the Mesh game engine to send a MSG_ARRIVED to this object when an object arrives at that location. This variable gets set by default to the settings

specified in the Class Attributes dialog for this class, but the values may be changed at runtime.

Departures: Same bit settings as with *Deaprted*, except that these are the locations where this object "cares" about departures. Only those locations with bits set will cause the Mesh game engine to send a MSG_DEPARTED to this object when an object departs from that location. This variable gets set by default to the settings specified in the Class Attributes dialog for this class, but the values may be changed at runtime. See the Arrow class in Hero's Hearts for an example of this.

Msg: the current message being executed by the class codes. Normally, you won't need to refer to this variable, but if you have a SUBS subroutine that is called from more than one message code block, then it's possible that something in that subroutine might want to behave differently depending upon the message being executed. This is how it can know.

MsgFrom: (READ-ONLY) a pointer to the object which sent the current message to this object. For most "system" (MSG_xxxx) messages, this is NULL. However, for some system messages, it's set to a useful value (such as with MSG_HIT and MSG_HITBY). However, for user-defined messages sent with SendMessage() or Broadcast(), MsgFrom *always* points to the object that executed the SendMessage() or Broadcast().

MsgArg1: (READ-ONLY)

MsgArg2: (READ-ONLY) Each message has two arguments which may or may not have pertinent values. These are retrieved via these two variables. The meaning of these two variables is dependant upon the message.

Boolean (0 or 1) variables:

Storing any non-zero value into these variables cause them to be set to 1, store zero into them to clear them to 0. (duh!)

Busy: setting this flag prevents the player from making any moves. Be VERY cautious when implementing classes that use this flag, as if the flag accidentally gets "stuck" in the set condition, the playing of the level will be locked up. This flag is used, for example, by the Heart White class to keep the player from moving while the "white heart to red heart" animation runs its course.

Destroyed: (READ ONLY) When an object is destroyed, it doesn't immediately get completely erased, but rather an internal flag gets set that says "this object has been destroyed", and most operations ignore all objects with this flag set. The "destroyed" objects are cleaned up later on at "opportune moments" in the operation of the game engine. An object, as its class codes are executing, might get destroyed. Execution of that objects class codes will continue, just as if the object had not been destroyed, until the end of the current message block within which the destruction occurred. On rare ocassions, it is important for an object to know whether it has been destroyed during some function call. Testing this bit is the way.

Invisible: Objects with this flag set are not drawn on the display. They continue to exist and interact with the other objects on the playfield in EVERY other way, they

simply aren't drawn on the display, so they are invisible to the player.

IsPlayer: (READ ONLY) is non-zero if the "Is the Player" is checked in the Class Attributes box, otherwise returns 0.

Stealthy: Objects with the Stealthy flag set don't cause MSG_ARRIVED or MSG_DEPARTED messages when it moves. For example, the Worm class code checks this flag on the player object before moving towards it. So, if the player object is Stealthy, it can walk right past Worms with impunity, the Worm just sits there. However, as long as this flag is set, no *other* "arrive/depart" actions happen either, so the player object can't pick up Red Hearts while it's stealthy, nor can it teleport, etc.

KeyCleared: a user-defined flag that is automatically cleared (in ALL objects) every time the player presses a key that gets sent to the "receives keys" object. If an object needs to do something once per turn, but not during the MSG_BEGIN_TURN message, this is a way to know if its been done. Just set this flag when the action has been done, and the flag will automatically be cleared when the next key is pressed. The Mesh game engine does nothing with this flag other than to clear it on key presses.

UserSignal: a user-defined flag that prevents the player from making moves as long as it's set to non-zero. The Mesh game engine does nothing with this flag other than to observe it before making moves.

UserState: a user-defined flag. The Mesh game engine does nothing with this flag, and the player CAN make moves regardless of whether it's set or cleared. This flag is a hold-over from early in the design, before the Misc4-Misc7 variables were added, which somewhat make this flag needless.

VisualOnly: when set, tells the game engine that this object is there purely for the players enjoyment (visually) and has NO effect on the other objects on the playfield. An object with this flag set will still receive most of the normal messages and will be returned from the ObjClassAt() function (if correct class is specified), but will NOT be returned by ObjTopAt, ObjBottomAt, ObjAbove, ObjBelow, etc, and the Move() function will NOT pay ANY attention to it when moving other objects. Other objects can move onto, through, and around as if it wasn't even there. There will be no MSG_HITS, etc, and no shoving. This flag should ONLY be set by an object when it TRULY will never again have any effect upon any other objects on the playfield. Otherwise, the replay feature of Mesh will be broken. An example of using this flag is Fire when it is extinguished and turns into "steam" for a brief time.

Global variables

The following variables are global to the Mesh game engine. There is only one copy of them. They are not associated with any given class or object.

AltImage: (READ-ONLY, 16-bits) this variable is incremented by pressing the F6 key and decremented by pressing the F7 key. It is used in Hero's Hearts to control which Hero image is displayed.

PuzzleSetNumber: (READ-ONLY, 32-bits) this variable is not used by the Mesh game engine, but is set according to the puzzle number from the puzzle set when

the puzzle set is loaded. This is most often used for keeping track of which puzzle set is loaded in a multi-puzzle set game. (ie, each puzzle set will have a unique puzzle set number which can be tested by class codes to determine which puzzle set is loaded.) However, it may be used for whatever you wish in YOUR puzzle sets.

MoveNumber: (READ-ONLY, 16-bits) this variable returns the number of moves that have so far been made on the current level. Starts at 0 until the first move is made, then goes to 1, etc. Get the idea? The primary purpose for which this is intended is for classes of objects that want to execute some behavior every N moves. This saves them from having to maintain their own counter. AVOID USING THIS FOR EXTREMELY OBSCURE BEHAVIOR, such as, "If you don't do exactly THIS on exactly the 13th move, you die." That's not a puzzle, at least not a good one. A GOOD puzzle allows the player to deduce or figure out the solution from what they see on the display, not "hidden" information that is unavailable to them.

GlobalBool: (READ-ONLY, Boolean) a user-defined flag. F8 toggles the state of this BOOLEAN value. It's saved in each players configuration file and is not used for anything currently, but is intended for future possible use of some global "feature" which DOES NOT AFFECT THE STATE OF GAME PLAY. Pressing F8 is NOT recorded in the "move list", so if changing the state of this boolean variable affects game play, it will BREAK the "replay" feature of the game. It is intended for "cosmetic" use ONLY (ala AltImage).

ExplainDeath: (READ-ONLY, Boolean) Set by the "*Options-Popup explanation of players destruction*" menu item. Checked by the Hero class codes to determine whether or not to display a popup window showing the object that killed the Hero.

Class Editing: Class Codes - Statements

MSG and SUBS code blocks consist of a series of zero or more *statements*. Each statement must be completely contained on a single line (with the exception of the *if-else* and *ForEachObjAt* multi-line constructs which are surrounded with {}'s). A statement may consist of any one of the following:

a blank line
code_label:
standard_variable = numeric_expression
objref.standard_variable = numeric expression
user_var = numeric expression
Animate(...)
Broadcast(...)
CallSub code_label
Create(...)
DelInventory(...)
Destroy(...)
FlushClass(...)
FlushObj(...)
ForEachObjAt(...) {}
Goto code_label
GotoLevel(...)
if(numeric_expression) statement
else statement
IgnoreKey()
JumpTo(...)
Link(...)
LocateMe()
Move(...)
Popup(...)
PopupColor(...)
PopupLoc(...)
PopupMisc()
return
return numeric_expression
SendMessage(...)
SetInventory(...)
Sound(...)
Trace(...)
WinLevel()

Blank lines are discarded by the parser. A code_label must be no more than 7 alphanumeric or underscore characters. The (...)'s merely imply arguments to those function calls.

Class Editing: Class Images

Overview

The Image Editor allows you to create and modify the images used to represent each class. It's accessed via the pushbutton with the profile of a face on the Class Editor toolbar. All images come in three sizes, to allow reasonable support of lots of different video resolutions without the images looking really cruddy on some resolutions. You MUST provide all three resolutions for ALL images. The sizes are 18, 24, and 32 pixels square. Each new class automatically gets one "new" image (and each class must have at least one image, you cannot delete the last image in a class). You can add addtional images with the "New" button on the toolbar.

Across the top is the *Image List Window* which shows all of the images for the current class in all three sizes. The images are "numbered" from 0 on up, left to right (this is useful to know, for when you're changing images in the Class Code or setting up animation sequences). There's a maximum limit of 128 images per class, and a limit of 512 images per puzzle set. ("One image" includes all three sizes of images, so if you look at it from that point of view, one class has a limit of 3*128 images, and the puzzle set has a limit of 3*512 images.) Beneath the three sizes of each image is a checkbox which controls whether or not the image will appear in the Level Editors "object selection" dialog. Images with this box checked WILL appear in the Level Editor. There is no requirements regarding how many or how few images appear in the Level Editor. For example, the HeroDead class doesn't appear at all, while all of the Arrow images appear, and only some of the Hero images appear. If you don't want a Level Designer to be able to place a particular image in a level, then be sure to uncheck the checkbox for that image. The currently selected image (ie, the one currently being edited in the Zoom window) is outlined with a black square.

On the left beneath the Image List Window" is the *Color Palette*. These are all of the colors available to you for creating images in the Mesh game engine. It is a fixed color palette, you can't add or change any of the colors. The purple bar across the top is the *transparent* color. Any pixels of this color in the image will be transparent, they will allow the pixels "beneath" them on the playfield to show through. The currently selected color is shown with a white outline and also in the the large rectangle beneath the Color Palette.

Below the Color Palette, the currently selected image is shown "full-sized" and "double-sized". This image is updated in "real time" as you edit the image in the Zoom. The image in the Image List Window retains its original state until you select another image, at which point the contents of the Zoom are copied back into the Image List Window. The images beneath the Color Palette allow you to see what your changes look like and compare them to the original (in the Image List Window).

To the right of the Color Palette is the Zoom window. This is where the editing actually takes place. Using the left mouse button on the Zoom will activate the currently

selected tool (from the toolbar). Clicking the right mouse button on the Zoom will sample the color from the pixel, causing that color to become the selected color in the Color Palette. (There is also a Color Sample tool on the toolbar, the pushbutton with the "eye-dropper" on it. However, it's rarely used, since the right button is so much more convenient.)

The Toolbar

The buttons on the toolbar are:

These are the "OK" and "Cancel" buttons. Selecting either of these causes the program to exit the Image Editor, returning to the Class Editor. Clicking on the OK button causes the changes to the images to be copied back into the class and saved (when you exit the class editor or exit the program). Be patient. This can take a few seconds, depending upon the speed of your machine, the color depth of the display, and how many images you have. Clicking on the "Cancel" (recycle) button causes the changes to be discarded. NOTE: this only discards changes to the images that existed when you entered the Image Editor, and does NOT include added images nor deleted images. When you add new images or delete images, those changes take effect IMMEDIATELY, and clicking the Cancel button will not undo those changes.


This rotates the image counter-clockwise. If an area is selected, only the selected area is rotated, otherwise the entire image is rotated. If the selected area is square, the image is rotated 90 degrees. If the image is not square (it's rectangular) then the image is rotated 180 degrees. This function is very useful when creating multiple images for a class that show the objects orientation or direction. Draw the first image, then add new images, copy the first one (see **Copying and Moving Images** below) to the new one, then rotate and/or flip the image to the desired orientation.


These flip the image either vertically or horizontally. If an area is marked, just the marked area is flipped. Otherwise, the entire image is flipped.


These are the tool selection buttons. They are:
Mark Area: holding the left mouse button down draws a dotted rectangle around a group of pixels. This "marked" group of pixels may then be copied (using CTRL-C or Edit-Copy), moved (by holding down the left mouse button within the marked rectangle and dragging it to its new location), rotated, or flipped. When pasting an image (using CTRL-V or Edit-Paste) if no area is marked, the image is pasted "full-sized" (1 pixel = 1 pixel). If an area is marked, however, then the pasted image is either stretched or compressed to fit within that marked area. This is a good way to get a "first pass" at the different sized images. When creating a new image, first draw one size of image, then copy it and (using Mark Area to mark the ENTIRE destination image) paste it into the larger or smaller images. It won't look perfect, but it

will give you something to start from rather than having to start all over from scratch. Of course, depending upon the nature of your image, sometimes it's easier to start each image from scratch than it is to clean up the stretched image.

Pencil: allows you to change one pixel at a time by clicking the left mouse button, or to draw free hand by holding down the left mouse button and dragging the mouse about.

Spray: like pencil, except it draw a pattern of pixels all at once.

Line: draws a line. Point at one end point of the line and hold down the left mouse button. Then drag the mouse to the other end-point of the line and release the left mouse button.

Rectangle Outline: Just like Line except it draws the outline of a rectangle.

Rectangle Solid-filled: Just like Rectangle Outline except that the interior of the rectangle is filled.

Oval Outline: I'll bet you've figured out already that this one is just like the Rectangle Outline except that it's an oval instead of a rectangle.

Oval Solid-filled: You're so smart that I don't think I even need to explain this one.

Color Replace: When you want to change all pixels in an image from their current color to a new color, select this tool, then select the new color from the Color Palette, then point at a pixel on the Zoom that is the old color which you wish to replace with the new color, and then click the left mouse button. Be careful though. If you have one area that's red and a second area that's blue, and you replace the red color with the exact shade of the second area's blue, then both areas will be the same color and will no longer be differentiatable (some word, huh?) by future color replacements.

Color Sample: This tool, combined with left mouse button clicks on the Zoom, achieves the same thing as right clicking at any time on the Zoom. It changes the currently selected color in the Color Palette to match the color of the pixel upon which you click.

 These four buttons control the size of your drawing tool, from one to four pixels wide.

 This is the Undo button (also available as CTRL-Z). Up to 50 levels of undo are available.

 Clicking on this button causes a new image to be added to the current class. (If you need to delete an image, that function is available under the Image Editors *Edit* menu.)

Copying and Moving Images

You can rearrange the order of the images in the class simply by pointing at the image you wish to move (in the Image List Window at the top), holding down the LEFT mouse button, and then dragging the image left or right to the destination location. When you release the mouse button, the other images will be shifted left or right as is appropriate and the moved image will be inserted at that point. While you're dragging the image to its new location, the letter 'M' appears on the image to remind you that you are Moving the image.

You can copy all three sizes of one image to all three sizes of another image (losing the original contents of those second images) simply by pointing at the image you wish to copy (in the Image List Window at the top), holding down the RIGHT mouse button, and then dragging the image left or right to the destination location. When you release the mouse button, the copied images will replace the original images at the destination. While you're dragging the image to its new location, the letter 'C' appears on the image to remind you that you are Copying the image.

Importing Images From Elsewhere

You can import images from outside of the Mesh program only by copying them to the clipboard (in another program) and then pasting them in the Mesh Image Editor.

If you have one or more images in one class of your puzzle set that you'd like to copy to another (either to use "as is" or to modify), the *Edit-Import image from class...* menu item is just your ticket. This allows you to select a class from your puzzle set, and then select one or all of the images from that class to ADD to the current class.

Message Tracing

What is TRACING, why is it useful, and how does it work?

Sometimes when developing a new class of objects, things get ***too*** complex, and you can't figure out ***what*** is happening (let alone ***when*** or ***why***)! There's a tool available under the main File menu called *Message trace window*. This brings up a window which can display the messages that are sent to objects during the playing of a level. The information recorded about each message consists of (in the order as their columns appear in the trace window):

the trace message number (just a meaningless, sequential number)
FromPtr (the value of MsgFrom)
FromClass (the class of the MsgFrom object)
X, Y (the Xloc,Yloc values for the MsgFrom)
ToPtr (the value of ***Self***, the pointer to the object receiving the message)
ToClass (the class of the object receiving the message)
X, Y (the Xloc, Yloc values for the object receiving the message)
Message (uh...the message numeric value and name, yeah, that's it)
MsgArg1 (if you can't figure this one out, you probably shouldn't use Trace)
MsgArg2

The reason for showing the FromPtr and ToPtr is that these values are sometimes passed as MsgArg1 or MsgArg2, and so they can be valuable in figuring out the meaning of those arguments. The trace message number is displayed in decimal, as are the X and Y locations. All other numbers in the trace window are displayed in hex.

When message tracing is turned on the game engine records in the trace buffer each message that meets the trace specification (more about that later). If tracing is turned on, the messages are recorded in the buffer whether the trace window is being displayed or not. When the buffer fills, it wraps around to the beginning and continues recording messages from the start of the buffer again, overwriting the earlier ones. The location in the trace buffer at which the system is about to write the next message is always shown with a black bar. You can not modify the information which is saved regarding each message, nor can you modify how it is displayed. One size fits all.

You *can*, however, control which messages are recorded in the trace buffer. This is done in the *Trace specification* dialog available under the Options menu of the trace window. This dialog allows you to ***exclude*** all messages *from* objects of selected classes, all messages *to* objects of selected classes, and/or selected messages from or to *any* object. Additionally, it lets you specify that NO messages are to be recorded until the player is at (and/or after) a specific move number. It also lets you specify that once the StartMove move number has been reached, that the first SkipFirst messages which meet all other criteria are not to be recorded. Finally, it lets you specify that, once StartMove and SkipFirst have been satisfied, only KeepNext messages are to be recorded, and then message recording ceases. (Oh, you can also tell it to ignore

"unreceived messages", which is explained below, and to have the trace buffer automatically cleared at the start of the level.) In other words, when a message is sent, the game engine decides whether to record the message in the trace buffer by first checking these criteria:

1) Is tracing turned on?
2) Is
Ignore unreceived messages NOT checked
OR
MsgTo != NULL
and MsgTo is not destroyed
and there is a message code block for this message in this class??
3) Is the player move number >= StartMove number?
4) Has SkipFirst messages already been skipped?
5) Is KeepNext == -1 or have we recorded fewer than KeepNext messages?
6) Is MsgFrom==NULL or a class of object that is NOT *excluded*?
7) Is MsgTo a class of object that is NOT *excluded*?
8) Is Message a message that is NOT *excluded*?

If the answer to ALL of the above questions is YES, then the message will be recorded in the trace buffer. If ANY of the answers to the above questions is NO, then the message will NOT be recorded in the trace buffer.

Tracing is turned ON (and OFF) under the Options menu. (That's a tough one.)

Ignore unreceived messages can be enabled in the Trace Specification dialog. A message is considered *unreceived* if the target of the message (the object to which it is being sent) is NULL (ie, it doesn't exist), or it's been destroyed (ie, its internal DESTROYED flag is set, but it hasn't been quite flushed from the system such as when an object executes a Destroy(self) function call), or if the target objects class doesn't have a message code block for the message being sent.

Classes and messages may be included and excluded individually by selecting the desired item in the appropriate list box of the Trace Specification dialog and then pressing the SpaceBar or clicking on the Include or Exclude pushbutton, or by merely double-clicking on the desired item. There are pushbuttons for including or excluding ALL items in the list box at once. Items which are *included* are displayed as green text, while items which are *excluded* are displayed as red text.

Under the trace window Options menu is the *Clear buffer* command, which ...uh... clears the trace buffer.

Also under the Options menu is *Size of trace buffer...* which brings up a dialog to let you specify how many thousands of entries you want in your trace buffer. Each entry takes 32 bytes of memory, so each count in the *Size of trace buffer...* dialog causes 32000

bytes to be reserved for recording message traces. The acceptable range of values for this setting are from 1 to 65 inclusive (requiring from 32,000 bytes up to almost 2 Mbytes). The only reason not to set this to a large value is if your system is short on memory. If the program is unable to allocate the buffer, an error message will appear. However, operation of the program may slow tremendously on systems without a lot of memory, as Windows will be swapping things back and forth to disk a lot (if some of the Mesh program data is too much to fit into memory and has to be virtualized to disk). If that seems to be the case, try setting the trace buffer size to 1, and if things improve, then nudge it up a ways until performance degrades again. Then, back it off so that performance improves again. On most systems with 16 Mbytes of RAM or more, this shouldn't be an issue, but it MIGHT be a problem on 8 Mbyte systems. (Remember, there's LOTS of other data kept in RAM, including object images, a copy of the screen, class codes and data, levels, etc. It all adds up. A megabyte here, a megabyte there, pretty soon you're talking *real* memory!)

Under the File menu is *Print* which ...uh... prints the contents of the trace buffer (assuming that you have a printer connected to your computer and it's turned on).

How to use TRACING most effectively

A ***lot*** of messages are sent during the playing of a single level, which can make it difficult to find the message sequences in which you're interested. Careful use of the Trace Specification can "weed out" a lot of the superfluous (ain't that a wonderful word?) messages. For example, you will often want to have *Ignore unreceived messages* checked, since MSG_INIT and MSG_POSTINIT are sent to EVERY object at the start of a level, and MSG_PLAYERMOVED is sent to EVERY object each time the player hits a key (or moves with the mouse), and MSG_PLAYERMOVING is sent to EVERY object each time the player object (the Hero) moves to a different location. Many objects don't care about those messages, so they can be safely ignored. ***However, if you enable the Ignore unreceived messages option, it is VERY important to remember that it's been enabled, or you may be confused by not seeing a message sent that really WAS sent.***

Now, I lied in the previous paragraph. MSG_INIT, MSG_POSTINIT, MSG_PLAYERMOVED, and other "broadcast" messages are NOT sent to EVERY object. Objects are grouped internally into two ...uh... groups, depending upon whether their class has ANY message block codes or not. Obviously, if a class has NO class code at all (other than empty VAR and SUBS blocks), then NO messages sent to objects of that class will have ANY effect, so why send them? Hence, when a message is "broadcast" it is ONLY sent to the objects in the "have class code" group. That's why you never see MSG_INIT and MSG_POSTINIT sent to "Field" objects or some other classes of objects.

Another thing to do to pare down the list of messages you have to look through is to *exclude* MSG_LASTIMAGE, MSG_PLAYERMOVED, and MSG_PLAYERMOVING

messages (again, *unless* they're critical to your particular problem you're trying to debug). If you DO need to see some of these messages, then you may be able to get rid of some of them by excluding some classes. For example, if your new class uses MSG_LASTIMAGE to do some things (that aren't working), so you need to see MSG_LASTIMAGE, but your level has a lot of animated objects that keep getting MSG_LASTIMAGE sent to them, any of those classes that aren't involved in the problem you're working on can safely be excluded (from the FROM and TO class lists). The other way to approach it is to exclude ALL classes EXCEPT the ones involved in the problem you're working on. ***Again, be VERY aware of the fact that you're hiding potentially important information from yourself!*** The fact that some aspect of the class you're working on is not working means that ***you don't fully understand the problem***. Because you don't fully understand the problem (it wouldn't be a problem if you did!) the problem may actually involve other objects or classes that you don't expect. So be careful (and very aware) as you exclude classes and messages from your traces.

If it's still too difficult to capture a trace that has the information you need, try creating a special TEST level that is as simple as possible, yet still exhibits the problem you're working on. This will minimize the number of messages sent, and thus simplify your task.

There's an old C programmers saying: **Never underestimate the power of printf() for debugging.** In other words, putting "print" statements at key places in your code to show the state of the program and its data can be a BIG help in debugging code. Mesh's Popup() function can sometimes be used for this, as can SetInventory(). However, only one Popup() window can be open at a time, so its use is limited. SetInventory() is similarly limited, in that only five items can be added to the inventory at a time, and all you get is a single image and a two-digit number. Message tracing can be used as a stand-in for "printf()". You can use the ***Trace()*** function to cause a dummy "message" to be entered into the trace buffer. The "To object" will be the object executing the Trace() function. The "From object" will be the value of the first argument of the Trace() function (which ***must*** be a valid object reference or 0, or a GPF will occur). The "message" value will be 0xFFFF, and will list as " *** TRACE ***". The second and third arguments to the Trace() function will be displayed as the values for MsgArg1 and MsgArg2. The arguments to the Trace() function have no other meaning (besides the fact that the first argument must be a valid object reference) than what meaning you assign to them.

Class Editing: Messages

When a level is played, the *class codes* for each object are given control at critical moments known as ***events***, giving the *class codes* an opportunity to modify the behavior of objects of that *class*. The *class codes* are given control by a series of ***messages***, which are sent to the class codes by the game. For example, when you start a level, all of the initial objects for the level are created on the playfield, and then a MSG_INIT message is sent to each and every object that has a MSG_INIT code block in its class codes. When you're creating a new class of objects, you can have the class respond to, or ignore, whichever messages you want.

The game predefines a set of *system messages*. Additionally, the class developer (that's you) can add additional messages which are sent by one object to itself or (most commonly) to other objects. These additional messages are called *user messages*, and they are created within the class code editor using the "envelope" pushbutton on the toolbar or the *New message...* item under the *Edit* menu. The *user messages* can be up to 31 characters in length, and may contain most characters except for space. They do not have to start with MSG_, that's merely the convention that the game system uses for its internal *system messages*. As a matter of fact, it's recommended that you DON'T use MSG_ as the prefix, but rather USR_ or some other prefix of your choice, or none at all.

Some system messages have *return values* that control the behavior of the game engine, but many of the system messages have no return value (although the *return* statement always requires a return value, even if the message doesn't require one). Whether the return value from *user defined* messages are used or not is up to you and how you design your class codes and user defined messages.

If a message is sent to an object and there is no corresponding message handler in that objects class for that message, then a value of 0 is automatically returned by the game engine as the return value from that message..

NOTE: Class codes *can* use the SendMessage() and the Broadcast() functions to send *system messages* to other objects, but this is not normally done, and should only be done under extreme circumstances when you can think of NO other way of accomplishing a particular task, as unforseen behavior may occur. Normally, only the game system should send *system messages.*

MSG_INIT - When initializing a level (for the player to start playing) *all* objects are first created on the playfield with no messages being sent. Once all objects have been created and linked into the playfield, then MSG_INIT is sent to *all* objects. The order in which objects receive the MSG_INIT is dependant upon where they are on the playfield and the sequence in which they were added to the playfield, and is too complex to easily predict. Therefore, do not depend upon one object always receiving MSG_INIT before or after another. If objects of your class need to do any initialization of user VARS, or send any "initialization" messages to other objects, it should be done in MSG_INIT (or MSG_POSTINIT). An example of this in Hero is when all Hearts

(regardless of color) broadcast a USR_HEART_PLUS message to all Exits. When an Exit receives the USR_HEART_PLUS message, it increments its *hearts* user variable, thus counting how many Hearts are on the level. (All user VARS are initialized to 0 when an object is created.) MsgFrom, MsgArg1, and MsgArg2 are always 0 for MSG_INIT. The return value is ignored.

MSG_POSTINIT
Ocassionally, you may need to send a message to another object at initialization time, but you don't want to send it until *after* the other message has been initialized. Therefore, once the game engine has initialized the level, immediately after sending the MSG_INIT message, it broadcasts MSG_POSTINIT to all objects on the playfield giving them a second chance at initialization after ALL objects have already had a first chance. The return value is ignored. MsgFrom, MsgArg1, and MsgArg2 are 0.

MSG_CREATE - Whenever an object is created (during the execution of the Create() function, *not* during the initial setup of the level), a MSG_CREATE message is sent to it after it has been created and linked into the playfield. MSG_INIT and MSG_CREATE serve very similar purposes, but they let you differentiate between an object that is created at the very beginning of a level and one that's created later during the playing of the level via a Create() function call made by another object. MsgFrom is a reference to the object which executed the Create() function, while MsgArg1 and MsgArg2 are always 0. The return value is ignored.

MSG_CREATED - Whenever an object is created (*except* the initial objects at the start of a level), a MSG_CREATED message is sent to all objects who have an ARRIVE flag set for the location in which the object was created. An example of when you might use this is if something causes Water to be created where a Rock is, then the Rock will want to know about it so that it can "sink" into the Water. Another example is that a Spider object might create Cobweb objects that only last for a short time, but are deadly to the Hero (or other "creatures"), and if a Cobweb gets created on top of the Hero (or another creature) then the Hero (or other creature) will want to know about it so it can die. MsgFrom is set to the object which was created, so you can get its location as (MsgFrom.Xloc, MsgFrom.Yloc), but MsgArg1 and MsgArg2 are also set to the new objects x,y. The return value is ignored. MSG_SUNK and MSG_FLOATED may also be sent to appropriate objects after the MSG_CREATE if the newly created objects Density caused it to sink below the top of the stack of objects.

MSG_DESTROY - Just before an object is destroyed, MSG_DESTROY is sent to the object. If the object returns 0 (the default value) from the MSG_DESTROY message, then the object is destroyed. If it returns non-zero, then the object is NOT destroyed. MsgFrom is always a reference to the object causing the destruction. MsgArg1 and MsgArg2 are always 0. An object is only destroyed in one of two ways: because of a call to the Destroy() function or because its Hardness is less than the Sharpness of an object which has moved against it. The return value is used as the return value of the Destroy() function.

MSG_DESTROYED - When an object is destroyed, a MSG_DESTROY is sent to that object, and then a MSG_DESTROYED is sent to all objects which have a DEPART flag set for the location at which the object was destroyed. The destroyed objects data structure (variables) are still usable during the MSG_DESTROYED message, so you can determine the location as (MsgFrom.Xloc, MsgFrom.Yloc), since MsgFrom contains a reference to the destroyed object. MsgArg1 and MsgArg2 are always 0. The return value is ignored.

MSG_PLAYERMOVING - Immediately before the Player object moves (whether because the player has pressed a key or because other objects are causing the Player object to continue to move afterwards), a MSG_MOVING is sent to the Player object. If non-zero is returned, the move is aborted. Otherwise, MSG_PLAYERMOVING is then sent to *all* objects (including the Player object). If *any* object returns non-zero for this message, the Player object will NOT move, and no further objects receive the MSG_PLAYERMOVING message. MsgFrom is a reference to the Player object which is about to move, and MsgArg1 is the X-location and MsgArg2 is the Y-location to which the Player object is about to move. The Player objects *current* location, before the move, is (MsgFrom.Xloc, MsgFrom.Yloc). The Player object will also receive its own MSG_MOVED message after it has moved.

This message allows an object to "track" the Player object (move with it) and to keep the Player object from moving (trapped).

MSG_BEGIN_TURN - This message is *only* sent after the player has pressed a key. It is sent regardless of whether the Hero actually moves in response to the key. It is this message that should be used to trigger the motion or action of objects which move every time the player does. MsgFrom is always a reference to the last object found with the "Is Player" flag set, and (MsgArg1, MsgArg2) is the (x,y) location of that object. The return value is ignored.

MSG_END_TURN - This message is sent to all objects when there are NO objects with pending actions (like Move, Arrive, Depart, etc). An internal "turn count" variable is set to 0 when MSG_BEGIN_TURN is sent. That "turn count" variable is passed as MsgArg1 of the MSG_END_TURN message. After each time MSG_END_TURN is broadcast to all objects, the internal "turn count" variable is incremented. Therefore, class codes can determine whether a particular MSG_END_TURN message is the first one after a MSG_BEGIN_TURN by testing for MsgArg1==0. MsgFrom and MsgArg2 are always 0. If any objects take actions during MSG_END_TURN processing which cause additional actions (like Move, Arrive, Depart, etc), then the player will be prevented from moving until THOSE actions settle down again, at which point another MSG_END_TURN will be broadcast. The player will be able to make another move *only* when there are NO pending actions AFTER the broadcast of the MSG_END_TURN message. If something is done during the processing of the MSG_END_TURN code block that should keep the player from making another move right away, then the MSG_END_TURN code block should return a non-zero value. If it's okay for the player to move (so far as THIS object is concerned) then the

MSG_END_TURN code block should return 0 (the default if there is no code block for this message). (The system also checks internal flags at return from this message for Busy, UserSignal, whether the object moved or jumped during the MSG_END_TURN, and whether something arrived or departed, and in those cases also disallows a player move until those flags have been resolved.)

NOTE: Once a class returns 0 for MSG_END_TURN, then it MUST NOT execute any Move(), JumpTo(), Create(), Destroy(), or other "game state" changing functions within the MSG_END_TURN code until MSG_END_TURN is called again with MsgArg1 equal to 0 (ie, until after the player has made another move). If you don't observe this rule, you will BREAK the "replayability" aspect of the game, because that means the player can make moves while your object is still finishing its previous move. Be VERY CAREFUL WITH THIS!!!

MSG_ARRIVED - This message is sent when an object arrives at a location which has been flagged in this objects Class Attributes as being sensitive to arrivals. The object will *usually* still be there when the MSG_ARRIVED is sent, but this is not guaranteed, as the object may have gotten destroyed as a result of its move to that location or, under some circumstances, it may have already moved on to a new location. MsgFrom, MsgArg1, and MsgArg2 are always 0. The Arrived variable will have bits set indicating the locations that had an arrival, but if multiple objects arrive only one MSG_ARRIVED is sent, regardless of how many objects might have arrived at the same time. The return value is ignored.

MSG_DEPARTED - This message is sent when an object departs from a location which has been flagged in this objects Class Attributes as being sensitive to departures. MsgFrom, MsgArg1, and MsgArg2 are always 0. The locations which have had a departure will have bits set in the Departed variable, but if multiple departures happen only one MSG_DEPARTED is sent, regardless of how many objects might have departed at the same time. The return value is ignored.

MSG_LASTIMAGE - An object can execute the Animate(ANI_ONCE...) function to show an animated image consisting of a sequence of the objects images. When the time comes to move to the next image, and the last image in the sequence is already displayed, then the animation stops (with that last image as the current image), and a MSG_LASTIMAGE is sent to the object. This can be used to start another animation sequence or execute any other code (except for calls to the Move() and JumpTo() fucntions). For example, the White Heart, when pushed into Fire, sets its BUSY flag to keep the player from moving, then starts an animation sequence with the Animate() function, showing it turn into a Red Heart. When the animation sequence finishes, the White Heart receives the MSG_LASTIMAGE message, at which point it clears its BUSY flag, destroys itself and the Fire, and uses Create() to create a Red Heart in its place. MsgFrom, MsgArg1, and MsgArg2 are always 0. The return value is ignored.

NOTE: The game does not allow Move() and JumpTo() functions to be executed while a MSG_LASTIMAGE message is being processed (no matter how deeply "buried" in

SendMessages() and CallSubs). An error message will appear during game play, and the Move() or JumpTo() function will do nothing. This is done because MSG_LASTIMAGE can happen at irregular times (ie, it's not 100% repeatable) and that could break the "replay" feature of the game. (See the following WARNING.)

WARNING: This game is designed to be non-reflex dependant (it should be just as playable by someone with only one finger as by a twitching, hyper-active action-fiend. In most instances, the game enforces this, there's nothing you can do about it. However, the MSG_LASTIMAGE is one place where this might break down, and you (the class designer) have the opportunity to break this aspect of the game. RESIST THE URGE. If, during your MSG_LASTIMAGE code you execute code which changes the state of the game (when the game is otherwise doing nothing but waiting for the player to press a key), then you'll break the "replay" mechanism. This is STRONGLY discouraged. Examples of things NOT to do during MSG_LASTIMAGE is calls to functions which change the states of objects, such as, Create(), Destroy(), etc. An exception to this is when the MSG_LASTIMAGE is called while your objects BUSY flag is set (which prevents the player from moving anyway). An example of this is when the White Heart is pushed into the Fire. It sets its BUSY flag at that point to keep the player from moving, then starts an animation sequence. When the animation sequence finishes, the White Heart receives its MSG_LASTIMAGE message, at which point it clears its BUSY flag, destroys the Fire beneath it, destroys itself, and creates a Red Heart in its place. The setting and clearing of the BUSY flag enables us to force the MSG_LASTIMAGE to occur within a setting wherein it's safe to modify the state of the game. (In other words, the setting of the BUSY flag during the White Hearts HEARTWHITE_TURN_RED user message, which gets sent to it by the Fire during the Fire's MSG_ARRIVE code, effectively extends the period of time covered by the players move to include the duration of the White Hearts animation. This makes the White Hearts MSG_LASTIMAGE message become part of the response to the player pushing the White Heart into the Fire.) However, be VERY cautious setting and clearing the BUSY flag, as you may find the player "locked out" and unable to move if the flag doesn't get cleared in a timely fashion.

MSG_MOVING - When an object is definitely about to move (either via a call to the Move() or JumpTo() functions or because it was *shoved* by another object), then the MSG_MOVING message is sent *immediately* before the move. This gives you an opportunity to do something *before* an object moves. (See MSG_MOVED if you need to do something *after* an object moves.) MsgFrom is always a reference to the object which caused the move, and (MsgArg1, MsgArg2) is the (x, y) location to which the object is about to move. (Xloc, Yloc) is the location from which the object is about to move. If you wish to allow the move to happen, return 0 for this message (0 is returned automatically for you if your class doesn't have a MSG_MOVING code block). If your class returns a non-zero value, then the move is aborted and the object stays where it is.

MSG_MOVED - When an object has moved from one location to another (either via a call to the Move() function by itself or another object, or because it was *shoved* by

another object as that second object tried to move), then the MSG_MOVED message is sent. This message is sent *after* the object is already located at the new location. (See MSG_MOVING if you need to do something *before* an object moves.) MsgFrom, MsgArg1, and MsgArg2 are always 0. The return value is ignored.

MSG_JUMPED - There is a JumpTo() function which moves an object to a given (x, y) location with no regard to the intervening locations, nor to the objects at the new location, nor their height. This is used by your class codes under special circumstances to move an object to a location to which it would not normally move using the Move() function (or to *jump* it to a location more than one location away, since Move() always moves only one location at a time). Immediately *before* the jump occurs, a MSG_MOVING is sent to the object (see MSG_MOVING). After an object has been *jumped*, the MSG_JUMPED message is sent to it. MsgFrom is always 0. (MsgArg1, MsgArg2) is the location from which the object has just jumped. The return value is ignored.

MSG_KEY - When the player presses a key, a MSG_KEY message is sent to all objects whose class has the "Receives keys" flag set. MsgArg2 is 0 for the first object receiving the key. The return value from each objects MSG_KEY code block then gets passed to the next object receiving keys as its MsgArg2. This return value is not used by the game itself, but if you have multiple classes that receive keys, you may want the first one to be able to tell the second or third one that the key has been handled or not. This return value is a mechanism for this signal. The value of the key that was pressed by the player is fetched with the ***Key*** function, and is also in MsgArg1. The game system reserves a few keys for itself which never cause a MSG_KEY message to be sent. These are F1, F2, F3, F4, F5, F6, F7, F8, Ins, Del, ESC, and the plus and minus keys on the numeric keypad. All other keys cause a MSG_KEY to be sent. The key will also be automatically entered into the *move list* (on the toolbar) for *replaying* purposes ***unless*** a "key receiving" object executes the IgnoreKey() function, which causes the game system to *not* enter the key into the move list. If the IgnoreKey() or Flush() functions are executed during the MSG_KEY message, then the MSG_PLAYERMOVED message will ***not*** be sent. MSG_KEY is also sent to an object that has created a Popup() or PopupMisc() window that contains \qX "quiz" formatting. See Popup().

NOTE: Do *not* use the *IgnoreKey()* function to ignore keys that have changed the state of the game, or you will break the ability to replay the moves!

MSG_SUNK - When an object is moved from one location to another, the *Density* of each object (declared in the Class Attributes dialog) controls the *stacking order* of the objects (which ones are above and which ones are below the others). When the object arrives at the new location, it starts at the "top of the stack" of objects by default. Then, if its density relative to the others causes it to sink down into the stack, a MSG_SUNK is sent to it and a MSG_FLOATED is sent to the ones above it. This has nothing to do with sinking or floating in a liquid, merely the changing of the order in which objects are drawn on the screen. The objects with the highest Density values are drawn first (on

the bottom of the stack) and the ones with the smallest Density values are drawn last (on the top of the stack). ObjAbove() and ObjBelow() functions can be used to find the objects which are above and below a given object at a given location. MsgFrom, MsgArg1, and MsgArg2 are all 0. The return value is ignored. MSG_SUNK and MSG_FLOATED may also be sent when an object is created, if its Density causes it to sink below the top of the stack of objects.

MSG_FLOATED - (See MSG_SUNK, except that MsgFrom is a reference to the object which caused the MSG_FLOATED by sinking beneath this object) The return value is ignored.

MSG_HIT - When object A tries to move against object B, and object B is too tall for object A to climb on top of, then JUST before the check to see if Sharpness/Hardness causes either of the objects to be destroyed, a MSG_HIT is sent to object A with MsgFrom set to object B and then a MSG_HITBY is sent to object B with MsgFrom set to object A. If object B gets shoved (moved) by object A, then object B will receive a MSG_HITBY almost immediately followed by a MSG_MOVING, eventually followed by a MSG_MOVED. If either object gets destroyed by the "hit", then it will receive a MSG_DESTROY almost immediately after receiving the MSG_HIT or MSG_HITBY. (Actually, if the shoving of object B causes object B to shove one or more other objects, then MSG_HIT and MSG_MOVING will be sent to those other objects in between the MSG_HIT and MSG_MOVING that are sent to object B.

When moving diagonally, object A may "hit" any of the three objects that are being "pushed" against (one in the diagonal direction, and one orthogonally to either side; for example, if object A is trying to move NE and there are objects located N and E from object A, then object A can't move NE unless NE is shorter than what object A can climb, AND the sum of objects N, E, and A's volumes are less than or equal to 10,000 (so that A can "squeeze between" N and E); if object A can't move NE because object N and object E are blocking it, then two MSG_HITs will be sent to object A, one each for objects N and E, but none for object NE, and a MSG_HITBY will be sent to each of objects N and E. If object A is NOT blocked by objects N and E, then a MSG_HIT is sent to object A with MsgFrom set to object NE, and a MSG_HITBY is sent to object NE with MsgFrom set to object A.

(Simple, yes?) The return value is 0 to continue as normal with the move and all checks. Otherwise, the individual bits in the return value cause these effects:

bit	effect
0	Don't send the MSG_HITBY message to this object
1	Don't do the Sharpness/Hardness check with this object
2	Don't try Shoving this object
3	Abort the move attempt immediately
4	Don't send MSG_HITBY to any further objects at this location.
5	Don't do the Sharpness/Hardness check with any further objects at this location.
6	Don't try Shoving any further objects at this location.

7 reserved, must be set to 0.
8 Send the MSG_HITBY and then abort the move attempt.
9 Send the MSG_HITBY and do the Sharpness/Hardness check, then abort the move attempt.
10 Send MSG_HITBY, do Sharpness/Hardness, try shoving, but then abort the move attempt.
11 reserved, must be set to 0.
12 reserved, must be set to 0.
13 reserved, must be set to 0.
14 reserved, must be set to 0.
15 reserved, must be set to 0.

MSG_HITBY - (See MSG_HIT.) Obviously, the condition of bits 0, 8, 9, and 10 in the return value have no meaning. Otherwise, the other bits have the same effect as with MSG_HIT.

See also: Messages and the internal code structure.

Messages are sent at two different times: immediately after something happens (INTERRUPT messages) and during the system "idle" loop (POLLED messages). These are the INTERRUPT messages:

MSG_DESTROY
MSG_CREATE
MSG_JUMPED
MSG_MOVING
MSG_SUNK
MSG_FLOATED
MSG_PLAYERMOVING
MSG_HIT
MSG_HITBY
MSG_DESTROYED
MSG_CREATED

These are the POLLED messages:

MSG_MOVED
MSG_ARRIVED
MSG_DEPARTED
MSG_KEY

These messages are effectively BOTH polled AND interrupt, because they're sent from within the "idle" loop, which is also where the corresponding action takes place.

MSG_BEGIN_TURN

MSG_LASTIMAGE
MSG_END_TURN
MSG_INIT
MSG_POSTINIT

Class Editing: Messages - User Defined

User defined messages allow you to expand the built-in functionality of the class codes, the messages they send, and the messages to which they respond. These messages are defined while editing class codes. There are two pushbuttons on the toolbar, both with the picture of an envelope (suggesting a "message"). The one without the red X is for creating (defining) a new message name. The message name may be any combination of alphanumeric characters or the underscore character. Try to be consistent in your message naming conventions, and try to choose a "generic" message name when it might be usable by more than one class for a similar function. For examples, see below.

There's a maximum limit of 1024 unique messages in any puzzle set, including the 20 or so "system" (predefined) messages.

The envelope pushbutton with the red X is for deleting the current message. If you delete a message, make sure that you first remove all references to it in the class codes. Otherwise, the game engine won't be able to properly decompile the class codes the next time you try to edit a class that references that (no longer existant) message. Also, internally the messages are assigned a number. When you delete a message, that number is freed for use by the next new message you create. If you delete a message without first removing all references to it in the class codes, then create a new message which gets assigned the old messages number, then when you edit the class code which still referred to the old message, it will decompile as the new message instead. This will be very confusing, so try to avoid it by properly deleting all references to a message before deleting the message.

The user defined messages currently defined are:

USR_HEART_PLUS: Broadcast at MSG_INIT/MSG_CREATE time by all Heart-type objects to all Exit objects, so that the Exits know how many Hearts are on the level.

USR_HEART_MINUS: Broadcast at MSG_DESTROY time by all Heart-type objects to all Exit objects, so that the Exits know how many Hearts are on the level, and thus know when to change to a "green exit."

USR_HEARTRED: Sent to a non-HeartRed heart object (like HeartBlue and HeartWhite) when something has happened that should cause that heart to turn into a red heart, such as a Creeper stepping on a HeartBlue, or a HeartWhite being pushed into Fire.

USR_BURN: Sent to Fire when a burnable object has been pushed into it, to cause the Fire to initiate its "smoking burning" animation sequence.

USR_SPLASH: Sent to Water when a "sinkable" object (like Rock, Ball, or Extinguisher) has been pushed into the Water, to cause the Water to initiate its "splash" animation sequence.

USR_DEC_INVENTORY:
USR_INC_INVENTORY:
These messages are used to tell the Hero class to add or subtract items from the

inventory, currently only Yellow and Blue keys. MsgArg1 is the Class of the item, and MsgArg2 is the number to add or subtract.

USR_DISABLE:

USR_ENABLE:

USR_TOGGLE: These three are used for various purposes by different classes. For example, Bridge objects will extend themselves when receiving USR_ENABLE, and retract when receiving USR_DISABLE (with MsgArg1 being the direction to extend/retract). Teleports and Rollers turn themselves off and on when receiving USR_DISABLE and USR_ENABLE. All of these "toggle" their state when receiving USR_TOGGLE.

USR_ARRIVING: Sent by a Teleport to another "target" Teleport, when an object has "arrived" at the first Teleport and is about to be jumped to the target Teleport. This is necessary because when the object arrives at the target Teleport it will trigger that Teleports "arrival" flags, causing a MSG_ARRIVED to be sent to the target Teleport. This would normally cause the "jumped object" to be passed on to any successive Teleports, but we only want one jump to occur at a time. To prevent the "chain reaction" jumping through a series of Teleports, the target Teleport has to know that an object is arriving from another Teleport. So, the sending Teleport first sends a USR_ARRIVING message to the target, with MsgArg1 as a reference to the object that is about to jump there. The target Teleport saves this object reference in a local user VAR, and then when the MSG_ARRIVED occurs, it compares the arriving object to the reference stored in the local user VAR. If it matches, nothing is done. If it doesn't match, then the object really is arriving at its first Teleport, and a jump occurs.

USR_SET_IMAGE: In order to keep the images for all Rollers "synchronized," one Roller is the "master" and all others are "slaved" to the master. The master Roller uses Animate() to time the progression of its image animation, and when each image is done and it's time to change to the next image, the master Roller receives MSG_LASTIMAGE. When this happens, it Broadcasts a USR_SET_IMAGE to all Rollers. ONLY the master is animating its image with Animate(), all other Rollers depend upon the USR_SET_IMAGE from the master to know when to change their images. Each Roller has a *master* VAR. In MSG_INIT, if *master* is 0 then the Roller knows that it's the first Roller to get initialized, and it Broadcasts a USR_SET_IMAGE message. All Rollers receive this (including the master) and set their *master* variable to MsgFrom.

USR_CALL_MASTER: If the master Roller is destroyed (by some class of objects added to the game), then all of the slave Rollers would stop being animated. So, in MSG_DESTROY the Roller checks to see if it's the master, and if so Broadcasts USR_CALL_MASTER to all other Rollers, with MsgArg1 referring to the master Roller. All Rollers respond to that message by sending a USR_AM_SLAVE message back to the master (MsgArg1). The master Roller stores the reference to the slave (MsgFrom) in a local VAR. Hence, when control returns to the master Roller after the Broadcast(USR_CALL_MASTER), the local VAR will be set as a reference to the last slave Roller to respond to the USR_CALL_MASTER (if any). The master Roller then sends a USR_BECOME_MASTER message to that Roller, making it the new master.

And the animation keeps on Rollering.

USR_AM_SLAVE: Sent by a slave object to its master object in response to the master object broadcasting a USR_CALL_MASTER to its slave objects.

USR_BECOME_MASTER: Sent by a master object to one of its slaves to make the slave the new master object.

Class Editing: Messages and the internal code structure

It's a little easier to understand the flow of the system messages, and how best to use them in your class codes, if you know what the internal code of the game engine is doing. Below are some "pseudo-code" listings of some of the sections which are critical to message sending. In this pseudo-code, when a "SendMessage()" is executed, MsgFrom is NULL, but when it's shown as "*SomeObj*.SendMessage()", then MsgFrom is a pointer to the *SomeObj* object.

MOVE

Move() is the most complex of the functions. Be prepared for mind-numb.

Internally, when Move() is called, the game engine gives the object being moved an Inertia equal to the Strength of the object doing the pushing (itself, if it's calling Move() on itself). One of the first things Move() does is subtract from its Inertia the Weight of the object that is being moved. The remaining Inertia is what is available for pushing objects that are in the way. Hence, if object A is going to push object B, then object A must have Strength >= (WeightA+WeightB). Object B's Strength doesn't come into play here, since it's not doing any pushing (even if Object B ends up trying to push an object C which is in the way, object B's Strength is still not used, ONLY object A's, because object B is being ACTED UPON by object A and is not voluntarily moving itself). In this second case, then object A's Strength must be >= (WeightA+WeightB+WeightC) or NOTHING will move.

ObjF is the topmost object in the *Forward* direction (dir), the object which *obj* is about to shove against or move on top of.

ObjLF and *ObjRF* are the objects next to and on either side of *ObjF*. ObjLF and ObjRF are named with respect to the direction in which *obj* is moving. When moving horizontally or vertically, this example shows the relationships:

ABC
RST
XYZ

If S is attempting to move to where T is, then T is *ObjF*, C is *ObjLF*, Z is *ObjRF*, B is *ObjL* and Y is *ObjR*. If S is attempting to move to where Y is, then Z is *ObjLF*, X is *ObjRF*, T is *ObjL*, and R is *ObjR*.

When moving diagonally, *ObjLF* and *ObjRF* are the objects next to and on either side of *ObjF*. For example, if we have four objects arranged thusly:

AB
CD

and C is attempting to move to where B is, then C's *dir* is DirNE, object B is the *ObjF*, object A is the *ObjLF*, and object D is the *ObjRF*. For the move to occur, C must be able to:

climb over B
AND squeeze between A and D

Height and Climb are used to determine if C can climb over B, and the combined

volumes (sum) of A, D, and C must be <= 10000 in order for C to squeeze between them.

As a second example, if B is attempting to move to C, then C is the *ObjF*, D is the *ObjLF*, and A is the *ObjRF*.

With all of that in mind:

```
BOOL Move(obj, dir) {
        if( obj is 0 ) goto movefailed
        Resolve dir from possible relative direction to an absolute direction
        obj.LastDir = dir
        if( obj.Weight > obj.Inertia ) goto movefailed
        obj.Inertia = obj.Inertia - obj.Weight
restart:
        if( dir is N, S, W, or E ) {
                HitVal = 0
                for( each object at ObjF's location ) {
                        if( EachObj.Height > 0 ) {
                                HitVal = HitVal & 0xFFF8
                                HitVal = HitVal | EachObj.SendMessage(obj, MSG_HIT,
EachObj.Xloc, EachObj.Yloc)
                                if( HitVal & 8 ) goto movefailed
                                if( 0==(HitVal & 0x11) ) HitVal = HitVal |
obj.SendMessage(EachObj, MSG_HITBY, obj.Xloc, obj.Yloc)
                                if( HitVal & 0x0108 ) goto movefailed
                                if( 0==(HitVal & 0x22) ) {
                                        if( obj.Sharpness > EachObj.Hardness ) {
                                                if( 0==obj.Destroy(EachObj) ) HitVal = HitVal |
0x8004
                                        }
                                        if( obj.Hardness < EachObj.Sharpness ) {
                                                if( 0==EachObj.Destroy(obj) ) HitVal = HitVal |
0x4C
                                        }
                                }
                                if( HitVal & 0x0200 ) goto movefailed
                        }
```

Here's the tricky part where one object pushes another object, and the pushing objects remaining Inertia is passed on to the pushed object as its Inertia. Notice that this is a RECURSIVE call to Move(), so the pushed object will subtract out its Weight from the Inertia, and then may pass that reduced Inertia on to another object in its way. As each object returns, if the move was successful, then that remaining Inertia (if any) is stored back into the "pushing objects" Inertia, thus reducing its remaining Inertia. The game engine does nothing with this remaining Inertia, but an objects class codes might.

```
                        if( 0==(HitVal & 0x44)
```

```
                        AND EachObj is shovable
                        AND obj.Inertia >= EachObj.Weight ) {
                          EachObj.Inertia = obj.Inertia
                          if( Move(EachObj, dir) ) {
                                if( EachObj has not been destroyed ) obj.Inertia =
EachObj.Inertia
                                HitVal = HitVal | 0x8000
                          }
                    }
                    if( HitVal & 0x0400 ) goto movefailed
              }
              if( HitVal & 8 ) goto movefailed
              if( HitVal & 0x8000 ) goto restart
              if( ObjF.Height <= obj.Climb ) {
                    if( 0==MoveTo(obj, ObjF.Xloc, ObjF.Yloc) ) goto movefailed
movesucceeded:
                    return 1
              }
              if( ObjRF is NOT 0,
                    AND obj can be SHOVED to the RIGHT (in relation to dir),
                    AND the shapes of obj and ObjF are rounded/sloped to allow
"sliding",
                    AND ObjRF.Height <= obj.Climb,
                    AND (ObjR.Height <= obj.Climb OR
(ObjR.Volume+ObjF.Volume+obj.Volume <= 10000)) ) {
                    for( each object at ObjRF's location ) {
                          do most of the same hit/hitby/sharpness checking as above,
but no shoving
                          if move fails for any reason, goto TryOtherSide
                    }
                    if( MoveTo(obj, ObjRF.Xloc, ObjRF.Yloc) ) goto movesucceeded
              }
TryOtherSide:
              if( ObjLF is NOT 0,
                    AND obj can be SHOVED to the LEFT (in relation to dir),
                    AND the shapes of obj and ObjF are rounded/sloped to allow
"sliding",
                    AND ObjLF.Height <= obj.Climb,
                    AND (ObjL.Height <= obj.Climb OR
(ObjL.Volume+ObjF.Volume+obj.Volume <= 10000)) ) {
                    for( each object at ObjLF's location ) {
                          do most of the same hit/hitby/sharpness checking as above,
but no shoving
                          if move fails for any reason goto movefailed
                    }
                    if( MoveTo(obj, ObjLF.Xloc, ObjLF.Yloc) ) goto movesucceeded
```

```
                }
        } else {
                if( ObjLF.Volume+ObjRF.Volume+obj.Volume <= 10000 ) {
                        if( 0 != ObjF ) {
                                for( EachObj at ObjF's location ) {
                                        if( EachObj.Height > 0 ) {
                                                HitVal = EachObj.SendMessage(obj,
MSG_HIT, EachObj.Xloc, EachObj.Yloc)
                                                if( HitVal & 8 ) goto movefailed
                                                if( (HitVal & 0x11)==0 )
obj.SendMessage(EachObj, MSG_HITBY, obj.Xloc, obj.Yloc)
                                                if( HitVal & 8 ) goto movefailed
                                        }
                                }
                                if( ObjF.Height <= obj.Climb ) {
                                        if( MoveTo(obj, ObjF.Xloc, ObjF.Yloc) ) goto
movesucceeded
                                }
                        }
                } else {
                        for( each object at ObjLF's location ) {
                                if( EachObj.Height > 0 ) {
                                        HitVal = EachObj.SendMessage(obj, MSG_HIT,
EachObj.Xloc, EachObj.Yloc)
                                        if( (HitVal & 1)==0 ) obj.SendMessage(EachObj,
MSG_HITBY, obj.Xloc, obj.Yloc)
                                }
                        }
                        for( each object at ObjRF's location ) {
                                if( EachObj.Height > 0 ) {
                                        HitVal = EachObj.SendMessage(obj, MSG_HIT,
EachObj.Xloc, EachObj.Yloc))
                                        if( (HitVal & 1)==0 ) obj.SendMessage(EachObj,
MSG_HITBY, EachObj.Xloc, EachObj.Yloc)
                                }
                        }
                }
        }
movefailed:
        obj.Inertia = 0
        return 0
}
```

MOVETO

The MoveTo() function is used by both the Move() and JumpTo() functions. It's purpose

is to move an object from one location to another with the initial MSG_MOVING as the only check to keep it from happening.

```
BOOL MoveTo(obj, newx, newy) {
        if( inside MSG_LASTIMAGE processing ) {
                display error message
                return FALSE
        }
        if( SendMessage(obj, MSG_MOVING, newx, newy) ) return 0
        if( obj Is_Player ) {
                for(all objects ) if( obj.SendMessage(EachObj, MSG_PLAYERMOVING,
newx, newy) ) return 0
        }

        unlink obj from current PlayField location
        DrawField(obj.Xloc, obj.Yloc)

        set internal OF_DEPART flags for objects-that-care around obj's old location
```

This next line is what causes the *Distance* standard variable to keep track of how far an object has moved since the last key press. (All objects *Distance* variable get set to 0 when a key press is sent to the "key receiver" objects.)

```
        obj.Distance += max(abs(newx-obj.Xloc), abs(newy-obj.Yloc))

        obj.Xloc = newx
        obj.Yloc = newy

        set internal OF_ARRIVE flags for objects-that-care around obj's new location

        link obj into new PlayField location, inserting into stack of objects already there
according to Density.

        if( any objects above obj at new location ) {
                SendMessage(obj, MSG_SUNK, 0, 0)
                for( all objects above obj ) obj.SendMessage(EachObj, MSG_FLOATED,
0, 0)
        }

        DrawField(obj.Xloc, obj.Yloc)

        set internal OF_MOVED flag for obj

        return 1;
}
```

NOTE: The internal OF_MOVED, OF_DEPART, and OF_ARRIVE flags are checked in the main idle loop, and the MSG_MOVED, MSG_DEPARTED, and MSG_ARRIVED messages are sent at that time.

JUMPTO

```
void JumpTo(obj, newx, newy) {
        if( (newx, newy) is within the limits of the playfield ) {
                oldx = obj.Xloc
                oldy = obj.Yloc
                if( MoveTo(obj, newx, newy) ) SendMessage(obj, MSG_JUMPED, oldx,
oldy)
        }
}
```

DESTROY

Destroy() is pretty simple and straight-forward. Note that the *obj*'s variables are still valid until we return to the "system" and garbage collection starts. Hence, an object can call Destroy() on itself and still continue to execute code and reference it's own variables afterwards *for the duration of the message block from within which it called Destroy() on itself*.

```
void Destroy(obj) {
        if( 0==SendMessage(obj, MSG_DESTROY, 0, 0) ) {
                set internal OF_DESTROYED flag for later "garbage collection"
                DrawField(obj.Xloc, obj.Yloc)
                for( any objects with DEPART flag set at the destroyed objects location ) {
                        obj.SendMessage(EachObj, MSG_DESTROYED, 0, 0)
                }
        }
}
```

CREATE

```
ObjectPointer Create(Class, X, Y, CurImage, FirstDir) {
        create a new object data structure of type Class
        initialize all user VARS to 0
        set its CurImage and FirstDir variables
        link it into the playfield at (X, Y)
        SendMessage(NewObj, MSG_CREATE, 0, 0)
        for( any objects with ARRIVE flag set at the created objects location ) {
                NewObj.SendMessage(EachObj, MSG_CREATED, X, Y)
        }
        if( NewObj's Density causes it to sink beneath other objects ) {
                SendMessage(NewObj, MSG_SUNK, 0, 0)
```

```
            for( any objects above NewObj ) {
                NewObj.SendMessage(EachObj, MSG_FLOATED, 0, 0)
            }
        }
        return reference to the new Object (or 0 if creation failed)
}
```

NOTE: **Create()** can return 0 because the initial allocation of data space failed, or more likely because some class code (during the execution of the MSG_CREATE, MSG_CREATED, MSG_SUNK, or MSG_FLOATED) executed a Destroy() on the new object.

Main Idle Loop

The main idle loop takes care of all key processing, and the dispatching of MSG_MOVED, MSG_DEPARTED, and MSG_ARRIVED messages. For the sake of clarity, I'll greatly simplify a lot of the code.

TurnCount is a global variable that gets set to 0 when the Player makes a move, and otherwise gets incremented once for each time MainIdleLoop executes. *BlockHero* is a Boolean variable that controls whether or not the Player gets an opportunity to make a move. *IgnoreKey* is a global boolean variable that gets set by the IgnoreKey() function (during the processing of the MSG_KEY message).

```
void MainIdleLoop() {

        BlockHero = FALSE
        for( each playfield row from 1 to 21 ) {
            for( each playfield column from 1 to 29 ) {
                for( each object at the current playfield location ) {
                    if( obj is destroyed (OF_DESTROYED is set) ) {
                        de-allocate obj's data storage
                    } else {
                        if( obj has OF_MOVED set ) {
                            clear OF_MOVED flag
                            SendMessage(obj, MSG_MOVED, 0, 0)
                            BlockHero = TRUE
                        }
                        if( obj has OF_DEPARTED set ) {
                            SendMessage(obj, MSG_DEPARTED, 0, 0)
                            clear OF_DEPARTED flag
                            clear obj.Departed flags
                            BlockHero = TRUE
                        }
                        if( obj has OF_ARRIVED set ) {
                            SendMessage(obj, MSG_ARRIVED, 0, 0)
```

```
                                        clear OF_ARRIVED flag
                                        clear obj.Arrived flags
                                        BlockHero = TRUE
                                }
                                if( obj is animated and it's time for the next update ) {
                                        advance obj.CurImage to the next image
                                        DrawField(obj.Xloc, obj.Yloc)
                                }
                                if( obj has any flags set to block Hero movements )
BlockHero = TRUE
                        }
                }
        }
        if( FALSE==BlockHero ) {
                for( all objects ) {
                        if( SendMessage(EachObj, MSG_END_TURN, TurnCount, 0) )
BlockHero = TRUE
                        if( EachObj has flags set to block Hero movements ) BlockHero =
TRUE
                }
                TurnCount = TurnCount + 1;
        }
        if( BlockHero == FALSE ) {
                if( pending Player key press or replay move ) {
                        IgnoreKey = FALSE
                        for( all objects ) {
                                EachObj.Distance = 0
                                clear KeyCleared and OF_DONE flags
                        }
                        ObjVal = 0
                        for( all objects ) {
                                if( in "popup quiz" ) {
                                        if( EachObj==PopupQuizObj ) {
                                                SendMessage(EachObj, MSG_KEY, keyval, 0)
                                        }
                                } else if( EachObj receives keys ) {
                                        ObjVal = SendMessage(EachObj, MSG_KEY, keyval,
ObjVal)
                                }
                        }
                        if( !IgnoreKey ) {
                                TurnCount = 0
                                BroadcastMessage(-1, MSG_BEGIN_TURN, PlayerObj-
>Xloc, PlayerObj->Yloc)
                                add key to moves list
```

```
                    },
                }
            }
        }
```

Class Editing: Problems (Q & A)

The following are some potential "gotchas" when creating new classes:

Hero (or other moving object) dies (or is destroyed) when it steps on or comes in contact your new class of object.

Most likely you haven't set the Temperature for that new object (the Hero dies when the Temperature of the object beneath it is less than 60 or greater than 154. Also, check to see how the Hardness of the "destroyed" object compares to the Sharpness of the "destroying" object. If all else fails, try using the Trace Window to see the exact sequence of messages that are occuring right before the destruction occurs.

I created a new class but when I right click on it during game play, no explanation of the object appears.

You probably haven't added the proper comments to the Attributes dialog box for that class. See Class Editing: Class Attributes (Comments).

My new class works fine except that when two of them are positioned diagonally from each other the Hero can walk between them even though their Height should be tall enough to block the Hero.

Movement diagonally between objects is controlled by the Volume, not the Height. If the sum of the Volumes of the two diagonal objects and the object that is trying to move between them is <= 10000, then the move will succeed, otherwise it will fail.

When I try running a level with my new class, I get a stack overflow.

Something in your class code is causing an infinite loop. Most commonly, it's doing something that causes an "interrupt" type message to be sent (see the end of Class Editing: Messages), and in the processing of that interrupt message, the class codes do something else that cause another "interrupt" type message to be sent, and so on. The easiest way to figure out what's happening is to use the Message Trace window to see what messages are being sent . See Class Editing: Message Tracing.

An example of this would be if you created a new class of object (let's call it Death) which Destroy()'d all objects which "stepped" on it. So, in the MSG_ARRIVED code block for Death, you execute a Destroy(ObjAbove(Self)). But, if an object gets created on top of a Death object, then you want that newly created object to be destroyed as well, so Death has a MSG_CREATED code block that either does a *goto* to the MSG_ARRIVED code block or executes its own Destroy(ObjAbove(Self)). Cool. But now, the bumbling Hero comes bumbling along, trips, and falls on the Death object. Bang! MSG_ARRIVED destroys the

Hero object. Hero gets a MSG_DESTROYED and says, "Uh-Oh!" and creates a HeroDead (tombstone) object. Uh-oh. Now Death receives a MSG_CREATED and destroys the Hero (which is still in the process of being destroyed from the first time), which causes Hero to receive another MSG_DESTROYED, so it creates HeroDead again, etc. Yes, there are several ways around THIS situation, but this is a fairly simple example of how "interrupt" type messages can quickly get you into an infinite loop.

Weird things are happening, and Message Trace doesn't make any sense.

One thing to be VERY cautious about is the use of *goto* to jump from one message block into another message block, ESPECIALLY ones involving "interrupt" type messages. Sometimes, you'll want to set a user-variable as a pointer to some object and then jump to common code. But, during that common code, you may do something that causes an "interrupt" message to be sent which causes (directly or indirectly) execution to re-enter another message code block for this same object, and THAT code block trashes the user-var that the previous code block had set up. This is one of the reasons that *goto* statements are outlawed in 39 states, they get you into trouble really fast.

I'm getting a "system error" (General Protection Fault) when my new class executes.

The usual, most likely cause of this is that you've set a user-variable (VARS) to a non-zero value which is also NOT a valid pointer to an object, and then you're using it to reference an object (frequently, fetching a standard variable of that non-object object). Think about what you've changed recently, and examime those class codes VERY carefully for all references to objects which are done via a user-defined variable.

For example, the statements with an asterisk in front of them will both cause GPFs:

```
VARS {
tmp
}
SUBS {
xcode:
tmp = 5
*       if( tmp.Class==Hero ) ...
tmp = ObjAbove(Self).Class
*       if( tmp.Class==Hero ) ...
}
```

In the first case, *tmp* just has some random value which is guaranteed NOT to be a pointer to an object. In the second case, a simple bug in the code (which frequently happens as you edit and change previously written code) has caused *tmp* to be assigned to an objects class rather than a pointer to the object, and

then the class is fetched from what should be a pointer to an object but is not.

I seem to be able to place more than one Hero object on a level at once. Is this legal?

Yes, this is perfectly legal. In the MS-DOS version of Hero, this was prevented, but in Mesh it is not only legal, but can make for some interesting puzzle challenges. If there is more than one Hero on a level, they must ALL survive to the end of the level (the end of the level occurs as soon as the first Hero reaches an Exit). If *any* of the Heroes die, you must restart the level.

Contents

Welcome to MESH: Hero's Hearts

The licensed version of this game costs $19.95 + $2 shipping (within North America, $4 shipping elsewhere). It contains over 1,000 levels with complete solutions. Check with us regarding other Mesh games.

Contact information for orders and support:

Everett Kaser Software
MESH:Hero's Hearts
PO Box 403
Albany, OR 97321-0117

Phone:
(541) 928-5259

email:
everett@kaser.com

Web page:
http://www.kaser.com

Credits

Although the MESH:Hero's Hearts game itself is pretty much the creation of one person, its richness and depth owes a great deal to many other people as well. The original DOS game (Hero Gold) was inspired by a game called PC Wanderer, which was a game inspired by another game called Boulderdash. There have been MANY games in this genre, and trying to track all of the lineages and cross inspirations is pretty much an insurmountable task. Very little in our world is truly new.

Many of the improvements that I made to the DOS Hero Gold game were helped along immensely by suggestions and relentless testing by William Pelletier, and it was his enthusiam for the game that started me thinking once again about implementing a new version of Hero that was closer to my original vision (which never really saw fruition in the DOS game). He also gave me significant feedback and encouragement during the development of MESH:Hero's Hearts, and is a gentleman and a... well... you get the idea.

Sam Stoddard sent me several puzzle sets for the DOS Hero Gold just as I was finishing up my previous game (Dinner With Moriarty), and it was my discussions with him via email that finally "broke my camel's back" and convinced me to finally start work on the "new Hero" game. He seems to have spent January through March of 1998 doing nothing but testing Mesh and making suggestions for improvements (although he claims to have held down a job and gone back to school at the same time). I am very grateful for his efforts, for without them Mesh would not have been nearly so robust or so capable. Best wishes to you and Darleen!

Finally, MESH:Hero's Hearts would have only a fraction of the number of puzzle levels if not for the work and contribution of many people. The following folks created these levels that were included in Hero's Hearts (roughly in the same order as I originally received levels from them):

Everett Kaser	HEARTS1	1-55, 116-124
	HEARTS2	1-3
	HEARTS3	129-130
	HEARTS4	135-139
Shane Kaser	HEARTS1	56-57
Megan Kaser	HEARTS1	58-66
Tim Emmerich	HEARTS1	67
Ed Tiley	HEARTS1	68-73
Jan K Hoag	HEARTS1	74-81
	HEARTS2	134-167
	HEARTS4	224-241
Rich Teets	HEARTS1	82
Annie Oakley	HEARTS1	83-87
Jean-Marc Genevey	HEARTS1	88-94
Carol Gortat	HEARTS1	95-96
Bonnie Raymond	HEARTS1	97-109

William T Pelletier and Jo,	HEARTS1	110-115
Sharon, Noah, and Stephen	HEARTS2	275-293
	HEARTS3	1-11
	HEARTS4	191-209
Patrick F McConnville	HEARTS2	4-93
Hein Mank	HEARTS2	94-133
William Degelmann	HEARTS2	168-177
Rob Taylor	HEARTS2	178-181
Sandy McCauley	HEARTS2	182-274
	HEARTS3	131-204
	HEARTS4	261-310
Russell Kennedy	HEARTS3	12-69
Germain Dube	HEARTS3	70-86
Charles S Knippenberg	HEARTS3	87
Jim Bush	HEARTS3	88-106
Teun, Piet-Jan, & Klaas Spaans	HEARTS3	107-128
John Gibbs	HEARTS3	205
Glenn Turner	HEARTS3	206-213
Jim Schuetz	HEARTS3	214-261
Lawrence J Coplin	HEARTS3	262-268
	HEARTS4	140-160
Albert H Them	HEARTS3	269-282
Sydney Burton	HEARTS3	283-294
Sam Stoddard	HEARTS4	1-132
Darleen Daniels	HEARTS4	133-134
Brett Bydairk	HEARTS4	161-177
Rich Beckwith	HEARTS4	178-190
Fran Anderson	HEARTS4	242-260

Thank you, one and all!!!

Function or Statement: Broadcast

Description:

Sends a message (almost *always* a user-defined message) to all objects currently on the playfield whose class matches the specified class.

Syntax: Broadcast(*class, msg, arg1, arg2*)

class	The class of object to all instances of which this message will be sent. If the *class* argument is -1, then the message is sent to ALL objects on the playfield regardless of their class.
msg	The message to be sent to the objects of that class.
arg1	The value that will be sent as MsgArg1 for that message.
arg2	The value that will be sent as MsgArg2 for that message.

Return value:

Returns the number of objects to which the message was sent.

Examples:

```
Broadcast(Roller, USR_SETIMAGE, CurImage, 0)
Broadcast(Water, USR_DRY_UP, 0, 0)
wormcnt = Broadcast(Worm, USR_COME_AND_GET_ME, Xloc, Yloc)
```

See also: SendMessage, Class Editing: Messages, Class Editing: Messages - User Defined

Function or Statement: Create

Description:

Creates an object of the specified class at the specified location on the playfield. A MSG_CREATE is sent to the object immediately after its creation so that the object may do any "one time" initialization that might be necessary. Then, all nearby objects that had their *Arrivals* flag set for that location will receive a MSG_CREATED with MsgFrom pointing to the newly created object and (MsgArg1, MsgArg2) equal to the playfield coordinates of the newly created object. Lastly, if the newly created object had a higher density than some of the objects already existing at that location, then a MSG_SUNK will be sent to the new object, followed by a MSG_FLOATED sent to all objects at that location with a lower density than the new object.

Relative directions (such as DirL, DirF, DirBR, etc) do not resolve until actually used by an object. Hence, they are only useful when an object already HAS a direction. Thus, when using the Create() statement, you should NOT pass relative directions as the *dir* argument, because the object being created has no direction yet. If you DO pass a relative direction to the Create() function, the object will be assigned a *dir* of 0.

Syntax: Create(*class, x, y, image, dir*)

class	The class of the object to be created.
x, y	The location on the playfield where the object is to be created.
image	The image number to store into the objects CurImage variable.
dir	The direction to store into the objects LastDir variable.

Return value:

0 if the creation failed for any reason, else a pointer to the created object. The most likely reason for the creation to fail is if, during the processing of the MSG_CREATE, MSG_CREATED, MSG_SUNK, or MSG_FLOATED messages, the Destroy() function is called on the newly created object.

Examples:

```
obj = Create(Worm, Xloc, Yloc, 0, 0)
if( obj ) Destroy(Self)
```

See also: Destroy

Function or Statement: Destroy

Description:

Destroys the specified object. If any nearby objects have their *Departures* flag set for the location of the destroyed object, then a MSG_DESTROYED is sent to those objects. (Be careful not to confuse MSG_DESTROYED with MSG_DESTROY. See Class Editing: Messages.)

The object which is about to be destroyed will receive the MSG_DESTROY message. If 0 is returned from this message (the default) then the object is destroyed and MSG_DESTROYED is sent to any objects with Departures bits set for that location. If a non-zero value is returned from the the MSG_DESTROY message, then the object is not destroyed.

Syntax: Destroy(*obj*)

obj A pointer to the object to be destroyed.

Returns:

The value returned by the MSG_DESTROY message.

Examples:

```
Destroy(Self)
Destroy(ObjAbove(Self))
if( Destroy(obj) ) Move(Self, LastDir)
```

See also: Create

Function or Statement: JumpTo

Description:

Moves the specified object to the specified location without "passing through" the intervening locations. Does not change LastDir. A MSG_MOVING will be sent to the object before the move. If 0 is returned from the MSG_MOVING (which is the default unless the object has a MSG_MOVING code block which specifically returns a non-zero value) then the move will occur, regardless of Height, Climb, Strength, etc. If the move occurs, then arrival and departure flags will be set in the appropriate objects surrounding the old and new locations (the actual MSG_ARRIVED and MSG_DEPARTED messages won't be sent until later, during the normal "idle" loop), and MSG_SUNK and MSG_FLOATED will be sent to the appropriate objects at the new location if the new object has a higher density than some of the objects already existing at the new location. Finally, if the move occurred, a MSG_JUMPED will be sent to the object immediately after the move with (MsgArg1, MsgArg2) being the coordinates of the objects *old* location *before* the jump. (Xloc, Yloc) is the coordinates of the *new* location.

Syntax: JumpTo(*obj, x, y*)

obj	A pointer to the object to be moved.
x,y	The destination location for the move.

Return value:

0 if the Jump failed, else non-zero.

Examples:

JumpTo(Self, Xloc-3, Yloc)

See also: Move

Function or Statement: Move

Description:

Moves the specified object to the adjacent location in the direction specified. ***Many*** messages may be sent as the result of this call. See Class Editing: Messages and Class Editing: Messages and the internal code structure, paying particular attention to these messages:
MSG_HIT
MSG_HITBY
MSG_MOVING
MSG_PLAYERMOVING (if it's the "player object" which is being moved)
MSG_DEPARTED
MSG_ARRIVED
MSG_SUNK
MSG_FLOATED
MSG_MOVED
Not all of these messages will be sent in every situation. There are many possible situations, and moving objects is the most complex aspect of the game.

Syntax: Move(*obj, dir*)

obj	A pointer to the object to be moved.
dir	The direction in which the object is to be moved. (It may be an *absolute* or a *relative* direction.)

Return value:

0 if the move failed, non-zero if the move succeeded.

Examples:

```
Move(Self, DirN)
if( Move(obj, DirL) ) Destroy(Self)
```

See also: JumpTo, Class Editing: Messages, Class Editing: Messages and the internal code structure

Function or Statement: SendMessage

Description:

Sends a message (almost *always* a user-defined message) to the specified object. Control does not return to the caller until the message has been processed by the target object.

Syntax: SendMessage(*obj, msg, arg1, arg2*)

obj	A pointer to the object to which the message is to be sent.
msg	The message to be sent.
arg1	The value to be sent with the message as MsgArg1.
arg2	The value to be sent with the message as MsgArg2.

Return value:

The value that is returned by the objects message code block for this message. If the object doesn't have a code block for this message, or if the code block doesn't have a *return* statement at its end, then 0 is returned.

Examples:

```
SendMessage(Roller, USR_SETIMAGE, CurImage, 0)
SendMessage(obj, USR_SPLASH, 0, 0)
s = SendMessage(obj, USR_FETCH_SOME_VALUE, 0, 0)
```

See also: Broadcast, Class Editing: Messages, Class Editing: Messages - User Defined

Function: Delta

Description:

Returns the absolute value of the difference between the two arguments.

Syntax: Delta(*a, b*)

a,b Two numeric values.

Return value:

If(a >= b) returns a-b
else returns b-a

Examples:

deltaX = Delta(Xloc, obj.Xloc)
deltaY = Delta(Yloc, obj.Yloc)

See also:

Function: HeightAt

Description:

Returns the *maximum* height of all objects at the given location. (ie, whichever object at the given location has the greatest Height, that objects Height value is returned as the value of this function.)

Syntax: HeightAt(*x, y*)

x,y The playfield location from which the maximum Height is returned.

Return value:

The ...uh... maximum height of the objects at the ...uh... specified location. Yeah, that's it.

Examples:

```
h = HeightAt(Xloc, Yloc-1)
if( 4000 > HeightAt(XDir(Self, LastDir), YDir(Self, LastDir)) ) Move(Self, LastDir)
```

See also: VolumeAt, Class Editing: Class Attributes

Function: Key

Description:

Only valid during the processing of the MSG_KEY message, this function returns the value of the key that was pressed. It can be compared to the KEY_xxx constants as defined in Class Editing: Class Codes - Constants.

Syntax: Key

The *Key* function has no arguments nor any following parentheses.

Return value:

The keycode of the key that was pressed which caused the MSG_KEY message.

Examples:

```
if( Key==KEY_BACKSPACE ) LocateMe()
if( Key==KEY_D ) Destroy(Self)
```

See also: Class Editing: Class Codes - Constants

Function: Level

Description:

Returns the number of the current puzzle set level. This number is actually one ***less*** than the value displayed on the toolbar. The level numbers internally are from 0 to LevelCount-1, whereas on the toolbar they are displayed as 1 to LevelCount.

Syntax: Level

The *Level* function has no arguments, nor any following parentheses.

Return value:

The number of the current level, where the first level is 0.

Examples:

GotoLevel(Level)
GotoLevel(Level+1)

See also: GotoLevel

Function: LevelCount

Description:

Returns the total number of levels in the current puzzle set.

Syntax: LevelCount

The *LevelCount* function has no arguments, nor any following parentheses.

Return value:

The number of levels in the current puzzle set.

Examples:

GotoLevel(Level)
GotoLevel(Level+1)

See also: Level

Function: NewX

Description:

Returns the x-coordinate of the playfield location in the specified direction from the specified x-coordinate location. NewX never return values <0 or >30. The direction argument is *anded* with 7. ie, only the lower three bits are observed, limiting the value to 0 through 7.

Syntax: NewX(*oldx, dir*)

oldx	The x-coordinate from which the new coordinate is derived.
dir	The direction from the old x-coordinate from which to find the new.

Return value:

The x-coordinate of the new location in the direction specified.

Examples:

```
if( NewX(Xloc, LastDir) < 15 ) Move(Self, LastDir)
x = NewX(obj.Xloc, obj.LastDir)
```

See also: NewY, XDir, YDir

Function: NewY

Description:

Returns the y-coordinate of the playfield location in the specified direction from the specified y-coordinate location. NewY never return values <0 or >22. The direction argument is *anded* with 7. ie, only the lower three bits are observed, limiting the value to 0 through 7.

Syntax: NewY(*oldy, dir*)

oldy	The y-coordinate from which the new coordinate is derived.
dir	The direction from the old y-coordinate from which to find the new.

Return value:

The y-coordinate of the new location in the direction specified.

Examples:

```
if( NewY(Yloc, LastDir) > 3 ) Move(Self, DirN)
x = NewY(obj.Yloc, obj.LastDir)
```

See also: NewX, XDir, YDir

Function: ObjAbove

Description:

Returns a pointer to the object *above* the specified object (at the same playfield location). *Above* means "out of the display" from the specified object, NOT "north" on the display from the object.

Syntax: ObjAbove(*obj*)

obj The object above which you wish to know the next object.

Return value:

0 if no objects exist at the same playfield location above the specified object, else a pointer to the next object above the specified object at the same location.

Examples:

```
obj = ObjAbove(Self)
Destroy(ObjAbove(obj)
```

See also: ObjClassAt, ObjBottomAt, ObjTopAt, ObjDir, ObjBelow

Function: ObjBelow

Description:

Returns a pointer to the object *below* the specified object (at the same playfield location). *Below* means "into the display" from the specified object, NOT "south" on the display from the object.

Syntax: ObjBelow(*obj*)

obj The object below which you wish to know the next object.

Return value:

0 if the specified object is the bottom-most object at its location, else a pointer to the next object below the specified object at the same location.

Examples:

```
obj = ObjBelow(Self)
Destroy(ObjBelow(obj)
```

See also: ObjClassAt, ObjBottomAt, ObjTopAt, ObjDir, ObjAbove

Function: ObjBottomAt

Description:

Returns a pointer to the *bottom-most* object at the given (x,y) playfield location.

Syntax: ObjBottomAt(*x, y*)

x,y	The (x,y) location on the playfield at which you wish to know the bottom-most object.

Return value:

0 if no objects exist at that playfield location, else a pointer to the bottom-most object at that location.

Examples:

```
obj = ObjBottomAt(Xloc-1, Yloc)
Destroy(ObjBottomAt(XDir(Self, DirF), YDir(Self, DirF)))
```

See also: ObjClassAt, ObjTopAt, ObjDir, ObjAbove, ObjBelow

Function: ObjClassAt

Description:

Searches a given (x,y) playfield location for an object of the a given class.

Syntax: ObjClassAt(*class, x, y*)

class	Usually a literal class name (like Hero, Rock, Arrow, etc), but may also be any legitimate numeric expression, so long as it resolves to a number which matches a valid class. Normally, the only thing you'd use here other than a literal class name is the name of a variable into which you've previously stored a literal class value.
x,y	The (x,y) location on the playfield where you want to look for an object of the specified class

Return value:

0 if none are found, else a pointer to the bottom-most object at that location whose class matches the given class.

Examples:

```
c = ObjClassAt(Worm, Xloc-1, Yloc)

if( !ObjClassAt(Water, XDir(Self, DirL), YDir(Self, DirL)) ) Move(Self, DirL)

c = Dirt
obj = ObjClassAt(c, varX, varY)
```

See also: ObjTopAt, ObjBottomAt, ObjDir, ObjAbove, ObjBelow

Function: ObjDir

Description:

Returns a pointer to the *top-most* object at the playfield location in the given direction from the given object (the adjacent location in that direction, i.e. one square away).

Syntax: ObjDir(*obj, dir*)

obj	The object from whose (Xloc, Yloc) and LastDir is used in conjunction with *dir* to arrive at the playfield location.
dir	The direction from the specified object.

Return value:

0 if no objects exist at that playfield location, else a pointer to the top-most object at that location.

Examples:

```
obj = ObjDir(Self, DirR)
Destroy(ObjDir(obj, DirN))
```

See also: ObjClassAt, ObjBottomAt, ObjTopAt, ObjAbove, ObjBelow

Function: ObjTopAt

Description:

Returns a pointer to the *top-most* object at the given (x,y) playfield location.

Syntax: ObjTopAt(*x, y*)

x,y The (x,y) location on the playfield at which you wish to know the top-most object.

Return value:

0 if no objects exist at that playfield location, else a pointer to the top-most object at that location.

Examples:

```
obj = ObjTopAt(Xloc-1, Yloc)
Destroy(ObjTopAt(XDir(Self, DirF), YDir(Self, DirF)))
```

See also: ObjClassAt, ObjBottomAt, ObjDir, ObjAbove, ObjBelow

Function: Self

Description:

Returns a pointer to the object whose class codes are currently executing. Useful when you need to pass a pointer to an object as the argument of a function like Move(), JumpTo(), Destroy(), etc.

Syntax: Self

The *Self* function has no arguments nor any following parentheses.

Return value:

A pointer to the "current object."

Examples:

Move(Self, DirN)
Destroy(Self)

See also: Class Editing: Class Codes (*Standard class variables, and pointers to objects*)

Function: VolumeAt

Description:

Returns the *maximum* volume of all objects at the given location. (ie, whichever object at the given location has the greatest Volume, that objects Volume value is returned as the value of this function.)

Syntax: VolumeAt(*x, y*)

x,y The playfield location from which the maximum Volume is returned.

Return value:

The ...uh... maximum volume of the objects at the ...uh... specified location. Yeah, that's it.

Examples:

```
v = VolumeAt(Xloc, Yloc-1)
if( 10000 > VolumeAt(XDir(Self, LastDir), YDir(Self, LastDir)) ) Move(Self, LastDir)
```

See also: HeightAt, Class Editing: Class Attributes

Function: XDir

Description:

Returns the x-coordinate of the location in the specified direction from the location containing the specified object.

Syntax: XDir(*obj, dir*)

obj	A pointer to an object on the playfield
dir	A direction from the specified object, absolute or relative.

Return value:

The x-coordinate of the location in the specified direction from the specified object.

Examples:

```
x = XDir(Self, LastDir)
JumpTo(Self, XDir(Self, LastDir), YDir(Self, LastDir))
```

See also: YDir, NewX, NewY

Function: YDir

Description:

Returns the y-coordinate of the location in the specified direction from the location containing the specified object.

Syntax: YDir(*obj, dir*)

obj	A pointer to an object on the playfield
dir	A direction from the specified object, absolute or relative.

Return value:

The y-coordinate of the location in the specified direction from the specified object.

Examples:

```
y = YDir(Self, LastDir)
JumpTo(Self, XDir(Self, LastDir), YDir(Self, LastDir))
```

See also: XDir, NewX, NewY

Game options

The primary game options are available under the *Options* menu (imagine that!) and the *View* menu. Also, the Hero's Hearts puzzle levels are divided up into four separate puzzle sets (HEARTS1, HEARTS2, HEARTS3, and HEARTS4). When you finish the last level in the first three of these, the game will automatically load in the next puzzle set and start you at the first level in that puzzle set. You can manually switch between these puzzle sets by using the *Select puzzle set...* menu item under the *File* menu.

Options-New player: this is how you add a new player to the game. Player names must be 1 to 8 alphanumeric characters. The various settings in the game for each player are kept separately, and are automatically reloaded when you select that player. These settings include things like puzzle set, level number, window size and position, image size, music and sound effects, etc.

> **[tech alert]** A subdirectory (or "folder" in currently politically correct terminology) is created in the Mesh directory for each player, the name consisting of the players name with .DIR tacked onto the end. Inside that subdirectory is a PLAYER.CFG file containing all of the settings for that player. Also stored in that subdirectory is a .MV file for each puzzle set that player has played. The .MV files contain the move lists for that player for that puzzle set. For example, if Hero's Heart is installed in C:\HEROHART, and one of the players is named Geek, then there will be a C:\HEROHART\GEEK.DIR directory, and a C:\HEROHART\GEEK.DIR\PLAYER.CFG file containing the players options settings, and a C:\HEROHART\GEEK.DIR\HEARTS1.MV file containing the move lists for Geek for puzzle set HEARTS1.MB. Should you want to delete, remove, or rub out a particular player, all you need to do is to delete that players .DIR directory.

Options-Select player: this is how you change players, if more than one player is setup in the game. When the game starts, if there is only one player then the game starts up normally. However, if there is more than one player, the game first displays the "Select player" dialog, with the current player highlighted as the default, to allow you to select a different player if you want to.

Options-Select puzzle set: this is how you change puzzle sets. All puzzle sets are stored in the main Mesh directory as .MB (Mesh Binary) files. The Mesh: Hero's Heart "splash screen" is displayed while the new puzzle set is loaded. The *Select puzzle set* option is also available under the *File* menu, since some folks may look for it in one place, while others might look for it in the other.

Options-Sound effects: this is a "toggle", turning sound effects on or off. If you can't figure this one out, you probably shouldn't be playing this game.

Options-Music: this is a "toggle", turning background music on or off.

> **[tech alert]** Only one MIDI music file is supported, MESH.MID. You may substitute your own MIDI file (if you have a favorite tune you'd rather play to) by renaming

MESH.MID to something like MESHMID.OLD, and then copying your favorite MIDI file into the Mesh directory as MESH.MID.

Options-Whatever speed: there are five different speed settings. These control how fast things move on the playfield. If something is happening too fast for you to see, try selecting a slower play speed. The actual delays introduced by these speed settings are specified in the *Speed & Replay options* dialog.

Options-Speed & replay options: this is where you can modify the actual speed of the five different speed settings (increasing the delay count causes things to slow down more). The defaults are 0, 5, 10, 15, and 30. This dialog also lets you specify the X in Restart-X, the number of moves left undone when you restart and automatically replay up to *just* before where you were when you did the Restart-X. Get that?

Options-Right click on REPLAY buttons does Restart-n: if that title doesn't say it all, what more can I say? There are four replay buttons on the toolbar: 1, 10, 100, and 1000. With this option enabled, RIGHT clicking on those buttons causes the level to be restarted and replayed up until 1, 10, 100, or 1000 moves BEFORE where you were when you right clicked on the button.

Options-Popup explanation of players destruction: when the Hero is "destroyed" (killed, in non-politically correct terminology), it is sometimes obvious what caused it, and sometimes it's not so obvious. With this option enabled, a popup window will appear showing what type of object (and at what x,y playfield location) killed the Hero. If this popup just gets in your way, then turn off this option.

View-Level startup text: The level startup text is a special block of text attached to each level. This text is entered by the creator of the level, and it is displayed the first time a level is played. Should you want to read the startup text again (during the playing of the level for example), then click on this menu item. Changing to a different level and then back to the current level will also cause the startup text to be redisplayed.

View-Images size...: There are three different sizes of images for each object, in order to give the best possible display on your monitor. However, there are a very wide range of monitors and possible display resolutions, and while the game makes a valiant attempt at guessing the appropriate size image for your display setup, it will frequently not be what YOU want. These six image size settings allow you to override the image size setting that the game has chosen.

View-Toolbar: Mesh puzzle levels come in different sizes, up to a maximum of 29 squares wide by 21 squares high. These largest puzzle levels BARELY fit on some displays at some resolutions. Hiding the Toolbar can buy you a little extra vertical space. However, if you do so, you'll not be able to see your move list, and will have to use the F1-F5 function keys for replaying.

View-Show toolbar tips: For the beginner, when you point at a button on the toolbar,

a small popup window appears telling you what that button does. For the more experienced player, these popup windows get in the way. They can turn off this option to hide those popup windows.

How to play the game

Who likes to read the manual? I'll try to keep this brief but informative...

Much of the information about the basic behaviors of the objects is provided in "popup" notes in the first few levels of the game. The rest is intentionally left vague, as this is part of the puzzle.

The Goal

...is to enjoy yourself. To do so, you must move the Hero around the playfield, pick up ALL of the Hearts, and reach the Exit. Red Hearts can be picked up merely by walking on them. White and Blue Hearts must first be turned into Red Hearts before they can be picked up.

Moving the Hero

The Hero can be moved either using the cursor pad or numeric keypad on the keyboard or with the mouse. I prefer the numeric keypad, but others prefer the mouse. The number of moves a player can make is limited to 32768 per level. (If you want to construct levels requiring more moves than that, I'm sorry. The game is intended to be a puzzle challenge, not an endurance challenge.)

The four arrow keys move the Hero north, east, south, and west, while the Home, End, PageUp, and PageDn keys move the Hero northwest, southwest, northeast, and southeast (ie, diagonally). Diagonal movements are NECESSARY in order to solve some levels. Pressing the 5 key on the numeric keypad or the SpaceBar causes the Hero to stand still, but gives all other objects on the playfield a chance to move (like Creepers and Worms).

If you point at a square with the mouse and click the left mouse button, the Hero will TRY to move directly to that location. The Hero doesn't know how to avoid objects or pitfalls, so be careful. Also, moving through Teleports or over Rollers may affect movement with the mouse. Clicking the left mouse button on the square where the Hero is located causes the Hero to stand still while giving all other objects on the playfield a chance to move. The mouse move gets translated into a series of keyboard moves. A single click of the mouse can generate 20 to 30 moves on the moves list. (This translation makes "replaying" more straight-forward and ensures that playing with the mouse doesn't give you an advantage as to the length of possible solutions, over someone playing with the keyboard.)

NOTE: if you RIGHT click the mouse on an object on the playfield, a popup note will appear giving you a brief description of that object.

As you move, your motions will affect other objects on the playfield. For example, each time you move, any Creepers on the level will also move once, as will any Worms. If you (or anything else) moves from within a certain "private space" of a rock, balloon, or arrow, the rock will attempt to fall, the balloon will attempt to rise, and the arrow will attempt to fly. The actions of these objects can then set off actions of other objects.

One move by the Hero can cause a massive amount of movement by other objects on the playfield or no movement at all. (Some objects, like Fire and Rollers, are "animated." They "move" all of the time while staying in the same square. This "motion" should not be confused with a "Move" from one location to another.)

Sometimes when you first start a new level, it can be difficult to locate the Hero, even though arms are waving and toes are tapping. Pressing the Backspace key causes the Hero object to show a "sonar" pattern on the screen which locates it.

Replaying moves

Solving a level usually requires hundreds, and sometimes thousands, of moves. Mess up once, and you'd have to repeat ALL of those moves. Bummer. Luckily, Mesh saves all of your moves and will repeat them for you automatically. The toolbar across the top of the window contains several pushbuttons and a "move list" window. The move list scrolls to the left as you make moves, recording each one. If you have to start over, the move list gets reset to the first move. You can either keep right on playing as normal from that point, or you can replay any number of your previous moves before resuming normal play. Previous moves can be replayed in increments of 1, 10, 100, or 1000 moves at a time by clicking on the appropriately numbered button on the toolbar, or by pressing F1, F2, F3, or F4 respectively. You can also insert moves into the move list (by pressing the Insert key) and delete moves from the move list (by pressing the Delete key).

The number on the left end of the move list window is the number of moves that have been made since the beginning of the level. The number on the right end of the move list window is the total number of moves in the move list. The colored box in the middle of the moves list indicates the next move that will be made during replay or where the next move will be stored while you're playing the level. The color of this box is red if you haven't solved the level (or are in the process of replaying it) and green if your player move list is a valid solution to the level. When in insert mode, it turns to a blinking yellow.

On the far left of the toolbar, there is a Restart button and a Restart-X button. The first just restarts the level, whereas the second not only restarts the level but also replays your moves up to the point X moves prior to where you were when you restarted it. The X is controlled by a setting in the *Speed and Replay options* dialog which is accessed under the *Options* menu. The Restart-X is useful when you mess up and want to back up just a few moves.

Ocassionally you may wish to try an "alternate" solution, but really don't want to lose the moves that you currently have been working on. Under the *File* menu are *Save move list as...* and *Load move list...* menu items which allow you to save your current move list into a file (with a name of your choice), and to load the move list from the file of your choice. You can save the move list at any time during the playing of the game without effecting the state of the move list or the playfield. However, when you load a move list from a file the game automatically re-initializes the level for you. If you accumulate a

lot of excess .MVL (move list) files in your player directory and want to delete some of them, you'll have to do it using a file manager or command prompt, as Mesh does not support deleting these move files. ***Be careful not to delete your .MV files, however, as those contain your move lists for ALL levels in the puzzle sets that you've played.***

Solutions: when you get stuck

Sometimes you'll get stuck on a level, and I don't want any lawsuits because you've pulled all of your hair from your head, so included with each level is a complete solution. It's not necessarily the shortest solution, merely one way of doing it. You can replay this solution by clicking the checkered-flag pushbutton on the toolbar. (It will stay depressed until you click it a second time.) As long as it's depressed (not just sad, but REALLY depressed!) then you will not be able to make any moves yourself, you will only be able to replay moves from the solutions move list. Each time the solution pushbutton is clicked, the level is re-initialized to the beginning, ready either for you to start playing again or for you to start replaying the solution again.

In the (very likely) possibility that you solve a level in fewer moves than the current solution, then YOUR solution will be stored into the data file and will thereafter show as the solution (on your computer) for that level. The player name is also stored with the solution and is displayed in the title bar (in place of the current player name) any time the solution pushbutton is depressed.

CAUTION: When you edit a level, Mesh automatically removes the solution for that level (since the solution quite likely will not be valid any longer). It does NOT remove the moves from the players move list, JUST the solution move list. So, if you modify a level in such a way that the players move list IS still a valid solution, merely replay your player move list, and the solution will be restored to the data file.

Changing levels

The current level number is displayed between the Restart-X pushbutton and the "1" pushbutton on the toolbar. You can move to the previous or next level by clicking on the arrows to the left or right of the level number. You can also jump to a specific level number by clicking the left mouse button on the level number window and then changing the level number (with the keyboard) to the desired number. Then, press the ENTER key.

You can move to the next and previous levels also by pressing the + and - keys on the numeric keypad.

Inventory

On some levels, you will find objects (such as Yellow and Blue Keys) that the Hero can pick up. When this happens, an "Inventory Window" will appear at the far right of the Toolbar, showing what objects the Hero has in his Inventory and how many of each.

Introduction

To paraprhase Charles Dickens, "It was the simplest of games, it was the most complex of games."

This is a simple game, just move your Hero around the playfield, pick up all of the Hearts, and reach the Exit. Each level is a self-contained puzzle, and quick reflexes are of no use, only quick wits will help you. Determine how the various objects behave and how to use objects in combinations to allow your Hero to reach all of the Hearts and the Exit, and you've mastered the game (or, at least, one puzzle). Some puzzles are trivial, some are *extremely* challenging.

The game can become more complex, if you wish to create your own puzzles. A level editor is included that lets you create new puzzle sets containing levels of your own design. Share them with family, friends, and enemies, challenging them to solve what you've created.

The game can become far more complex, if you wish to add new classes of objects to the game. These new classes can have their own unique images and their own unique behavior. Entirely new types of puzzles can be created, or you can make "simple" modifications of existing classes. A class editor is included that allows you to specify the *attributes* for you new class, draw the *image(s)* for you new class, and write the *code* that can override, modify, or extend the behavior specified by the *attributes*. **NOTE:** class editing is ***not*** a simple thing to do. It requires a great deal of thought, care, logic, and other rare commodities (like warm bathroom floors and honesty on a first date). Although the program supports class editing and there is a great deal of information regarding it in this help file, class editing is an ***UNSUPPORTED*** feature. If you have questions about it, please ask and I'll try to answer, but don't be surprised or hurt if I say, "I'm sorry, I can't help with that." Life is short, and class editing is complex. *ALL* of my time could easily be taken up helping people develop new classes, and then there'd be no new games from me, and my family would starve. Do you want THAT on your conscience???

What is Mesh?

Mesh is the name of the "game engine," the basic program itself. Mesh allows the creation of classes of objects, puzzle "levels" containing objects of those classes, and the playing (or execution) of those classes and levels. Mesh by itself isn't too useful. It's like a box full of tools sitting in the corner. But put it together with some blood, sweat, and tears (not to mention some creative ideas), and you can build some wonderful games with it. A *mesh* is a tightly interwoven tapestry of individual threads, the combination making something very different from the individual components. Any single object in Mesh is usually quite simple, but when combined with others, a very different experience results.

What is Hero's Hearts?

Hero's Hearts is the name of a collection of puzzles that run on the Mesh game engine. It is the first, but not the last, game implemented with the Mesh game engine. The

puzzles are stored in a large data file (referred to hereafter as a *puzzle set*). When you run Mesh:Hero's Hearts, the Mesh game engine reads in the puzzle set and executes it ("performs", not "kills"). There are four individual puzzle sets in Hero's Hearts, and together they contain over 1,000 puzzle levels. These puzzles were originally created for the DOS game *Hero Gold*, by a lot of different people, so there is a wide variety of styles and difficulty levels.

If you create new puzzle sets that you'd like to share with others, send a copy to:

Everett Kaser Software
PO Box 403
Albany OR 97321-0117

or email them to:

everett@kaser.com

and I'll try to make them available to others via my web page or by diskette. My web site is:

www.kaser.com

Level Editing

Creating new puzzle levels of your own can be as much fun as playing puzzle levels created by others, and is a very different kind of experience. When playing levels, you're presented with a complete, whole tableau which you must observe, pick apart, and figure out how it works, how to get your Hero through it. When creating a puzzle level, you start with a blank screen, must think of the kernel of a puzzle idea, add that to the level, then think of ways to complicate and obfuscate (hide, make less than obvious, present a challenge, stump the player, pull the wool over, and otherwise befuddle), and you must do that without adding "unintentional solutions", ways of solving the puzzle that are easier and simpler than what you had intended.

Creating your puzzle set

The first step in creating your own puzzle levels is to create a new puzzle set. This is done by (how else?) selecting *Create new puzzle set* under the main *File* menu. Leave the "radio buttons" set to "Retain class definitions" and enter a name for your new puzzle set. You will now have a new puzzle set containing all of the class definitions (objects) from the previous puzzle set, but none of the levels. There will be one level, Blank. At this point, you should select the *Edit levels* item under the File menu. Watch the toolbar, as that's the only place you'll see much change when you go from "game playing" mode to "level editing" mode.

If you create multiple puzzle sets that are linked together, be sure to set the PuzzleNumber for each puzzle set to a different value so that your class codes can know which puzzle set is currently loaded. You do this with the Level Editors *Edit-Set puzzle set number...* menu item. See Link and Class Editing: Class Codes - Standard Variables for more information about the Puzzle Set number. **If you're not creating any new classes and are just creating new levels with existing classes, it is recommended that you ignore this feature.**

You're now ready to start adding objects to the first level in your new puzzle set. To do this, you will first need to select the object to be added. If you look at the toolbar, you will see these pushbuttons and controls:

- *Return to Game*: this button tries to show four itsy-bitsy objects on a playfield, and will return the program to "game play" mode (ending your level editing session).
- *Class Editing*: this button takes you to Class editing mode (surprised?) Don't click on this button unless you REALLY want a challenge and think you know a lot about "programming" types of things.
- *Level status/selection*: this control is the same as the one on the main game menu.
- *Level Startup Text*: this brings up a text edit dialog that allows you to enter/modify the startup text for this level, the text that appears in a popup window when the level is first played.
- *Border Colors*: this is the button with the blue square on it. It brings up the Border Colors dialog. The Mesh playfield is automatically surrounded by a "border" which can be different for each level. For more information on this subject, see

Level Editing: Borders and Color Schemes.

Object Selection: this is the button with the tiny "Hero" image on it. It brings up the Object Selection dialog, which allows you to select the class, and image of the object which you wish to add to the level.

Most Recently Used Objects List (MRU list): this shows the most recently used objects, and is a shortcut method for selecting the object you wish to add to the playfield. Each time you select a new object from the Object Selection dialog, it gets added to the front of the MRU list. At any time while editing the level, you can change the currently selected object simply by clicking on its image in the MRU list. The currently selected object is outlined with a black box.

Playfield Location: this shows the X:Y location on the playfield of the square that the mouse cursor is pointing at. This is reference information only, but can be useful with some classes of objects that might need an X:Y "target" location stored in their Misc variables (see Level Editing: Misc Vars).

Selecting an object and adding it to the playfield

With that under your belt, click on the *Object Selection* pushbutton, and find the "Field" object (the green square, the "background" object). It should be about the 10th object in the default set of Hero's Hearts objects. As you click once on each object, the class name for that object will appear below, just above the window containing all of the possible images for that class.
You can either click once on the Field object and then click the Close button, or you can just double-click on the Field object and that will automatically close the dialog.

You will now notice that the Field object has been added to the MRU list on the toolbar, and is the selected object. Point anywhere on the playfield (which is probably all black at this point, as there are no objects on the level) and click the left mouse button to add a Field object. You can also hold down the left mouse button and "paint" with the current object by dragging the mouse around the playfield. If you drag the mouse over a location that already has an object of that type as the topmost object, it will NOT add another object of that type at that location. You cannot even add a duplicate object by clicking the left mouse button on the location, except under one of two situations:

1) There is another object of a different class on top of the first object which you're trying to duplicate.
2) You've enabled the *Allow duplicate objects at same location* item under the *Options* menu.

So, if you place a Field object at a location, then place a Hero object on top of it, then you could place a second Field object at that location. However, you wouldn't know that you'd done that, because objects (classes) have *Density* (which is specified when creating a Class), and the Density controls the order that objects are drawn, with the "heaviest" (most dense) objects sinking to the bottom. Why you'd want two Field objects at the same location is beyond me, but there's some weird people out there... However, this applies to all classes, not just Field objects.

Whew. Who said this was complex?

Now that you've place some Field objects on the playfield, there's an easier way to get the basic "groundwork laid" for a level. Under the *Edit* menu, you will find *Fill 23x15 playfield* and *Fill 29x21 playfield*. These menu items will place objects of the currently selected class at ALL playfield locations that don't currently contain ANY objects, within the limits of the the 23x15 or 29x21 playfield sizes. Your levels don't have to contain rectangular playfields, they can be any shape or size you want. But the most common puzzles are rectangular, and these functions make it easier to get started. The 29x21 size is the largest playfield possible, but means that the objects on the display will be smaller and more difficult for some folks to see (if they want the entire puzzle to fit on the display at one time without scrolling). The 23x15 puzzle size allows the player to use larger images (selected under the *View* menu with the *Images sizes...* items) while still seeing the entire puzzle on the display. The 29x21 size matches the size of the playfield in the older DOS game Hero Gold, from which most of the puzzles in Hero's Hearts were acquired. The 23x15 puzzle size is expected to be the default size for most newer puzzles.

Copying, moving, and deleting levels

While you're looking at the *Edit* menu, you should also note the other items available there. Some are "menu" versions of the the pushbuttons on the toolbar, such as *Level Startup Text, Level Border colors,* and *Select class/image to add...* There are also:

New level: adds a new blank level at the "end" of the already existing levels.
Cut level: copies the current level to the clipboard and deletes it from the puzzle set.
Copy level: copies the current level to the clipboard.
Paste level: inserts the level from the clipboard into the puzzle set BEFORE the current level.
Delete level: deletes the current level without copying it to the clipboard.
Clear playfield: removes ALL objects from the current level, without actually deleting the level.

Cut, Copy, and Paste levels can be used to rearrange levels, moving them to a different sequence, or making a copy of a level when you want to try a variation without losing the original design. These can also be used to move a level from one puzzle set to another, although great care should be taken doing this, because if the two puzzle sets don't have identical Class definitions, the results may not be you expect or desire. To do this, you would Copy (or Cut) the desired level to the clipboard, then return to game play mode and select the other puzzle set. Once the other puzzle set is loaded, then return to Edit Levels mode, change to the level in front of which you wish to insert the level from the first puzzle set, and then select the Paste Level function.

The Object Selection dialog

The object select dialog allows you to select the objects you wish to add to your level, and also lets you specify what functionality you wish to be mapped to various buttons on your mouse. Three button "mice" are supported, but if you have a two button mouse,

you can emulate the middle button by holding down the SHIFT key while using the left mouse button. You can select a function for each of the three mouse buttons, as well as for each of the three mouse buttons with the CTRL key held down. *Add* and *Delete* should be fairly obvious. *Move* allows you to move objects that have already been placed on the playfield without having to delete them from one location and then add them to another. Just point at the object to be moved, then hold down the button to which you have mapped the *Move* function, and then drag the object to the new location. *This dialog* refers to the Object Selection dialog, and is an alternate way of bringing up the dialog (instead of clicking on the pushbutton on the toolbar). *Obj MiscVars* brings up the Misc Vars dialog with values set according to the object that you're pointing at when you invoke that functionality. For more on Misc Vars, see Level Editing: Misc Vars. *Obj Help* causes a window to pop up that contains information about the object that you were pointing at when you clicked this button. This help is defined by the person who created the class, and may or may not have anything important to say. It's intent is to tell the level designer (that's you) about any special requirements necessary for objects of this class. These usually pertain to the setting of the Misc Vars, but may also mention any limits on the numbers of objects of this class placed on a level, or how they're to be used in combination with other classes of objects. You can also bring up this help while in the Object Selection dialog by clicking once on an image in the image selection window (to select it) and then clicking on the *How to use* pushbutton.

When you add an object to the playfield, you're specifying the Class of the object, the first image that it should use at the start of the level (frequently there is only one image, but some objects like Rollers and Exits have multiple images from which you can select), the LastDir value (last direction in which the object has moved, in this case the direction in which it is "pointing" at the beginning of the level), and the Misc Vars for the object. Many objects don't care about LastDir or the MiscVars, in which case all you need do is select its image and add it to the playfield. But if an object cares about any of those additional values, then you should ensure that they are set correctly before adding the object to the level.

Editing the level Startup Text

The Startup Text for the level is edited by selecting the *Edit-Level Startup Text...* menu item. (I try to keep these things obvious.) This brings up a dialog that includes some brief notes about special formatting that can be included in the startup text. All special formatting sequences are started with a backslash ('\') character, and the backslash is followed by another character which determines what the special formatting is. This is known as WYSINWYG (What You See Is Not What You Get) text editing. Experiment with the formatting 'til your heart's content. The startup text is limited to less than 4096 characters, which should be enough for all but the most verbose of you.

Saving your new levels

When you return to game play mode, or when you quit the program, if you have made any changes to the levels or classes, they will automatically be saved to disk. Depending upon the state of the *Options-Prompt before saving edits* menu item, you

may or may not be prompted first whether you wish to save the changes. ***All*** information related to a puzzle set is stored in a single file whose name is the name of your puzzle set with a file extension of ".MB" (Mesh Binary). It is stored in the main Mesh directory (folder), and it includes level definitions, class definitions (images, attributes, code, user defined sounds, etc), and any existing solutions to the levels. If you wish to share your new puzzle set with someone, that single .MB file is all you need to give them. However, you may wish to compress it first into a ZIP file, as the uncompressed file can be quite large. Generally, .MB files will compress down to almost 1/10 of their uncompressed size.

The maximum number of levels in a single puzzle set is 1024, although in reality you'll never want to have more than a couple of hundred, as the .MB file becomes VERY large VERY fast as you add more and more levels.

NOTE: There are four "exit sounds" built into the Hero's Hearts game. The sound that is played when the Hero steps on the Exit is determined by the position of the Exit on the playfield. It's a combination of the X and Y locations of the Exit. So, if you want a particular "exit sound" to be played on a particular level, then position the Exit first, and move it around 1 place up and down, left and right, until you get the sound you want. Then, build the rest of the level around it. Kind of a backwards way of doing things perhaps, but that's an option that you have.

Borders and Color Schemes

Borders are drawn in all playfield locations that contain no objects but DO have objects located directly next to them. The border is drawn using colors specified in the *Border Colors* dialog, which is accessible in the Level Editor, either under the Edit menu or via the button on the toolbar that has a blue colored border drawn on it.

Border colors are specified separately for each level, so you may change the border colors with each level if you wish, or have all levels in a puzzle set share the same border colors. I recommend the latter, to give the puzzle set a cohesiveness, a common tone, unless changing the border colors gives some indication of change in the puzzle set (such as how far you've progressed through the puzzle set or how difficult the particular level is). However this choice is, of course, completely up to you as the level designer. Frequently, you may wish to coordinate the border color with an image used on that level for the "Field" (or ground) object.

To see how the border colors are used to draw the border, let's look at a simple example. Imagine that, in the diagram below, the 'b' represents locations having NO objects NOR any adjacent objects, 'B' represents locations having NO objects but HAVING adjacent objects, and 'P' representing locations HAVING objects.

```
bbb
BBB
PPP
```

Then 'b' locations will be completely filled with the *right-most* border color from the Border Color dialog. The 'B' locations will be filled by drawing a series of horizontal lines, starting from the bottom (next to the 'P') and moving upwards (towards the 'b'). The colors used will vary with the size of images the player has selected. The Border Color dialog shows you an example for each of the six possible image sizes. Only the half of the 'B' location that is closest to the 'P' location will be filled with this series of differently colored horizontal lines. The upper half (in this example) of the 'B' location will be filled entirely with the *right-most* border color from the Border Color dialog, just like the 'b' location. This provides the ability to control the "background" color of locations when the playfield is not completely full, and to have the border gradually fade into that background, if you wish.

The bottom portion of the Border Colors dialog shows which colors have been selected for the 33 possible border colors, and also shows which of those 33 colors are actually used at different image sizes. The image sizes are listed on the left, and a black dot beneath a color indicates that the color is used for drawing borders at that image size. Care must be taken when specifying the border colors so that the border will look similar and reasonable (ie, good) at all image sizes. This will frequently mean duplicating the same border color in two or more adjacent locations, but that is entirely up to you. Usually it's best if the four *left-most* colors are the same, as that will ensure that that color will appear at ALL image sizes next to the playfield, providing a consistent "edge" between the playfield and the border. Remember that the *right-most* color is used as

the background. It seems to work best if you specify the left-most and right-most colors first, and then work your way "inwards" to the center of the border.

You select a color from the color palette by left-clicking on the desired color. The current color is denoted with a white rectangle. The border colors are changed by left-clicking on the border colors at the bottom of the dialog. Right-clicking on a border color at the bottom of the dialog will select that color in the color palette.

The DEFAULTS pushbutton loads the default "neon blue" border. The IMPORT pushbutton opens a dialog that lets you select a border from any of the other levels in the current puzzle set.

Level Editing: Misc Vars

In order to include flexibility in the Mesh game engine for the addition of new classes that might have special "initialization requirements", each object on the playfield has three "miscellaneous variables" that are not pre-defined (nor used) by the game engine. These three variables are called (surprise, surprise) Misc1, Misc2, and Misc3. These contain values between 0 and 65,535, and can represent just a numeric value, a class, a message, or (in the case of Misc1 only) a user-defined string.

When setting one of these variables to a class, message, or user string, the value edit box is replaced by a combo box that lets you select the class, message, or user string of your choice.

In the case of user strings, two pushbuttons are available: *NewStr* and *EditStr*. NewStr adds a new user defined string to this level, whereas EditStr edits an already existing string. These strings are very similar to the Level Startup Text, with the additional feature that you can include numeric formatting of the Misc2 and Misc3 values into the popup window. These user strings are used by an objects class code to pop up a window under certain circumstances. An example of this is the UserNote class which, when the Hero steps on the UserNote, displays the user string pointed to by Misc1 for that particular UserNote.

Another example of the use of Misc Vars, although it's not made use of in the objects that come with Hero's Hearts, is the Teleport and Roller classes. Setting Misc1 to a non-zero value causes the Teleport or Roller to be "off", that is disabled. This feature is included because future classes of objects will offer the ability for the Hero to turn selected Teleports and Rollers on and/or off. The Misc1 variable is a means for controlling whether the Teleport or Roller is on or off *at the beginning of the level*. This way, you can have some Teleports and Rollers on the level ACTIVE (on) at the beginning of the level, and some of them INactive (off).

The settings of the Misc Vars are "global" in the sense that when you change their settings and then add objects to the level, those Misc Var settings will be the same for ALL objects added from that point on until you change the Misc Var settings again. Normally, this won't matter. However, if you (for example) added a UserNote with a string that was NOT the first ("zero-th") string for the level (so that Misc1 would be set to a non-zero value), and then added a Teleport or Roller without going into Misc Vars and changing Misc1 back to 0, then that Teleport or Roller that you added would be disabled (off).

If you need to change the Misc Var settings for an object *after* it's been added to the level, you can do that by clicking the appropriate mouse button combination (as defined in the Object Selection dialog) for invoking the *Obj MiscVars* dialog while pointing the mouse at the object which you wish to edit. This will cause the Misc Vars dialog to be displayed with the Misc Vars values set to that particular objects current state.

Another thing worth mentioning is that most classes of objects that have a "direction" to

them (such as Rollers, Creepers, Worms, and Arrows) don't use the "LastDir" setting to initialize their starting direction, but rather (in their Class Codes) look at the "First Image" that you selected when you added them to the level, and set their LastDir to match the image. This saves you from having to make sure you have LastDir set correctly all the time with these "common" objects, and ensures proper behavior of the objects. MOST classes of objects that have a directional sense AND have images that match each of those possible directions SHOULD be implemented to work this way in order to simplify the level creators job.

Statement: Animate

Description:

Animates (displays a series of images at regular time intervals) an object on the playfield. A number of animation methods are available:

1) Single time through the sequence, using ascending image numbers (*firstimg*<*lastimg, flags*=ANI_ONCE)
2) Single time through the sequence, using descending image numbers (*firstimg*>*lastimg, flags*=ANI_ONCE)
3) "Oscillate" once through a sequence, first ascending, then descending, then stopping when the sequence reaches the starting point again (*firstimg*<*lastimg, flags*=ANI_ONCE | ANI_OSC)
4) "Oscillate" once through a sequence, first descending, then ascending, then stopping when the sequence reaches the starting point again (*firstimg*>*lastimg, flags*=ANI_ONCE | ANI_OSC)
5) Loop indefinitely through an ascending sequence, starting over at *firstimg* after *lastimg* has been reached and displayed. (*firstimg*<*lastimg, flags*=ANI_LOOP)
6) Loop indefinitely through a descending sequence, starting over at *firstimg* after *lastimg* has been reached and displayed. (*firstimg*>*lastimg, flags*=ANI_LOOP)
7) "Oscillate" indefinitely through a sequence, first ascending, then descending, then ascending again, etc. (*firstimg*<*lastimg, flags*=ANI_LOOP | ANI_OSC)
8) "Oscillate" indefinitely through a sequence, first descending, then ascending, then descending again, etc. (*firstimg*>*lastimg, flags*=ANI_LOOP | ANI_OSC)
9) Stop all animation, setting the current image to *firstimg*. (*firstimg*=the desired image to be shown when the animation stops, *flags*=ANI_STOP)

The ONLY way to stop a looping animation sequence is with an Animate(ANI_STOP...) call. The *CurImage* standard variable may be set at any time during an animation, but that may confuse things a bit. It is recommended that you only use *CurImage* to change the current image when Animate() is NOT active. Setting *CurImage* while an Animate() is active will NOT stop the animation sequence.

NOTE: use animation where it has the most effect. An entire level of animated objects will run slowly, jerkily, on slower computers. Also, there are only 512 images available per puzzle set, so if you have 16 images per class, that limits the number of classes in your puzzle set to 32.

Syntax: Animate(*flags, firstimg, lastimg, timing*)

flags	any one of ANI_ONCE, ANI_LOOP, or ANI_STOP. You may also optionally combine ANI_ONCE or ANI_LOOP with ANI_OSC using the bitwise OR operator.
firstimg	the first image in the animation sequence

lastimg the last image in the animation sequence
timing the delay between successive images in the animation sequence. The count is the number of 1/100ths of a second. The animation may actually run SLOWER than that, depending upon the processing power of the computer and the amount of animation being done on a particular level, but it will never run any faster than that, regardless of the speed of the computer.

Examples:

Animate(ANI_ONCE, 1, 7, 10)
Animate(ANI_LOOP, 0, 3, 5)
Animate(ANI_ONCE | ANI_OSC, 1,15, 10)
Animate(ANI_ONCE, 5, 0, 8)

See also: Class Editing: Class Codes - Standard Variables (CurImage)

Statement: CallSub

Description:

Calls (passes control to) the specified subroutine in the SUBS code block. When execution of that subroutine encounters a *return* statement or the end of the SUBS code block, execution will return to the statement following the *CallSub* statement. The label which is the target of the *CallSub* must be in the SUBS code block, not in a MSG code block. *CallSub* can be executed from within the SUBS code block to another subroutine (or itself, if you want to get recursive) in the SUBS code block.

No parameters are passed to a subroutine, nor any values returned, except through the use of the global User Variables (VARS) for the class.

Syntax: CallSub *label*

label The target subroutine entry point in the SUBS code block. This label is limited to 7 characters, and may contain only alphanumeric characters and the underscore character.

Examples:

```
CallSub chkdead
CallSub MoveMe
```

See also: return

Statement: DelInventory

Description:

If the inventory contains an item with the specified *class* and *image* number, then that inventory item is deleted.

Syntax: DelInventory(*class, image*)

class	The class of the inventory item to delete.
image	The image number of the inventory item to delete.

Examples:

DelInventory(KeyYellow, 0)

See also: SetInventory

Statement: FlushClass

Description:

For all objects on the playfield with the specified class, clears all "moved/arrived/departed" flags and sets *Inertia* variable to 0. Use with caution, as the "moved/arrived/departed" flags are how the system knows when it's supposed to send MSG_MOVED, MSG_ARRIVED, and MSG_DEPARTED messages to an object.

Syntax: FlushClass(*class*)

class The class of the objects to flush. If -1, then ALL objects are flushed, and also prevents a MSG_BEGIN_TURN from being sent if HeroClass is the one doing the FlushClass() as the result of a MSG_KEY. Normally, IgnoreKey() is used rather than FlushClass().

Examples:

FlushClass(Rock)

See also: FlushObj

Statement: FlushObj

Description:

Clears the "moved/arrived/departed/busy" flags for the specified object and sets the *Inertia* variable to 0. Use with caution, as the "moved/arrived/departed" flags are how the system knows when it's supposed to send MSG_MOVED, MSG_ARRIVED, and MSG_DEPARTED messages to an object.

Syntax: FlushObj(*obj*)

obj A pointer to the object to flush.

Examples:

FlushObj(Self)

See also: FlushClass

Statement: ForEachObjAt

Description:

Executes a section of code on each object at the given location. A specified user-defined variable (from the VARS block) is set as a pointer to the "current object" at the beginning of each loop, so the code must be written to reference that user variable in order to operate upon each successive object at the given location.

The ForEachObjAt() statement requires the use of {}'s, single line statements after the ForEachObjAt() statement are not supported.

You may use Goto to break *out* of a ForEachObjAt loop, but should NOT use Goto to jump *into* one. The variables involved would be undefined in this case, and the behavior would NOT be what you would like.

ForEachObjAt clears all of the (internal) OF_DONE flags for all objects at the target location (ALL objects get their OF_DONE flag cleared at each key press along with the KeyCleared flag). OF_DONE is used ONLY by the ForEachObjAt function to keep track of which objects have been processed and which ones haven't, as objects may be moved, created, or destroyed during the execution of any loop of the ForEachObjAt. Then, ForEachObjAt jumps to the "bottom of the loop" (an invisible token at the end of the loop that does the "find the next object and jump to the
top of the loop" code. It searches from either the bottom or the top of the stack of objects (depending upon the upflag that's passed in) for the next object that doesn't have the OF_DONE flag set. If one is not found, it falls out of the loop. If one is found, it sets the OF_DONE flag for that object, sets the user-defined variable pointing to that object, and then jumps to the top of the loop and executes the loop code on that object.

So, any new objects created at the location during the execution of any passes through the loop will ALSO get processed by the loop, but any objects moved away from the location or destroyed BEFORE their pass through the loop will NOT get processed by the loop. Obviously, ALL objects created in the loop will get processed by the loop, but you have little control or knowledge of whether a moved or destroyed object has been processed in the loop yet or not (if you care).

Syntax: ForEachObjAt(*upflag*, *userVAR*, *x, y*) {
...inane statements of your choice...
}

upflag non-0 to go through the "stack" of objects from the bottom-up, 0 to got through the "stack" of objects from the top-down.

userVAR The VARS user-defined variable which will be set pointing to each successive object at the start of each time through the loop.

x, y The (X,Y) playfield location of the objects on which the loop will occur.

Examples:

```
ForEachObjAt(1, obj, Xloc, Yloc) {
if( obj!=Self && obj.Class!=Field ) Destroy(obj)
}
```

See also: Goto

Statement: GotoLevel

Description:

Causes the Mesh game engine to initialize the game to the beginning of the specified level. Goto(Level) will restart the current level. Goto(Level+1) will go to the next level.

NOTE: Execution of the class codes does NOT resume after the GotoLevel() statement. Starting a new level causes the destruction (deallocation) of data storage for ALL objects on the current level, so that the new objects on the new level can be allocated and linked into the playfield. Do NOT expect any further code to happen after executing a GotoLevel.

Syntax: Goto(*levelnum*)

levelnum The level number of the level to be initialized. This number is "0 based", ie. it's one less than the level number displayed on the toolbar. Valid values are 0 through LevelCount-1.

Examples:

```
GotoLevel(Level)
GotoLevel(Level-1)
GotoLevel(Level+1)
GotoLevel(10)
```

See also: Level, LevelCount, WinLevel

Statement: IgnoreKey

Description:

To be used during the execution of the MSG_KEY code block, this statement causes the current key to be ignored by the Mesh game engine (not recorded in the players move list). Exercise GREAT caution using this. If you ignore a key which changes the state of the game, then the "replay" feature will be broken.

Syntax: IgnoreKey()

Examples:

```
if( Key==KEY_BACK ) {
IgnoreKey()
LocateMe()
return 1
}
```

See also:

Statement: Link

Description:

Used to link two puzzle sets together. Causes the Mesh game engine to load the specified puzzle set, replacing the current one, and initialize the specified level number. If you have multiple puzzle set files that link together, it is recommended that you assign a unique PuzzleNumber to each one. This is done in the Level Editor under the *Edit* menu, the *Set puzzle set number...* menu item.

NOTE: Execution of the class codes does NOT resume after the Link() statement. Loading a new puzzle set (which loads completely new class codes) causes the destruction (deallocation) of data storage for ALL objects on the current level, so that the new objects on the new level can be allocated and linked into the playfield. Do NOT expect any further code to happen after executing a Link().

Syntax: Link(*puzsetname, levelnum*)

puzsetname The name (as a quoted string) of the target puzzle set.

levelnum The level number to start in the target puzzle set. This number is "0 based", with 0 being the same as the level shown as 1 on the toolbar.

Examples:

```
MSG_ARRIVED {
if( ObjClassAt(Hero, Xloc, Yloc) ) {
        WinLevel()
        if( Level==LevelCount-1 ) {
                if( PuzzleNumber==0 ) Link("HEARTS2", 0)
                else if( PuzzleNumber==1 ) Link("HEARTS3", 0)
                else if( PuzzleNumber==2 ) Link("HEARTS4", 0)
                else Popup("Major Bummer, Dude!  You're out of levels!!!")
        } else GotoLevel(Level+1)
}
}
```

See also: GotoLevel, WinLevel, Class Editing: Class Codes - Standard Variables (PuzzleNumber)

Statement: LocateMe

Description:

Draws a "sonar" on the playfield to help locate the player object (the Hero). See the Hero class code, the MSG_KEY code block where it tests for the KEY_BACK key for an example of this. Although this function was included to aid in locating the Hero object at the start of a level, this statement can be used by any class for any purpose.

Syntax: LocateMe()

Examples:

```
if( Key==KEY_BACK ) {
IgnoreKey()
LocateMe()
return 1
}
```

See also:

Statement: Popup

Description:

Displays a window containing whatever you would like it to contain, just so long as what you want it to contain is something that is supported by the Mesh game engine.
All of the same formatting options may be used in level Startup Text strings, except for those requiring arguments from an argument list (%u, %x, and %i).
The location of the popup window (within the Mesh window) can be specified using the PopupLoc() statement. The color of the window and the colors available for use as text within the window can be specified with the PopupColor() statement.

The *string* argument is a quoted string containing text and/or formatting sequences. The text is displayed "as is". Popup's are limited to not more than 50 lines and not more than 4096 characters total. Allowed special formatting sequences:

%u	formats the next argument list item as a decimal number
%x	formats the next argument list item as a hexidecimal number
%i	uses the next two argument list items (as *class* and *image number*) to display a class image.
\n	causes a "new line" (following text begins at the start of the next line)
\c	causes the current and following lines to be centered horizontally.
\l	(lowercase L) causes the current and following lines to be left justified.
\iCLASSNAME:IMGNUM\	inserts the specified image, similarly to %i, except that the literal class name is specified along with the actual image number.
\b	causes a new line, then a horizontal line drawn across the message box, with following text starting at the beginning of the line below the horizontal line.
\qX	causes a Quiz button in the Popup window, where X is the character that selects that button. Multiple \qX's could be used within a single popup to provide a mulitple choice question. The key pressed in response to this popup is sent to the object which executed the Popup() statement as a MSG_KEY message, even if that class of object does not have "Receives keys" checked in it's Attributes dialog. This is intended for use in "educational" test puzzles, for selecting items from the Hero's inventory, or whatever your heart desires. You can also click on the Quiz buttons with the mouse to select the one you want.
\0	causes the following text to be BLACK
\1	causes the following text to be BLUE
\2	causes the following text to be GREEN
\3	causes the following text to be CYAN

\4 causes the following text to be RED
\5 causes the following text to be VIOLET
\6 causes the following text to be YELLOW
\7 causes the following text to be WHITE

The actual colors selected by \0 through \7 *may* be different if your class codes have previously executed a PopupColor() statement.

Syntax: Popup("*string*" [,*arg1* [,*arg2* [,...]]])

string a quoted string containing the text and formatting information to be displayed in the message window.

arg1, arg2, etc optional arguments as required by the formatting in *string.*

Examples:

```
Popup("You've made a BIG mistake!")
Popup("Height=%u Climb=%u", Height, Climb)
Popup("My class is %i", Class, 0)
Popup("The Exit looks like this \iExit:0\ or like this \iExit:1\")
```

See also: PopupMisc, PopupLoc, PopupColor

Statement: PopupColor

Description:

Allows you to change the colors used by a Popup() or PopupMisc() window. The defaults are exciting technicolor grays, so I'm not sure why you would *want* to change them, but here's the statement that will take care of you!

There are 11 different color settings (indexes 0 through 10), which include the 8 possible text colors, the window background color, and the window highlight/lowlight "edge" colors. The default values are:

INDEX	COLOR	R	G	B	USED FOR
0	black	0	0	0	text \0 color
1	blue	0	0	255	text \1 color
2	green	0	255	0	text \2 color
3	cyan	0	255	255	text \3 color
4	red	255	0	0	text \4 color
5	violet	255	0	255	text \5 color
6	yellow	255	255	0	text \6 color
7	white	255	255	255	text \7 color
8	white	255	255	255	window highlight
9	gray	187	187	187	window background
10	black	0	0	0	window lowlight

If you wish to change these colors for the Level Startup Text, it is recommended that you execute the appropriate PopupColor() statements in the MSG_INIT or MSG_POSTINIT code blocks of your "player object" (Hero), as those messages are sent *just* before the Startup Text for the level is displayed. The popup colors are reset to their default values at the start of each level.

Syntax: PopupColor(*index, red, green, blue*)

index	Which popup color you're changing (0-10)
red, green, blue	The RGB components of the color to be used (0-255).

Examples:

```
PopupColor(0, 255, 255, 255)
PopupColor(10, 255, 0, 255)
```

See also: Popup, PopupMisc, PopupLoc

Statement: PopupLoc

Description:

Determines where subsequent Popup() and PopupMisc() windows will appear relative to the Mesh game window. The PopupLoc() arguments are combined to specify these locations:

FLAGS	X-positioning	Y-positioning
0	centered in view	centered in view
1	right edge at x	centered in view
2	left edge at x	centered in view
3	centered at x	centered in view
4	centered in view	bottom edge at y
5	right edge at x	bottom edge at y
6	left edge at x	bottom edge at y
7	centered at x	bottom edge at y
8	centered in view	top edge at y
9	right edge at x	top edge at y
10	left edge at x	top edge at y
11	centered at x	top edge at y
12	centered in view	centered at y
13	right edge at x	centered at y
14	left edge at x	centered at y
15	centered at x	centered at y

Syntax: PopupLoc(*flags, x, y*)

flags	what type of positioning to use
x, y	possible playfield location to use for positioning the window.

Examples:

```
PopupLoc(0, 0, 0)
PopupLoc(11, Xloc, Yloc+1)
```

See also: Popup, PopupMisc, PopupColor

Statement: PopupMisc

Description:

Displays a popup window whose text is the Level user-defined string specified by the Misc1 variable for the object executing the PopupMisc() statement. In pseudo-code, PopupMisc() is effectively the same as Popup(LevelString(Misc1), Misc2, Misc3). There is no such function as LevelString(), that's why it's pseudo-code. However, this demonstrates that the user-defined string can contain two %u or %x formatting sequences and Misc2 and Misc3 will get formatted there. Otherwise, the formatting string is very much the same as for the Popup() function.

The location of the popup window (within the Mesh window) can be specified using the PopupLoc() statement. The color of the window and the colors available for use as text within the window can be specified with the PopupColor() statement.

The *string* argument is a quoted string containing text and/or formatting sequences. The text is displayed "as is". Popup's are limited to not more than 50 lines and not more than 4096 characters total. Allowed special formatting sequences:

%u	formats the next argument list item as a decimal number
%x	formats the next argument list item as a hexidecimal number
%i	uses the next two argument list items (as *class* and *image number*) to display a class image.
\n	causes a "new line" (following text begins at the start of the next line)
\c	causes the current and following lines to be centered horizontally.
\l	(lowercase L) causes the current and following lines to be left justified.
\iCLASSNAME:IMGNUM\	inserts the specified image, similarly to %i, except that the literal class name is specified along with the actual image number.
\b	causes a new line, then a horizontal line drawn across the message box, with following text starting at the beginning of the line below the horizontal line.
\qX	causes a Quiz button in the Popup window, where X is the character that selects that button. Multiple \qX's could be used within a single popup to provide a mulitple choice question. The key pressed in response to this popup is sent to the object which executed the Popup() statement as a MSG_KEY message, even if that class of object does not have "Receives keys" checked in it's Attributes dialog. This is intended for use in "educational" test puzzles, for selecting items from the Hero's inventory, or whatever your heart desires. You can also click on the Quiz buttons with the mouse to select the one you want.

\0	causes the following text to be BLACK
\1	causes the following text to be BLUE
\2	causes the following text to be GREEN
\3	causes the following text to be CYAN
\4	causes the following text to be RED
\5	causes the following text to be VIOLET
\6	causes the following text to be YELLOW
\7	causes the following text to be WHITE

The actual colors selected by \0 through \7 *may* be different if your class codes have previously executed a PopupColor() statement.

Syntax: PopupMisc()

Examples:

PopupMisc()

See also: Popup, PopupLoc, PopupColor

Statement: SetInventory

Description:

This adds an item to the inventory window (if it's not already there) and sets the count value of an item in the inventory window. This is for user interface purposes only. The Mesh game engine does nothing with this (other than display the window). It is up to your class codes to make sure that the contents of the inventory window match your version of reality. The inventory window is automatically cleared (all items deleted) at the start of each level.

Syntax: SetInventory(*class, imagenum, count*)

class	The class of the image to be displayed.
imagenum	The image number to be displayed.
count	The count (from 0 to 99) to be displayed beside the image.

Examples:

SetInventory(KeyYellow, 0, ykeycnt)

See also: DelInventory

Statement: Sound

Description:

Makes a sound. The sound can either be a predefined sound (internal to the Mesh game engine) or one added by you as a user-defined sound (a WAV file). See Class Editing: Class Codes - Constants for a list of the available internal sounds.

To add your own user defined sounds, use the *Edit-Sounds...* menu item or the "musical note" pushbutton in the Class Editor to bring up the Sounds dialog. This dialog has pushbuttons for adding, deleting, replacing, renaming, and exporting user-defined (WAV file) sounds. These user-defined sounds that are added to the game are stored in the .MB file, so the individual WAV files do not need to be kept or distributed with the .MB file. **BE CAREFUL ADDING USER-DEFINED SOUNDS: THEY CAN BE QUITE LARGE AND CAUSE "FILE BLOAT" TO OCCUR.**

Note: The predefined sounds are "grouped" in the Sound dialog roughly by "type". All "human" sounds are together, all "animal" sounds are together, etc. This organization is intended to make it a little easier to find a sound that fits your situation.

Syntax: Sound(*snd_name, interrupt*)

snd_name	The name of the sound to be played.
interrupt	non-zero if this sound is to interrupt (pre-empt) any sound already playing. Only do this on VERY important sounds, as otherwise the sound becomes very "hashed". For most sounds this should be 0.

Examples:

```
Sound(SND_UH_OH, 1)
Sound(USR_HI_THERE, 1)
Sound(SND_HEARTBEAT, 0)
```

See also: Class Editing: Class Codes - Constants

Statement: Trace

Description:

Cause a fake MSG to be entered into the Trace Window trace buffer. The message will be displayed as "**** TRACE ****", MsgTo will be the object executing the Trace() statement, MsgFrom will be the *obj* argument of the Trace() statement, and MsgArg1 and MsgArg2 will be the *arg1* and *arg2* Trace arguments. This is intended for use as a debugging tool (and, indeed, is pretty worthless for anything else). See Class Editing: Message Tracing for more information.

Syntax: Trace(*obj, arg1, arg2*)

obj	A pointer to an object. MUST not be 0 or a GPF will occur.
arg1, arg2	Displayed in the Trace Window as MsgArg1 and MsgArg2 for this "message".

Examples:

```
Trace(Self, somevar, whatval)
Trace(ObjBelow(Self), ObjBelow(Self).Class, ObjBelow(Self).Temperature)
```

See also: Class Editing: Message Tracing

Statement: WinLevel

Description:

Tells the Mesh game engine that the level has been solved, and that the players move list should be copied into the solution move list for the level if it's shorter than the previous solution or if there was no previous solution. Does not advance the game to the next level. GotoLevel must be called AFTER WinLevel in order to change the level.

Syntax: WinLevel()

Examples:

```
MSG_ARRIVED {
if( ObjClassAt(Hero, Xloc, Yloc) ) {
        WinLevel()
        if( Level==LevelCount-1 ) {
                if( PuzzleNumber==0 ) Link("HEARTS2", 0)
                else if( PuzzleNumber==1 ) Link("HEARTS3", 0)
                else if( PuzzleNumber==2 ) Link("HEARTS4", 0)
                else Popup("Major Bummer, Dude!  You're out of levels!!!")
        } else GotoLevel(Level+1)
}
}
```

See also: GotoLevel

Statement: goto

Description:

Causes execution to continue at the specified code label. If the *goto* statement is in the SUBS code block, then the target label must also be in the SUBS block. If the *goto* statement is in a MSG code block, then the label must also be in a MSG code block, but doesn't necessarily have to be in the same MSG code block. Frequently, you want the same code to be executed for two different messages. The *goto* statement allows one message code block to jump to another. Very structured, very object oriented. If you don't like it, don't use it. :-)

Syntax: goto *label*

label The label in the MSG or SUBS code block which is the target of the *goto* statement.

Examples:

```
MSG_MOVED {
trymore:
...misc statements...
goto trymore
}
```

See also: CallSub, return, if-else

Statement: if-else

Description:

This is the conditional execution statement. Similar to the C language if-else statement EXCEPT that it does not use "lazy evaluation". The ENTIRE conditional expression is evaluated instead of aborting as soon as it becomes false. Also, due to a weakness in the decompiler, excess parentheses are sometimes thrown in. Sorry about that.

The *if* and *else* statements can be simple or compound. Simple statements (one-line statements) must follow the *if* or the *else* on the same line. Compound (multi-line) statements must have the opening brace { on the same line as the *if* or the *else*, with the conditional statements on the following lines, and then the closing brace } on a line by itself. The exception to that is when the closing brace } is followed by *else* in which case the else must be on the same line as the preceding closing brace.

You may use *goto* to jump into and out of *if-else* statement blocks.

Syntax:

if(*expression*) statement

```
if( expression ) {
statement
statement
...
}
```

```
if( expression ) statement
else statment
```

```
if( expression ) statement
else {
statement
statement
...
}
```

```
if( expression) {
statement
...
} else {
statement
...
}
```

Well, you get the idea, I hope.

Examples:

After the above you want examples??? Sigh.

```
if( Create(HeroDead, Xloc, Yloc, 0, 0) ) Destroy(Self)
```

How's that?

See also: ForEachObjAt, goto

Statement: return

Description:

In a SUBS block, *return* causes class code execution to resume with the next statement following the most recent CallSub statement. In MSG blocks, *return* *value* causes the execution of the MSG code block to be terminated and the *value* to be returned as the message ...uh... return value.

Syntax:

return
return *retval*

retval a numeric expression whose value is returned from the MSG code block.

Examples:

```
SUBS {
somesub:
return
}
MSG_DESTROY {
CallSub somesub
return 0
}
```

See also: CallSub, Class Editing: Messages