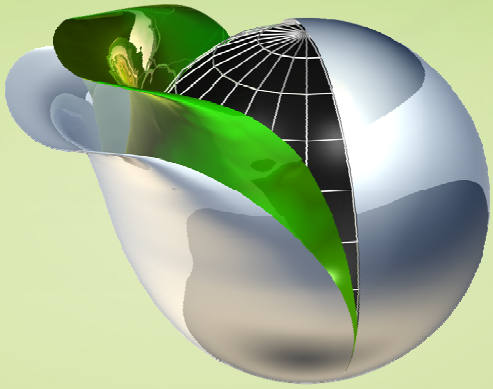
User's Guide

NVIDIA Shader Debugger



June 2008

Table of Contents

Overview

The NVIDIA Shader Debugger

The NVIDIA Shader Debugger is a plug-in for FX Composer 2.5 that enables debugging of HLSL, CgFX, and COLLADA FX Cg pixel shaders. It seamlessly integrates into FX Composer's shader authoring environment, allowing you to debug shaders while creating them.

The debugger offers numerous advantages to shader authors:

- Makes it easy to understand shader algorithms and control logic
- Improving productivity by removing the need to embed additional debugging functionality into shaders
- Quickly understanding shaders written by other artists, developers, or shader authoring software
- Learning shader authoring and shading language features

Features

Overview

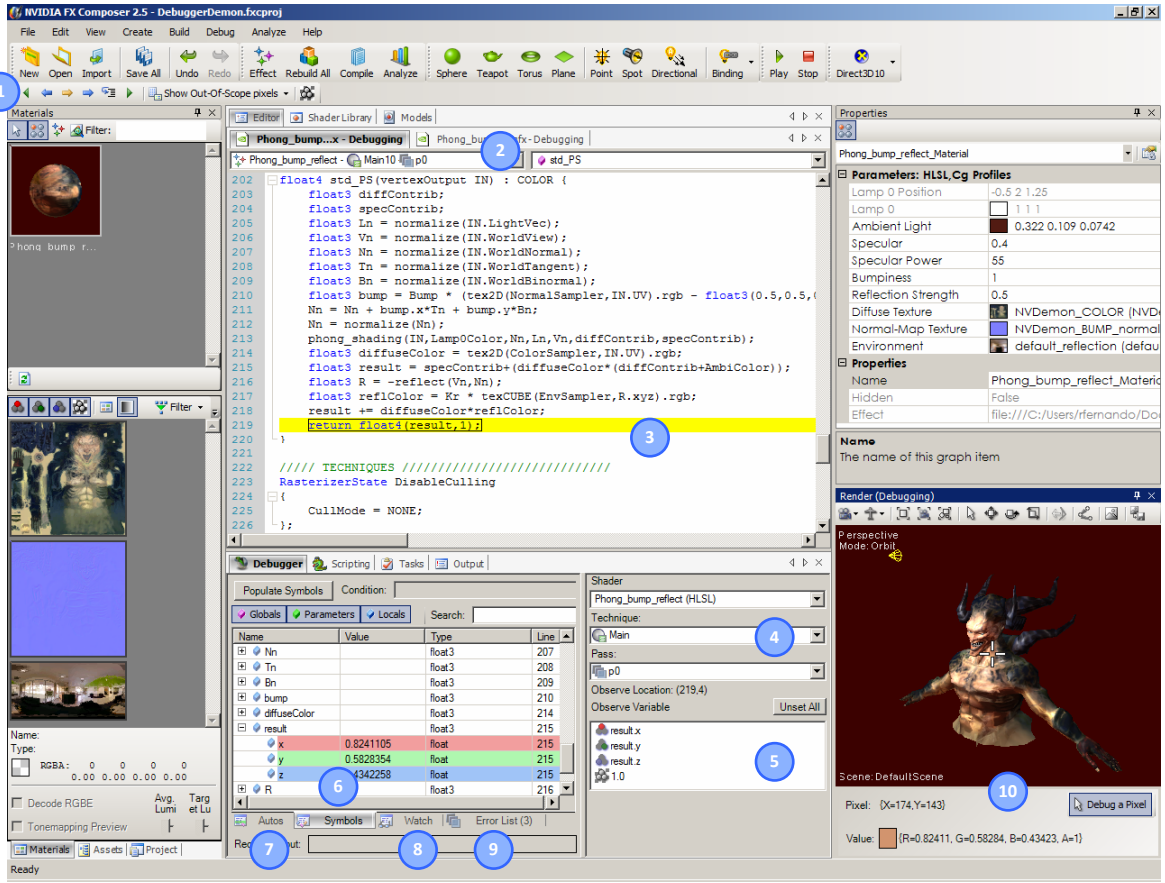


Figure 1: Screenshot of FX Composer in a typical shader debugging session. The numbers in the blue circles identify key debugger features, which are listed in the next page.

Feature List

The NVIDIA Shader Debugger contains several features to debug and understand shaders. These features include:

1. **Debugger Toolbar.** This toolbar contains several convenient buttons:
    a. **Run to next/previous statement (F10/Shift+F10):** Used to step through the shader code, while debugging. Pressing “next statement” button makes the

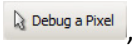
debugger set scope at the next statement. The “previous statement” button is analogous.

b. **Show observe point (F12).** Jumps to the current observe point. (This is useful if you have scrolled far away from the observe point.)

c. **Run to previous/next bookmark (F8/Ctrl+F8):** These commands will move the current observe location to the next/previous code bookmark. Note that bookmarks are not breakpoints as on a CPU, because many threads are running in parallel. Bookmarks are simply locations in a text file.

d. **Run to cursor (Ctrl+F10).** Executes the shader up to the current cursor position.

e. **Kill fragment mode:** Toggles between 3 modes: off, kill fragment, and highlight killed fragments. The latter two modes can be used to determine the fragments that do not reach a particular observe location in the code.

2. **Shader/Technique & Function Selector for Editor.** On the left, provides a summary of all the techniques and their respective passes for the current effect. On the right, provides a dropdown of all functions in the current shader file.
3. **Editor in Debug Mode.** Line numbers are highlighted if debugging is active. The highlight is green (for parallel debugging) or gray (for single pixel debugging). The current observe location is highlighted in yellow.
4. **Shader/Technique/Pass Selector in Debugger Panel.** Allows you to choose which shader, technique, and pass to debug.
5. **Output Remapper.** Allows you to visualize variable component values in arbitrary ways.
6. **Symbols Tab.** Lists all variables affecting the code currently being debugged. The Globals, Parameters, and Locals filters allow you to reduce the number of variables you are viewing at one time.
7. **Autos Tab.** Lists only those variables that in the immediate preceding proximity to the current observe location.
8. **Watch Tab.** Allows you to enter expressions and set them as observe variables.
9. **Error List.** Lists any errors or warnings generated by the debugger. Note that you can disable the “Set focus to error list on warnings and errors” checkbox if you don’t want to be disturbed by the error list while you work. This option is recommended only for advanced users.
10. **Render Panel in Debug Mode.** Includes a special status bar to indicate the pixel currently being debugged (if any), as well as a swatch and value indicator to show pixel values as you mouse over them. Also includes the “Debug a Pixel” button 

, which allows you to specify a pixel to debug.

Using the Debugger

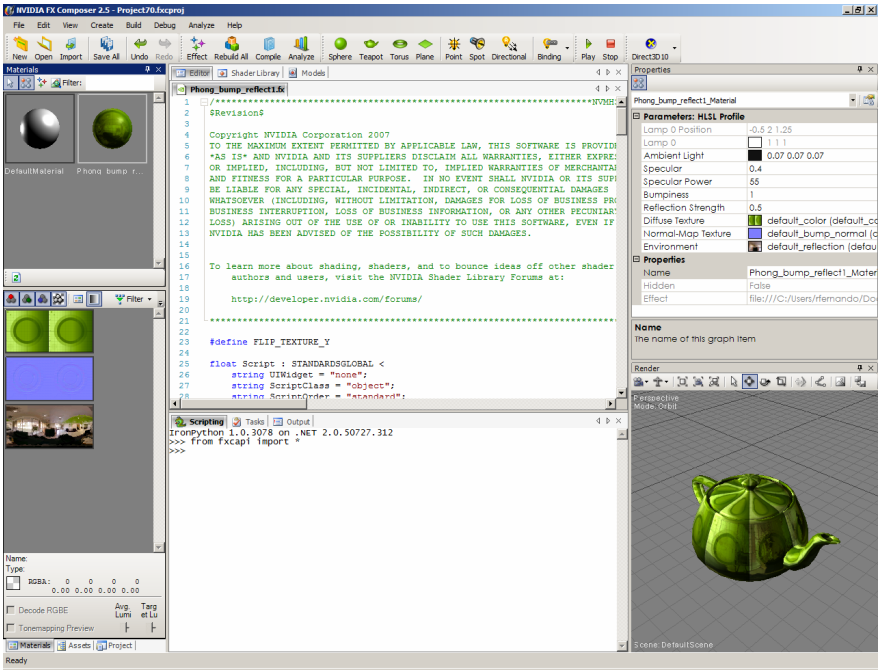
Getting Started

In this section, you'll be able to follow a simple set of steps to learn about basic shader debugging functionality.

Let's start by creating a new material and assigning it to an object to debug:

1. Start FX Composer
2. Create a new project
3. Create a teapot by clicking on the main toolbar's Teapot icon
4. Click on the "Add Effect…" button in the main toolbar
5. Select HLSL FX (or CgFX if you're working in OpenGL)
6. Click Next
7. Choose Phong Bump Reflect.
8. Click Finish
9. Apply your new Phong Bump Reflect material to the teapot by dragging-and-dropping it from the Materials panel to the Teapot (in the Render panel)

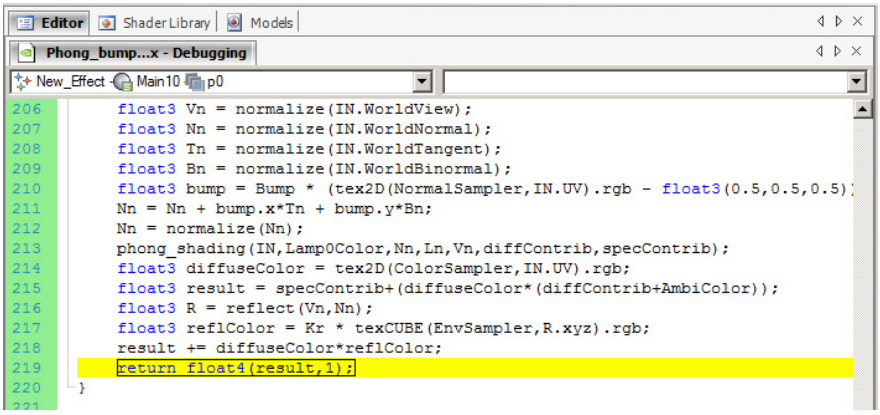
Now you're ready to start debugging. Your FX Composer window should look similar to this:



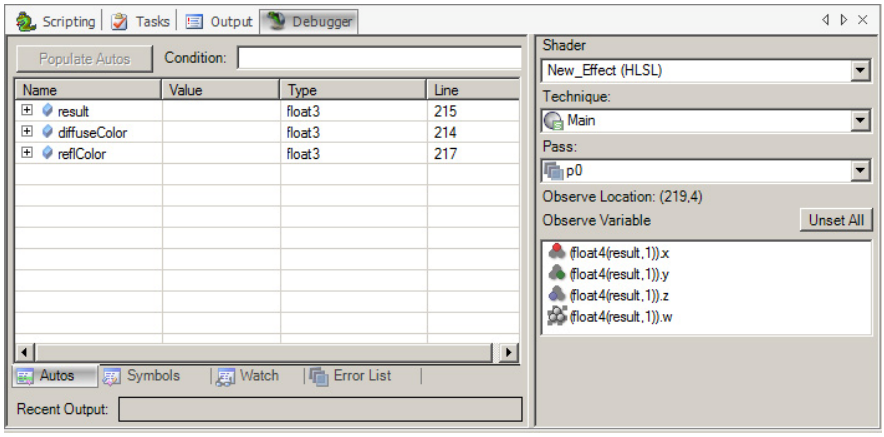
Now let's debug this material's shader:

10. Right-click on the green Phong Bump Reflect sphere in the Materials panel. Choose "Start Debugging" from the context menu. (You can also start debugging at any time in the editor by right-clicking and selecting "Start Debugging".)

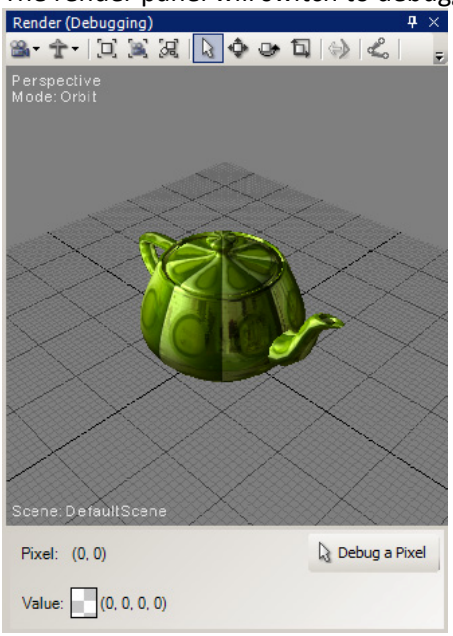
    a. The shader code will now be shown in the shader editor with a green left margin which indicates that the shader is being debugged. The return statement of the entry function of the pixel shader is highlighted in yellow to signify that the current "observe location" is at the return statement.



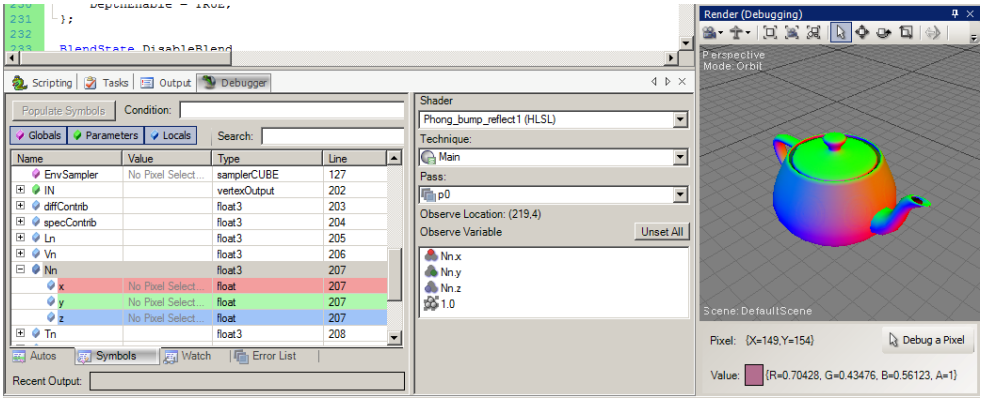
    b. The Debugger panel also brought into focus (it is also available at any time through the View Menu). By default, the "Autos" tab is showing. This tab lists variables that are used in proximity to the current observe location.

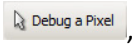


The render panel will switch to debugging mode, with a pixel picker sub-panel



11. Select the “Symbols” tab in the Debugger panel. This lists all variables that are in scope. (As a convenience you can unselect the “Global” and “Parameter” filters to limit the listing to local variables).
12. Double-click on Nn to visualize the teapot’s normals. You can double-click on any variable to visualize its value in the Render panel.



13. Move the mouse around the Render panel. The status area shows the coordinate of the pixel pointed by the cursor, as well as the value currently being visualized. (In this case, Nn).
14. You’ll notice that none of the values in the Symbols panel are populated. This is because the debugger doesn’t yet know which pixel you’re interested in. Pick a pixel using the “Debug a Pixel” button 

, located in the status area below the render panel. The values for the Nn vector will now be populated.

15. You can double-click on other variables to populate their values as well. The "Populate Symbols" button will populate all symbols, but please note that this operation can take some time if there are many symbols in your shader.
16. That's it! You have just experienced a simple shader debugging session. You'll discover more debugger features in the next section.

Make Your Life Easier

This section outlines some helpful functions that the Shader Debugger offers to make debugging easier.

Choosing a Technique to Debug

While debugging, you may change the current technique you wish to debug using the drop down "Technique" menu, located in the upper-right corner of the debugger panel. Selecting one of the techniques from this drop down menu will immediately set the "observe point" to the chosen technique, within the editor.

Remapping Output Channels

The Channel Remapper

The Channel Remapper shows you how variable values are mapped to specific color channels in the rendered scene, and allows you to change those mappings. The Channel Remapper is displayed along the right hand side of the debugger panel.

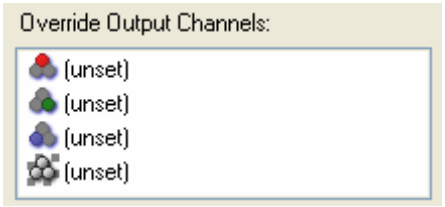


Figure 2. The Channel Remapper

Visualizing Variables Automatically

Whenever you double-click on a variable in the "Autos", "Symbols", or "Watch" tabs to visualize it, each component of the variable will automatically be mapped to the red, green, blue, alpha channels, in that order. If a variable has fewer components, the remaining components will be set to 0.0, and the Alpha will be set to 1.0.

Visualizing Variable Components Individually

Sometimes, you may want to mix-and-match components of different variables. For example, you may want the x and y components of one variable in the red and green output channels, but the x component of another variable in the blue channel.

To do this, right-click on a component (such as Output.x) in the "Autos", "Symbols", or "Watch" tabs and choose the channel mapping you want. (For example, "Set as Red Channel".)

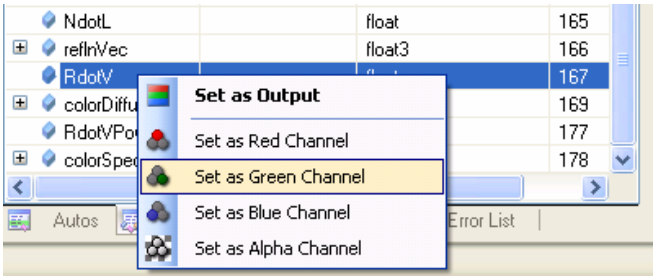


Figure 3. Visualizing a Variable

Note that the "Autos", "Symbols", and "Watch" tabs use red, green, and blue highlighting to show which variable components are being matched to which output channels. If a variable is mapped to multiple output channels, it will be highlighted in yellow. An example is shown below.

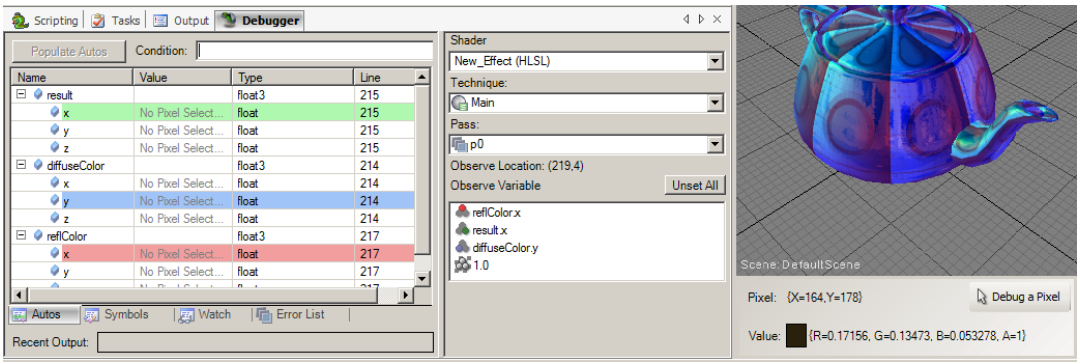


Figure 4. Visualizing Components from Multiple Variables

The Autos Tab

The "Autos" tab and the "Watch" tab can be found within the debugger panel and can be very useful while debugging.

The "Autos" tab displays variables much like the "Symbols" tab, except that it shows only variables used in the current observe location and the preceding line. This keeps you from having to constantly scroll through a long list of variables while working on a particular part of a shader.

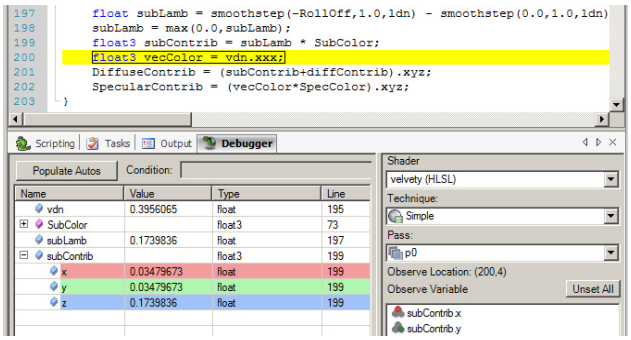


Figure 5. The Autos Tab of the Debugger Panel

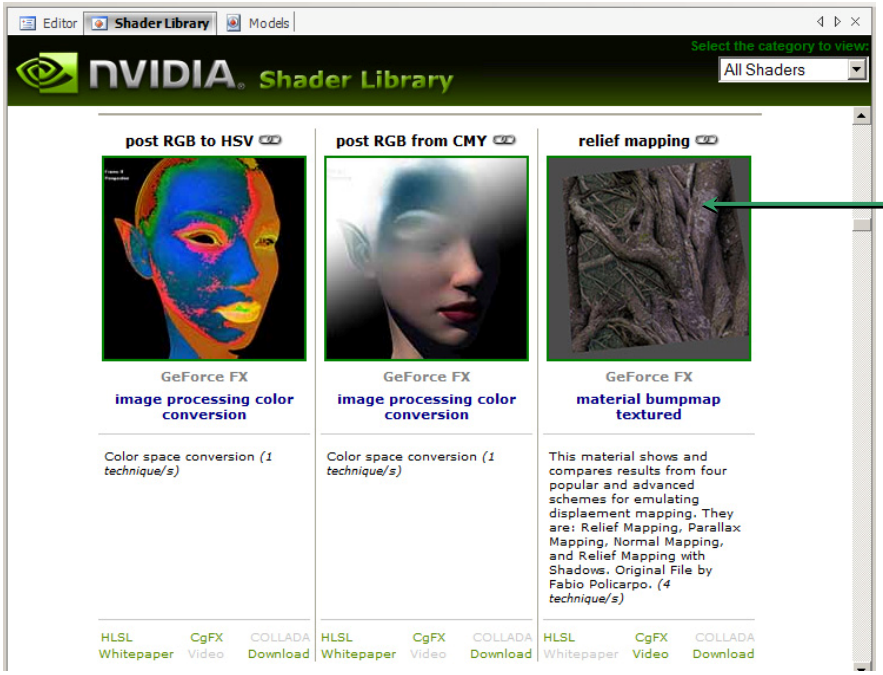
The Watch Tab

The “Watch” tab allows you to debug your own expressions, formed from currently existing variables and mathematical operators. Double-click on an empty line to enter an expression. Once entered, you can visualize an expression by double-clicking on it (just like variables in the “Autos” or “Symbols” tabs).

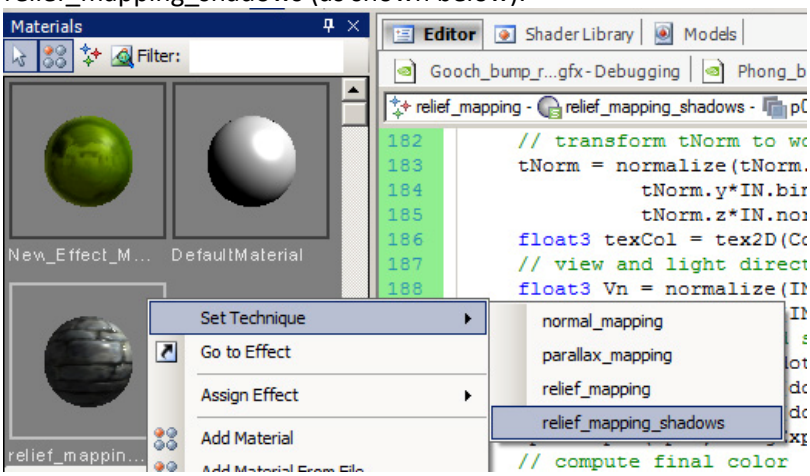
Conditional Debugging

In this section we will do a debugging walkthrough of a slightly more complex shader that contains multiple techniques.

1. Create a new project.
2. Add a teapot.
3. Apply the Relief Mapping shader from the Shader Library to the teapot. To do this, select the Shader Library tab (next to the Editor tab), scroll down a few rows, and then drag-and-drop the image for Relief Mapping onto the teapot.



4. Right-click on the Relief Mapping material and choose Set Technique -> relief_mapping_shadows (as shown below).



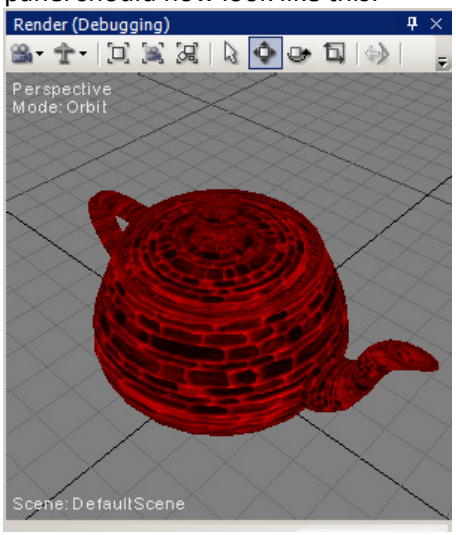
5. Left-click on the Relief Mapping material to see its properties. Change the "Tile Repeat" parameter to 1 instead of 8.

6. Right-click on the Relief Mapping material and choose Start Debugging.

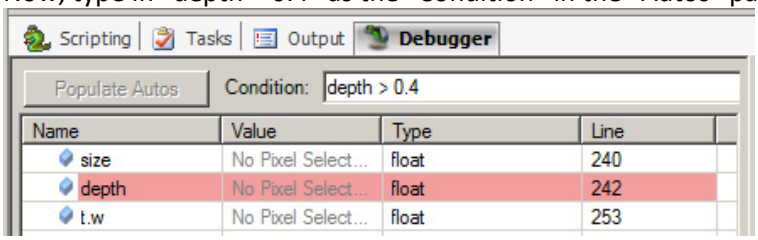
7. In the "Technique" drop-down of the Debugger panel, choose "relief_mapping_shadows".

8. In the editor, scroll to line 255 in the shader code. This line is within a loop in the "ray_intersect_rm" helper function. Right-click to bring up the editor context menu and choose "Run to cursor". This will set the "observe location" in the code to line 255.

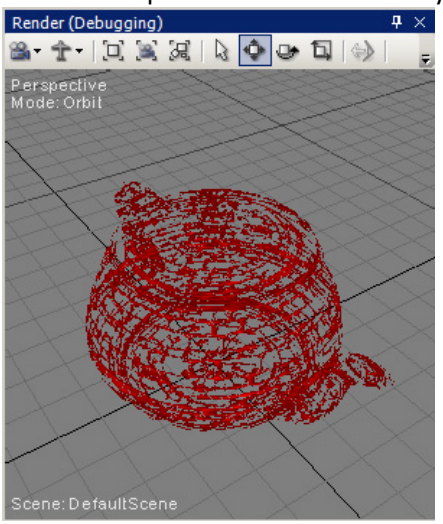
9. In the debugger's "Autos" panel, double-click on "depth" to visualize it. The Render panel should now look like this:



10. Now, type in “depth > 0.4” as the “Condition” in the “Autos” panel, as shown below.



The Render panel will now show only fragments with depth greater than 0.4:



Debugging Multi-Pass Effects

Multi-pass (or “full-scene”) effects are debugged in much the same way as other effects. The only change here is that when you right-click on a multi-pass effect and select “Start Debugging”, you will see a sub-menu where you can choose the specific pass to debug. This is illustrated below.