

Introduction

Debuggifier is a computer program that makes it easier to debug a plugin written in the Ruby language for use with Google Sketchup ® version 8 or above. The program inserts commands where necessary in the Ruby plugin to call a debugging tool.

Later, when the “Debuggified” plugin is invoked, the debugging tool shows to the user a history of code lines, and variable values, and watches, and breakpoints, that were encountered as the plugin executed. The results are shown in the Ruby console and are printed in a text file (a ‘log file’). The user can set conditional watches and conditional breakpoints as the session progresses. The user can examine and set values of variables of the plugin. As as the Ruby plugin execution progresses, the user can determine where the program goes wrong. This is program debugging without an IDE. By adding many debugging commands to the plugin’s source code, the code thereby is ‘debuggified’. After debugging is complete, the user ‘Undebuggifies’ the plugin code automatically.

What Debuggifier can do

Debuggifier can

- Trace through the plugin line by line, showing the original Ruby code lines and the variable names and their values used in each line.
- Watch selected variables and pause the program when a selected variable is assigned a value, changes value, or conditionally changes value. Variables can be added to the ‘watch list’ at any breakpoint.
- Break at any code line or method, to allow the user to inspect variables, set watched variables, set additional breakpoints, and perform other actions. Breakpoints can be added to the ‘breakpoint list’ any time the program pauses at some other breakpoint.
- List the code lines being debugged
- Save and load filename settings
- Execute the plugin in ‘Go’ mode until a watch or breakpoint suspends the execution
- Abort debugging, letting the plugin run by itself

What Debuggifier cannot do

Debuggifier cannot

- Be an integrated development environment. By the way, we know of no IDE for Ruby in Sketchup.
- Change code lines

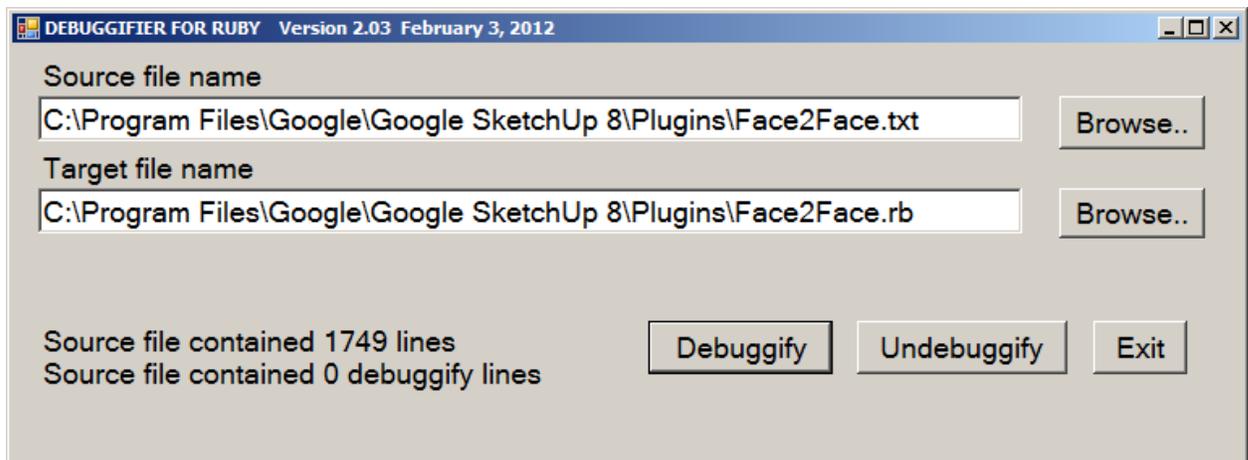
- Prevent Ruby from exiting suddenly (“bug splat”) when it encounters an error. However, examining the debug output file shows the last successful statement encountered, therefore determining the offending code.
- Handle complicated Regex expressions well. It usually ignores them.
- Handle uncommon Ruby syntaxes. It usually ignores them.
- Make distinctions for different scopes (bindings) of variables
- Debug more than one plugin file at a time.

Example

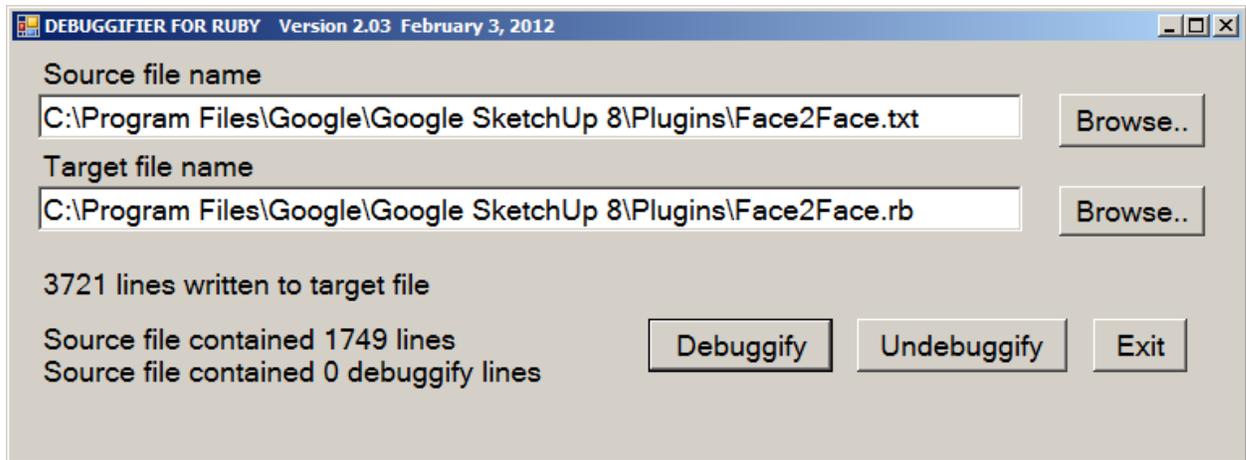
Here is a flavor of how it works. A Ruby program (that is, a plugin) ‘Face2Face.txt’ is the ‘source file’, and gets run through the Debuggifier.exe program to insert all the debug routine calls and create the ‘target file’ Face2Face.rb. Then Sketchup is invoked to use the preprocessed plugin’s new code in the target file. As the code runs, the text of the code lines and the values of the variables used in the code lines are displayed on the Ruby console, and are logged to a text file. At breakpoints, a debugger inputbox allows you to enter debugging commands, and to continue stepping through or jumping through the code.

Here are some screenshots of a typical debugging cycle.

First, the plugin source file code is run through the preprocessor.



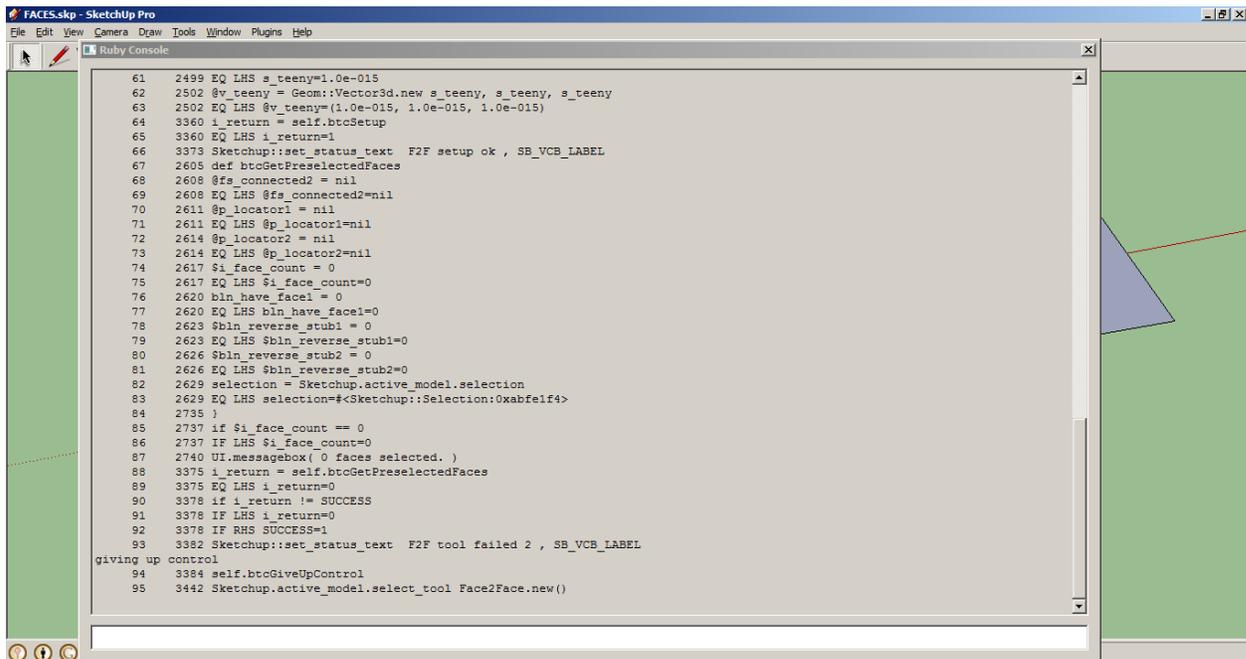
The user clicks the “Debuggify” command button, and then the code in file Face2Face.txt has all the required debug lines added to it and is written to Face2Face.rb, as shown here –



Notice that the line count has increased to 3271 due to the added debugging lines.

Then Sketchup is run on any model, and the new target file Face2Face.rb plugin is included with all of the plugins that Sketchup reads in (including Debbugifier.rb which implements the debugging calls). The user invokes the the Face2Face plugin by whatever mechanism the plugin provides, in this case, by selecting Sketchup menu items 'Plugins' and then 'Extrude faces'. The plugin starts running, and Debuggifier.rb intercepts it, and outputs tons of information to the Ruby console, and to a log file 'DEBUGGIFIER.TXT' in the same folder that the model was located.

Here is a sample of what the Ruby console might show –



And here is a partial closeup:

```
78 2623 $bln_reverse_stub1 = 0
79 2623 EQ LHS $bln_reverse_stub1=0
80 2626 $bln_reverse_stub2 = 0
81 2626 EQ LHS $bln_reverse_stub2=0
82 2629 selection = Sketchup.active_model.selection
83 2629 EQ LHS selection=#<Sketchup::Selection:0xabfelf4>
84 2735 }
85 2737 if $i_face_count == 0
86 2737 IF LHS $i_face_count=0
87 2740 UI.messagebox( 0 faces selected. )
88 3375 i_return = self.btcGetPreselectedFaces
89 3375 EQ LHS i_return=0
90 3378 if i_return != SUCCESS
91 3378 IF LHS i_return=0
```

Sequence numbers and program line numbers from the modified Face2Face.rb are shown followed by the text of each code line, then values of the Left Hand Side (LHS) and Right Hand Side (RHS) of each line if appropriate.

When the plugin encounters its first debug call, the debugger inputbox appears –



Here is where the real usefulness starts. The user can just hit 'enter' to step through the code that is after the breakpoint, and keep just hitting the <Enter> key to continue stepping. Or the user can set more breakpoints or set watches on any variables by making entries into the 'Breakpoints' and 'Watches' boxes. After doing so, hitting 'OK' (or the <Enter> key) will allow the plugin to continue.

When the user reaches the code lines or data values of interest, the intended event occurs wherein the user realizes the source of the problem with the plugin code. Then the user exits Sketchup and makes the necessary bug fix in the code Face2Face.rb with the user's favorite editor, and the sequence is repeated -

- Run Sketchup on a model, which reads the revised code in the .rb file, followed by
- Steps and go's, inspection of variables at the desired breakpoints and watches, and in the Ruby console. Read back the log file with an editor if that will help.

- Make code changes, based on the user's new enlightenment, to make the plugin's code work correctly

This will lead to the desired bug fix(es). When the code fixes are made, the user can 'Undebugify' the code using Debuggifier.exe to remove debugging lines and get back to 'clean' code.

Debugging slows the execution speed of the plugin being developed. **Comment out** most of the debug statements by globally replacing "Debuggifier.dfrMyPrint" with "#Debuggifier.dfrMyPrint". Then **uncomment** these debugging statements in the areas of the code that you need to debug. This will greatly improve execution speed.

Preprocessor - insertion of Debuggifier calls

To illustrate by example, the code snippet 1 (a small part of a larger Ruby program) gets changed by Debuggifier into the code snippet 2. The outputs to the Ruby console are shown previously.

Code snippet 1- original source code:

```
$bIn_reverse_stub1 = 0
$bIn_reverse_stub2 = 0
selection = Sketchup.active_model.selection
```

Code snippet 2 the Debuggified code:

```
$bIn_reverse_stub1 = 0
  Debuggifier.dfrMyPrint ' 2623 ' + '$bIn_reverse_stub1 = 0'
  Debuggifier.dfrMyPrint 'EQ LHS ' $bIn_reverse_stub1,'$bIn_reverse_stub1'
$bIn_reverse_stub2 = 0
  Debuggifier.dfrMyPrint ' 2626 ' + '$bIn_reverse_stub2 = 0'
  Debuggifier.dfrMyPrint ' EQ LHS ' ,bIn_reverse_stub2,'$bIn_reverse_stub2'
selection = Sketchup.active_model.selection
  Debuggifier.dfrMyPrint ' 2629 ' + 'selection = Sketchup.active. . .
  Debuggifier.dfrMyPrint ' EQ LHS ' , selection,'selection'
```

There are calls to Debuggifier.rb calls after any assignment/if/while/case statement. The first inserted statement is to print the source code line so the user can follow the program flow. The subsequent inserted statements show the values used in the statement evaluation if that is appropriate.

All of your comments (denoted by #) are included in the target file. Sometimes literals have their singlequote ' or doublequote " delimiters removed in the printing of the code line.



Actions

Selections from the action box dropdown list are:

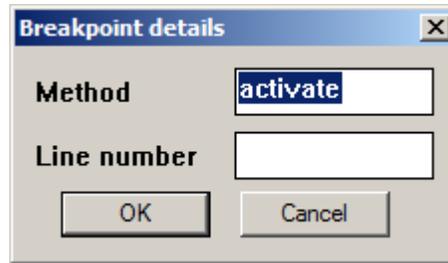
- “Step” runs the plugin to the next of its lines, presumably having a Debugger all after it so it can be paused again.
- “Abort” cancels debugging and lets the plugin run to its completion.
- “Go” lets the plugin run until a breakpoint is reached or a watched variable is assigned/changed.
- “Save” writes the current watch list and breakpoint list to “DEBUGGIFIER_SETUP.TXT”, in the folder where the Sketchup model is found.
- “Load” gets last-saved watch list and breakpoint list.
- “List” lists the code lines N1 through N2 on the Ruby console. N1 is 5 lines previous to the current line. N2 is 5 lines after the current line.

Breakpoints

The debugger will pause and show the debugging inputbox **after** the relevant breakpoint line is executed.

Selections from the Breakpoints box dropdown list are:

- Show all – displays currently set breakpoints in a messagebox
- Clear all – deletes all breakpoint settings
- Add – Brings up another inputbox, shown below, to allow the user to specify a new breakpoint at either a method entry or a plugin code line number as shown below. The code line numbers are those that are generated and made part of the debug call in the **modified** plugin.
- Delete – Brings up another inputbox to remove a breakpoint setting



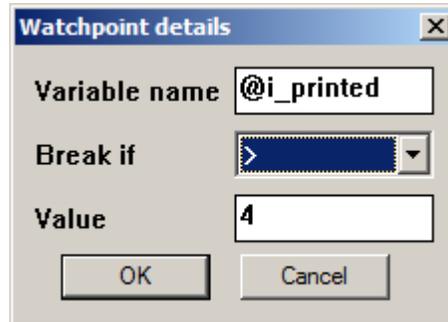
Watches

The debugger will pause **after** the any statement with the relevant watched variable on the left side, if the value of the variable meets the criterion.

Selections from the Watches box dropdown list are:

- Show all – displays currently watched variables and their latest values in a messagebox
- Clear all – deletes all watches
- Add – Brings up another inputbox, shown below, to allow the user to specify a new watch as shown below. The possible watch criteria are “ASSIGNED”, “CHANGED”, “>”, “<”, “==”, and “!=”.
- Delete – Brings up another inputbox to remove a watch

Text comparisons can be made as well as numeric comparisons.



Variables

The debugger stores variable name and value pairs as they are encountered in the debug statements in the plugin code.

Selections from the Variables box dropdown list are:

- Show all – displays currently known variables in a messagebox. This number of known variables increases as the plugin code executes more and more lines.
- Examine – brings up another inputbox asking for the name of the variable. The variable does not need to be in the variables list yet. The debugger then tries to **get** the value associated with the variable name.

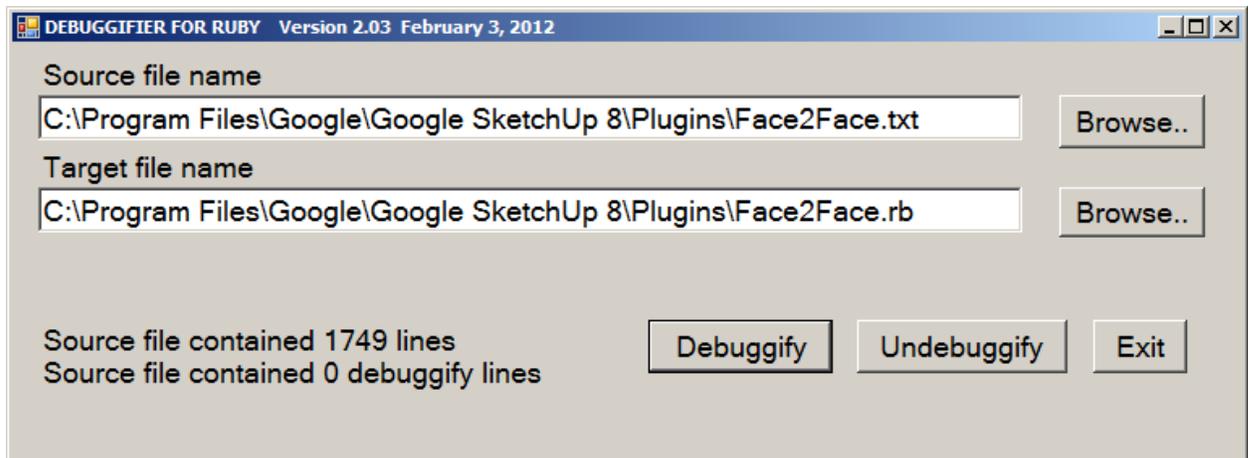
- Set – brings up another inputbox asking for the name of the variable. The variable does not need to be in the variables list yet. The debugger then tries to **set** the value associated with the variable name.

Updates

The program continues to be developed. It is written in VB.net. As each new version of the Debuggifier is released, updates to this manual will also be released. The current version is a 32-bit application that runs under Microsoft Windows XP and Windows 7 (most of which are 32-bit versions). It may run well on other versions of Windows as well but this has not yet been tested.

Installation

1. Determine that your computer has .NET framework 4 or above.
2. Download the Debuggifier zipped package from the website www.sketchucation.com and unzip it
3. Run ...deploy\setup.exe. The installation program will install Debuggifier.exe wherever you like.
4. If a desktop icon has not been created on your desktop create one and point it to the newly created DEBUGGIFIER.EXE, wherever you chose to install it.
5. Doubleclick the icon. You will get the main screen as follows:



The first time that you run Debuggifier, some default filenames appear in the textboxes, instead of the names shown above. A setup file (C:\Users\DEBUGGIFIER_SETUP.TXT) keeps track of the user preferences for path and file names. Change the filenames by typing in the textboxes or using the Browse buttons.

Source file name textbox

The source file is the Ruby program which is saved as a text file.

Target file name textbox

The target file is the Ruby program with many print statements added to it for debugging, i.e., the Debuggified file.

Browse command buttons

The user can use the <Browse> command buttons to navigate to and select the files. It is **not advisable** to use the same file names for the source and the target, but Debuggifier will allow it. It gets very confusing and errors are easy to make if you do this.

Debuggify command button

This button starts the processing of inserting debug statements and writing the target file.

Undebuggify command button

This button removes all debug statements **from the source file** and rewrites the target file. This is done after the Ruby **source file** script is debugged and fixed. Then the source / target file names can be changed in the textboxes, or the source / target files can be renamed, to remove the debug lines and restore your fixed source file so it has no more debug lines in it.

Exit command button

This button saves the settings and ends the program, at which time the user should run Sketchup and start debugging.

Pseudoprofiling

The Debuggifier output file can be used to determine the most-used statements in your program. Drop the log file contents into Excel, then sort on program line number. Count the number of occurrences of each line. Then sort the counts from high to low to discover the most-used lines. Nondebuggified lines will not be counted.