The Debugger

Introduction.

The debugger is a tool to find errors or speed losses in the script that's executed. The power of a debugger is that you can monitor every single instruction that must be executed within the total sequence of instructions. This way you can actually see the changes caused by a single instruction, instead of seeing the changes after many instructions. Like the animation is the visualization of your simulation, the debugger is the visualization of the execution. And just like the animation slows down your simulation, the debugger slows down all execution. This is why the debugger is turned off when starting Enterprise Dynamics.
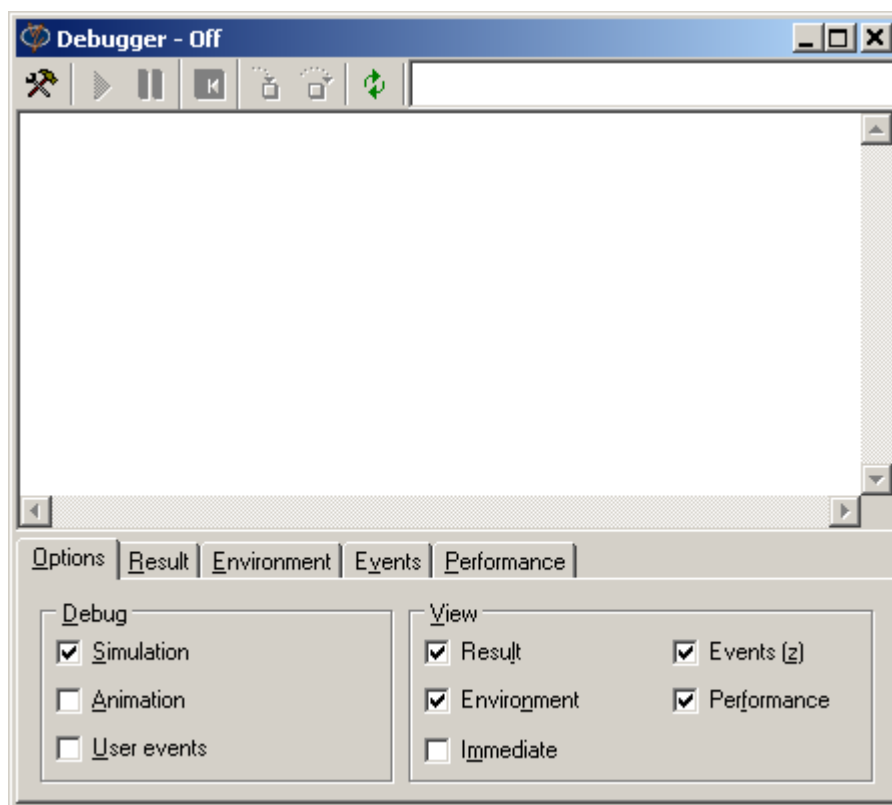
Like all other functionality in ED, the debugger has it own window. To show this window, you can use the 4DScript word DisplayDebugger with only one parameter: Show it (1) or hide it (0). To move or resize the window you can use the window reference Debugger.

EXAMPLE

```
Do(
 DisplayDebugger(1),
 Maximize(Debugger)
)
```

Description.

If you show the debugger you'll see the window below.

The window is splitted into three parts, from top to bottom: The toolbar, the source code displayer and the information. We will discribe each of these.

Part 1: The Toolbar.

The toolbar is at the top of the debugger window and it has seven buttons and one information field, and it looks like this:

From left to right, the buttons have the following functions:

Switch to turn the debugger on and off. In this picture the debugger is turned on (pressed). If the debugger is turned off, the other buttons are gray (disabled);

Run. Press this button to continue the normal execution. Pressing {F5} has the same effect;

Stop. Press this button to stop the execution as soon as there is code the be executed. Pressing this button when the system is idle causes the next code to be interrupted;

Cancel. Press this button to abort execution of the current script. If the debugger isn't executing any code, this button has no effect. Be carefull using this button, because the current execution is aborted completly, but when simulating in Unlimited speed, the next event will be executed immediatly.

Step-in. When debugging a sequence of instructions, this button forces the debugger to go into the instruction, and debug the instructions called inside it. Normally these other instructions are parameters of an instruction. Pressing {F7} has the same effect.

Step-over. When debugging a sequence of instructions, this button forces the debugger to skip the instructions used inside the current instruction. Pressing {Shift + F8} has the same effect.

Update. Press this button to update information shown in the information pages at the bottom of the window. Which pages there are, what's in these pages and how it works, is described in the third part of the window ('Information pages'). On some pages the keyboard can be used to press this button, for instance the Environment page, described below.

Part 2: The source code displayer:

This part can be found in the middle of the debugger window and looks like this:
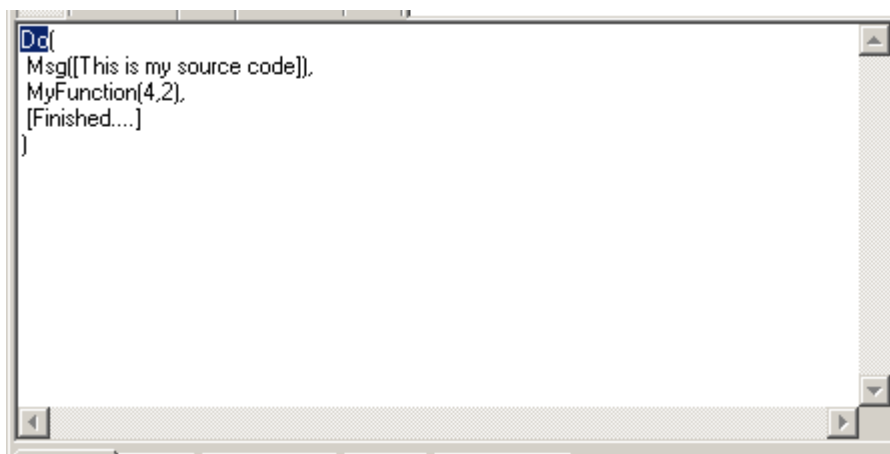
It shows the 4DScript code that you are debugging. This code differs from the code in the editors, because it is generated by the debugger. That's why the layout is set automatically, comments are skipped and code is written in the same form as the execution take place. This means that attributenames can be translated into indexed attributes, mathmatic operations can be rearranged, atomlabels and ':=' instructions are translated into functions. It looks different, but the result is the same.

If the selected part is a single word, like in this picture, then that word is the instruction that's about to be executed. If the selected part is a sequence of instructions, then this sequence is executed, but the debugger allows you to see the result. This result is shown in the Result page, described in the third part of the window ('Information pages').

The difference between a single word selection, and a sequence selection can be shown with the following example:

EXAMPLE

We'll show the source, when debugging:

2*(3+4)

After we step-in twice, it looks like this:

The debugger is about to execute the adding function '+', as a parameter of the multiplying function '*'. Now we step-in three times, or just step-over once, and both parameters '3' and '4' are execution. Now the selection looks like this:

This means that the function '+' is executed, and the result will be returned to the '*' function. If we step further, we will see that '*' will be finished as well:


The code inside the displayer can not be changed by the user, but you can change the selected part, just like in any other editor. However, changing the selection doesn't have any effect.

Part 3: The information pages:

The information pages can be found at be bottom of the debugger window. Every page can be shown by selecting the tab which is connected to the page and has its name on it. The tabs look like this:


All pages contain information, and sometimes it is better to resize the height of the pages. This can be done by putting your mouse between the source code displayer and the tabs, and when the mousecursor changes into a sort of resize cursor press the left mouse button and mouse the mouse up, until the pages are high enough and release the left mouse button.
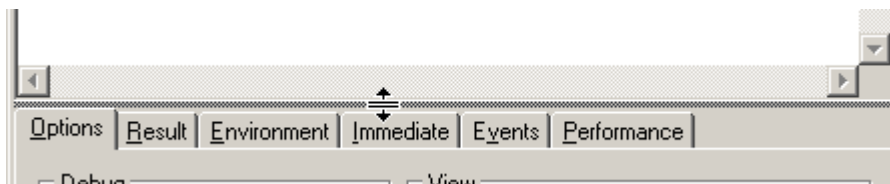
To explain why and how to use each page, we describe each page from the left tab to the right.

- **Options**:
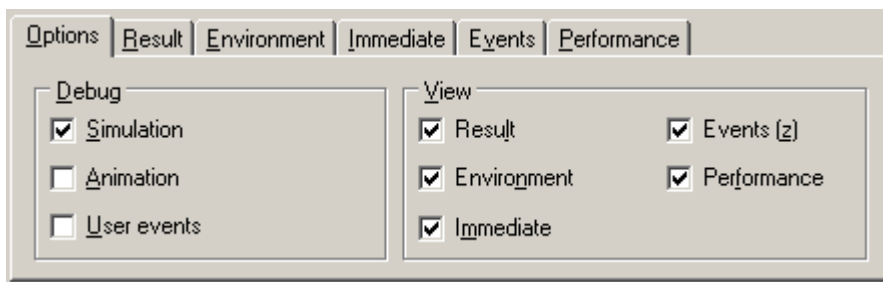
The options page has two groups of options: Debug and View.

The last group, the view options, contains switches to show and hide the corresponding pages. A checked item means that that page is available and the tab is shown. Unchecked items means that that page is not available and the tabs is hidden. In this picture all items are checked, so all pages are available. The options tab can not be hidden.

The first group, the debug options, contains switches for the kind of code the you want to debug. These kinds of code are:

1. *Simulation* code

   This is the default debug option. This kind contains the atom events OnEvent, OnEntering, OnEntered, OnExiting, OnCreation, OnDestruction, OnReset, OnOcReady, OnIcReady, OnMessage and OnInit. If this options is turned on, the entire simulation can be debugged.

2. *Animation* code

This kind contains the atom events On2dDraw and On4dDraw. If this option is turned on, the animation can be debugged, but only if there is at least one animation window. The animation windows are refreshed very often, so don't turn this option on, unless you really want to debug the animation.
3. *User event* code
This kind contains all code that must be executed as a result of the users action, like pressing an item in the menu, of pressing a button of a speedbar, but also the OnUser events of the atoms, the code typed in the Interactive window and all GUI events.
4. Not debuggable code
This kind contains all internal code, like DDE- and OLE-callbacks, and reading code from a file. The callbacks can not be debugged, because it could stall other applications. Also the code in the Immediate page, described below, can not be debugged.

- **Result**:

This page contains the result of last executed instruction. The result value is displayed in the upper box, the result text is displayed in the lower box. In this picture both results are available: value = 1, and text = 'Text result'.
The results are updated every time an instruction is finished. In the source code displayer this instruction is not selected anymore: the selection refers to the next waiting instruction, or the instruction that uses this instruction.
Changing the results is not possible, because the results are already returned to the calling instruction.
Use this page to monitor result while stepping through the 4DScript code, for instance when you want to find the reason for 'strange behaviour'.

- **Environment**:

This page contains atom information. Only the Active atom (4DScript word a), the Current atom (c) and the Involved atom (i) are available, and the order is fixed.
If the atom exists, it shows:
- the reference of the atom, which is the same as the result of execution the corresponding 4DScript word,
- the name of the atom,
- the ID of the atom, and
- the ID and name of the mother atom, if any.

It is possible that an atomreference is invalid, for instance when the involved atom is destroyed somewhere in the code of the event. In that case 'Invalid reference' is shown.

The information is updated after every Step-in or Step-over, but can be done manually by pressing the Update button in the toolbar.

This manually updating can also be done by pressing {F5}, if and only if the Environment information is active, which looks like:

Description	Reference	Name	ID	Mother
Active atom	1071513...	My Active Atom	2	1: My Mother At...
Current atom	1081999...	My Current Atom	3	2: My Active Atom

All columns can be resized by selecting the edges of the headers, are resize the column to the desired width. Double clicking on one of these edges automatically resizes the column, so all data should fit.

Description	Reference	Name	ID	Mother
Active atom	107151359	My Active Atom	2	1: My Mother At...
Current atom	108199935	My Current Atom	3	2: My Active Atom
Involved atom	109248511	aMy Involved Atom	4	3: My Current At...

The scrollbar at the bottom can appear if not all columns can be shown entirely. Resizing other columns or the entire window can remove it again.

- **Immediate**:

Execute
```
1
Text value
Msg[[This code is executed immediately]]
```

This page looks the same as the Interactive window. It also works the same way, even the {F5} key to execute the entered code, but only if the code editor is active.
There are a few differences.
- When debugging an instruction, the Interactive window can also be executed, but the atomreferences shown in the Environment page can not be changed. If the same code is executed in the Immediate page, these references can be changed.
- The code typed in the Immediate page can never be debugged. If it's an endless loop, it can not be interrupted.
Using this page can be usefull, if you want to change the atomreferences while debugging some code.

- **Events**:

Time	Code	Priority	Atom ID	Name	Involved
1	1	0	2	My Active Atom	No Invol...
1.5	2	0	3	My Current Atom	4
3	2	1000	5	My New Atom 2	No Invol...
3	1	0	2	My Active Atom	3
9	3	0	4	My New Atom 1	3

This page shows information about the waiting events. This information consists of the **time**, the **eventcode**, the **priority**, the **ID** and **name** of the atom and the ID of the **involved** atom. If there isn't an involved atom, 'No Involved Atom' is displayed.
This list is not updated automatically, so it has to be updated manually. This can be done by pressing the Update button in the toolbar or pressing {F5} if and only if the Event list is active, which looks like this:

Time	Code	Priority	Atom ID	Name	Involved
1	1	0	2	My Active Atom	No Invol...
1.5	2	0	3	My Current Atom	4

The list can be sorted by pressing the headers 'Time', 'Code', 'Priority', 'Atom ID', 'Name' or 'Involved'. This way searching events of an atom can be simplified.

All columns can be resized by selecting the edges of the headers, as described at the Environment page.

- **Performance**:

This page is usefull to monitor the performance of your simulation. The page has two important parts: analysis of time used for simulation vs. animation (Processortime), and analysis of 4DScript word usage (Count commands).
If you want to update this page automatically, select an **Update frequency**: Off (not updated), Very Low (once every 10 seconds), Low (once every 5 seconds), Normal (once every 2 seconds), High (every second) or Very High (twice every second). By default, the update frequency is Off.
Manually updating is done by pressing {F5}, but only if this page is active.

As we wrote above, this page has two parts, which allows us to monitor the performance in three different ways:
1. Performance monitor: Simulation vs. Animation
To start analyzing, make sure the Processortime switch is checked, and the page looks like this:

The best way to use it, is to choose Unlimited simulation speed, and open just one animation window, and just run your simulation.

EXAMPLE
First we turn on the Processortime switch on the Performance page and start the simulation. After some time your analysis look something like this:

Conclusion: Most of the time is used for simulation, and only a small part is used for the animation.

However, the performance page uses counters to monitor the performance. So if you're running for a long time, the simulation counter doesn't change much comparing to the animation counter. So if your model is slowing down after several simulation hours, it doesn't appear in this page. You have to press the Reset button to reset the counters: the green and red bar has turned white again. If this page is updated it should look like in the example above.

EXAMPLE
We'll do the same as in the previous example. But what if it looks like this:
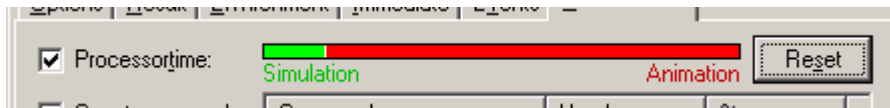
Conclusion: Most of the time is used for animation, instead of simulation. If you didn't use Unlimited simulation speed, this happens very often. But if you're running in Unlimited simulation speed, this means that the animation is much too time consuming, so it's better turn some parts off, or turn it entirely off.

Let's go on with the second part of this page, the Count of commands:

2. Performance monitor: 4DScript word usage

Let's start with the simple version. If you used the Processortime described above, please turn it off first. To turn on the counters for every 4DScript word, just make sure the Count command switch is checked. The page should look something like this:
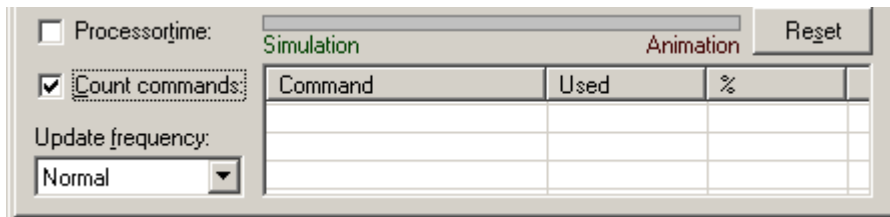
Whenever the page is updated, the used wordcounters are put in the list. If the word isn't in the list it's added at the bottom. The Command column contains the word, and the Used column the number of times this word is used. The last column stays empty. Also values, texts and variables are put in the list. After a while the list will be filled with data. After a while the list will look like:
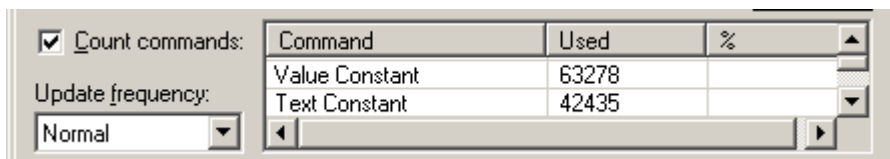
To show more words than the two shown above, just resize the Information pages as described at the beginning of the Information pages.

You can sort the data by pressing the headers 'Command' or 'Used', to sort them. The commands are sorted by alphabet, the counters are sorted from high to low (most used first). If the data ends with '…', you can resize the columns by selecting the edges of the headers, so the columns are wide enough for the data to fit, as described at the Environment page.

The counters will keep increasing, so after some time you should use the Reset button to clear the list and start counting from zero again.

The list can give you some information about the usage of 4DScript words. You'll see that a lot of words are used only if animation is turned on, and other words are only used if the simulation is running. But what if we want to see which commands use a lot of time:

3. Performance monitor: 4DScript Processortime usage

Combining both features described above, we can trace the time used for every instruction. To do this, turn on both Processortime and Count commands. It is possible that some counters have increased already before both switches have been turned on. If this happens the debugger doesn't trace the processortime for each instruction. So to make sure all counters are zero just press the Reset button.

After a while the list will be fill, and the %-column as well. For every word, this column is fill with the percentage of the total time needed to do all executions of that word.

First we turn on both Processortime and Count commands. Just to be sure, we press Reset, so all counters are set to zero. After some time, the performance page looks like this:
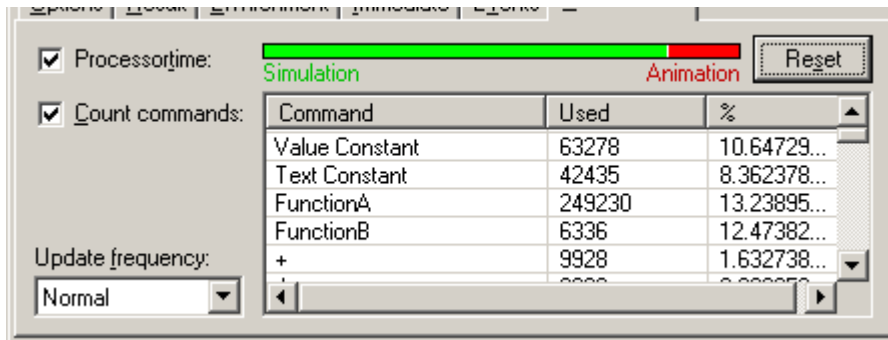
'FunctionA' is call 249230, and it took 13.238953…% of the total time. We compare this with 'FunctionB', which was executed only 6336 times. That it took 12.47382…% of the total time. Conclusion: A single FunctionA is faster than a single FunctionB.
Now we have three columns filled, which all three can be sorted by pressing the headers 'Command', 'Used' or '%'. The commands are sorted by alphabet, the counters and percentages are sorted from high to low.

When using these counters for a while, without resetting, very small values can appear. For example 0.0000012345678%: this will be displayed as 1.2345678E-6. The 'E' can be hidden, because the column is too small. This looks strange when sorting by percentage, because that value looks bigger than for instance 0.1, but this 0.1 is put somewhere above 1.2345….

To make sure all digits are visible, just resize the columns by selecting the edges of the headers, as described at the Environment page.

All parts of the debugger window are described. But there are more features:

Executing 4DScript while debugging other 4DScript

It is possible to execute some 4DScript code, while you debugging some other code. For example, when you are debugging some simulation code, all the items in the menu still work. This means you have to be careful:

Make sure that the code executed doesn't affect the code you're debugging, or about to debug. This can cause unpredicable behaviour, like *access violations* and *invalid pointer operations*. So don't press reset, new model, don't close any animation windows and so on, while debugging simulation or animation.

The best way of preventing this, is to turn off the debugger, using the switch in the toolbar. If you want to stop the simulation, just type Stop in the immediate and press the Execute button first. Then wait till it finishes execution. The instruction Exit can be used to terminate the execution at once and closes ED without any warning. An alternative of turning off the debugger is to press the Cancel button in the toolbar. This will abort the execution of the entire sequence.
If your animation takes too much time, don't close any animation windows while debugging, but follow the steps described above, and finally close the animation windows when ED is not debugging anymore. Then wait till the windows are closed.

How do I create a breakpoint?

Breakpoints can be set, using the 4DScript word EnterDebugger. Whenever this command is executed is checks whether the debugger is turned on. If you use the command with one parameter, not equal to zero, the debugger is forced to turn on. If the debugger isn't visible, it's shown and you can start debugging at the 4DScript code to be executed right after the EnterDebugger command.

 EXAMPLE
Do(
 Trace([This code is executed, without interruption]),
 EnterDebugger(1),
 Trace([This is the first statement to debug])
)

Stop execution at any time

If the debugger is turned on, the {Ctrl + Break} combination forces the debugger to stop execution as soon as possible. Then you can use the debugger to step through the current 4DScript code, or abort execution of the sequence. The last is usefull when you're in an endless loop.

Protected code

If you open an encrypted file, the code inside it is protected. This means that you can't debug it. However, you can stop it by pressing {Ctrl + Break}. The debugger asks you whether you want to abort this execution or continue. Stepping through this code is not possible.

This also applies to code executed while reading a file. You can abort it, but you can not debug it.

If there's a breakpoint in the protected code, which forces the debugger to turn on (like in the example), the error monitor will pop up and the error '**Can not enter debugger**' will appear.