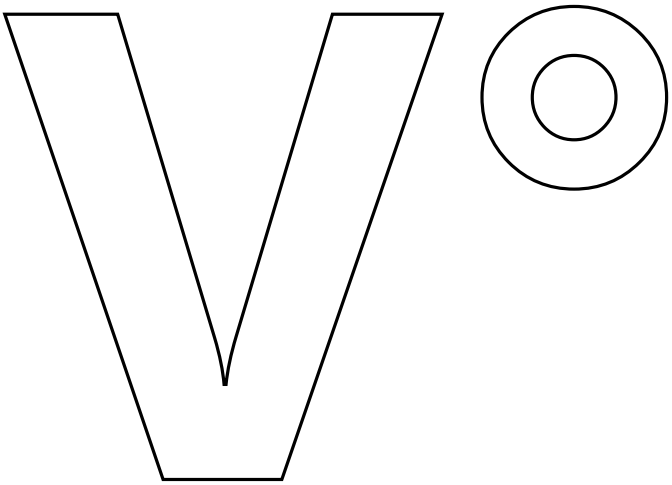
miniVivisection

See your script's data in Rhino while you debug

A plug-in for the Script Editor and Grasshopper of Rhino 8



Rhino 8
Script Editor
Grasshopper

Version 0.2.0
build 325
experimental

2026-10-23
formerly
ScriptEditorWatchViz

Experimental. miniVivisection relies on internal code of Rhino 8's Script Editor; a Rhino update can break parts of it.

Contents

I Getting to know miniVivisection 1

1 Introduction 2
1.1 Why it exists . . . 2
1.2 What it does, in six ideas . . . 2
1.3 Requirements . . . 3
1.4 How this manual is organised . . . 3
1.5 Conventions . . . 4

2 Features at a glance 5

3 Concepts 9
3.1 Debugging in two minutes . . . 9
3.2 The paused frame . . . 9
3.3 Drawn, not added . . . 10
3.4 Write-back . . . 10
3.5 Rhino's UI thread and FreePause . . . 11
3.6 Debug runs and plain runs: pause or stop . . . 11
3.7 Threads . . . 12
3.8 Grasshopper: components run inside a solution . . . 12
3.9 Markdown as documentation . . . 12
3.10 Prompts and AI . . . 13
3.11 The V names and the two switches . . . 13

4 Getting started 14
4.1 Installing . . . 14
4.2 Where it shows . . . 15
4.3 Your first session . . . 15
4.4 The feature test . . . 16
4.5 Editor conveniences . . . 16

II In the Script Editor 17

5 V°Watch: values drawn in Rhino 18
5.1 Adding a watch . . . 18
5.2 The list . . . 19
5.3 What is drawn, and how . . . 19
5.4 Interpret as . . . 20
5.5 Selection, both ways . . . 20
5.6 Watches in every thread . . . 20
5.7 C# and IronPython scripts . . . 20

6 Values in the code 21
6.1 Tooltips . . . 21
6.2 Links in the tooltip . . . 21
6.3 Hover preview . . . 22
6.4 Values at the ends of lines . . . 22
6.5 The editor's Watch tab evaluates expressions . . . 22
6.6 Live Variables and Watch tabs . . . 22
6.7 Context menus while paused . . . 22

7 Edit in Rhino, and Bake 24
7.1 Edit in Rhino . . . 24
7.2 Auto edit . . . 25
7.3 Bake . . . 25

8 Visualizers and Interpret as 26
8.1 The order . . . 26
8.2 Writing a visualizer . . . 26
8.3 Where to register them . . . 27
8.4 Choosing an interpretation . . . 27

9 Breakpoints with conditions 28
9.1 Setting them . . . 28
9.2 Conditions . . . 28
9.3 Log messages (logpoints) . . . 29
9.4 In the code and the Breakpoints tab . . . 29
9.5 Stepping ignores conditions . . . 29
9.6 Stop once per line . . . 29
9.7 Turning it off . . . 29

10 Running, pausing, stopping 30
10.1 Run and Debug buttons . . . 30
10.2 Break now: Ctrl + Alt + B . . . 30
10.3 FreePause: Rhino usable while paused . . . 31
10.4 The editor stays open while debugging . . . 31
10.5 No freeze from another script's pause . . . 31
10.6 Scripts started from Rhino's command line . . . 31
10.7 ScriptEditor Run, with Debug=On . . . 32
10.8 When an editor can't be reached . . . 32

11 Threads 34
11.1 V°Threads . . . 34
11.2 Watches in every thread . . . 34
11.3 The freeze guard . . . 35
11.4 What isn't traced . . . 35

12 Prompts: V°Python, V°PowerShell, V°Claude 36
12.1 Working in a prompt . . . 36
12.2 V°Python . . . 36
12.3 V°PowerShell . . . 38
12.4 V°Claude . . . 38

12.5 V°Log . . . 40
12.6 V°Settings . . . 40

III In Grasshopper 42

13 Debugging script components 43
13.1 Three modes . . . 43
13.2 Run and Debug in the component's editor . . . 43
13.3 Ctrl+Alt+B on a component . . . 43
13.4 Hidden editors . . . 44
13.5 Editors follow the definition . . . 44
13.6 Which run is paused . . . 44
13.7 Which branch failed . . . 44
13.8 Runtime errors and wiring . . . 45
13.9 Recomputes held back . . . 45
13.10 async:true in a component . . . 46

14 Grasshopper while a component is paused 47
14.1 The canvas works . . . 47
14.2 Changes are computed at once, downstream . . . 47
14.3 What depends on the paused component is marked . . . 47
14.4 What waits, and what is refused . . . 47

15 Scripts and Markdown in Panels 49
15.1 Scripts in Panels shown as code . . . 49
15.2 Editing a script Panel in place . . . 49
15.3 After Ctrl+Alt+B stopped a Panel's script . . . 50
15.4 Scripts from a component's script input . . . 50
15.5 A component with a script input . . . 51
15.6 Show as . . . 51
15.7 Markdown in Panels . . . 52
15.8 Panels fed by another . . . 52

16 The components: V°Edit, V°Text, V°Protect 54
16.1 V°Edit: Grasshopper geometry edited in Rhino . . . 54
16.2 V°Text: a Panel's text as Rhino text . . . 55
16.3 V°Protect: is it still the original? . . . 57

17 Prompts in Grasshopper 59
17.1 V°Claude in a definition . . . 59
17.2 V°Python in Grasshopper . . . 60
17.3 V°PowerShell in Grasshopper . . . 61
17.4 Settings . . . 61

IV In Rhino 62

18 Markdown as Rhino text 63
18.1 Making Markdown text . . . 63

18.2 What the Markdown becomes . . . 63
18.3 Editing it . . . 63
18.4 The parts belong together . . . 64
18.5 Changes made in Rhino are taken in . . . 64
18.6 Links . . . 65

19 Links in Markdown 66
19.1 Targets . . . 66
19.2 Into another file, and back . . . 67
19.3 Following a link . . . 67
19.4 Their look . . . 67

V Reference 69

20 The lessons 70
20.1 Where they are . . . 70
20.2 The overview . . . 70
20.3 Script lessons . . . 70
20.4 Grasshopper lessons . . . 71
20.5 The Rhino example . . . 72

21 Commands, keys and files 73
21.1 Commands . . . 73
21.2 Keys . . . 73
21.3 Logs and files . . . 74
21.4 Environment variables . . . 75

22 Settings 76
22.1 Tab *General* . . . 76
22.2 Tab *Rhino* . . . 82
22.3 Tab *Grasshopper* . . . 83

23 Limits and troubleshooting 87
23.1 Limits . . . 87
23.2 Troubleshooting . . . 87
23.3 Reporting a problem . . . 88

Part I

Getting to know miniVivisection

1 Introduction

miniVivisection extends the debugger of Rhino 8's Script Editor, and it brings that debugger into Grasshopper. When a script stops at a breakpoint, you see what its variables hold *as geometry in Rhino's viewports*, not only as lines of text. You can change that geometry with Rhino's own tools and write it back into the running script. The values appear where you read the code: in the tooltips, and at the ends of the lines. Rhino stays usable the whole time, so you can select, measure and run commands while the script waits.

1.1 Why it exists

Scripts that make geometry are hard to debug with a text debugger. A list of four `Point3d` values can look right number by number and still describe a rectangle whose corners are in the wrong order. On the screen, the mistake is obvious: the outline crosses itself (the cover picture). Rhino's Script Editor has a good debugger. It shows values as text, though, and while a script is paused Rhino is locked in a **Select Action** prompt: you can't select an object, measure a distance or look at the model from another side.

Other problems come up again and again in teaching and practice:

- A script started from Rhino's command line hangs in a loop, and the only way out is to kill Rhino and lose the model.
- A Grasshopper script component fails for three items out of a hundred, and Grasshopper says only *"'NoneType' object is not iterable"*, not which item, branch or run.
- Notes about a definition sit in Panels as centered plain text, and scripts kept in Panels have no line numbers, so an error's line number points nowhere.

miniVivisection addresses each of these, inside the tools you already use. It adds nothing you have to learn before you can start: set a breakpoint, press F5, and the values are drawn.

1.2 What it does, in six ideas

See. A paused script's values are drawn in Rhino: points, curves, surfaces, meshes, planes, vectors and lists of them, each with a name. The code shows its values in tooltips and at the ends of the lines, and what changed since the last step (Chapters 5 and 6).

Change and keep. Turn a value into Rhino objects, edit them with the Gumball or control points, and the paused script continues with your edit. Or bake what was computed into the document before the script crashes (Chapter 7).

Stop where you want. Breakpoints with conditions, hit counts and log messages. Ctrl + Alt + B pauses a running script at its next line, even one that holds Rhino's UI thread, or stops a plain run and records where it was (Chapters 9 and 10).

Ask the paused script. A Python prompt runs in the paused frame and can change the script's variables. A PowerShell prompt and a chat with Claude work on the same paused script

(Chapter 12).

Debug Grasshopper. Script components pause at breakpoints and Ctrl + Alt + B. Failing runs are named by branch and item, and can be debugged one by one. The canvas stays usable while a component is paused (Chapters 13 and 14).

Document in Markdown. Panels show scripts as code and notes as rendered Markdown. Markdown becomes Rhino text, and links lead from a note to a component, a Rhino object, a named view, another file or a script (Chapters 15, 18 and 19).

1.3 Requirements

Rhino	Rhino 8 for Windows, with its Script Editor (RhinoCode). Rhino 8 runs on .NET; IronPython 2 scripts need the .NET 8 runtime (Rhino's `SetDotNetRuntime`).
Python 3 scripts	Get every feature. This is what miniVivisection is made for.
IronPython 2 scripts	(`#! python2`) Get the debugger features, drawing of plain variable names and conditional breakpoints; not expressions, inline values, editing in Rhino or the Python prompt in the paused frame (Chapter 23).
C# scripts	Watches of plain variable names and conditional breakpoints (a small C# evaluator); largely untested.
Grasshopper	Rhino 8's Python 3 script components. The legacy GhPython (IronPython) component is not covered.
V°Claude (optional)	Claude Code, the `claude` command line program, installed and signed in. RhinoAI (McNeel's Rhino MCP Platform) lets Claude work in the model and in Grasshopper.
V°PowerShell	PowerShell 7 (`pwsh`) or Windows PowerShell.

> **Experimental** miniVivisection 0.2.0 is experimental. The Script Editor has no extension interface, so the plug-in reaches into its internal code. A Rhino service release can change that code and switch parts of the plug-in off; it then does nothing rather than fail. It was called ScriptEditorWatchViz until October 2026.

1.4 How this manual is organised

Part I, Getting to know miniVivisection: this introduction, all features at a glance (Chapter 2), the concepts the features are built on (Chapter 3) and a first session (Chapter 4). Read Chapter 3 once: most questions about why something works the way it does are answered there.

Part II, In the Script Editor: every feature for scripts, from drawing values to threads and the prompts.

Part III, In Grasshopper: debugging script components, the canvas while paused, Panels as code and Markdown, the components V°Edit, V°Text and V°Protect, and the prompts window.

Part IV, In Rhino: Markdown as Rhino text, and links between Rhino, Grasshopper and scripts.

Part V, Reference: the lessons, commands and keys, every setting, limits and troubleshooting.

The Markdown manual that **V°Help** opens in the Script Editor (`miniVivisection_Manual.md`) has the same features in short form. This manual explains them at length.

1.5 Conventions

Edit in Rhino	A button, menu item, tab or label as it appears on screen.
V°Watch ▸ **Interpret as**	A path through tabs and menus.
Stop once per line	A setting, in **V°Settings** or **Tools** ▸ **Options** ▸ **miniVivisection** (Chapter 22).
Ctrl + Alt + B	Keys to press together.
`crv.PointAtStart`	Code: Python, a command, a value.
`examples`	A file or folder.
`04_edit_in_rhino.py`	A lesson that teaches the feature (Chapter 20).
V°Watch	One of miniVivisection's names (§ 3.11).

2 Features at a glance

Every feature in one list, grouped by what it is for. The last column links to the section that explains it.

Seeing values while paused

V°Watch	Watched values drawn in Rhino: points, curves, surfaces, meshes, planes, vectors, lists; name dots; Show, Bake, selection both ways.	§ 5
Expressions in watches	Any Python expression: `curves[0]`, `crv.PointAtStart`, comprehensions, constructors.	§ 5.1
Interpret as	Raw data drawn as points, polylines, curves, lines, vectors or triangles.	§ 5.4
Visualizers	Your own drawing, and editing, for your own classes.	§ 8
Tooltip values	A name's value as a line of code, its facts, "was" after a step, links to watch, edit, bake, Tree.	§ 6.1
Watch a call	Over a class called on the line: watch the whole call, `rg.Point3d(…)`.	§ 6.2
Value tree	A small window with a value's members, updated at every pause.	§ 6.2
Hover preview	The geometry under the mouse in the code, drawn in yellow in Rhino.	§ 6.3
Values at line ends	Gray values after each line of the paused function; changed ones orange.	§ 6.4
Watch tab expressions	The editor's own Watch tab evaluates any Python expression.	§ 6.5
Context menus	Watch, Bake, Edit in Rhino from the code and the Variables and Watch tabs.	§ 6.7

Changing and keeping

Edit in Rhino	A value as Rhino objects; edit them, and the change is written back into the paused script, with Undo.	§ 7.1
Auto edit	Every shown watch editable at each pause.	§ 7.2
Bake	Drawn values copied into the document; `BakeAll`.	§ 7.3

Stopping where you want

Conditions, hit counts, log messages	Breakpoints that pause only when they should, or print instead.	§ 9
Thread and run variables	`thread_name`, `is_main_thread`; in Grasshopper `gh_run`, `gh_branch`, `gh_item`.	§ 9.2
Stop once per line	Continue and Step over finish a line that runs code several times.	§ 9.6
Run and Debug buttons	Run always runs plainly, Debug always debugs, breakpoint or not.	§ 10.1
Break now: Ctrl+Alt+B	Pauses a debug run at its next line, also on Rhino's UI thread; in your script only; a sound answers.	§ 10.2
FreePause	Rhino stays usable while a script is paused.	§ 10.3
Editor stays open	While debugging and for async scripts, so V°Threads can be used.	§ 10.4
No freeze from another pause	A debug run on the UI thread waits, Rhino usable, while another script is paused on its own thread.	§ 10.5

Scripts started from Rhino's command line

Stop and record	`RunPythonScript` at full speed; Ctrl+Alt+B stops it, and the editor opens it at the line with its values.	§ 10.6
Debug mode	`RunPythonScript` in the debugger, the editor invisible until the script pauses.	§ 10.6
ScriptEditor Run, Debug=On	The same for `ScriptEditor Run` and toolbar buttons.	§ 10.7

Threads

V°Threads	Every thread and where it is; its stack; Break now; Debug.	§ 11.1
Where is everyone?	The same while the script runs, for hangs; "deadlock?".	§ 11.1
Watches in every thread	Each watch drawn once per thread, `pt @Worker-1`, each with its eye.	§ 11.2
Freeze guard	A worker's pause that would freeze Rhino is skipped, and you are told why.	§ 11.3

Prompts and helpers

V°Python	Python in the paused frame; `show`, `bake`, `watch`, `edit`, `breakpoints`.	§ 12.2
V°PowerShell	A PowerShell prompt in the script's folder.	§ 12.3
V°Claude	Claude Code on the paused script: reads, evaluates, debugs, edits; code waits for `/run`; Tab completes commands.	§ 12.4

RhinoAI	Claude in the Rhino model and Grasshopper; offered for install.	§ 12.4.4
V°Log, V°Settings, V°Help	What happened, in plain words; every setting, on the tabs General (with Markdown), Rhino and Grasshopper; the manual.	§ 12.5
The V names	V°Watch, V°Python, ...: the mark between V and name is a setting.	§ 3.11
Drop a folder	Opens in the editor's Explorer.	§ 4.5

Grasshopper: debugging

Three modes	Normal, Stop and record (default), Always debug; the V° toolbar button goes round them.	§ 13.1
Ctrl+Alt+B on a component	Pauses it with its editor open; stops it and opens its editor at the line when closed.	§ 13.3
Hidden editors	Under Always debug every Python 3 component can pause, even if its editor was never opened.	§ 13.4
Editors follow the definition	Another definition opened or switched to: the script editors of the one left close.	§ 13.5
Which run is paused	A badge names the run, its branch or item, the inputs' paths.	§ 13.6
Which branch failed	Failing runs named; copy their paths; debug exactly that run, by data or by number.	§ 13.7
Wiring without pausing	An error right after rewiring doesn't bring up the editor.	§ 13.8
Recomputes held back	No "expired during a solution" dialog.	§ 13.9
async:true in a component	Its runs on a thread of their own: Rhino and Grasshopper stay usable; its threads pause.	§ 13.10

Grasshopper: while paused

Canvas usable	Pan, drag, hover, select while a component is paused.	§ 14
Compute downstream	A change is computed at once with what is downstream, from the paused component's last data.	§ 14.2
Marks	What depends on the paused component gets an orange veil and dashed wires.	§ 14.3

Grasshopper: Panels and script inputs

Panels as code	Scripts in Panels with line numbers, colors, no wrapping; edited in place.	§ 15.1
Panel at the stop	After Ctrl+Alt+B a Panel's script opens at the line, with what happened.	§ 15.3
Script inputs	A Panel's or a file's script edited in the Script Editor, saved back; debugged.	§ 15.4
Library input	Double-click: its files and its script open in the Script Editor.	§ 15.5
Show as	Autodetect, Python, C#, Markdown, Plain text, per Panel.	§ 15.6
Markdown in Panels	Rendered; double-click edits the source at that word; Markdig or the plug-in's parser.	§ 15.7

Grasshopper: components and prompts

V°Edit	Edit Grasshopper geometry in Rhino and feed it back: items, lists, trees; saved with the definition.	§ 16.1
V°Text	A Panel's text as Rhino text: Markdown, code in colors, plain text; Bake.	§ 16.2
V°Protect	Is a definition still the original? Green or red, and Revert.	§ 16.3
Prompts window	V°Claude, V°Python, V°PowerShell for the definition; each Claude answer one undo step.	§ 17

Rhino: Markdown and links

MarkdownText	Markdown as Rhino text; double-click to edit it, the same objects updated; changes in Rhino taken in; Undo a word at a time in the dialog, the whole edit in Rhino.	§ 18
Links	`gh:`, `ghview:`, `rhino:`, `rhinoview:`, `script:`, files and the web; into another file and back. a click, Ctrl+click or off (one setting for Rhino and Grasshopper); blue words, a line under them.	§ 19

Learning

Lessons	17 script lessons, 12 Grasshopper lessons, a Rhino example, an overview; your own copies.	§ 20
Feature test	`miniVivisection_Test.py`: automatic checks and a guided tour.	§ 4.4

3 Concepts

The features in this manual rest on a handful of ideas. Once these are clear, the rest follows: why a value can be drawn but not edited, why Ctrl + Alt + B sometimes pauses and sometimes stops, why a Grasshopper component needs its script editor to be paused. Each idea gets a short section here; the chapters after it give the details.

3.1 Debugging in two minutes

A *debugger* runs your script and stops it at a line you choose. While it is stopped, or *paused*, you can look at every variable and see what the script really computed, not what you think it computed.

Term	What it means
Breakpoint	A red dot left of a line number. Click there to set or remove one. The script pauses when it reaches that line.
Debug F5	Runs the script in the debugger, so it stops at breakpoints. (F5 without a breakpoint is a plain run; miniVivisection's **Debug** button always debugs, § 10.1.)
Paused	The script waits. The highlighted line is the *next* one to run; it has *not* run yet.
Continue F5	Runs on to the next breakpoint, or to the end.
Step over F10	Runs one line, then pauses again.
Step into F11	If the line calls a function, goes into it and pauses at its first line.
Step out Shift + F11	Finishes the current function and pauses where it was called.
Stop Shift + F5	Ends the run.
Variables tab	The editor's list of the paused script's variables, at the bottom.

The method is always the same: guess where things go wrong, set a breakpoint just before that place, look at the values, step forward, and narrow it down until you see the line that makes a wrong value. miniVivisection makes the *look* part faster: you see values as geometry, in the code, and in Rhino at once.

3.2 The paused frame

When the debugger pauses, the script is in the middle of a function, a *frame*, with its local variables, the globals of its module and everything it imported. Nearly everything miniVivisection does happens in this frame:

- A watch such as `curves[0].PointAtEnd` is a Python expression, evaluated in the paused frame at every pause. It sees the same names the paused line sees.

- A tooltip reads a name *as Python sees it at the hovered line*. Over a global `crv` at module level you get the global curve, even while paused inside a function with a local `crv`.
- **V°Python** runs what you type in the paused frame. An assignment changes the script's variable, and the script continues with the new value.
- Editing in Rhino writes the edited geometry back with an assignment in the paused frame.

Because watches run again at every pause, an expression with a side effect, such as `curves.pop()`, would act each time. Keep watches to expressions that only read.

Python evaluation needs the *paused thread*: the frame is found by walking up that thread's stack. When the script runs on Rhino's UI thread, the plug-in is already there. When it runs on a thread of its own, the plug-in hands the work to that thread through FreePause's wait loop (§ 10.3).

3.3 Drawn, not added

What **V°Watch** shows in the viewports is *drawn*: a display conduit paints copies of the values on top of the scene at every redraw. Nothing is added to the document, nothing is saved in the `.3dm`, and nothing has to be cleaned up. When the run ends, the drawing goes away.

Two features make real Rhino objects instead, on purpose:

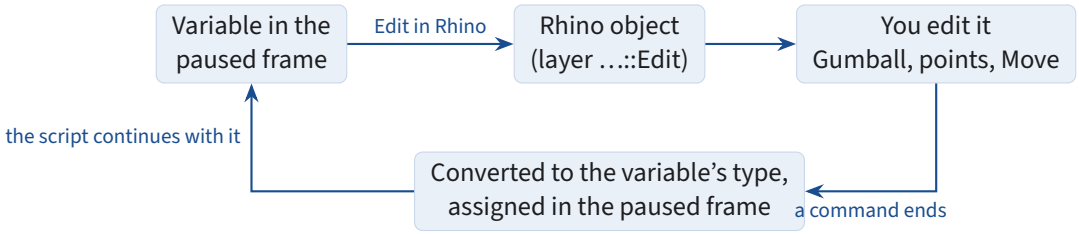
Bake copies drawn values into the document, on the layer `Script Editor Watch::Baked`, to keep them (§ 7.3).

Edit in Rhino turns a value into objects on the layer `Script Editor Watch::Edit`, so you can edit them with Rhino's tools. They are removed when the run ends (§ 7.1).

The same principle applies in Grasshopper: V°Edit's edit objects belong to the component, not to the model. They are never saved in the `.3dm` and don't mark the model as changed (§ 16.1).

3.4 Write-back

Editing in Rhino is a round trip:



The edited geometry is turned back into the variable's own type: a moved point into a `Point3d`, a curve that is still a line into a `Line`, and so on. A list item is written back as `curves[2] = …`. Rhino's Undo and Redo write the earlier or later geometry back too.

> **A write-back lasts for this run** What you change while the script is paused, by dragging points or typing into **V°Python**, changes the running script's data, not its code. When you have found the fix, change the code.

3.5 Rhino's UI thread and FreePause

Rhino has one *UI thread*: it draws the viewports, reads the keyboard and mouse, and runs commands. A script started from the Script Editor normally runs on that same thread. When such a script pauses, Rhino's own debugger keeps the thread busy in a **Select Action** prompt on the command line: typed text goes to that prompt, objects can't be selected, and only toolbar macros get through.

FreePause replaces that prompt. While the script is paused, Rhino sits at its normal `Command:` prompt and keeps its message loop going. You can select, run commands, use the Gumball and orbit the view. You continue and step with the Script Editor's buttons or keys (F5, F10, F11, Shift + F11, Shift + F5).

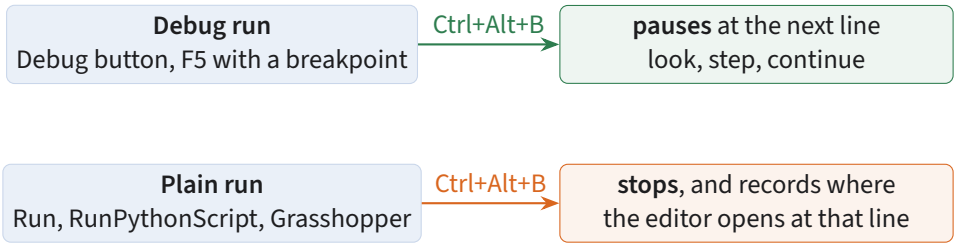
A script can also run on a thread of its own: add `# async:true` after its first line (`#! python 3`). Rhino then stays free even while the script runs, not only while it is paused. Such a script mustn't use Rhino's UI directly. The difference matters for threads (§ 11.3) and for Grasshopper (§ 13.10).

3.6 Debug runs and plain runs: pause or stop

A script can run in two ways, and the difference decides what Ctrl + Alt + B can do with it.

A debug run is traced by the debugger line by line. It stops at breakpoints, and Ctrl + Alt + B *pauses* it at its next line: you can look at it, step, and continue.

A plain run runs at full speed, without the debugger. Breakpoints are ignored. It can't be paused, but with miniVivisection Ctrl + Alt + B can *stop* it, and the plug-in records where it was: the file, the line, the call stack and the values of that moment. The Script Editor later opens the script at that line and explains what happened (*Stop and record*).



Which kind a script gets depends on how it is started:

- In the Script Editor, F5 debugs only when the script has a breakpoint, otherwise it is a plain run. miniVivisection's **Run** and **Debug** buttons make it explicit (§ 10.1).
- `RunPythonScript` and `ScriptEditor Run` run plainly, unless their *Debug* mode is chosen (§ 10.6).
- Grasshopper script components run plainly, unless *Always debug* is on or the run was started from the component's script editor with **Debug** (§ 13.1).

Ctrl + Alt + B is a Windows hotkey, taken only while a script runs. A script on Rhino's UI thread holds that thread, so nothing on screen can change; the hotkey answers with a sound instead. A soft sound means it paused or stopped something; an error sound means there was nothing it could do.

3.7 Threads

Python scripts can start threads (`threading.Thread`). The debugger traces each of them. When one thread reaches a breakpoint and pauses, the others stop at their next line, until you continue. **V°Threads** lists every thread and where it is.

A paused *worker* thread is dangerous when the script runs on Rhino's UI thread: the UI thread waits for the worker (in `join()`, say), the worker waits for you, and you can't click anything, because the UI thread is the one that would handle the click. Rhino freezes. miniVivisection's *freeze guard* skips such a pause and says why (§ 11.3). To debug threads, run the script with `# async:true`.

3.8 Grasshopper: components run inside a solution

A Grasshopper script component is not a script you run once. It runs once per item or per branch of its input data, inside a Grasshopper *solution*, on Rhino's UI thread. Three things follow:

Runs have names. A component that fails for some data fails in particular runs. miniVivisection names them by run number, branch (`{0;3}`) and item (`{0;3}[2]`), and a breakpoint can pause for exactly that data (§ 13.7).

The debugger lives in the component's editor. A component's breakpoints and stepping belong to its script editor window. With the editor closed, there is no debugger to pause in, so Ctrl + Alt + B can only stop the component. Under *Always debug*, miniVivisection gives every Python 3 component a hidden editor, so it can pause even if you never opened it (§ 13.4).

A pause holds the solution. While a component is paused, its solution is unfinished. Grasshopper doesn't allow recomputing during a solution. miniVivisection keeps the canvas usable, computes changes downstream with the paused component's last data, marks what depends on the paused component, and runs everything properly when the pause ends (Chapter 14).

The three modes for script components choose between these behaviours: *Normal* (plain runs, left alone), *Stop and record* (plain runs, Ctrl + Alt + B stops a hang, the default) and *Always debug* (every recompute in the debugger).

3.9 Markdown as documentation

Markdown is plain text with light marks: `# Heading`, `**bold**`, `*italic*`, `` `code` ``, `- list`, `[text](target)`. miniVivisection uses it in three places, with the same parser and the same link targets:

- **Panels in Grasshopper** show Markdown rendered and scripts as code (Chapter 15).
- **Markdown text in Rhino**, made by the `MarkdownText` command, is real Rhino text that you edit by double-clicking it (Chapter 18).
- **V°Text** turns a Panel's text into Rhino text in Grasshopper (§ 16.2).

Links connect these documents with what they describe: a note on the canvas can lead to a component, a Rhino object, a named view, another definition, a model or a script, and back (Chapter 19). The lessons use this: each ends with links to the one before and after, and an overview links to all of them.

3.10 Prompts and AI

Three prompts work on the paused script, in the Script Editor and in a window of their own for Grasshopper:

V°Python Python in the paused frame. Try a fix on live data before you change the code.

V°PowerShell A shell in the script's folder, for files, git and tools.

V°Claude A chat with Claude Code. Claude can read the script, the paused values and the run's errors, set breakpoints, step, propose edits and, with RhinoAI, work in the model and in Grasshopper.

Claude acts only through tools the plug-in gives it, and you keep control. Code it wants to run waits for your `/run`, edits to a paused script wait until the run ends, RhinoAI's destructive tools stay off until `/allow on`, and in Grasshopper each answer is one undo step (§ 12.4).

3.11 The V names and the two switches

Everything miniVivisection adds is named with a V, so it is told apart from the editor's own: **V°Watch**, **V°Threads**, **V°Python**, **V°PowerShell**, **V°Claude**, **V°Log**, **V°Settings**, **V°Help**, the Grasshopper components **V°Edit**, **V°Text** and **V°Protect**, and the switches **V°AI** and **V°Debug**. Its items in Grasshopper's menus start with **V°** too (**V° Show as**, **V° Debug this component**).

The mark between the V and the name is a setting (*Mark in the V names*): ° (the default), |, ǀ (a bar that stays on the line), or either bar with thin spaces (**V ǀ Watch**). It applies to Script Editors opened afterwards, and to V°Edit and V°Text components placed afterwards. This manual writes V°Watch. To search for it, type V° (Alt + 0176 on the number pad) or just the name.

Two switches in the Script Editor's top bar turn miniVivisection on and off as a whole:

V°Debug	Every debugging feature: the V tabs and pages, drawing, tooltip values, line-end values, the hover preview, watch expressions, conditional breakpoints (off: every breakpoint pauses), FreePause, the menu items. Off, they are all gone; on, everything is back as it was, with the settings unchanged.
V°AI	The **V°Claude** tab and page. Off, they are hidden in every Script Editor; their sessions keep running.

4 Getting started

4.1 Installing

miniVivisection runs in Rhino 8 (8.36 or later) on Windows. It comes as a Rhino package, `minivivisection-<version>-rh8_36-win.yak`, from Rhino's Package Manager or from the web page. The package holds the plug-in `miniVivisection.rhp`, its Grasshopper part `miniVivisection.GH.gha`, the libraries they need, the lessons in `examples`, and this manual.

Installing the package. In Rhino 8, run `PackageManager`, search for **miniVivisection**, tick **Include pre-releases** (it's a beta) and click **Install**. Or drag the `.yak` file from the web page onto Rhino 8's window: Rhino installs it as a package all the same. Restart Rhino: it loads from then on, its Grasshopper components too. `PackageManager` offers newer builds, however miniVivisection was installed (keep **Include pre-releases** ticked). To remove it: `PackageManager`, **Installed**, miniVivisection, **Uninstall**. Each version goes into a folder of its own, so keep your own `rhino_tools.json` in the plug-in's settings folder (`%APPDATA%\McNeel\Rhinoceros\8.0\Plug-ins\miniVivisection (…)\settings`), where it comes before the one next to the `.rhp`.

From the zip to the package. Earlier versions came as a zip. Not both: Rhino would find the plug-in twice. Close Rhino and double-click `miniVivisection_Remove_Old_Install.cmd` (from the web page, and in the package's `misc` folder). It lists what it finds – Rhino's registration of the zip copy, its Grasshopper link, copies of the Grasshopper part in Grasshopper's Libraries – asks, and moves them into a backup folder (registry keys as `.reg` files), so nothing is lost. The settings stay; the package uses them too. `-DeleteFiles` moves the zip copy's files out as well, `-ResetSettings` the settings, `-ListOnly` only lists. In Rhino, `PlugInManager` (miniVivisection, **Uninstall**) removes the registration too.

From a build folder. For development, the build folder `bin\Release\net7.0` holds the same files. Remove the package first, in `PackageManager`. Then:

1. Close Rhino if it runs an older miniVivisection: Rhino locks a loaded plug-in.
2. Start Rhino 8 and drag `miniVivisection.rhp` onto its window, or install it with `PlugInManager`. Do this once; from then on it loads when Rhino starts.
3. The command line confirms it: `miniVivisection 0.2.0 (build N), <date> loaded (experimental). …`
4. The Grasshopper components need nothing else. At its first load, the plug-in tells Grasshopper where they are (a `miniVivisection.ghlink` in Grasshopper's Libraries folder, if needed; a package needs none, and an old link is removed then); they appear from Grasshopper's next start, on the tab **miniVivisection**.

> **Coming from ScriptEditorWatchViz** After the rename, drag `miniVivisection.rhp` onto Rhino once: Rhino still looks for the old `ScriptEditorWatchViz.rhp`. The settings are kept: Rhino stores them by the plug-in's id, which didn't change.

4.2 Where it shows

Open the Script Editor (command `ScriptEditor`). miniVivisection adds to it in four places (Figure 4.1):

The top bar shows **miniVivisection 0.2.0 · 2026-10-09 14:05**: the version and when it was built, in your time zone. Next to it are the switches **V°AI** and **V°Debug** and the **V°Help** button, which opens the Markdown manual in the editor.

The bottom tabs **V°Watch**, **V°Threads**, **V°Python**, **V°PowerShell** and **V°Claude**, next to the editor's own Terminal, Problems, Variables and Watch tabs.

The left panel gets pages below Explorer, Search and Help: **V°Claude** (an orange speech bubble), **V°Watch** (a blue eye), **V°Threads**, **V°PowerShell** (a blue prompt), **V°Log** (a gray page) and **V°Settings** (a gear). Click an icon to show its page; click the shown page's icon to fold the panel. They are separate instances from the bottom tabs, laid out for the narrow panel.

The toolbar gets fixed **Run** and **Debug** buttons in place of the editor's switching Run button (§ 10.1).

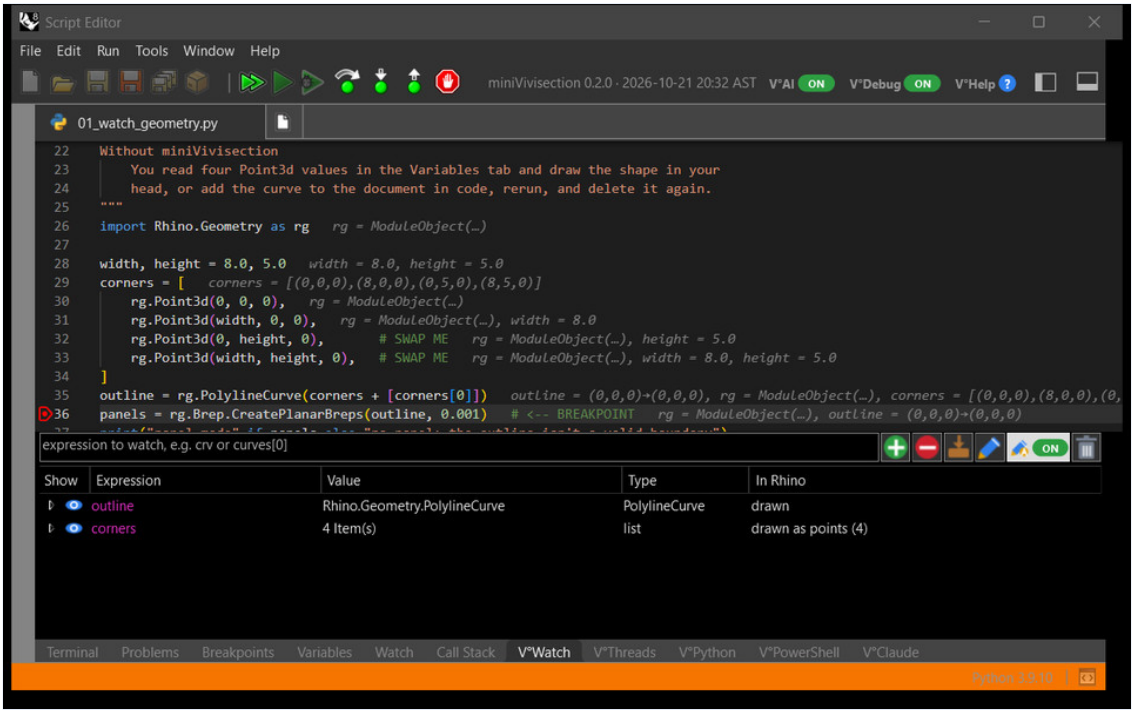


Figure 4.1: The Script Editor paused in lesson 01. Values at the ends of the lines (gray), **V°Watch** below, the V tabs along the bottom, and in the top bar the switches **V°AI** and **V°Debug**, **V°Help**, and the version.

Each tab and page can be turned off in **V°Settings** ▸ **Editor tabs and pages**. A Grasshopper component's script editor uses a *compact layout*: its left panel's icons sit in the status bar at the bottom, and the V pages get buttons there too, after the editor's own (Search, Templates, Examples, Help). Click one to show its page, again to fold the panel.

4.3 Your first session

Lesson 01 shows the core idea in two minutes.

1. In the Script Editor's left panel, open **Examples** ▸ **miniVivisection** and double-click `01_watch_geometry.py`. (Or open it from the `examples` folder next to the plug-in.)
2. Read the text at the top: *The bug*, *Try this*, *Without miniVivisection*.
3. Click left of the line marked `# <-- BREAKPOINT` to set a breakpoint, and press F5.
4. The script pauses. The ends of the lines show their values in gray.
5. In **V°Watch**, type `outline` and press Enter, then `corners`. Both are drawn in Rhino's viewports, each with a name dot. The outline is a bow tie, not a rectangle: two corners are swapped (cover picture).
6. Hover `corners` in the code. The tooltip shows its value and facts, and Rhino highlights it in yellow.
7. Press Shift + F5 to stop, swap the two corners in the code, and run again.

The other lessons go on from there, one feature each (Chapter 20). New to debugging? Start with `00_Examples_Guide.pdf` in the `examples` folder.

4.4 The feature test

`miniVivisection_Test.py`, next to the `.rhp`, checks an installation:

Part A runs by itself (Run or Debug): the plug-in is loaded, its build time and switches, and the drawing pipeline with real Rhino geometry: curves, Breps, meshes, points, polylines, lists, edit modes, raw write-back, and a visualizer with its inverse. One PASS or FAIL line per check, and a summary, in the Terminal.

Part B is a guided tour: set the breakpoints marked `<- BREAKPOINT A` to `D` and debug; the STEP comments say what to try at each.

4.5 Editor conveniences

Drop a folder anywhere on the Script Editor (tabs, panels, trays): it opens in the editor's Explorer in the left panel, as its **Explore Path** does, with an undo step (Ctrl + Z goes back to the folder before). Files dropped still open as scripts. Setting: *Drop a folder: open it in the editor's Explorer*.

Version The version, build number and build time in your time zone are shown in three places: the top bar, the load message on the command line, and **About V°Edit** in Grasshopper. Quote them when you report a problem.

V°Help opens the Markdown manual in the Script Editor, rendered. A second click shows its tab again. It works in a Grasshopper component's script editor too.

Part II

In the Script Editor

5 V°Watch: values drawn in Rhino

V°Watch is a list of expressions that are drawn in Rhino's viewports at every pause. It is a tab at the bottom of the Script Editor and a page in its left panel, next to the editor's own Watch tab, which shows values but doesn't draw them. Each editor has its own list, and it is kept between Rhino sessions.

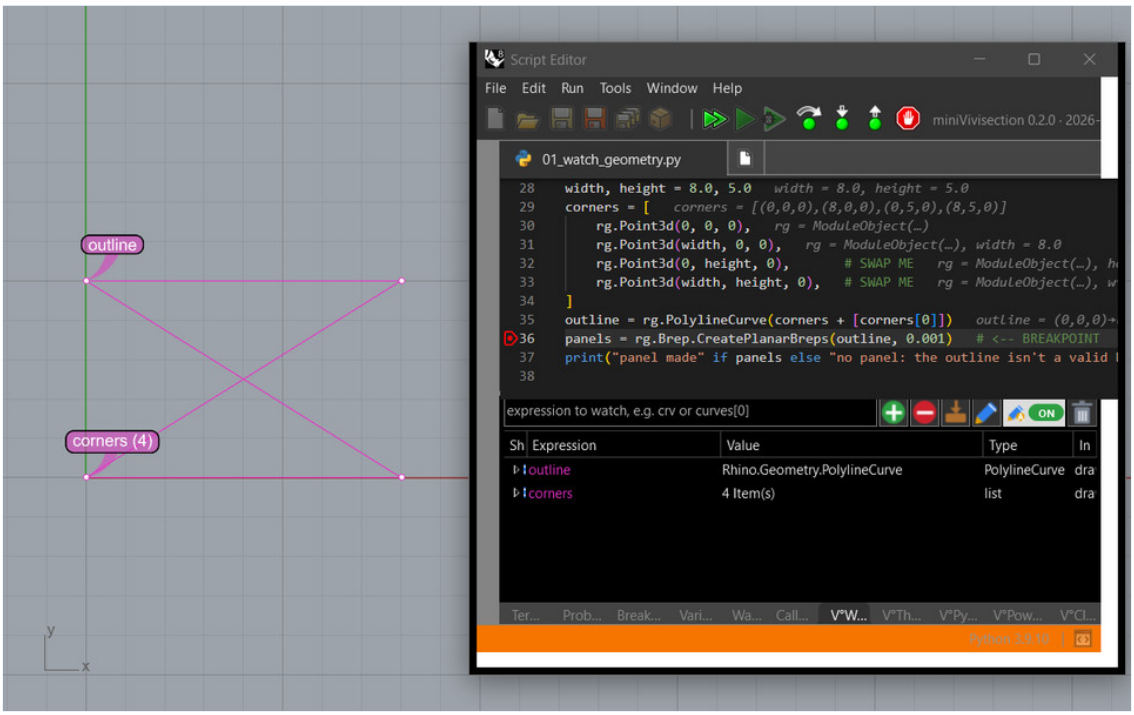


Figure 5.1: Lesson 01 paused: **V°Watch** lists `outline` (a PolylineCurve, *drawn*) and `corners` (a list, *drawn as points (4)*). Rhino's Top view draws them with their name dots: the outline crosses itself.

5.1 Adding a watch

- Type an expression in the box above the list and press Enter: a name (`crv`), an item (`curves[0]`), a slice (`pts[1:3]`), a member (`crv.PointAtStart`), a comprehension (`[c.PointAtEnd for c in curves]`) or a constructor (`rg.Line(a, b)`).
- Right-click a name in the code, or a row in the editor's Variables or Watch tab, and choose **Watch in Rhino**. A nested row counts as its path, such as `curves[0]` or `node.x`.
- In a tooltip, click **Watch in Rhino** (§ 6.2).
- In **V°Python**, `watch('crv.PointAtStart')`.
- Right-click a member row inside a watch (`Degree`, `[0]`) and choose **Watch in Rhino** to make it a watch of its own.

A watch added while paused is drawn at once. Each watch gets its own color.

5.2 The list

The list is a tree, like the editor's Variables tab.

Show	An eye: open and blue means drawn, closed and gray means hidden. Click it to hide a watch without removing it. A hidden watch's row is grayed out.
Expression	In its drawing color. Click the cell or press F2 to edit it in place; the watch is drawn again with the new expression.
Value, **Type**	As the editor's Variables tab shows them. Rows expand into the value's members: a curve's `Degree`, `Domain`, `PointAtStart`; a list's `[0]`, `[1]`, ... Rhino objects show their data as in the Variables tab.
In Rhino	What the last pause found: `drawn`, `drawn (3)` for three items, `drawn by show_sensors (12)` for a visualizer, the value's type with *nothing to draw*, or the error.

A row is grayed when the last pause couldn't evaluate it, and every row is grayed while the script runs or after it ended (*running*, *not running*).

Toolbar. One row above the list: the expression to add, then **Add**, **Remove** (or Del), **Bake**, **Edit in Rhino**, **Auto edit** (a switch with a green **ON** or gray **OFF** label) and **Clear all**. Their tooltips name them; the right-click menu has the same actions.

5.3 What is drawn, and how

Value	Drawn as
`Curve`, `Line`, `Polyline`, `Arc`, `Circle`, `Ellipse`, `Rectangle3d`	the curve
`Point3d`, `Point3f`, `Point`	a point
`Vector3d`, `Vector3f`	an arrow from the world origin
`Plane`	its axes, x red, y green, z blue, 60 pixels long at any zoom
`Box`, `BoundingBox`, `Brep`, `Surface`, `Extrusion`	shaded, translucent in the watch's color, with isocurves
`Mesh`, `SubD`	shaded, with mesh wires
`PointCloud`, `TextDot`	as such
a `Guid` of a Rhino object (rhinoscriptsyntax)	that object's geometry, in the watch's color
a .NET collection, a Python list, tuple or set	every drawable item, up to 4 levels deep and 10 000 items
your own class	what its visualizer returns (Chapter 8)
raw data: numbers in tuples and lists	by **Interpret as:** points, a polyline, vectors, ... (§ 5.4)

Other values aren't drawn; the **In Rhino** column says what they are. A generator or other iterator is left alone, since reading it would consume it.

Name dots. Each drawn value has a dot with its name (and item count), up and to the right of it, with a tail that curves down from the dot's bottom edge to a point at the value, like a speech bubble. That way the dot never covers a point, at any zoom. Four settings in **V°Settings ▸ Rhino ▸ Watches in Rhino**: *Name dots* (on or off; the values stay drawn), *Name dot offset* (pixels; 0 puts the dot on the value), *Name dot opacity* (%; lower lets geometry show through) and *Name dot pointer* (the curved *Tail* or a straight *Line*).

5.4 Interpret as

Raw data, such as `[(0,0,0), (5,0,0), (5,5,0)]`, can mean several things. Right-click a watch and choose **Interpret as**: *Auto* (the first, checked by default), or one of the choices that fit the value, read from the paused script. A triple can be a point or a vector; a list of triples points, a polyline, a closed polyline, an interpolated curve or vectors; pairs of triples lines or anchored vectors; six-number rows anchored vectors; point triples triangles; containers are taken item by item. *Plain data* draws nothing. See Chapter 8 for the full order.

5.5 Selection, both ways

- Selecting a row, in **V°Watch** or in the editor's Variables and Watch tabs, draws its geometry in Rhino's selection color, and selects its objects being edited.
- Clicking drawn geometry in a viewport selects its row (a list item's row when its list is expanded) and its variable. *Select a watch by clicking its drawing*: a plain click (the default), Alt + click, or off.
- Selecting an object being edited in Rhino highlights its watch and selects its variable (*Selecting edited objects highlights their watch*).

5.6 Watches in every thread

While a script runs more than one thread, each watch is also evaluated in the other threads and drawn once per thread, as `pt @Worker-1`. The details are in § 11.2.

5.7 C# and IronPython scripts

For C# scripts and IronPython 2 scripts, watches of plain variable names are drawn and baked; expressions need Python 3.

> **Try it: lessons 01, 06, 07** `01_watch_geometry.py`: V°Watch. `06_watch_expressions.py`: expressions in watches. `07_raw_data_interpret_as.py`: Interpret as.

6 Values in the code

While the debugger is paused, the code itself shows what the script computed: in the tooltips, at the ends of the lines, and as a preview in Rhino of whatever is under the mouse. You read the values where you read the code, without opening the Variables tab or adding watches name by name.

6.1 Tooltips

Hover a name in the code. The tooltip has three parts:

Its value, as a line of code, syntax-colored and compact: `count = 5`, `pts = [[0,0,0],[5,0,0],… +2]`, `p = (1,2,3)`, `node = Node(x=3,label='n1')`, `def scale(v,k=2)`. Long collections show their first items and how many more: `many = [0,1,2,… +247]`.

Facts, in gray, for collections and Rhino geometry: *250 items*; *closed · degree 3 · length 31.4 · 12 control points*; *solid · 6 faces · volume 1000*; *closed · 482 vertices · 960 faces*. After a step, *was 4* when the value changed.

The editor's own text, below a line: the type and its documentation.

Where a name has no value (not paused, not in scope) only the editor's own tooltip shows.

Scope. Names are read as Python sees them at the hovered line (§ 3.2). A global `crv` and a function's local `crv` show their own values. A local of a function that isn't running shows nothing.

6.2 Links in the tooltip

Under the value, the tooltip offers what can be done with it:

Watch in Rhino	Adds it to **V°Watch** (anything but plain numbers and text).
Edit in Rhino	Turns it into Rhino objects you can edit (geometry that can be written back, § 7.1).
Bake	Copies it into the document (§ 7.3).
Tree	Opens the value's members in a small window (below).
Watch rg.Point3d(…)	Over a class called on the line, such as `Point3d` in `pts.append(rg.Point3d(ri * math.cos(th), 0, h))`: the whole call, the value it makes, onto **V°Watch**, computed again at every pause. Offered only for classes: a watch of `pts.append(…)` would append at every pause. For any other expression, select it and right-click **Watch in Rhino**.

The Tree window opens at the mouse with the value's members, expandable like the Variables tab. It stays open while you step and reads the value again at every pause, keeping the rows you opened. It is grayed while the script runs, and closes when the run ends or with Esc.

6.3 Hover preview

The geometry of the name under the mouse, such as `crv`, `crv.PointAtStart` or `curves[0]`, is drawn in yellow in Rhino until the mouse moves on. It stays while the mouse is on the tooltip, so you can read the values and click its links, and while the code's context menu is open, so **Watch in Rhino** can take it from there. Setting: *Hover preview*; command `miniVivisection HoverPreview=Off`.

6.4 Values at the ends of lines

Each line of the paused function, from its start to the paused line, shows the values of the variables written on it, in gray after the line: `total = 6, k = 3`, as in VS Code and PyCharm.

After a step, values that changed are orange, and the tooltip says what they were before (*was 5*). The comparison is within the same function, and per thread: stepping into another function starts afresh. Settings: *Values at the ends of lines*, *What changed since the last pause*.

6.5 The editor's Watch tab evaluates expressions

The Script Editor's own Watch tab only resolves plain variable names and shows anything else as an error. With miniVivisection it evaluates any Python expression in the paused frame: `1+2` shows `3`, `crv.PointAtStart` its point, `[c.PointAtEnd for c in curves]` a list to expand. A Python error shows as the row's value (`NameError: name 'x' is not defined`). This tab isn't drawn: drawing is **V°Watch**'s job, and the two lists are separate. Setting: *Expressions in the editor's Watch tab*; command `miniVivisection WatchExpressions=Off`. C# keeps plain names, and an outer frame chosen in the Call Stack tab keeps the editor's own lookup.

6.6 Live Variables and Watch tabs

After a value is written back from Rhino, after a **V°Python** command, or after Claude changed something, the editor's own Variables and Watch tabs show the new values right away; expanded rows stay expanded. A moved curve's end point coordinates update while still paused.

6.7 Context menus while paused

Right-click a name in the code (the items act on the selection, or else on the name under the cursor), or a row in the Variables or Watch tab (the items act on that row's path):

- **Watch in Rhino**, and **Watch in Rhino as** ▸ with a chosen interpretation (§ 5.4);
- **Bake to Rhino**;
- **Edit in Rhino**.

The menus show only what can be used right now. While paused, most of the code view's own items are disabled, since the script is read-only then.

> **Try it: lesson 02** `02_values_in_the_code.py`: tooltips, values at line ends, *was*, the Tree; stepping with F10.

7 Edit in Rhino, and Bake

7.1 Edit in Rhino

Some fixes are easier to find by hand than with numbers: drag a control point until the curve looks right, then read off what it should have been. **Edit in Rhino** makes a paused script's value into real Rhino objects for that. Change them with any Rhino tool, and the change is written back into the paused script (§ 3.4). Python 3 scripts only.

Starting it. Choose **Edit in Rhino** in **V°Watch**'s toolbar, in a tooltip, or in the right-click menu of a name or a Variables row; or type `edit('crv')` in **V°Python**. The value becomes objects on the layer `Script Editor Watch::Edit`, one object per list item. Its drawing is hidden meanwhile, and its name dot moves to the edited objects and with them.

Editing. Move them, drag control points, use the Gumball, `Scale`, `Rebuild`, even Undo. When a command ends, its edits are written back together, with one redraw. The new geometry is turned back into the variable's type and assigned in the paused frame:

The variable was	Written back from
`Point3d`	a point
`Line`, `Polyline`, `Circle`, `Arc`	a curve, if it still is one
`Box`	a Brep
`Vector3d`	its arrow
`Curve`, `Brep`, `Mesh`	as they are
a list item	written as `curves[2] = …`
raw points, polylines (Interpret as)	written into the list; lists stay the same lists, tuples become tuples
your class with a visualizer	through its `@rhinoviv.inverse` function (Chapter 8)

Undo. Ctrl + Z and Redo write the earlier or later geometry back too.

At every pause the objects take the script's current values. **When the run ends**, they are removed; `miniVivisection StopEditing` removes them earlier.

Limits.

- Write-back works only while paused. With the script on a thread of its own, it needs FreePause.
- An edit that changes the kind of geometry, a circle made into a free-form curve, say, is reported and not written back.
- Planes can't be edited.
- Other interpretations of raw data (Interpret as) are drawn only.

7.2 Auto edit

With **Auto edit** on (the default), every shown watch whose value is geometry becomes editable objects at each pause, and right away for a watch added or shown while paused. It is **V°Watch**'s toolbar switch, and one setting for all editors (*Auto edit*): every **V°Watch** tab and the settings show the same state. Turning it off removes those objects.

7.3 Bake

Bake copies drawn values into the document, on the layer `Script Editor Watch::Baked`. Each copy is named after its expression and uses its color; a list's items are grouped. It is one undo step.

Bake from **V°Watch**'s toolbar (the selected watch, or per thread row, § 11.2), from a tooltip, from the right-click menu (**Bake to Rhino**), with `bake(value)` in **V°Python**, or with `miniVivisection BakeAll`, which bakes everything drawn right now.

> **Pause before a crash** A script that crashes after a long computation loses its results. Set a breakpoint just before the line that fails, and bake what was computed so far.

> **Try it: lessons 03, 04** `03_bake_before_a_crash.py`: Bake. `04_edit_in_rhino.py`: Edit in Rhino, Auto edit.

8 Visualizers and Interpret as

Rhino geometry draws itself. Your own classes, such as a `Sensor`, a `Room` or a `Truss`, don't: miniVivisection can't know which of their fields are coordinates. A *visualizer* tells it. Raw data, numbers in tuples and lists, can mean several things; *Interpret as* chooses one.

8.1 The order

What a watch is drawn as is decided in this order:

1. **A visualizer** registered for the value's type.
2. **Rhino geometry**, as it is (§ 5.3).
3. **Raw data** by Interpret as: a triple as a point or vector; a list of triples as points, a polyline, a closed polyline, an interpolated curve or vectors; pairs as lines or anchored vectors; six-number rows as anchored vectors; point triples as triangles; containers item by item; or plain data, not drawn.

8.2 Writing a visualizer

`rhinoviv.py` ships with the plug-in (in `python` next to the `.rhp`) and can be imported in Rhino's Python. A visualizer is a function that returns geometry, one item or a list, for a value of its class:

```
import rhinoviv
import Rhino.Geometry as rg

class Sensor:
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

class SensorArray:
    def __init__(self, sensors):
        self.sensors = sensors

@rhinoviv.visualize(SensorArray)
def show_sensors(array):
    return [rg.Point3d(s.x, s.y, s.z) for s in array.sensors]

@rhinoviv.inverse(SensorArray)        # optional: makes it editable in Rhino
def update_sensors(array, edited):    # edited: one geometry, or the list in drawing
    order
    for s, pt in zip(array.sensors, edited):
        s.x, s.y, s.z = pt.X, pt.Y, pt.Z
```

With the visualizer registered, a watch of an array draws its sensors as points; **In Rhino** says `drawn by show_sensors (12)`. With the inverse, **Edit in Rhino** and Auto edit work on it: drag a point,

and the sensor's coordinates change in the paused script. A value without an inverse can't be edited.

8.3 Where to register them

In the script itself, as above.

In V°Python while paused: they count at once.

In visualizer scripts: *Visualizer scripts* in **V°Settings** ▸ **Visualizers** lists Python files, separated by `;`. They run in the paused script's context at the first pause of every run, and again when saved (*Reload a visualizer script when it's saved*).

In **V°Python**, `visualizers()` lists what is registered and `reload_visualizers()` runs the visualizer scripts again.

8.4 Choosing an interpretation

- **V°Watch** ▸ **right-click** ▸ **Interpret as:** the choices that fit the value, *Auto* first.
- **Watch in Rhino as** ▸ in the editor's context menus adds a watch with one.
- `show(pts, as_="polyline")` in **V°Python** draws with a chosen interpretation.

When raw points, polylines and single points are edited in Rhino, they are written back into the list: lists stay the same lists, tuples become tuples. Other interpretations are drawn only.

> **Try it: lessons 07, 10** `07_raw_data_interpret_as.py`: Interpret as. `10_visualizers.py`: visualizers.

9 Breakpoints with conditions

A plain breakpoint pauses every time its line runs. In a loop of a thousand rounds, where the 731st goes wrong, that is a thousand pauses. miniVivisection adds conditions, hit counts and log messages, as in VS Code.

9.1 Setting them

Right-click a line in the code and choose one of these. A small box opens at the mouse: Enter applies, Esc cancels, an empty text clears it. A breakpoint is made on the line if there is none.

Breakpoint Condition…	Pauses only when the expression is true, such as `i % 3 == 0`.
Hit Count…	`5` (the 5th hit), `>= 5`, `> 5`, `<= 5`, `< 5`, `% 10` (every 10th). Counted from the run's start, when the condition holds.
Log Message…	Prints a line to the Terminal instead of pausing: `i = {i}, n = {len(pts)}`.
Remove Condition	Back to a plain breakpoint.

9.2 Conditions

Python scripts: any Python expression, evaluated in the script's frame at that line: `i > 3 and crv.IsClosed`.

C# scripts: a small evaluator over the frame's variables: literals, names, members, indexers, method calls (and `Math.`), arithmetic, comparisons, and `&&`, `||`, `!`, which short-circuit, so `crv != null && crv.IsClosed` works. No LINQ, lambdas, casts or generics.

IronPython 2 scripts: the same evaluator, which also reads Python's `and`, `or`, `not` and `True`, `False`, `None`: `i > 3 and name == 'a'`.

A condition that fails, by a typo or a missing name, *pauses*, as without a condition. The marker turns red, and **V°Log** says why, once per run.

Variables only conditions know.

`thread_name`	`"main"`, `"script"` for an async script's thread, or the thread's name: `thread_name == 'Worker-3' and i > 5`
`thread_id`	the thread's id
`is_main_thread`	`True` on the script's main thread
`gh_run`	in Grasshopper: the component's run number
`gh_branch`	in Grasshopper: the run's branch, `"{0;3}"`

`gh_item`	in Grasshopper: the run's item, `"{0;3}[2]"`; `gh_item == "{0;3}[2]"` pauses only for that data

9.3 Log messages (logpoints)

A log message prints instead of pausing, with each `{expression}` filled in; `{{` and `}}` are literal braces. The lines go to the Terminal and to **V°Log**. While the script runs threads of its own, each line starts with its thread, `[Worker-2] i=3` (*Logpoints name their thread*). In Grasshopper, `run {gh_run} {gh_item}: n={n}` labels every line with its run.

9.4 In the code and the Breakpoints tab

A breakpoint with a spec shows it after its line: ◆ `if i > 3 · hit >= 2 · 4 hits`. Its tooltip has the details. The specs move with their lines while you edit, and are kept per script file between sessions. The editor's Breakpoints tab gets a column with the same text, a right-click menu with the same four items, and a double-click on it edits the condition. In **V°Python**, `breakpoints()` lists them with this run's hits. Claude's `set_breakpoint` takes a condition, hit count and log message too.

9.5 Stepping ignores conditions

F10, F11 and Shift + F11 step as usual; only continuing (F5) is filtered by conditions, as in VS Code.

9.6 Stop once per line

Python's tracer reports a line for every round of code within that line: each item of a comprehension such as `y = [r.Line for r in rungs]`, a generator expression or lambda, each round of a one-line loop. A breakpoint there paused once per item, and **Step over** inside a comprehension too. That is how RhinoCode's debugger, like Python's `pdb`, works.

With *Stop once per line* (on by default), such a line pauses once. **Continue** and **Step over** then finish the line instead of stopping for every item. **Step into** still goes into the comprehension, and a breakpoint with a condition, hit count or log message still decides for every item.

9.7 Turning it off

Conditional breakpoints, hit counts and logpoints in **V°Settings** ▸ **Debugger**, or the **V°Debug** switch, turns all of it off: every breakpoint pauses, and the markers are hidden.

> **Try it: lesson 05** `05_conditional_breakpoints.py`: conditions, hit counts, log messages.

10 Running, pausing, stopping

This chapter is about control: how a script is started, how to pause it when it doesn't stop by itself, and how to get out of a hang without losing Rhino. The idea behind it is in § 3.6: a debug run can be paused, a plain run can only be stopped.

10.1 Run and Debug buttons

The Script Editor's own Run button switches: F5 debugs when the script has a breakpoint, and otherwise runs it plainly, so **Break now** can't pause it. miniVivisection puts two fixed buttons in its place:

Run always runs normally, also when there are breakpoints.

Debug always runs in the debugger, also without a breakpoint, so Ctrl + Alt + B can pause it. For an `async:true` script it is **Debug (Async)**.

While debugging, the editor's own button shows as **Continue**. F5 is unchanged. Both buttons look like the editor's and gray out while a script runs. The same goes for a Grasshopper component's script editor (§ 13.2). Setting: *Run and Debug buttons*.

10.2 Break now: Ctrl + Alt + B

Ctrl + Alt + B pauses a script running in the debugger at its next line. This works even for a script without `async:true`, which holds Rhino's UI thread so that Rhino handles no other keys: the hotkey is taken by Windows, on a thread of its own, and only while a script runs.

- **It answers with a sound at once**, since nothing on screen can change while a script holds the UI thread: a soft *asterisk* sound when it pauses or stops something, an error sound when there is nothing it can do. **V°Log** says why.
- **It stops in your script.** While the script is inside a module it imported, it runs on until it is back in the script (*Break now stops in the script only*).
- **It works again after Continue**, also without breakpoints.
- **On a plain run** it stops the script instead, as *Stop and record* says (§ 10.6).
- The key is a setting: *Break now hotkey*: Ctrl + Alt + B (the default), Ctrl + Shift + B, Pause, Ctrl + Break, or Off.

Break now in V°Threads. **V°Threads** has a **Break now** button that pauses at the next line in any thread or the selected one, and a **Debug** button (the editor's **Run** ▸ **Debug**) that starts the script in the debugger without a breakpoint. From the command line: `miniVivisection BreakNow`.

> **Try it: lessons 15, 16** `15_deadlock.py`: Where is everyone?, Break now. `16_long_running_watch.py`: Break now in a running script, fix it while paused, run on.

10.3 FreePause: Rhino usable while paused

With FreePause (on by default), Rhino stays at its normal `Command:` prompt while the script is paused, instead of the debugger's **Select Action** prompt (§ 3.5). You can select objects, run commands and use the Gumball. You continue and step with the editor's buttons or keys: F5, F10, F11, Shift + F11, Shift + F5.

> **Commands while paused** Commands run while paused are nested inside the paused script. They can change or delete objects the script holds, and they become part of whatever undo record the run has open.

`miniVivisection FreePause=Off` brings back the original prompt from the next pause.

> **Try it: lesson 11** `11_rhino_usable_while_paused.py`: FreePause.

10.4 The editor stays open while debugging

The Script Editor minimizes itself when a script runs and comes back at a pause. While debugging, and for `# async:true` scripts, it now stays open, so **V°Threads** (Where is everyone?, Break now) can be used while the script runs. A plain run of other scripts still minimizes. Setting: *Keep the Script Editor open while debugging*.

10.5 No freeze from another script's pause

Rhino's debugger has one lock for every script it debugs. While a script is paused on a thread of its own (`# async:true`), a debug run starting on Rhino's UI thread, in another editor or a Grasshopper component, would wait for that lock and freeze Rhino. With miniVivisection it waits with Rhino usable, and goes on once you continue the paused script.

10.6 Scripts started from Rhino's command line

`RunPythonScript` runs a script without the debugger: breakpoints never pause, and a script that hangs used to mean killing Rhino. The setting *RunPythonScript command* chooses what happens:

Normal It runs plainly, as before. Ctrl + Alt + B can't stop it.

Stop and record (the default) It still runs plainly, at full speed and without the editor, but Ctrl + Alt + B **stops it**, and where it stopped is recorded: the file, the line, the function, its values. An error that ends the script is recorded too. The command line says so. **Open the Script Editor** afterwards (within two hours): it opens the script, goes to the line, and an overlay explains what happened and why, with the call stack and the values of that moment (Figure 10.1). No breakpoint is set. Run it again with **Debug**, and Ctrl + Alt + B pauses it; or click the line's margin to stop there.

Debug A Python 3 script (first line `#! python 3`) opens in the Script Editor and runs in its debugger, also without a breakpoint. The editor stays out of sight while the script runs. When a breakpoint or Ctrl + Alt + B pauses it, the editor comes up at that line and you debug on as usual. A run without a pause leaves it minimized.

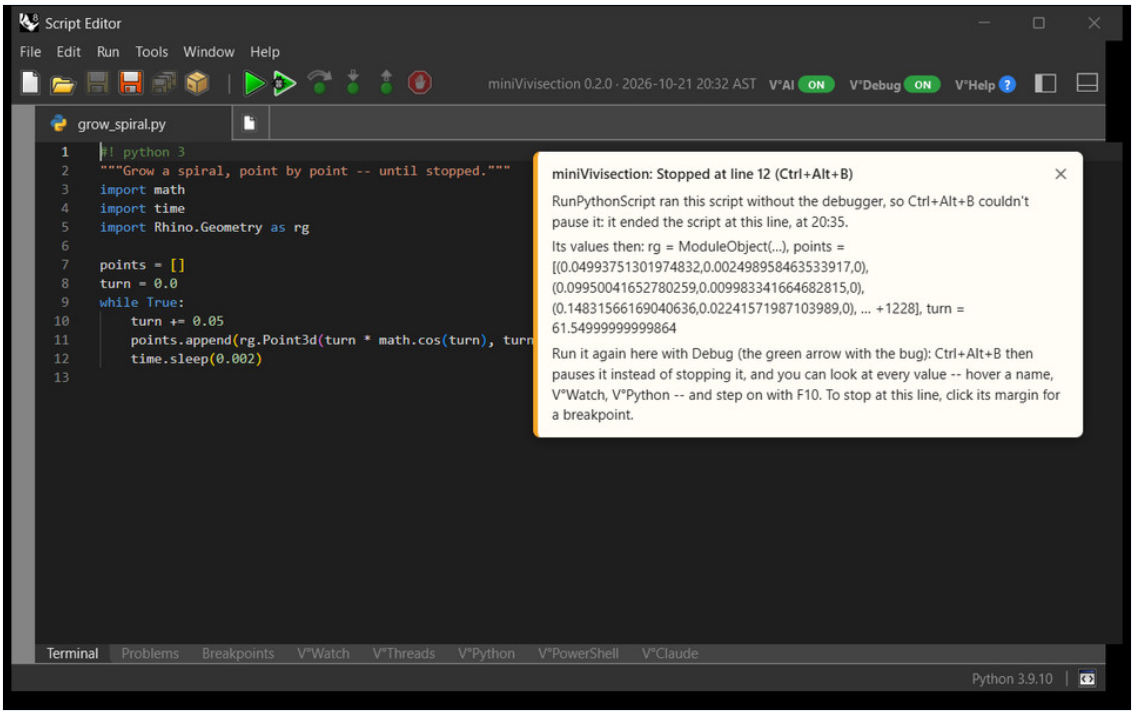


Figure 10.1: Stop and record: a script with an endless loop, run by `RunPythonScript` and stopped with Ctrl + Alt + B. The Script Editor opens it at line 12; the overlay shows the values of that moment (`points`, over a thousand by then, and `turn`), the call stack, and how to debug it instead.

Each run starts with a line on the command line: Python 2 or 3, and whether Ctrl + Alt + B can pause it, with a hint to switch the setting. Scripts without `#! python 3` run in RunPythonScript's own IronPython engine, as before; it has no debugger to stop.

> **The invisible editor** In *Debug*, the Script Editor that runs the script is made invisible while it runs, unless you had it open yourself; an editor left open by the last run's pause is hidden again. It draws, loads and runs normally and keeps its place and size, but shows nothing. It comes up only when the script pauses.

10.7 ScriptEditor Run, with Debug=On

Rhino's `ScriptEditor` command with **Run** (also used by toolbar buttons) gets a **Debug** option:

```
_-ScriptEditor _Run _Debug=_On "C:\path\script.py"
```

Off runs it plainly, as Rhino does. *On* opens it in the Script Editor and runs it in the debugger, the editor invisible until a breakpoint or Ctrl + Alt + B pauses it, as RunPythonScript's *Debug*. The default follows *RunPythonScript command*.

10.8 When an editor can't be reached

`miniVivisection StopPaused` stops every paused script: a way out when its editor is gone or hidden. A paused script whose editor window is closed for good is stopped too.

> **Try it: lesson 17** `17_runtime_error.py`: the debugger stops at a runtime error; values and call stack; Stop and record.

11 Threads

Python scripts that start threads (`threading.Thread`) are debugged thread by thread: a breakpoint pauses the thread that reaches it, and the others stop at their next line until you continue. Add `# async:true` after `#! python 3` so the script runs on a thread of its own and Rhino stays free (§ 3.5).

```
#! python 3
# async:true
import threading, time

def measure(n):
    time.sleep(0.1 * n)
    print("done", n)              # a breakpoint here pauses in each worker

workers = [threading.Thread(target=measure, args=(i,), name="Worker-%d" % i) for i in
    range(4)]
for w in workers: w.start()
for w in workers: w.join()
```

11.1 V°Threads

V°Threads, a tab at the bottom and a page on the left, has one row per thread at each pause, with where it is:

paused here	the thread that hit the breakpoint
stopped by the debugger	a thread held at its next line while another is paused
waiting in join(), wait(), with	waiting for another thread, an event, or a lock
in library code	outside the script's files
running	not stopped

Expand a row for its stack; double-click it to show its line in the open script.

Where is everyone? The same snapshot while the script runs, for hangs. When every thread waits, it says *deadlock?*, with their lines. From the command line: `miniVivisection WhereIsEveryone` (any run: it prints each thread and where it is).

Break now and Debug are on **V°Threads** too (§ 10.2).

11.2 Watches in every thread

While the script runs more than one thread, each watch is also evaluated in the other threads, read only, and drawn once per thread: `pt @Worker-1`, each in a color of its own. A thread that

hasn't got the name yet is listed as *not (yet) in*.

- Expand the watch: one row per thread, *(paused thread)* first, then `@Worker-1`, ... with each value; expand a row for its members.
- Each thread's row has its own eye: hide or show that thread's value. Hidden threads are kept by name with the watch, so a thread stays hidden at the next pauses and runs.
- The paused thread's value is labeled `pt @Worker-3 (paused)`.
- Selecting a row highlights that thread's drawing; clicking a drawing selects its row. Bake works per row; Edit in Rhino only for the paused thread.
- **V°Watch** ▸ **right-click** ▸ **In every thread** turns it off or on for a watch.

> **Note** A paused thread holds Python's lock: threads not started yet can't start until you continue. A breakpoint without a condition in the first worker pauses before the others exist.

11.3 The freeze guard

A script without `# async:true` runs on Rhino's UI thread. A breakpoint in one of its worker threads would freeze Rhino: the UI thread waits for the worker, the worker waits at the breakpoint, and the editor's **Continue** can't be clicked (§ 3.7). Such a pause is skipped, and the Terminal and **V°Log** say why and how to debug threads instead. Setting: *Freeze guard for threads*.

Grasshopper components always run on Rhino's UI thread, inside the solution. Their workers' breakpoints are skipped unless the component's script has `# async:true` (§ 13.10).

11.4 What isn't traced

Python code run by .NET threads, a `Task.Run` of a Python function for example, isn't traced: there are no breakpoints there. Such threads still show in **V°Threads**.

> **Try it: lessons 14, 15** `14_threads_async.py`: V°Threads, thread conditions, watches in every thread. `15_deadlock.py`: Where is everyone?, Break now.

12 Prompts: V°Python, V°PowerShell, V°Claude

Three prompt tabs work on the open, and often paused, script. A fourth page, **V°Log**, tells what happened in plain words.

12.1 Working in a prompt

The three tabs work like a terminal: the prompt (`>>>` , `PS C:\path>` , `>`) is printed after the output, and you type right after it. There is no separate input box.

Enter	runs the input
Shift + Enter	a new line (several lines run together, like a notebook cell)
↑ ↓	earlier inputs
Esc	clears the input
Home	the input's start
Ctrl + C	copies a selection; with nothing selected it interrupts: stops Claude's answer or the PowerShell command, or drops the Python input
Ctrl + mouse wheel	font size, for all three tabs (*Prompt font size*)
Tab	**V°Python**: four spaces; **V°Claude**: completes the `/` commands

Typing with the caret in the output adds to the input. The output can be selected and copied but not changed; the right-click menu has **Copy**, **Paste**, **Select all** and **Clear**. What you type in **V°Python** and **V°PowerShell**, and code blocks in Claude's answers, are shown in the Script Editor's syntax colors, light or dark theme. Each tab can switch that off (*Syntax colors: …*).

12.2 V°Python

The Script Editor's Terminal only shows output. **V°Python** is a Python prompt.

While the debugger is paused, input runs in the paused frame, with its variables, modules and functions. Assignments change the script's variables, which it continues with, and the watch drawing updates right away. A line says where code runs now whenever the script pauses or continues.

When not paused, input runs in the tab's own namespace, kept between inputs.

It works like Python's own prompt: expression results are printed, as are `print` output and errors; `for`, `if` and `def` blocks continue with `...`, and an empty line runs them; several lines at once run together and a final expression's value is printed.

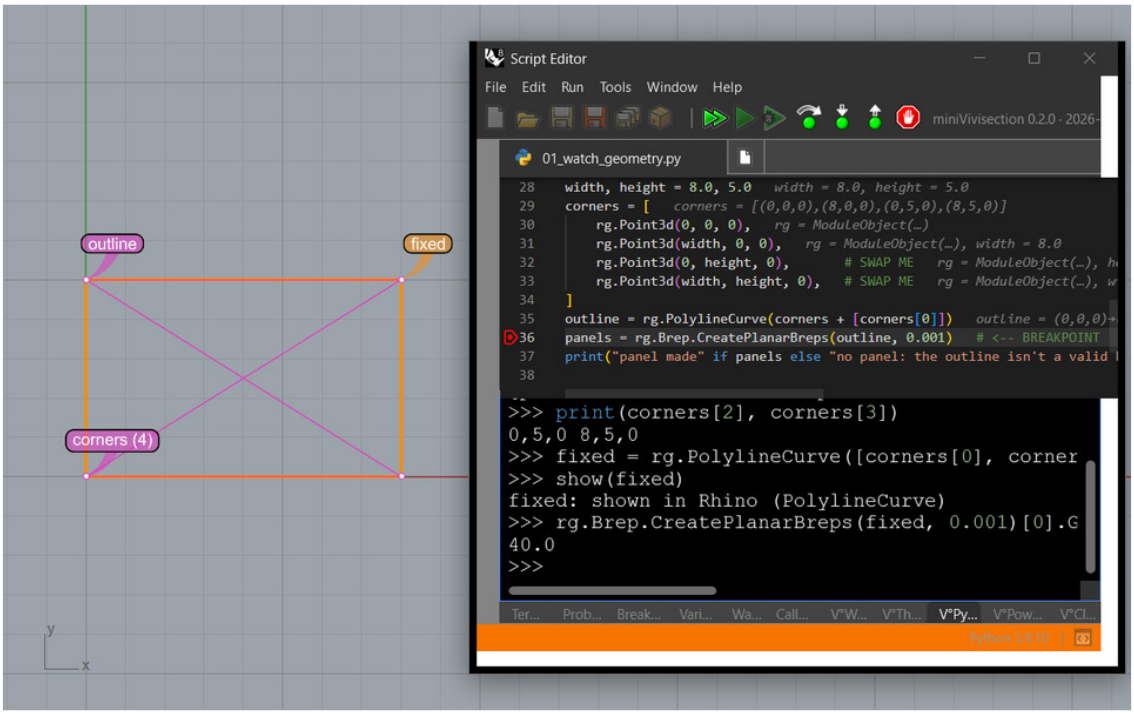


Figure 12.1: **V°Python** in lesson 01's paused frame: the swapped corners printed, a fixed rectangle made from them and drawn with `show(fixed)`, its area computed. The fix is tried on the live data before the code is changed.

Commands. Placed in Python's `builtins` only while an input runs, so they never end up in the script's variables; a script variable of the same name takes precedence. `help(show)` explains each.

`show(crv, curves[0])`	Draws values in Rhino until `clear()`, labelled with what was typed. In a loop, repeated calls count up: `c`, `c #2`, … `name=` sets the label.
`show(pts, as_="polyline")`	Draws raw data with a chosen interpretation.
`hide("crv"), clear()`	Removes shown values.
`bake(crv)`	Copies values into the document.
`watch("crv.PointAtStart")`	Adds an expression to **V°Watch** (while paused).
`edit("crv")`	Edit in Rhino, with write-back (while paused).
`visualizers(), reload_visualizers()`	Lists the registered visualizers; runs the visualizer scripts again.
`breakpoints()`	Lists the breakpoints' conditions and this run's hits.

Good to know.

- New variables created in a paused *function* don't survive into the function. That is a Python limit; changes to existing ones do.
- When the script runs on its own thread, the tab needs FreePause to reach the paused script.
- The editor's Variables and Watch tabs keep the values from the pause until the next step, and are refreshed after a command changed something.

- While an IronPython 2 script is debugged, an amber notice says that this Python 3 can't reach its variables; input runs in the tab's own namespace.

> **Try it: lesson 08** `08_python_terminal.py`: V°Python.

12.3 V°PowerShell

A PowerShell prompt: `pwsh` (PowerShell 7) if installed, else Windows PowerShell (*PowerShell executable*).

- One process per tab, started when the tab is first shown, in the open script's folder (else your user folder). Variables, functions and the current location persist between inputs.
- Output, errors, warnings and `Write-Host` are shown as PowerShell formats them. `cls` clears the tab.
- It is no real console: a program that waits for keyboard input (a REPL, a password prompt) blocks the prompt. Ctrl + C ends the process, and the next input starts a new one.
- No profile is loaded unless *Load the PowerShell profile* is on.

> **Try it: lesson 12** `12_powershell_terminal.py`: V°PowerShell.

12.4 V°Claude

V°Claude is a chat with Claude Code, Anthropic's command line agent, about the open script. Claude Code must be installed and signed in (the `claude` program); the **V°AI** switch shows or hides the tab.

12.4.1 What Claude can do

Claude gets the Script Editor's own tools from the plug-in, and, when RhinoAI is installed, RhinoAI's tools for the Rhino model.

Read	the open script with line numbers, the cursor and selection, the Terminal output, the Problems tab, the errors recent runs left behind (exceptions with type, message, line and traceback), the debugger's state and the paused variables, and the plug-in's own log.
Evaluate	a read-only look at an expression in the paused frame: type, value, and for geometry its key data (point coordinates, curve ends and length, bounding boxes); lists item by item.
Debug	set and clear breakpoints (with conditions, hit counts, log messages), start debugging, continue, step over, into and out, stop. Each step waits for the next pause and answers where it stopped and which variables changed; it shows in the tab as `[debugger] …`. (*Claude may control the debugger*.)
Draw	add watches, set Interpret as, register visualizers.

Run Python	in the paused frame, or in a scratch namespace: only after your `/run` (below).
Edit	the script, by exact replacement or by replacing the selection.
In Rhino	with RhinoAI: the document, objects, the view as an image, Grasshopper's canvas and components.

Each message carries a one-line summary of the editor (script, cursor, selection, debugger), and the selected code when it is short.

12.4.2 Commands

`/explain [question]`	the selection, or the whole script when nothing is selected
`/fix [note]`	reads what the last run left behind and the script, and fixes the script
`/why [question]`	(while paused) why the script stopped here: the exception, or how it got to this line
`/debug [question]`	(while paused) Claude inspects the paused frame
`/replace <instruction>`	an instruction for the selected code; the answer, one code block, replaces the selection, undoable with Ctrl + Z
`/run`, `/deny`	answer a request to run code
`/pending`, `/apply`, `/reject`	edits waiting: show, apply, drop them
`/undo [force]`	the script as it was before Claude's last changes; only while it is as Claude left it, `force` also after you changed it
`/history`	this chat's messages
`/allow on|off`	RhinoAI's destructive tools for this chat (off by default)
`/new`, `/clear`, `/help`	a new chat; clear the tab; the commands

Tab completes them: `/ex` and Tab gives `/explain` and a space. When several fit (`/re`), Tab completes as far as they agree and lists them with their help lines. It completes the arguments too (`/allow o` lists `on|off`, `/undo f` gives `/undo force`), and on an empty prompt it lists every command.

12.4.3 You stay in control

Running code asks first. Code Claude wants to run in Rhino can change the script, the model or files. With *Claude may run Python* on *Ask* (the default), the code is shown in the tab, with **Run** and **Deny** buttons above the chat and a `>` prompt, while Claude's answer goes on: click a button, or type `/run` or `/deny`. No answer within two minutes denies it. Plain expressions, a name, an attribute, an index, are read without asking. *Off*: never; *On*: without asking.

A running script isn't changed. While a run holds the script – running or paused in the debugger, or stopped at a runtime error until **Stop** – the run keeps the code it started with. Claude's edits are queued and shown as a change (`- old` / `+ new`); when the run has ended and the editor is back to editing, they are applied (*Claude's edits to the script*: *Automatic*), or wait for `/apply` (*Ask*).

Edits stay with their script. An edit is for the script the message was about. With another tab active meanwhile, it waits; `/apply` after switching back.

RhinoAI's destructive tools are off (`run_command`, its own `run_python`, opening and saving documents, materials, clearing canvases) until `/allow on` or *Allow RhinoAI's commands and scripts*. Read-only and modify tools, such as listing objects, the view image and placing Grasshopper components, are allowed.

Claude's reach is limited. Claude Code runs with only the Script Editor's tools, RhinoAI's allowed ones, and read-only file tools (Read, Glob, Grep). Bash, file edits, web fetches and searches are always refused, whatever your own Claude settings allow. It reads only your user settings, and runs in a folder of the plug-in's own, so a `.claude` folder or `CLAUDE.md` that came with a downloaded script isn't loaded.

12.4.4 RhinoAI

RhinoAI (McNeel's Rhino MCP Platform) gives Claude the Rhino model and Grasshopper. When it isn't installed, miniVivisection offers to install it once Rhino has started (**Install**, **Not now**, **Don't ask again**; *Offer to install RhinoAI*). It loads at Rhino's next start.

RhinoAI's server must run in this Rhino for Claude to reach the model. Start it with RhinoAI's `MCPStart` command, or let the tab start it before each message (*Start RhinoAI's server for Claude*; off by default, because the server listens on localhost without a token, so any program on this computer could use it). Otherwise the chat says once that Claude can't work in the model until you start it.

12.4.5 Sessions

Each script keeps its own Claude Code conversation, also between Rhino sessions; `/new` starts it over. The MCP servers' connection state is printed when Claude Code starts, and each answer's time and cost go to **V°Log**. Settings in **V°Settings** ▸ **Claude**: Claude Code's path, RhinoAI's router, the model, and the options above.

> **Try it: lesson 13** `13_claude.py`: V°Claude.

12.5 V°Log

A page in the left panel with what matters while working, in plain words: runs started and ended, where the debugger paused, exceptions, values written back from Rhino, objects baked, Claude's answers with their time and cost, PowerShell started or ended, warnings and problems. Colored by kind; right-click for **Copy**, **Clear** and **Open log folder**. Warnings and errors also go to Rhino's command line (*User log on the command line*). The files are listed in § 21.3.

12.6 V°Settings

A page in the left panel (the gear) with every setting, on three tabs: *General* (what the Script Editor and Grasshopper's script components share, and *Markdown*), *Rhino* (watches drawn in Rhino, selection, the `RunPythonScript` command, FreePause) and *Grasshopper* (script components, Panels, the prompts window), applied at once: tabs and pages appear or go, inline values and markers

are redrawn, Auto edit starts or stops. The same settings are in **Tools** ▸ **Options** ▸ **miniVivisection**. All of them are listed in Chapter 22.

Part III

In Grasshopper

13 Debugging script components

Rhino 8's Grasshopper script components have a script editor of their own, the same Script Editor with a compact layout. miniVivisection works there as in the Script Editor (Part II), and adds what a component needs: it runs once per item or branch, inside a solution, mostly without a debugger (§ 3.8). This chapter covers Python 3 script components. The legacy GhPython (IronPython) component isn't covered.

13.1 Three modes

Grasshopper script components (**V°Settings** ▸ **Grasshopper**) decides how components run:

Normal	Components run plainly. A component's breakpoints pause only when its solution was started from its script editor's **Debug**, and Ctrl + Alt + B leaves a component alone.
Stop and record (the default)	They run as plainly and fast, but Ctrl + Alt + B stops a component that runs (one that hangs), and its editor opens at the line where it was, with an explanation and the values then.
Always debug	Every recompute (a slider, an input) runs in the debugger, stops at the component's breakpoints, and Ctrl + Alt + B can pause it. With hidden editors, this works even for components whose editor you never opened.

The V° button. Grasshopper's toolbar gets a **V°** button on the right, next to the preview buttons. Each click goes to the next mode: *Stop and record* (V° with a green Run badge) → *Always debug* (V° with the Debug badge) → *Normal* (V°, gray and faint) → *Stop and record*. Setting: *Grasshopper toolbar button*.

13.2 Run and Debug in the component's editor

A component's script editor gets the same **Run** and **Debug** buttons (§ 10.1). **Run** recomputes the component plainly, breakpoints ignored, except under *Always debug*, which puts every solution in the debugger, Run's too. **Debug** recomputes it in the debugger, breakpoint or not. The editor's status bar has buttons for the V pages (§ 4.2).

13.3 Ctrl+Alt+B on a component

Its editor open: Ctrl + Alt + B pauses a running component at its next line (its debugger lives there). Continue, step and stop as usual.

Its editor closed (*Stop and record*, or without hidden editors): Ctrl + Alt + B stops a running Python 3 component, its remaining runs and the solution too, then opens its editor at the line it was on,

with an overlay that explains it. Press **Debug** to run it again in the debugger; Ctrl + Alt + B then pauses it. No breakpoint is set.

Ctrl + Alt + B stops a component that runs plainly first, whatever else is in the debugger: that component is what holds Rhino.

13.4 Hidden editors

A component's debugger lives in its script editor. With *Hidden editors for Grasshopper components* (on by default), *Always debug* gives every Python 3 script component of the open definitions its editor, out of sight: one every two seconds, without taking the focus. Its recomputes then run in the debugger too: a breakpoint or Ctrl + Alt + B *pauses* it, and its editor comes up at that line.

- Opening its editor yourself shows it; closed again, it is hidden again.
- Switching to Normal, or the option off, closes the hidden ones; a deleted component's goes with it.
- Hidden editors stay out of the taskbar and Alt + Tab until they come up.
- Costs: each one is a Script Editor window (at most 24), and every run is in the debugger, which is slower.
- A definition's very first solution, right when it is opened, runs before its editors are there.

13.5 Editors follow the definition

Open another definition, or switch the canvas to another open one, and the script editors of the definition you leave close first – as their close buttons would, so Grasshopper may still ask about changes not applied. The editors on the screen do; hidden ones (above) stay with their components, and the editor of a script *paused* in the debugger stays open (closing it would end the pause; the command line says so). *Close script editors when the definition changes* (on by default) switches it off.

13.6 Which run is paused

While a script component is paused, a badge at the top of its code shows the run: *Grasshopper: 'Py3' run 4 · {0;3}[2] · x {0;3}[2], y {0}*: the component, the run number, its branch or item, and every input's path. **V°Log**'s pause line and **V°Python**'s status line say the same; in **V°Python**, `ghenv.Component.RunCount` is the run number.

Conditions and log messages know the run too: `gh_run`, `gh_branch` and `gh_item` (§ 9.2).

13.7 Which branch failed

When runs fail, a remark on the component names them: run number, branch `{0;3}`, item `{0;3}[2]`, the line. The data named is each input's own, as Grasshopper had it for that run. The component's right-click menu then offers:

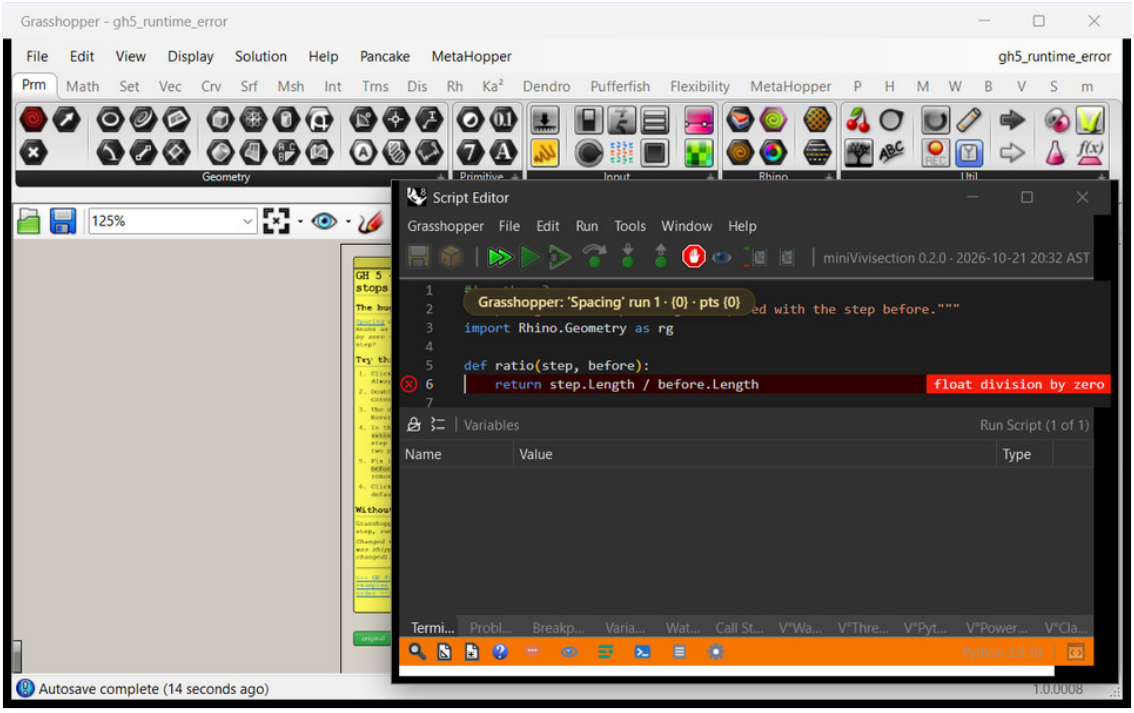


Figure 13.1: Lesson GH 5: *Spacing* paused at its error, line 6 (*float division by zero*). The badge names the run: *Grasshopper: 'Spacing' run 1 · {0} · pts {0}*.

V° Copy failing branches / Copy failing items One path per line: paste it into a Panel, and feed Tree Branch's *Path*, or V°Edit's *Index*, to isolate exactly that data.

V° Debug failing run N – data {0;3}[2] Its editor opens, a one-time breakpoint is set at the error line for that run only, with the condition `gh_item == "{0;3}[2]"`: the same data, wherever it is in the order now. The component recomputes in the debugger.

V° Debug failing run N – run number The same, with `gh_run == N`: the same run number.

V° Debug this component Recomputes it in the debugger, pausing at its first line.

13.8 Runtime errors and wiring

Under *Always debug*, the debugger stops at a runtime error in a component, as it does in the Script Editor, with the values and the call stack of that moment.

A runtime error right after wires were connected or disconnected, a Ctrl + drag that takes an input away for example, doesn't pause, so the editor doesn't come up in the middle of the wiring. The component shows the error, **V°Log** says why, and recomputing it pauses at the error as usual.

13.9 Recomputes held back

While a component's script is paused, Grasshopper must not recompute anything: it would show *Grasshopper breakpoint: … expired during a solution*. Whatever asks for that meanwhile, a slider, a save, an edit, is done right after the solution ends; **V°Log** says so. With *While paused: compute changes downstream*, changes are computed at once instead (Chapter 14).

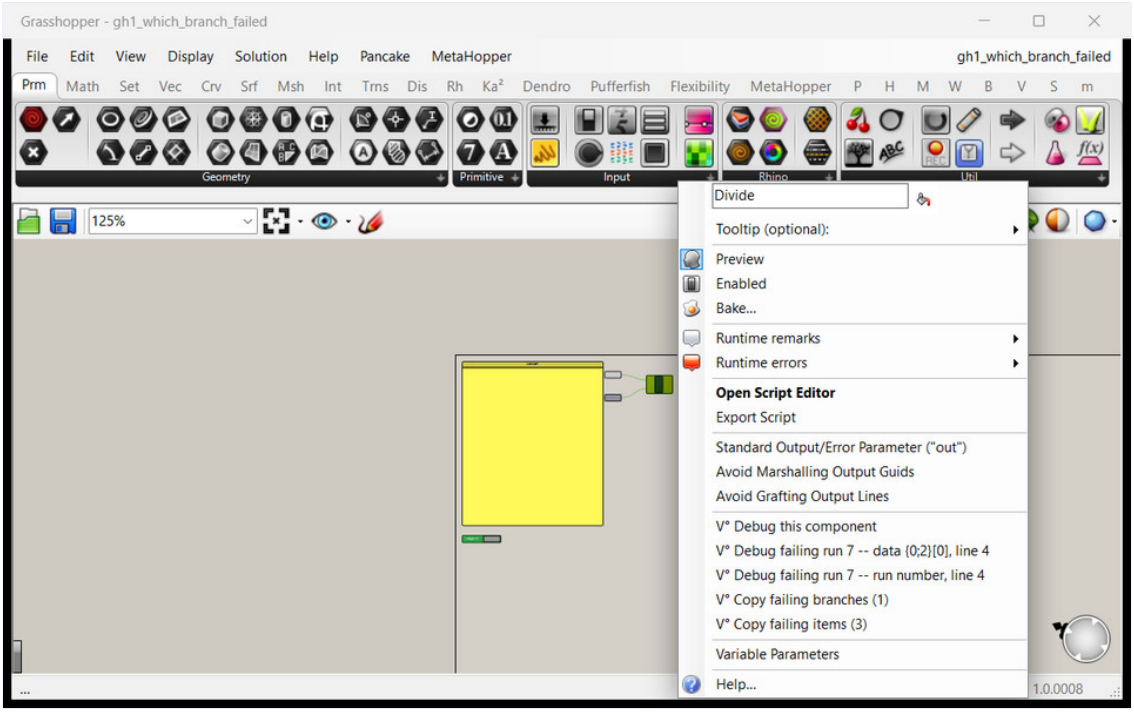


Figure 13.2: Lesson GH 1: *Divide* fails for some of its 12 runs. Its menu names the failing run by its data, *{0;2}[0]*, and by number, and copies the failing branches and items.

13.10 async:true in a component

Grasshopper runs a component on Rhino's UI thread, so a long-running component freezes Rhino's views and the canvas, and a breakpoint in one of its threads can't pause (the freeze guard skips it, § 11.3). Add `# async:true` to its script, the line after `#! python 3`:

- Its runs, plain ones and in the debugger, go to a thread of their own, as a Script Editor script's would. Rhino's views and Grasshopper's canvas stay usable while it computes.
- Its threads pause, step and show in **V°Threads**.
- The definition holds still meanwhile: changes recompute when it is done; deleting waits.
- Its script mustn't use Rhino's UI or document where they need the UI thread.

Setting: *async:true runs off the UI thread*.

ghpythonlib.parallel. `parallel.run` computes its first item on the component's own thread: a breakpoint in the worker function pauses for that one (lesson GH 7).

> **Try it: Grasshopper lessons** `gh1_which_branch_failed`: failing runs. `gh2_the_one_wrong_item`: Always debug, a condition, the badge. `gh3_stop_a_hang`: Ctrl + Alt + B with the editor closed. `gh5_runtime_error`: the debugger at a runtime error. `gh6_threads_in_a_component`, `gh7_parallel_run`: threads. `gh8_long_running`: Ctrl + Alt + B pauses a long-running component, a fix in V°Python, it runs on.

14 Grasshopper while a component is paused

A paused script component is paused inside a Grasshopper solution (§ 3.8). Grasshopper itself ignores the mouse while its solution runs, and forbids recomputing anything until it ends. miniVivisection keeps the definition usable during the pause, and keeps its data honest.

14.1 The canvas works

While a component is paused you can pan, zoom, drag components, select, and hover for wire and parameter data, as when nothing runs.

14.2 Changes are computed at once, downstream

A slider moved, a wire changed or a Panel edited while a component is paused is *computed at once with what is downstream of it*:

- The paused component isn't run again. Its outputs show the data of its last finished run, and what is downstream of it computes from that.
- Other script components aren't run either: one would start a second debug run.
- When the pause ends, everything is computed properly.

Setting: *While paused: compute changes downstream*. Off: all of it waits for the pause to end (§ 13.9).

14.3 What depends on the paused component is marked

The data downstream of a paused component most probably isn't right any more: it comes from the run before. So it is marked meanwhile: an orange veil and a dashed border on each object, *out of date? (paused upstream)* under it, and its wires dashed orange. Setting: *While paused: mark what depends on the paused script*.

14.4 What waits, and what is refused

A new solution waits for the pause to end: a paste, a component placed from the search box, an Alt + drag copy.

Deleting is refused until the pause ends, since the paused solution may still need the objects; **V°Log** says so. Adding is fine.

> **Try it: lesson GH 2** `gh2_the_one_wrong_item`: Always debug, a breakpoint condition, the run badge, a slider moved while paused.

15 Scripts and Markdown in Panels

A Panel is Grasshopper's text box. People keep two kinds of text in Panels: scripts, fed to a script component's *script* input, and notes. Grasshopper shows both as centered plain text. miniVivisection shows scripts as code, edits them in place, and renders notes as Markdown.

15.1 Scripts in Panels shown as code

A Panel holding a Python or C# script shows it in Grasshopper's monospace font, with line numbers and syntax colors, unwrapped. A long line runs on past the Panel's edge; a scroll bar along the bottom (drag it, or click the track) brings it into view, next to the Panel's own scroll bar on the right.

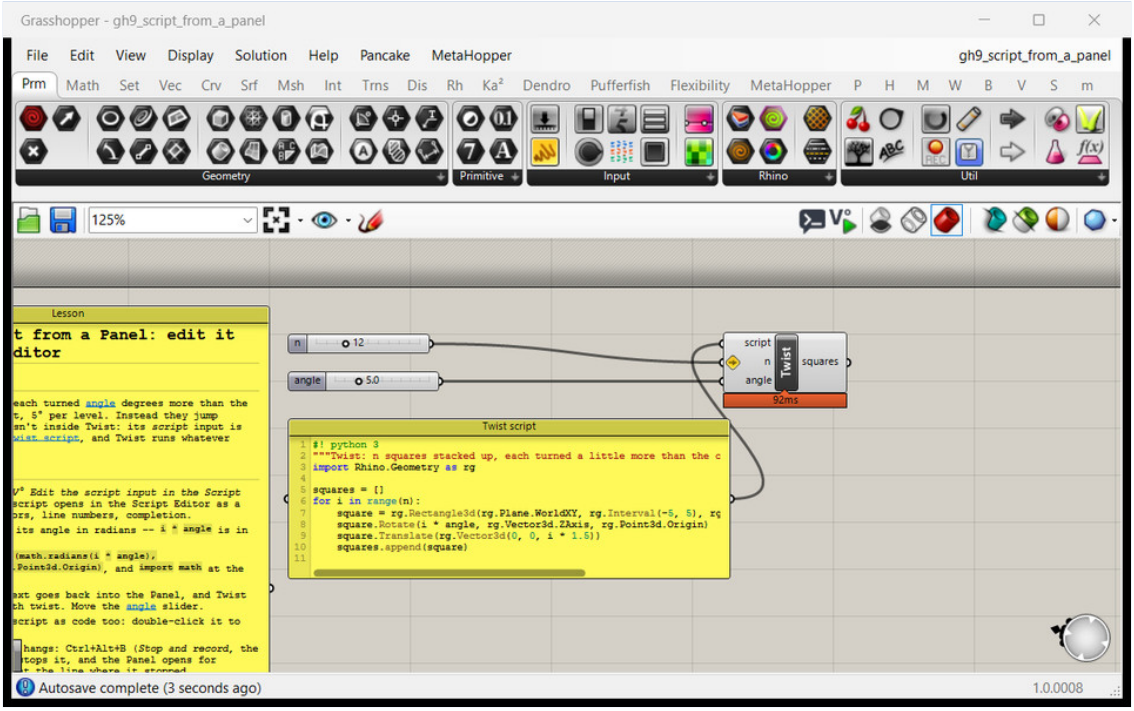


Figure 15.1: Lesson GH 9: the *Twist script* Panel holds the script the *Twist* component runs. It is shown as code, with line numbers and syntax colors.

When a Panel counts as a script. When its first line is RhinoCode's marker (`#! python 3`, `#! python 2`, `// #! csharp`); when it is connected to a script component's *script* input (that component's language); or when its lines are plainly Python or C#: imports, `def` and `class` lines, `using` lines, statements ending in `;`. Notes stay text. Setting: *Scripts in Panels shown as code*.

15.2 Editing a script Panel in place

Double-click a script Panel to edit it right there, looking the same:

mouse, Shift+arrows	caret and selection
Ctrl + Z, Ctrl + Y	undo, redo
Ctrl + C, Ctrl + X, Ctrl + V	clipboard
Enter	keeps the indentation; indents after a `:` in Python, a `{` in C#
Tab, Shift + Tab	indent, outdent
Ctrl + /	comments lines in or out
wheel, Shift + wheel	scroll, scroll sideways
click elsewhere, Ctrl + Enter, Ctrl + S	keep the changes: one step of Grasshopper's Undo
Esc	drop them

Its right-click menu has **V° Edit the script here**, and **Edit the script in the Script Editor**: a working file (`.py` or `.cs`) with completion and everything else, saved back into the Panel. Shift + double-click opens Grasshopper's own text box, without wrapping, in the monospace font, Tab as four spaces; Ctrl + double-click its dialog.

15.3 After Ctrl+Alt+B stopped a Panel's script

When Ctrl + Alt + B stopped a component that runs a Panel's script, the Panel opens for editing at the line where it stopped: the line marked, and an explanation over it, top right: what happened, the run's data, the values then, the call stack. Click it to put it away. The Panel keeps the line marked, with a line about it along its bottom, until its text changes. When the Panel isn't on screen, the Script Editor shows it all.

15.4 Scripts from a component's script input

A script component's *script* input can be fed by a Panel's text or by a file's path. Grasshopper then gives the component no editor of its own. miniVivisection opens the script in the Script Editor instead, as a file:

A file's path opens that file. Saving it recomputes the component.

A Panel's text opens as a working file. Saving writes the text back into the Panel, as an undo step, and the component recomputes. If the Panel changed meanwhile, the save is refused: open it again.

Text from anywhere else opens as a copy to read.

Open it with **right-click the component ▸ V° Edit the script input in the Script Editor**, or by double-clicking the component. After Ctrl + Alt + B stopped such a component, its script opens by itself at the line where it stopped.

Debugging it. **Debug the script input** in the component's menu runs it once in the debugger, pausing at its breakpoints, or at its first line when it has none. Under *Always debug* every run does, as long as the file holds what Grasshopper runs (a Panel's working copy equal to its text).

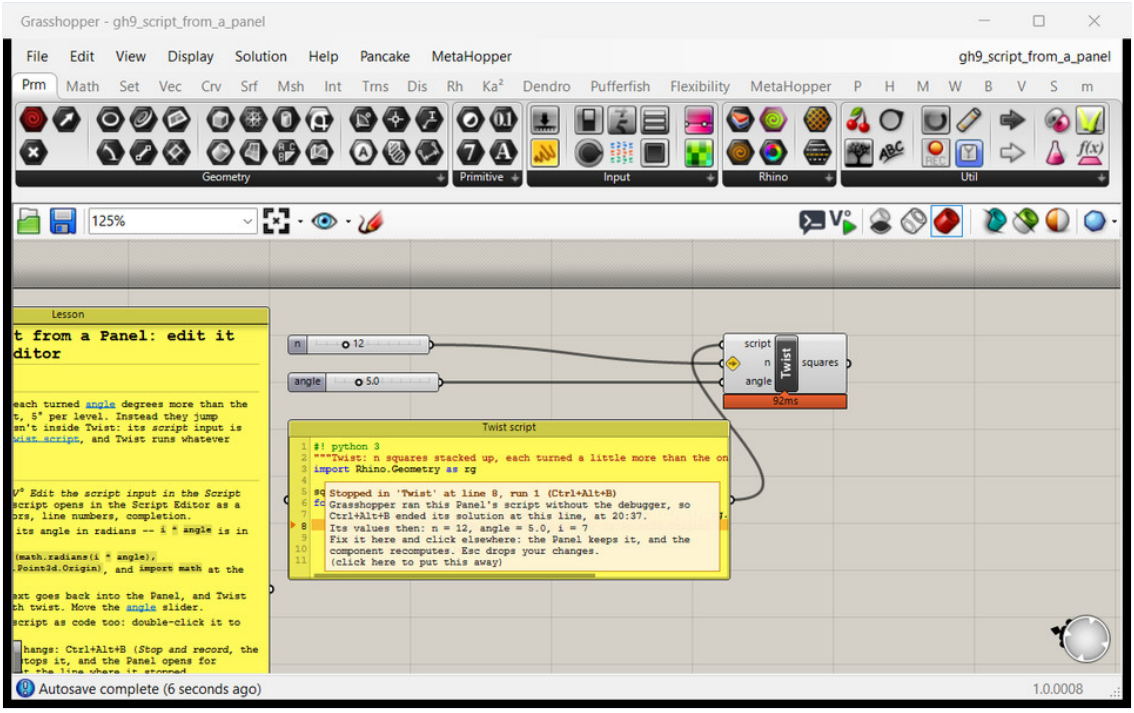


Figure 15.2: The script Panel open for editing at the stop: line 8 marked, and the explanation: *Stopped in 'Twist' at line 8, run 1 (Ctrl+Alt+B)*, the values `n = 12, angle = 5.0, i = 7`, and what to do.

15.5 A component with a script input

Grasshopper switches its own editor off for such a component. **Double-click it**, and its *library* input's files (Python or C#, up to 20) and its script open in the Script Editor, the script's tab in front. Its menu's **Open Script Editor** does the same, and **Export Script** saves its script to a file (Grasshopper grays both out for such a component). Any script component with a library input has **V° Open the library's files in the Script Editor** in its menu.

15.6 Show as

Every Panel's menu has **V° Show as**, saved with the definition:

Autodetect (the default)	code when it is a Python or C# script; rendered when it is plainly Markdown, that is two kinds of strong signs such as **bold**, `code`, links, lists, quotes or code fences, or one of them and a # heading; else as Grasshopper shows it
Python code, **C# code**	any Panel of its own text shown and edited as code, also one nothing marks as a script
Markdown (rendered)	rendered (below)
Plain text	never code or Markdown: shown from the top left, as code is, in Grasshopper's font and wrapped as its *Wrap Items* says (Grasshopper centers a Panel's text), with the same scroll bars

15.7 Markdown in Panels

A Markdown Panel shows headings, **bold**, *italic*, `code`, links, lists, quotes, code blocks, tables (as fixed-width text) and rules, laid out in the Panel's width.

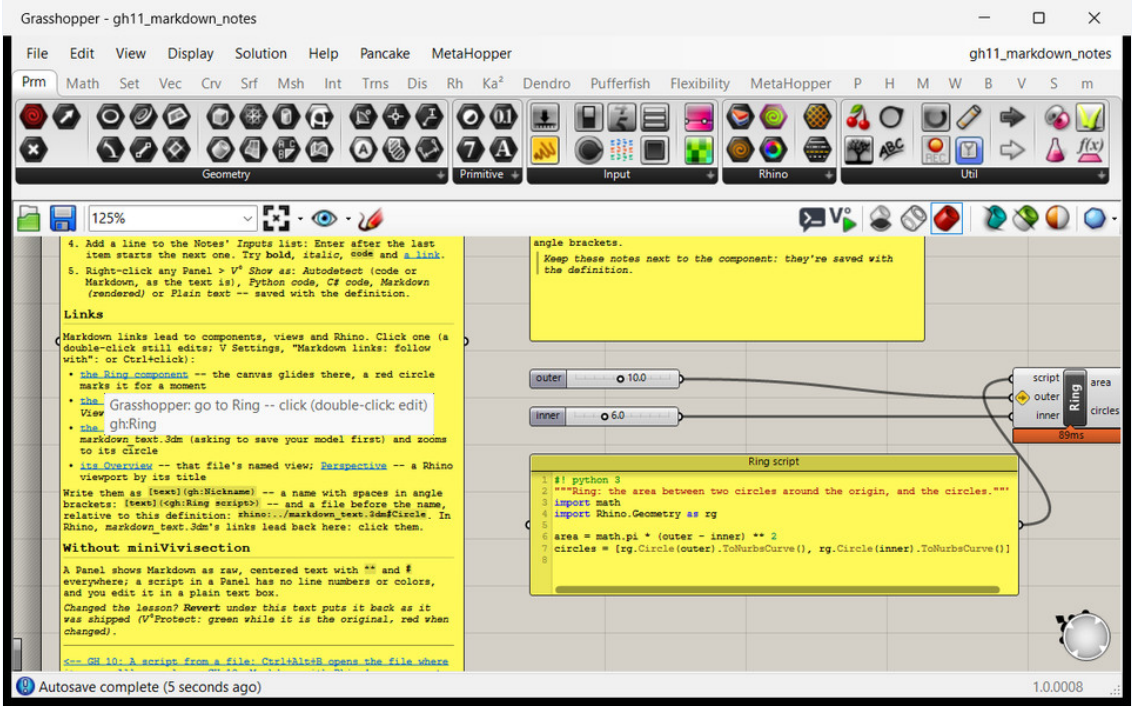


Figure 15.3: Lesson GH 11: the lesson Panel and the *Notes* Panel rendered as Markdown. The mouse is over a link; its tooltip says where it goes and how to follow it.

- **Double-click a word**, and the Markdown source opens in the Panel's editor with the caret at that very place. Enter goes on with a list or a quote. Click elsewhere and it is rendered again.
- The menu also opens it in the Script Editor, which shows Markdown too.
- **Click a link** to follow it – or Ctrl + click, as *Markdown links: follow with* says (Chapter 19). The pointer becomes a hand, and a tooltip says where it goes and how; a double-click still edits.

The parser. *Markdown parser*: *Rhino* (the default) is Markdig, Rhino's own, as in the Script Editor's preview: CommonMark's rules, nested lists and quotes, task lists with ☐ and ☑, footnotes, tables only with their header rule. *miniVivisection* is the plug-in's simpler parser. The same parser serves `MarkdownText` and V°Text. Setting *Markdown in Panels* off: Markdown Panels show their text as Grasshopper does.

15.8 Panels fed by another

A Panel whose input is connected shows its data, one item a line, the same way, when it opens like Markdown (a heading, a list) or starts with a script's marker line; **V° Show as** in its menu decides otherwise. It isn't edited in place: its data comes from upstream.

> **Try it: lessons GH 9 to GH 12** `gh9_script_from_a_panel`: a script from a Panel, saved back. `gh10_script_from_a_file`: a script file, Ctrl + Alt + B opens it at its line. `gh11_markdown_-`

> `notes`: Markdown notes, editing where you double-click, links. `gh12_markdown_parser`: Markdig's task boxes, nested lists, footnotes.

16 The components: V°Edit, V°Text, V°Protect

miniVivisection's Grasshopper part, `miniVivisection.GH.gha`, adds three components on the tab **miniVivisection – V°** when Grasshopper's tabs don't fit: Grasshopper shortens tabs from the right until they do (with the mark chosen for the V names, Section 3.11).

16.1 V°Edit: Grasshopper geometry edited in Rhino

Tab **miniVivisection ▸ Debug.** Some geometry is easier to fix by hand than with numbers. V°Edit sits in a definition between two components, puts chosen items of its input into Rhino as ordinary objects, and feeds your edits back into the definition. They are saved with the `.gh`.

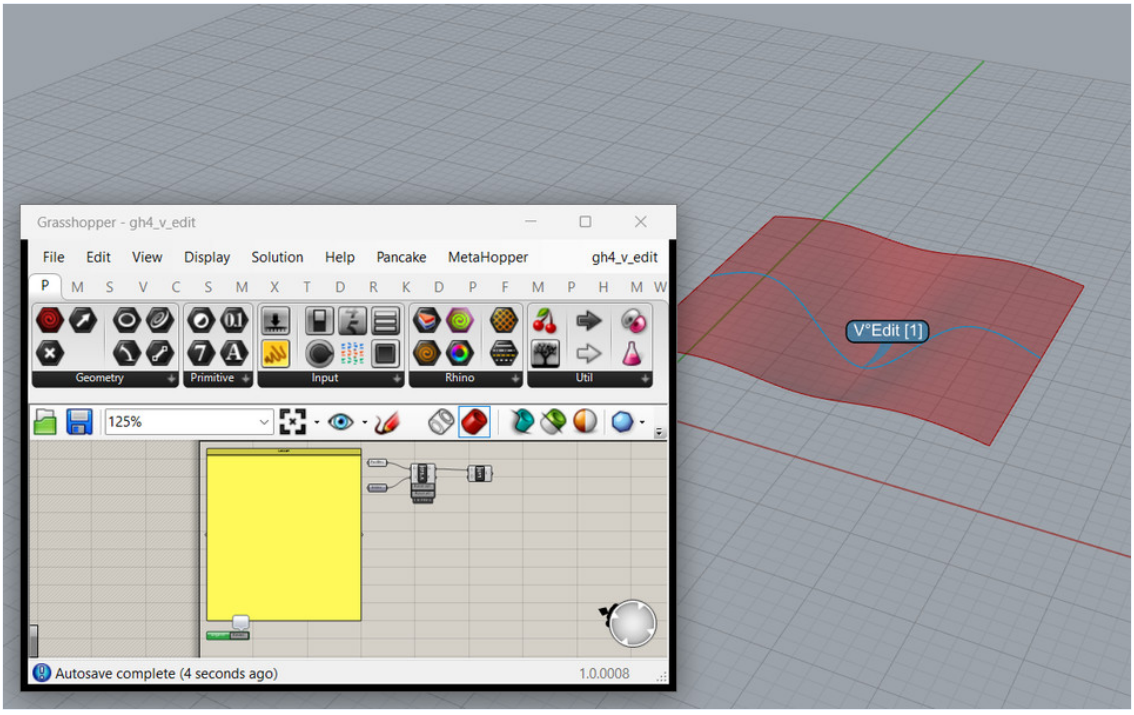


Figure 16.1: Lesson GH 4: V°Edit between *Profiles* and *Loft* puts profile [1] into Rhino as an edit object (*1 in Rhino*), with the name dot *V°Edit [1]*. **Reset sel.** and **Reset all** drop edits.

16.1.1 Inputs and outputs

Geometry an item, a list or a tree.

Index which items to edit, any mix of:

- **indices:** positions in every branch; negative counts from the end. Or per branch, with an index *tree* matched by path;

- **paths:** `{0;2}` as text, or a Path parameter: only those branches, all their items, or with indices the items at them;
- **path with index:** `{0;2}[3]`, that one item.

Nothing connected: item 0.

Outputs, in the input's exact tree: *Geometry*, with your edits in place, and *Edited*, True or False for every single item.

16.1.2 Editing

1. The items at *Index* show in Rhino as ordinary objects, **blue**, each with a name dot and tail showing the component's nickname. Rename the component, and the dots follow at once.
2. Edit them with anything: Gumball, control points, Move, Scale, Rebuild.
3. When you finish, the edited geometry **replaces** its item, and the definition recomputes from the component on.
4. The object turns **red-orange**, its dot says *edited*, the component counts the edits, and a balloon on it explains.

Every later solution uses your edits instead of the incoming items, also when the input changes. When the input changes so that another kind of item is at an edited place, the edit isn't used there: a warning names it, and **Reset all** drops it.

16.1.3 Undo, reset, save

- **Reset sel.** drops the edits of the edit objects selected in Rhino (orange while one is selected); **Reset all** drops every edit (orange while there are any).
- Undo in Rhino back to the input's geometry drops an edit; Grasshopper's Undo takes back an edit made in Rhino. Grasshopper's save prompt includes the edits.
- The menu has **Select edit objects in Rhino**, **Edit objects in Rhino** (show or hide them), **Name dots in Rhino**, and **About V°Edit…**, with its version, build and build time, and the plug-in's.
- *Preview*: Grasshopper's own preview of the Geometry output, as for other components.

> **The edit objects belong to the component** They are not part of the model: they go away while Grasshopper is minimized or hidden, another definition is on the canvas, or the component is disabled; when it is deleted or the definition closed. They are never saved into the `.3dm`, hidden or locked ones neither, and don't mark the model as changed. (Export Selected exports them when they are selected.)

16.2 V°Text: a Panel's text as Rhino text

Tab **miniVivisection ▸ Text.** V°Text makes Rhino text of a Panel's text, the way the Panel shows it: Markdown laid out, a script as code, plain text with its lines kept. Notes and scripts become drawing annotation in the model, in the format they have on the canvas.

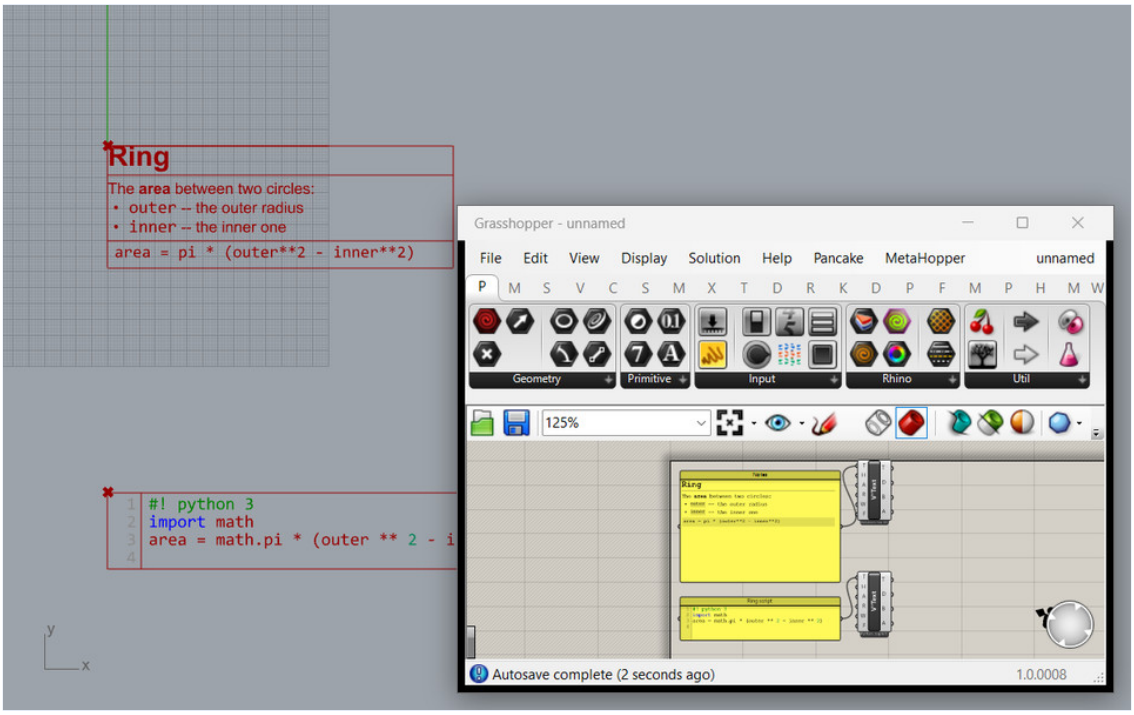


Figure 16.2: A Markdown *Notes* Panel and a Python *Ring script* Panel, each feeding a V°Text. Rhino shows the Markdown laid out and the script as code. The red frames are the *Bounds* outputs, the crosses the anchors.

16.2.1 What it makes

Markdown as `MarkdownText` lays it out (Chapter 18): headings, bold, italic, code, lists, task boxes, quotes, code blocks, tables, rules, links with their bars.

A Python or C# script as code: Consolas, line numbers in a gutter, a frame, each token in its syntax color (one text object per color, lined up).

Anything else as plain text, its lines kept.

V° Show as in its menu: *Autodetect* follows the Panel the text comes from, its own Show as, else what it is; or Markdown (rendered), Python code, C# code, Plain text.

16.2.2 Inputs

Text	connect a Panel; a list is taken as its lines
Height	a paragraph's or a line's text height (Rhino's text height: capital letters). **10 pt** by default: points made the model's units (10 pt = 3.53 mm), or model units (menu: *Sizes in points / in model units*)
Anchor	a point; the menu chooses the text's corner at it: *top left* (the default), *top right*, *bottom left*, *bottom right*
Angle	turned about the anchor in the world's XY plane (radians; *Degrees* in the input's menu)
Width	the width Markdown and plain text wrap in; 0: no wrapping. Without a Width it wraps about as wide as the Panel (plain text only when the Panel wraps); making the Panel wider or narrower, or changing its Show as, updates it

Font	the base font of Markdown and plain text (code stays in Consolas); nothing: the dimension style's. Right-click **Set one Text** lists the installed fonts, each shown in itself: type to filter, double-click or OK to take one

16.2.3 Outputs

Text the Rhino text objects as the component draws them, a token or a part each.

Decorators everything else it is drawn with: the lines of code frames, rules, quotes' bars, tables' grids, and the links' bars (hatches). Both placed and turned as the text is; their previews are off, since the component draws the text in its colors.

Bounds the text's bounding rectangle, aligned with it.

Anchor the anchor point.

The component's message says what it shows and where it is anchored (*Markdown, top left*). The preview draws the text in Grasshopper's preview color, code in its syntax colors, green while selected.

Bake (right-click **Bake**) makes Rhino objects, grouped; a token's color is its object's color. Baked Markdown keeps what `MarkdownText`'s parts keep: double-click any of it to edit its Markdown in place, and its links work. Links in the preview aren't clickable: bake it.

16.3 V°Protect: is it still the original?

Tab **miniVivisection** ▸ **Files**. Lessons, templates and handed-out definitions get changed by accident. V°Protect tells whether a definition, or other files, are still the originals, and puts them back. It has no inputs and no outputs; its menu sets everything, saved with the definition.

16.3.1 Setting it up

Files None set: the definition's own file. **Add files…** (several at once), **Add this definition**, each file's **Remove**, **Remove all**.

Where the originals go *Archive*: by default `(the definition's name).protect.zip` next to it; **Choose…** another (an archive that exists is added to), or back to the default. Or **Keep the originals in the definition itself**: no archive. The other files' originals are kept in the component, saved with the definition, which grows by their size; the definition keeps its own original by a fingerprint of every object's saved state.

Store now stores at once. A definition not saved yet can't be kept; it says so.

In setup, the files are hashed (MD5) and copies stored in the archive whenever the component solves: placed, opened, a setting changed, **Store now**. Two files of the same name in different folders are two entries; a file changed since is replaced, an unchanged one left as it is. Its **message balloon** says per file *added*, *replaced*, *unchanged* or *missing* (a warning); its tooltip lists the files, the archive and the whole log.

16.3.2 Protect and Revert

Protect (the button under it, or its menu): the definition is saved and stored as the original, with the component protecting it. From then on the component looks like a Grasshopper Button: in front it says **original**, the component green, while the file on disk is the original and the open definition has no changes; **modified**, the component red, when either isn't so. Then a gray **Revert** button. Its balloon and tooltip say which, and how the other files stand.

Revert (while *modified*) asks first. When the file on disk isn't the original, the archived copy is written over it; then the definition is opened again from the file, and the changes of the open one are dropped.

Back to setup Right-click anywhere on it for its menu.

> **Note** V°Protect has no password and no write protection: it tells, and it puts back.

> **Try it: the overview and every Grasshopper lesson** `gh0_overview.gh` has a V°Protect to try: move its slider, it turns red; **Revert** puts the overview back. Every Grasshopper lesson has one under its text, keeping the original in the lesson itself. `gh4_v_edit`: V°Edit.

17 Prompts in Grasshopper

The speech-bubble button on Grasshopper's toolbar (right, next to the V° button), or the command `miniVivisection GrasshopperPrompts`, opens a window with three tabs for the definition on the canvas: **V°Claude**, **V°Python** and **V°PowerShell**. The window stays above Grasshopper's and is minimized with it. Closing it only hides it, so the chat, Python's namespace and PowerShell keep going.

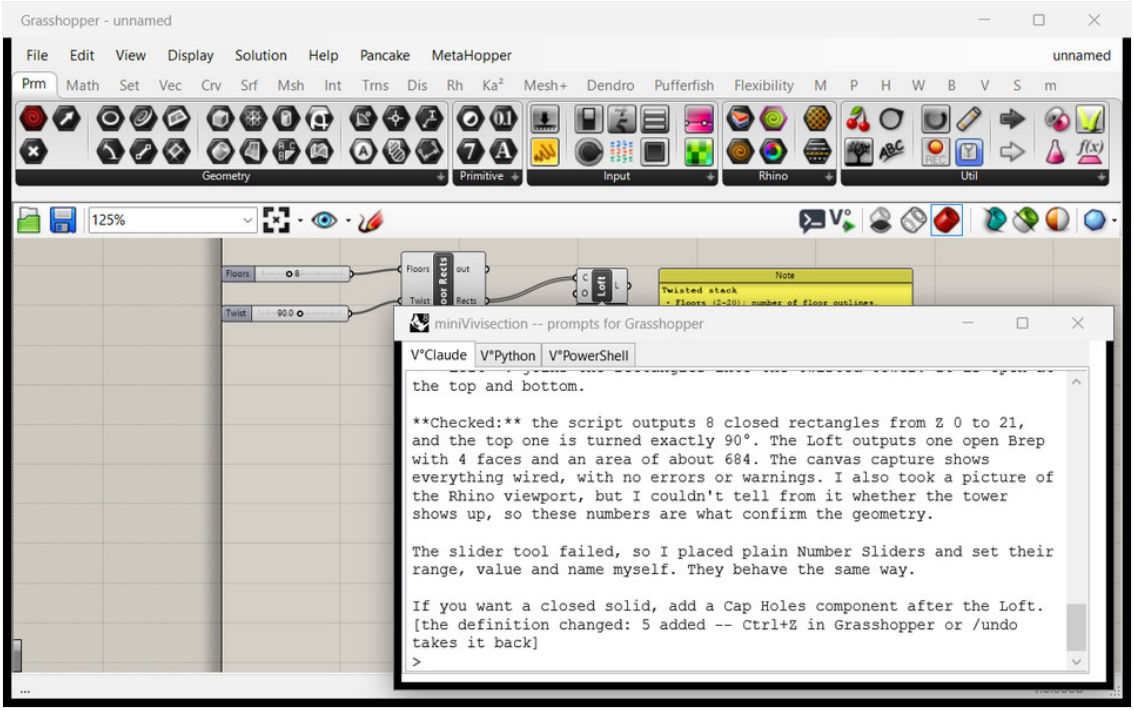


Figure 17.1: One message in an empty definition: two sliders, a rectangle per floor, each turned by Twist, lofted, and a note. Claude placed the sliders, wrote the *Floor Rects* script component, added Loft, wired them and wrote the note; its answer says what it checked, and that Ctrl + Z takes it all back.

17.1 V°Claude in a definition

Each message names the definition, whether the solver is on, and **the components selected on the canvas.** Select what you mean, then ask: "explain these", "add a loft between these curves and a slider for the count".

17.1.1 What Claude uses

RhinoAI's Grasshopper tools (RhinoAI installed and its server running, § 12.4.4): the whole canvas with every object's wires and messages, the component library, placing components and sliders, wiring, deleting, solving; the Rhino view as an image.

miniVivisection's own tools, for what RhinoAI doesn't do: the selection in detail; a parameter's data in full, each item with its facts (a point's coordinates, a curve's length and whether it is

closed, a Brep's area, volume and whether it is solid, the bounding box); a script component's code; setting sliders (value, range), Panels (text, Show as), toggles, value lists, a parameter's stored values and the Rhino objects it references; writing a script component's code with its inputs and outputs; nicknames, enabled, preview, moving, groups; removing wires; **Markdown notes** next to what it built; an image of the canvas.

Claude checks its work, the components' messages, the data, the canvas image, before it answers. Without RhinoAI, Claude can read and set values and write scripts and notes, but not place, wire or delete components: it says what to do instead.

17.1.2 One undo step per answer

Whatever one answer changed, components added, deleted, set, rewired, is one step in Grasshopper's undo list, named *V°Claude: (your message)*. Ctrl + Z takes it all back and Ctrl + Y brings it again; `/undo` in the chat does the same when the last step is Claude's. The chat says what changed (*2 changed, 1 added*). Setting: *One undo step per answer*.

17.1.3 An answer stays with its definition

Claude's tools work on the definition the canvas shows. Showing another one while Claude answers stops the answer, and the chat says so, so that nothing changes the wrong definition. What it did until then is still one undo step in its own definition. There is one conversation per definition, continued when you come back to it.

17.1.4 Commands

`/explain` (the selected components, or the whole definition), `/fix` (the components with errors or warnings), `/undo`, `/history`, `/allow on|off` (RhinoAI's commands and scripts), `/new`, `/clear`, `/help`. Tab completes them.

Read only. With *Claude may change the definition* on *Read only*, Claude reads and explains, and proposes changes in words.

17.2 V°Python in Grasshopper

While a Grasshopper Python component is paused, input runs in its paused frame, as **V°Python** does in the Script Editor. Otherwise it runs in the tab's own namespace, which has these helpers (*V°Python: Grasshopper helpers*):

`ghdoc()`	the definition, a `GH_Document`
`ghselected()`	the selected objects
`ghcanvas()`	the canvas
`ghfind('Name')`	objects by nickname
`show()`, `hide()`, `clear()`, `bake()`	as in the Script Editor (§ 12.2)

17.3 V°PowerShell in Grasshopper

A PowerShell prompt, started in the definition's folder (Documents for an unsaved one; *V°PowerShell: start in the definition's folder*).

17.4 Settings

On the tab *Grasshopper*, in **V°Settings** and **Tools** ▸ **Options** ▸ **miniVivisection:** the toolbar button, each tab on or off (**V°Claude** also needs the **V°AI** switch, the others **V°Debug**), whether the window stays with Grasshopper, what Claude may change, undo, the selection, RhinoAI's tools, how many items Claude reads per parameter, whether it may look at the canvas, V°Python's helpers and V°PowerShell's folder (§ 22.3). Claude Code's path, the model, starting RhinoAI's server and `/allow`'s default are **V°Claude**'s settings on the *General* tab.

Part IV

In Rhino

18 Markdown as Rhino text

The command `MarkdownText` makes Markdown into Rhino objects: headings, paragraphs, lists, tables and links laid out as real text, lines and hatches in the model. Double-click any part of it later, and its Markdown opens again for editing; the same objects are updated in place.

18.1 Making Markdown text

Run `MarkdownText`, type or paste Markdown in its dialog, and click where its top left goes, in the active construction plane. The dialog has:

Width	the width text wraps in, in mm; 0: no wrapping
Size	the base size in pt: a paragraph's text height; headings are larger
Font	the base font

`-MarkdownText` takes the Markdown from a file, with the options *Height* and *Width* in model units. Option *Markdown parser*: Rhino's Markdig, or the plug-in's, as for Panels (§ 15.7).

18.2 What the Markdown becomes

Markdown	In Rhino
`# Heading`	larger bold text; the first two levels with a line under them
paragraphs	wrapped rich text: **bold**, *italic*, `code` in Consolas
`[text](target)`	a link: its words in the link color, a line under them (Chapter 19)
`- item`, `1. item`, `- [ ] task`	bullets, numbers, task boxes ☐ ☑
`> quote`	a bar along it
code blocks	a frame around them
tables	a grid of cells, the header row bold
`---`	a line

The whole text is one group.

18.3 Editing it

Double-click any part of it – a link too –, or select it and run `MarkdownText`, or run the command and click a part. Its Markdown opens in the dialog:

- A part double-clicked or clicked, or selected alone with Ctrl + Shift + click, puts the caret at the start of its section and shows the other parts' Markdown in gray, editable too. The whole text selected opens it all.
- The whole text is selected in Rhino meanwhile.
- **OK** updates the **same objects** in place: their ids, layers and colors stay. Only when the text needs more parts than it has are new ones added, and parts it doesn't need any more are removed.

Undo and Redo. In the dialog, Ctrl + Z undoes a word at a time (a step ends at a space, a line break, a tab, a punctuation mark or a pause in the typing), back to the Markdown as it opened; Ctrl + Y or Ctrl + Shift + Z redoes. After **OK**, Rhino's Undo takes the whole edit back – the objects and their Markdown as they were – and Redo brings it again; a new text is one Undo too.

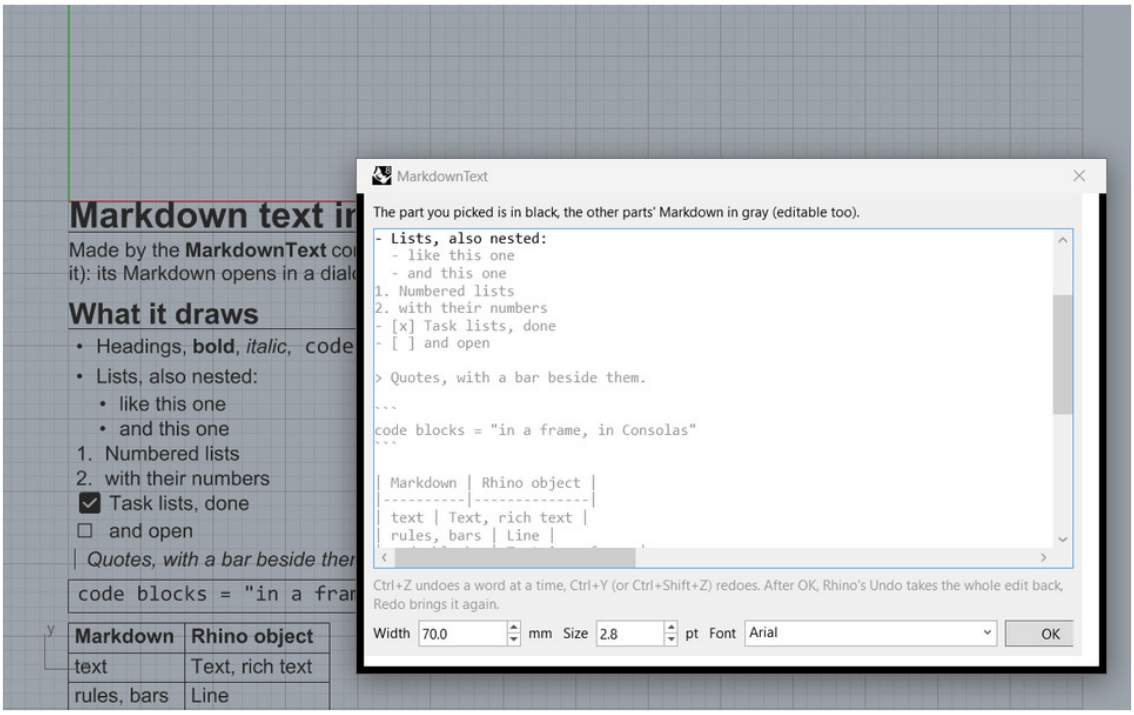


Figure 18.1: A double-click on the nested list opens the Markdown: that part in black at the caret, the other parts in gray. Width, size and font at the bottom.

18.4 The parts belong together

The parts of a text are tied together by the text's id, stored as user text on each object, not only by the group: ungroup it, and any part still edits the whole text. A copy (Copy, Ctrl + C and Ctrl + V) has its own parts: edited, it parts ways with the original.

18.5 Changes made in Rhino are taken in

Before the dialog opens, what was done to the parts with Rhino's own tools goes into the Markdown:

A text's words edited (Rhino's text editor, or Properties): put where they came from, the formatting around them kept.

The whole text moved, turned or scaled: it stays there.

All of a block's parts deleted: its Markdown is taken out.

A line on top of the dialog, and the command line, say what was taken in.

18.6 Links

Click a link to follow it (*Markdown links: follow with*: or Ctrl + click, or off); a double-click on it edits. Over a link, a hint by the mouse says where it goes and how to follow it (Chapter 19).

> **Try it: the Rhino example** `markdown_text.3dm`: Markdown made into Rhino text. Double-click any of it to edit; click its links: a named view, a viewport, a named circle, GH 11's *Ring* component and canvas view (Grasshopper opens it), lesson 01 (the Script Editor opens it), the overview, a web page.

19 Links in Markdown

Links make notes into a guide. A Panel's note can lead to the component it talks about; Markdown text in a model can open the Grasshopper definition that made it and go to a component there, and that definition's notes can lead back. Links work the same in Panels, in `MarkdownText`, and in baked V°Text.

19.1 Targets

A link is `[text](target)`, the target one of:

`gh:Name`	a Grasshopper object by its nickname (or name, or instance id; a group by its name too): the canvas glides to center it, and a red circle marks it for a moment. The active definition first, then the other open ones.
`ghview:Name`	a named view of the Grasshopper canvas (**View** ▸ **Named Views**), restored.
`rhino:Name`	a Rhino object by its name (or id): the view zooms so it takes about a tenth of the view, its surroundings in sight, and a red circle marks it.
`rhinoview:Name`	a named view of Rhino's, restored in the active viewport; or, when there is none of that name, the viewport with that title (`rhinoview:Perspective`) made the active one.
`script:file.py`	a script (`.py`, `.cs`) or a Markdown file (`.md`) opened in the Script Editor, not run.
`https://…`, `mailto:…`, `file:…`	the browser, or the app for it.

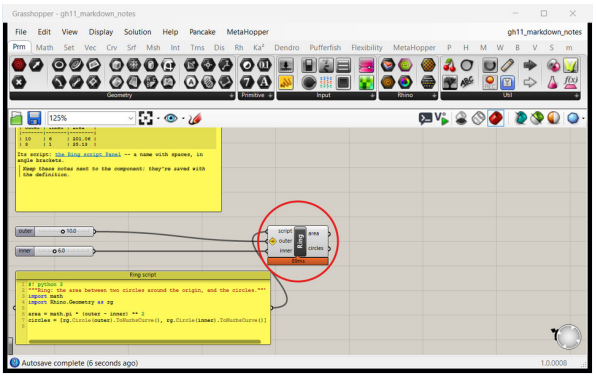


Figure 19.1: `gh:Ring` followed: the canvas centered on *Ring*, a red circle marking it.

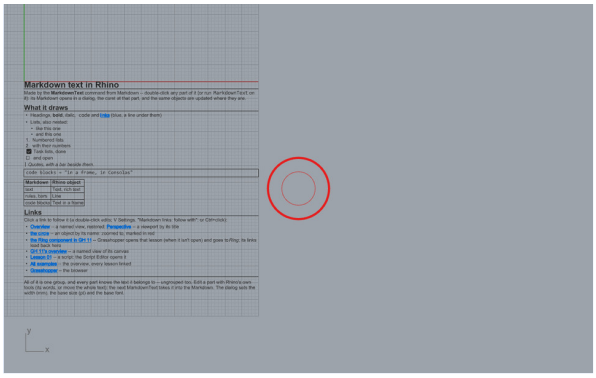


Figure 19.2: `rhino:Circle` followed: the view zoomed so the circle takes about a tenth of it.

How much of the view a Rhino object takes is a setting: *Links to Rhino objects: their size in the view (%)*, 10 by default; 100 fills the view.

19.2 Into another file, and back

Put the file before the name, with `#`:

- `gh:grasshopper/gh11_markdown_notes.gh#Ring` opens that definition in Grasshopper when it isn't open (starting Grasshopper when it isn't running), puts it on the canvas and goes to *Ring*. `ghview:file.gh#View` likewise.
- `rhino:../markdown_text.3dm#Circle` opens that model in Rhino when it isn't the one open (Rhino asks to save yours first) and zooms to *Circle*. `rhinoview:file.3dm#View` likewise.
- The file alone, `gh:file.gh`, just opens it.

A relative path is from the file holding the link: the `.3dm` for Markdown text in Rhino, the `.gh` for a Panel. So save that file first. The examples link both ways: `markdown_text.3dm` into GH 11, GH 11 into `markdown_text.3dm`.

A file without a scheme is a link too: `[next lesson](gh3_stop_a_hang.gh)` opens that definition, or shows it when it is open; `[the model](../markdown_text.3dm)` that model; `[lesson 1](../01_watch_geometry.py)` that script in the Script Editor. The same as `gh:`, `rhino:` and `script:` with the file alone. A full path (`C:\models\a.gh`) works as well.

Names with spaces: `[the twist](<gh:My Twist>)` or `gh:My%20Twist`.

19.3 Following a link

Click the link – in Rhino and in a Panel shown as Markdown alike. *Markdown links: follow with* (one setting for both) chooses:

Click (the default): a plain click follows it, once the double-click time has passed without a second click.

Ctrl+click: a click with Ctrl held follows it at once; a plain click selects, as on any object.

Off: links aren't followed.

A **double-click** always edits – on a link too: in Rhino the Markdown dialog opens (Section 18.3), in a Panel its source. Over a link, a hint by the mouse says where it goes and how to follow it – *click*, *Ctrl+click*, or, when following is off, where to switch it on (Figure 19.3); in a Panel the pointer becomes a hand (Figure 15.3).

A target that isn't found is reported on Rhino's command line.

> **Safety** A link that opens a file asks first, unless the file is near the linking one (its folder, or the one above). A `file:` link always asks. Nothing opens while a script is paused.

19.4 Their look

Links are blue: *Markdown links in color* (on) and *Markdown links: color* (`#0066CC`; a name such as `Blue`, `#RRGGBB` or `R,G,B`). A link's words and the **line under them** are in the link color. The line is as long as the link's words, *Markdown links: line thickness* high (10 % of the text height by default,

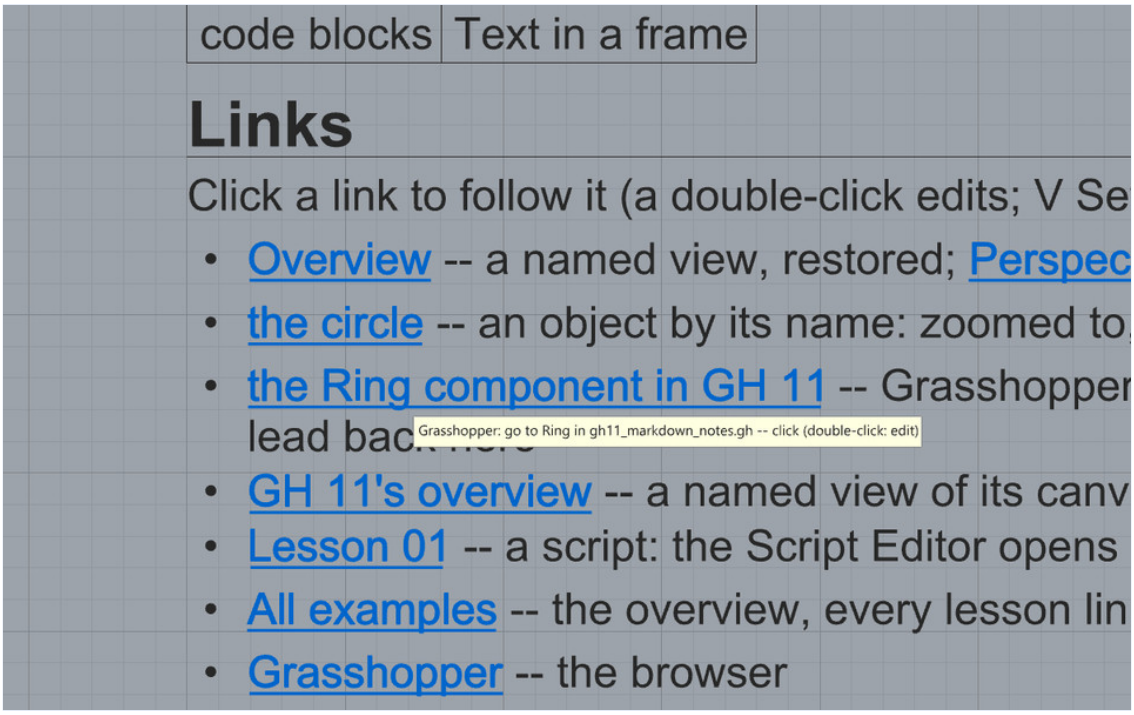


Figure 19.3: `markdown_text.3dm`: the mouse over a link into Grasshopper. The hint says where it goes and how to follow it.

1 to 50), its top *Markdown links: line distance* under the baseline (15 % by default, an underline's place; about 30 % clears the descenders of g, p, y); in Rhino for texts made or edited afterwards. In Rhino the line is a solid hatch, one under each line of the link – where Rhino wraps it, and at a hard line break in it; it belongs to the text: it moves, edits and bakes with it (V°Text: in *Decorators*). A Rhino text object has one color, so the plug-in **draws** the link's letters over it, filled in the link color – drawn, not objects: a click on a link's word still picks just the text, and a change of the color shows at once. Not while the text is selected (its highlight shows), nor after its words were changed in Rhino until it is edited again; without the plug-in the words show in the text's color. Colors off: the line in the text's color (by its layer).

Part V

Reference

20 The lessons

miniVivisection comes with 17 short script lessons, 12 Grasshopper lessons, a Rhino example and an overview of all of them. Each lesson has a bug in it that one feature finds or fixes, with instructions at the top of the file: *The bug*, *Try this*, *Without miniVivisection*. New to debugging? Start with `00_Examples_Guide.pdf` in the `examples` folder; it explains debugging for beginners and lists the lessons.

20.1 Where they are

In the Script Editor: the left panel's **Examples** ▸ **miniVivisection** (*Lessons in the Examples tab*).

On disk: the `examples` folder next to the `.rhp`, the Grasshopper lessons in `examples\grasshopper`.

Your own copies. The Examples tab opens your own copies, in `%LOCALAPPDATA%\miniVivisection\lessons`, and the Grasshopper lessons and their script files are there in `grasshopper`: save your fixes there, the shipped lessons stay as they are. A copy you haven't changed follows a new version of the lesson. When a new miniVivisection brings a new version of a lesson you changed, it is put beside your copy as `(name) (new)`, for example `gh3_stop_a_hang (new).gh`. To start a lesson over, delete your copy; it is copied again when the editor next opens. Open the Grasshopper copies with Grasshopper's **File** ▸ **Open**.

How to do a lesson.

1. Open it, and read the text at the top (and a *Needs:* line under the title when it needs something first).
2. Set a breakpoint on the line marked `# <-- BREAKPOINT`, and press F5, unless *Try this* says otherwise: lessons 15 to 17 start with **Debug** and no breakpoint; lesson 04 has two breakpoints.
3. Follow the steps. When you have found the bug, stop, fix the code, and run again to check.

20.2 The overview

`00_overview.3dm` (Markdown text in Rhino) and `grasshopper\gh0_overview.gh` (a Panel) – click a link – link to every lesson: the 17 script lessons open in the Script Editor, the Grasshopper lessons in Grasshopper, the Rhino example in Rhino. Each links to the other. `gh0_overview.gh` also has a slider to change and a V°Protect to try.

20.3 Script lessons

File	Feature	
`01_watch_geometry`	V°Watch: variables drawn in Rhino	§ 5

`02_values_in_the_code`	tooltips, line-end values, “was”, Tree; stepping	§ 6
`03_bake_before_a_crash`	Bake	§ 7.3
`04_edit_in_rhino`	Edit in Rhino, Auto edit	§ 7.1
`05_conditional_breakpoints`	conditions, hit counts, log messages	§ 9
`06_watch_expressions`	expressions in watches	§ 5.1
`07_raw_data_interpret_as`	Interpret as	§ 5.4
`08_python_terminal`	V°Python	§ 12.2
`09_step_over_other_files`	stepping across files	§ 3.1
`10_visualizers`	visualizers	§ 8
`11_rhino_usable_while_paused`	FreePause	§ 10.3
`12_powershell_terminal`	V°PowerShell	§ 12.3
`13_claude`	V°Claude	§ 12.4
`14_threads_async`	V°Threads, thread conditions, watches in every thread	§ 11
`15_deadlock`	Where is everyone?, Break now	§ 11.1
`16_long_running_watch`	Break now in a running script, fix it while paused, run on	§ 10.2
`17_runtime_error`	the debugger stops at a runtime error; values and call stack; Stop and record	§ 10.6

20.4 Grasshopper lessons

Twelve definitions in `examples\grasshopper`, each with its lesson in a Panel on the canvas, ending with links to the lesson before and after and to the overview, and under it a V°Protect: changed something? **Revert** puts the lesson back as it was shipped. GH 10’s script file, `gh10_spiral.py`, is next to them.

File	Feature	
`gh1_which_branch_failed`	failing runs, Copy failing branches, Debug failing run N	§ 13.7
`gh2_the_one_wrong_item`	Always debug, a breakpoint condition, the run badge, a slider moved while paused	§ 13.6
`gh3_stop_a_hang`	Ctrl + Alt + B with the component’s editor closed	§ 13.3
`gh4_v_edit`	V°Edit	§ 16.1
`gh5_runtime_error`	the debugger stops at a runtime error in a script component (Always debug)	§ 13.8
`gh6_threads_in_a_component`	threads in a component: results in finishing order, V°Threads, the freeze guard	§ 13.10
`gh7_parallel_run`	`ghpythonlib.parallel`: `parallel.run` that doesn’t flatten, the pause at the error, a breakpoint inside the worker function	§ 13.10

`gh8_long_running`	a long-running component: Ctrl + Alt + B pauses it (Always debug), a fix in V°Python, it runs on	§ 13.3
`gh9_script_from_a_panel`	a script from a Panel: edited in the Script Editor, saved back into the Panel	§ 15.4
`gh10_script_from_a_file`	a script file named in a Panel: Ctrl + Alt + B stops the hang and opens the file at its line	§ 15.4
`gh11_markdown_notes`	notes in Markdown, rendered in a Panel; editing the source where you double-click; a script Panel edited in place; links to components, a canvas view and the Rhino example	§ 15.7
`gh12_markdown_parser`	Markdown with Rhino's parser: task boxes, nested lists and quotes, a footnote; a task and a table that need fixing	§ 15.7

20.5 The Rhino example

`markdown_text.3dm`: Markdown made into Rhino text by `MarkdownText`. Double-click any of it to edit its Markdown, and click its links: a named view, a viewport, a named circle, GH 11's *Ring* component and canvas view, lesson 01, the overview, a web page (Chapters 18 and 19).

21 Commands, keys and files

21.1 Commands

miniVivisection. Without options it shows how many editors are followed and how many items are drawn and edited. Its options:

`Draw=On|Off`	draw watches in Rhino
`FreePause=On|Off`	Rhino usable while paused, from the next pause (§ 10.3)
`HoverPreview=On|Off`	the hover preview (§ 6.3)
`WatchExpressions=On|Off`	expressions in the editor's Watch tab (§ 6.5)
`BreakNow`	as the hotkey: pause a debug run at its next line (§ 10.2)
`WhereIsEveryone`	print every thread of a running script and where it is (§ 11.1)
`StopPaused`	stop every paused script, a way out when its editor can't be reached
`BakeAll`	bake everything drawn right now (§ 7.3)
`StopEditing`	remove the objects being edited in Rhino (only while there are some)
`GrasshopperPrompts`	open the prompts window for Grasshopper (Chapter 17)
`Settings`	the Options dialog's miniVivisection page
`LogFolder`	open the log folder

MarkdownText makes Markdown into Rhino text, or edits a Markdown text: run it with one selected, or run it and click a part. `-MarkdownText` takes the Markdown from a file (options *Height*, *Width*, *Markdown parser*). See Chapter 18.

ScriptEditor Run gets a *Debug* option: `_-ScriptEditor _Run _Debug=_On "file"` (§ 10.7).

RunPythonScript follows *RunPythonScript command* (§ 10.6).

21.2 Keys

Script Editor	
F5	Debug (with a breakpoint) or Run; Continue while paused
F10, F11, Shift + F11	Step over, Step into, Step out
Shift + F5	Stop
Ctrl + Alt + B	Break now: pause a debug run, stop a plain one (a setting, § 10.2)
F2	edit a **V°Watch** expression in place

Del	remove the selected watch
Esc	close the Tree window
Ctrl + mouse wheel	prompt font size in **V°Python**, **V°PowerShell**, **V°Claude**
Tab	complete a / command in **V°Claude**
Ctrl + C	with nothing selected in a prompt: interrupt
Grasshopper	
double-click a script Panel	edit it in place (§ 15.2)
double-click a Markdown Panel	edit its source at that word
click a link in a Panel	follow it
Shift + double-click, Ctrl + double-click	Grasshopper's own text box, its dialog
Ctrl + Z	also takes back a whole V°Claude answer
Rhino	
double-click Markdown text	edit its Markdown at that part; on a link: follow it
Ctrl + Shift + click	pick one part of a Markdown text
Alt + click on a drawing	select its watch, when *Select a watch by clicking its drawing* says so

21.3 Logs and files

`%TEMP%\rhinovivisection\minivivisection`	the log folder (`miniVivisection LogFolder`)
`user.log`	**V°Log**'s lines
`watchviz.log`	the detailed log, by area and level, for diagnosis; new each session
`*.previous.log`	the last session's logs
`watchviz.crashed-(time).log`	the detailed log of a session that ended without Rhino shutting down (the last five); V°Claude's steps in Grasshopper are written at once, so the last thing before a crash is in it
`last_stop.json`	where Stop and record last stopped
`%LOCALAPPDATA%\miniVivisection\lessons`	your copies of the lessons
`(definition).protect.zip`	V°Protect's archive, next to the definition
`rhino_tools.json`	next to the `.rhp`: RhinoAI's tools V°Claude may use, by group; edit it when RhinoAI's tools change, no rebuild needed

The detailed log's level is a setting (*Detailed log*: Off to Trace), with levels per area (*Detailed log: levels per area*, for example `Pause=Trace; Claude=Debug`). **Copy detailed log** in

Tools ▸ **Options** ▸ **miniVivisection** copies its last 2000 lines.

21.4 Environment variables

`MINIVIVISECTION_CLAUDE`	the path to `claude.exe`
`MINIVIVISECTION_RHINO_MCP`	the path to RhinoAI's MCP router

The old names, `SCRIPTEDITORWATCHVIZ_…`, are still read.

22 Settings

Every setting is in two places: the **V°Settings** page in the Script Editor's left panel (applied at once; text fields when you press Enter or leave the box), and **Tools** ▸ **Options** ▸ **miniVivisection**, in collapsible groups, each setting with a tooltip – on three tabs: *General*, what the Script Editor and Grasshopper's script components share (and *Markdown*: MarkdownText, V°Text and Panels alike); *Rhino*, what happens in Rhino (watches drawn in the viewports, selection, the `RunPythonScript` command, FreePause); *Grasshopper*, script components, Panels and the prompts window. They are kept between Rhino sessions. A setting marked *(new windows)* applies to Script Editors opened afterwards.

The two switches in the Script Editor's top bar, **V°Debug** and **V°AI**, aren't listed here (§ 3.11). While **V°Debug** is off, the debugging features read as off whatever their settings say, and come back as they were.

The tables below are generated from the plug-in's source, so they list exactly what this build has.

22.1 Tab *General*

Watches and values

Setting	What it does
Hover preview default: **on**	While paused, show the geometry of the name under the mouse in the code.
Values in the code's tooltips default: **on**	While paused, the tooltip over a name starts with its value (count = 5) and key facts.
Links in the tooltips default: **on**	Under a value in the code's tooltip: Watch in Rhino, Edit in Rhino, Bake (for geometry) and Tree (the value's members, in a small window).
Values at the ends of lines default: **on**	While paused, the values of the variables on each line of the paused function (up to the paused line) in gray after the line, as in VS Code and PyCharm.
What changed since the last pause default: **on**	After a step: changed values in the inline values are colored, and the tooltip says what a value was before ("was 5").
Expressions in the editor's Watch tab default: **on**	Evaluate any Python expression in the editor's own Watch tab, not only variable names.

Visualizers

Setting	What it does
Visualizer scripts default: **(empty)**	Python files with @rhinoviv.visualize / @rhinoviv.inverse functions, separated by ';'. Run at the first pause of every run, in the paused script's context.
Reload a visualizer script when it's saved default: **on**	While paused right away, else at the next pause.

Debugger

Setting	What it does
Conditional breakpoints, hit counts and logpoints default: **on**	A breakpoint's condition, hit count or log message decides whether it pauses (right-click a line, or the Breakpoints tab). Off: every breakpoint pauses.
Run and Debug buttons default: **on**	Two fixed buttons instead of the editor's switching Run button: Run always runs normally (also with breakpoints), Debug always runs in the debugger (also without a breakpoint, so Break now can pause it). The editor's own button shows while debugging, as Continue. Off: the editor's Run button as it was.
Keep the Script Editor open while debugging default: **on**	The editor minimizes itself when a script runs and comes back at a pause. While debugging, and for scripts with '# async:true', it stays open, so V°Threads (Where is everyone?, Break now) can be used while the script runs. A plain Run of other scripts still minimizes.
Break now hotkey default: **Ctrl+Alt+B** Ctrl+Alt+B · Ctrl+Shift+B · Pause · Ctrl+Break · Off	Pauses a script running in the debugger at its next line – also one on Rhino's UI thread, which handles no other keys while it runs. A Windows hotkey, taken only while a script runs.
Break now stops in the script only default: **on**	Break now (its hotkey, V°Threads) pauses at the next line of the script itself: while the script is inside a module of its own or another file, it runs on until it's back in the script. Off: at the next line anywhere.
Stop once per line default: **on**	A line that runs several times (a comprehension, a one-line loop) pauses once. Off: once per item, as pdb does.
Logpoints name their thread default: **on**	A logpoint's line starts with its thread ([Worker-2] i=3) while the script runs threads of its own.
Freeze guard for threads default: **on**	A script without '# async:true' runs on Rhino's UI thread: a breakpoint hit in one of its worker threads would freeze Rhino. Such a pause is skipped, and V°Log and the Terminal say why.

Setting	What it does
Wait for the paused script (ms) default: **2000**	How long the tabs wait for the paused script's thread to answer (Watch tab expressions, members, Claude's tools).

Editor tabs and pages

Setting	What it does
Mark in the V names (new windows) default: **°** ° · | · I (on the line) · | with thin spaces · I with thin spaces	What stands between V and the name: °, a bar, or a bar with thin spaces. For Script Editors opened afterwards, and V°Edit and V°Text components placed afterwards.
Prompt font size default: **10**	The text size of V°Python, V°PowerShell and V°Claude. Also Ctrl+mouse wheel in any of them.
Syntax colors: V°Python default: **on**	What you type in V°Python, in the Script Editor's Python colors.
Syntax colors: V°PowerShell default: **on**	What you type in V°PowerShell, in the Script Editor's colors for PowerShell.
Syntax colors: V°Claude default: **on**	Code blocks in Claude's answers (```python, ```powershell), in the Script Editor's colors.
Tab (bottom): V°Watch default: **on**	V°Watch at the bottom of the editor.
Tab (bottom): V°Threads default: **on**	V°Threads at the bottom of the editor: the script's threads, where each one is, its stack, Break now and "Where is everyone?".
Tab (bottom): V°Python default: **on**	V°Python, the Python prompt, at the bottom of the editor.
Tab (bottom): V°PowerShell default: **on**	V°PowerShell, the PowerShell prompt, at the bottom of the editor.
Tab (bottom): V°Claude default: **on**	V°Claude, the Claude chat, at the bottom of the editor.
Page (left): V°Claude default: **on**	V°Claude in the editor's left panel.
Page (left): V°Watch default: **on**	V°Watch in the editor's left panel.
Page (left): V°Threads default: **on**	V°Threads in the editor's left panel.
Page (left): V°PowerShell default: **on**	V°PowerShell in the editor's left panel.
Page (left): V°Log default: **on**	V°Log, the user log, in the editor's left panel.

Setting	What it does
Lessons in the Examples tab default: **on**	The miniVivisection lessons (the examples folder next to the plug-in) in a 'miniVivisection' folder of the editor's Examples tab (left panel); they open your own copies (in %LOCALAPPDATA%\miniVivisection\lessons).
Drop a folder: open it in the editor's Explorer default: **on**	A folder dropped anywhere on the Script Editor opens in its Explorer (left panel). Files dropped still open as scripts.
Page (left): V°Settings default: **on**	These settings in the editor's left panel, applied at once. Shown whatever the V°AI and V°Debug switches say.

Setting	What it does

Claude

Setting	What it does
Claude Code (claude.exe) default: **(empty)**	Claude Code's executable. Empty: found automatically (MINIVIVISECTION_CLAUDE, the PATH, the usual install folders).
RhinoAI's MCP router default: **(empty)**	rhino-mcp-router.exe. Empty: found automatically (MINIVIVISECTION_RHINO_MCP, RhinoAI's folders).
Offer to install RhinoAI default: **on**	When Rhino has started and RhinoAI (Claude working in the Rhino model) isn't installed, offer to install it (Rhino's package Rhino-MCP-Platform). Once per Rhino session, while the V°AI switch is on.
Claude may run Python default: **Ask** Off · Ask · On	Code Claude wants to run in Rhino (it can change the script, the model, files). Ask: runs after /run. Off: never. On: without asking. Reading a value is always allowed.
Start RhinoAI's server for Claude default: **off**	Before each message, start RhinoAI's MCP server in this Rhino when it isn't running (RhinoAI's MCPStart). It listens on localhost without a token: any program on this computer can then use RhinoAI's tools. Off: Claude works in the model only while you've started it yourself.
Claude may control the debugger default: **on**	Breakpoints, continue, step, stop and start debugging. Each step is shown in V°Claude.
Wait for a step (s) default: **30**	How long Claude's continue/step waits for the next pause before it answers "still running".
Claude's edits to the script default: **Automatic** Automatic · Ask	Automatic: applied right away (while paused: when the run ends). Ask: shown as a change, applied with /apply.
Claude model default: **(empty)**	Claude Code's –model (e.g. sonnet, opus). Empty: Claude Code's default.
Allow RhinoAI's commands and scripts default: **off**	Claude may use RhinoAI's destructive tools (run_command, run_python, ...) without asking. The /allow command changes it for a chat.

PowerShell

Setting	What it does
PowerShell executable default: **(empty)**	pwsh.exe or powershell.exe. Empty: PowerShell 7 if installed, else Windows PowerShell.

Setting	What it does
Load the PowerShell profile default: **off**	Start PowerShell with the user's profile (slower; a profile that waits for input blocks the tab).

Markdown

Setting	What it does
Markdown parser default: **Rhino** Rhino · miniVivisection	For Markdown in Grasshopper Panels and the MarkdownText command. Rhino: Markdig, Rhino's own – as the Script Editor's preview: CommonMark's rules, nested lists and quotes, task lists, footnotes. miniVivisection: the plug-in's own, simpler parser.
Markdown links: follow with default: **Click** Click · Ctrl+click · Off	How a link in Markdown text is followed – in Rhino (MarkdownText, V°Text baked) and in Grasshopper Panels alike. Click: a click on it (a double-click still edits the text). Ctrl+click: a click with Ctrl held (a plain click selects, as on any object). Off: links aren't followed; the hint over one says how to switch them on.
Markdown links in color default: **on**	Links in Markdown text (MarkdownText, V°Text, Markdown in Panels) in the link color below: their words and the line under them (in Rhino solid hatches: the letters filled over the text's, Rhino's text having no colors of its own). Off: in the text's color.
Markdown links: color default: **#0066CC**	The links' color: a name (Blue, RoyalBlue), #RRGGBB, or R,G,B.
Markdown links: line thickness (% of the text height) default: **10**	The line under a link's words, in Rhino and in Panels, as a share of the text height (Rhino's text height: its capitals'). 1 to 50.
Markdown links: line distance (% of the text height) default: **15**	How far under the baseline the line under a link's words starts, as a share of the text height: 15, an underline's place; about 30 clears the descenders (g, p, y). 0 to 100. In Rhino for texts made or edited afterwards.
Links to Rhino objects: their size in the view (%) default: **10**	A Markdown link to a Rhino object (rhino:Name) zooms the view so the object takes about this share of it – its surroundings in sight – and marks it with a red circle. 100: it fills the view.
Markdown in Panels default: **on**	A Panel set to Show as Markdown – or on Autodetect, and plainly Markdown – shows it rendered (headings, bold, italic, code, lists, quotes, links, tables, rules); a double-click edits its source where you clicked. Off: such Panels show their text as Grasshopper does.

Logs

Setting	What it does
User log on the command line default: **Warnings and errors** Warnings and errors · All · Off	Which lines of V°Log also go to Rhino's command line.
User log lines kept default: **1000**	How many lines V°Log keeps.
Detailed log default: **Info** Off · Error · Warning · Info · Debug · Trace	The detailed log's level (watchviz.log, for diagnosis). Off: no detailed log.

22.2 Tab *Rhino*

Watches in Rhino

Setting	What it does
Draw watches in Rhino default: **on**	Draw the V°Watch tab's values in the viewports while the debugger is paused.
Name dots default: **on**	A dot with the watch's name (and item count) next to each drawn value.
Name dot offset (pixels) default: **30**	How far up and right of its value a name dot sits, with a tail curving down from its bottom edge to the value, so it doesn't cover a point. 0: on the value.
Name dot pointer default: **Tail** Tail · Line	From the dot to its value: Tail, a curved shape like a speech bubble's; Line, a thin straight line (the simplest to draw).
Name dot opacity (%) default: **50**	100: solid; lower lets the geometry behind a dot show through.
Auto edit default: **on**	At every pause, every shown watch's geometry becomes Rhino objects you can edit; changes go back into the script.

Setting	What it does

Selection

Setting	What it does
Selecting a row highlights it in Rhino default: **on**	Rows selected in the V°Watch tab or the editor's Variables and Watch tabs highlight what's drawn for them and select their objects being edited.
Select a watch by clicking its drawing default: **Plain click** Plain click · Alt+click · Off	Plain click: also when Rhino selects an object there. Alt+click: only with Alt held, taken by the plug-in. Off.
Selecting edited objects highlights their watch default: **on**	Objects being edited (Edit in Rhino, Auto edit) selected in Rhino highlight their watch and select their variable.

Running and pausing in Rhino

Setting	What it does
RunPythonScript command default: **Stop and record** Normal · Stop and record · Debug	Normal: runs plainly; the Break now hotkey (Ctrl+Alt+B by default) can't stop it. Stop and record: runs plainly; the hotkey stops it and the Script Editor then opens it at that line (errors too). Debug: Python 3 scripts run in the editor's debugger, the editor out of sight until they pause. Also ScriptEditor Run's Debug option.
Keep Rhino usable while paused (FreePause) default: **on**	Rhino keeps its normal command prompt while the debugger is paused, instead of the Select Action prompt. From the next pause.
Message loop while paused (diagnosis) default: **Rhino** Rhino · Eto	How Rhino keeps going while a script pauses on its UI thread. Rhino: the default. Eto: for diagnosis only.

22.3 Tab *Grasshopper*

Script components and Panels

Setting	What it does
Grasshopper script components default: **Stop and record** Normal · Stop and record · Always debug	Normal: plain runs; the Break now hotkey (Ctrl+Alt+B by default) leaves them alone. Stop and record: the hotkey stops a hanging component and opens its editor at the line. Always debug: every recompute runs in the debugger (editor open or hidden).
Hidden editors for Grasshopper components default: **on**	With Always debug: every Python 3 script component gets its script editor, out of sight, so the Break now hotkey (Ctrl+Alt+B by default) can pause it (not only stop it) and a breakpoint pauses it even with its editor closed. The editor comes up at the pause, or when you open it. Each one is a Script Editor window, and every run is in the debugger (slower).
async:true runs off the UI thread default: **on**	With '# async:true' a component runs on a thread of its own: Rhino and Grasshopper stay usable and its threads can pause. Its script mustn't use Rhino's UI.
Close script editors when the definition changes default: **on**	Opening another definition, or switching the canvas to another open one, first closes the script editors of the definition it leaves (an editor paused in the debugger stays). Off: they stay open.
Grasshopper toolbar button default: **on**	The V° button on Grasshopper's toolbar: each click goes to the next mode (Stop and record, Always debug, Normal).
Scripts in Panels shown as code default: **on**	A Panel holding a Python or C# script shows it as code (line numbers, colors, no wrapping). Double-click to edit it in place.
While paused: compute changes downstream default: **on**	While a script component is paused in its solution, a change elsewhere (a slider, a wire, a Panel) is computed at once with what is downstream of it – the paused component's outputs as its last finished run left them, script components not run. Everything is computed properly when the pause ends. Off: all of it waits for the pause to end.
While paused: mark what depends on the paused script default: **on**	While a script component is paused, everything downstream of it gets an orange veil and dashed wires on the canvas: its data most probably isn't right any more.

Setting	What it does

The prompts window

Setting	What it does
Prompts button on Grasshopper's toolbar default: **on**	A button on Grasshopper's toolbar (next to the V° button) opens miniVivisection's prompts window: V°Claude, V°Python and V°PowerShell for the definition.
Tab: V°Claude default: **on**	Claude working in the Grasshopper definition: it reads the selection, the data and the scripts, places, wires and sets components (with RhinoAI), writes scripts and notes. Also needs the V°AI switch on.
Tab: V°Python default: **on**	The Python prompt: in the paused frame of a paused Grasshopper Python component, else in its own namespace.
Tab: V°PowerShell default: **on**	The PowerShell prompt.
The window stays with Grasshopper default: **on**	The prompts window stays above Grasshopper's window and is minimized with it. Off: a window of its own. For windows opened afterwards.

V°Claude in Grasshopper

Setting	What it does
Claude may change the definition default: **Read and change** Read and change · Read only	Read and change: Claude places, wires, sets and deletes components, writes scripts and notes. Read only: Claude reads and explains; changes it proposes in words.
One undo step per answer default: **on**	Everything one answer changed in the definition is one step in Grasshopper's undo list (Ctrl+Z, or /undo in the chat).
The selection goes with each message default: **on**	Each message names the components selected on the canvas, so 'this' and 'these' mean them.
Use RhinoAI's Grasshopper tools default: **on**	RhinoAI's g1 tools: the canvas graph, the component library, placing, wiring, deleting, solving. Without them Claude can read and set values, but not place components.
Items Claude reads per parameter default: **50**	How many items of a parameter's data Claude reads at once (gh_get_data), each with its facts (coordinates, length, area, volume, closed).
Claude may look at the canvas default: **on**	Claude can capture the Grasshopper canvas as an image and look at it (gh_capture_canvas). The Rhino view: RhinoAI's get_viewport_image.

V°Python and V°PowerShell in Grasshopper

Setting	What it does
V°Python: Grasshopper helpers default: **on**	ghdoc() the open definition, ghselected() the selected objects, ghcanvas() the canvas, ghfind('name') an object by nickname.
V°PowerShell: start in the definition's folder default: **on**	PowerShell starts in the open definition's folder (else your Documents folder).

23 Limits and troubleshooting

23.1 Limits

Experimental. miniVivisection relies on internal code of Rhino 8's Script Editor (RhinoCode 8.36 when it was written). A Rhino service release can change it. The plug-in then follows nothing; it doesn't fail.

Python 3 first. Inline values, tooltips with scope, the Python prompt in the paused frame and Edit in Rhino are for Python 3 scripts only.

IronPython 2 (`#! python2`): FreePause, step over across modules, **V°Watch** drawing and baking of plain variable names, the hover preview and tooltip values of plain names, the value tree, conditional breakpoints, hit counts and logpoints work. Left out, quietly: expressions in watches and the Watch tab, scoped tooltips, values at the ends of lines, "was", Edit in Rhino and Auto edit, visualizers and Interpret as for raw Python data, the Python prompt and Claude's evaluation in the paused frame. While one is debugged, an amber notice on **V°Python** and **V°Claude** says so. Full IronPython 2 support is planned.

C#: watches of plain variable names and conditional breakpoints with a small evaluator; largely untested.

Two watch lists. The editor's Watch tab shows values; **V°Watch** draws. Each has its own list.

Grasshopper: Python 3 script components through their own script editor. The legacy GhPython (IronPython 2) component isn't covered. With a component's editor closed and no hidden editor, Ctrl + Alt + B can only stop it, not pause it.

Not traced: Python code run by .NET threads.

23.2 Troubleshooting

What happens	What to check
Nothing is drawn, no values in the code	Is the script paused? Is **V°Debug** ON in the top bar? Is *Draw watches in Rhino* on? Is the watch's eye open?
F5 doesn't stop	Without a breakpoint, F5 is a plain run. Use **Debug** (§ 10.1).
Ctrl + Alt + B does nothing	An error sound means nothing could be paused or stopped; **V°Log** says why. A plain run with *RunPythonScript command* on *Normal*, or a Grasshopper component in *Normal* mode, is left alone.
A breakpoint in a thread doesn't pause	The freeze guard skipped it: the script runs on Rhino's UI thread. Add `# async:true` (§ 11.3).
A condition always pauses	It fails: a typo or a missing name. Its marker is red, and **V°Log** says why.
The editor of a paused script is gone	`miniVivisection StopPaused`.

Claude says “the rhino MCP didn’t answer”	RhinoAI’s server isn’t running in this Rhino: run `MCPStart`, or turn on *Start RhinoAI’s server for Claude* (§ 12.4.4).
Claude doesn’t work in the model	RhinoAI isn’t installed, or its server isn’t running; or the tool is a destructive one: `/allow on`.
Grasshopper shows “expired during a solution”	Something recomputed while a component was paused; it should be held back (§ 13.9). Report it with the detailed log.
A Grasshopper component can’t be deleted	A component is paused; deleting waits for the pause to end.
After a Rhino update, a feature is gone	The Script Editor’s internal code may have changed. Look at the detailed log, and for a newer miniVivisection.

23.3 Reporting a problem

Note the version and build (the top bar, or the load message), what you did, and what happened. Open the log folder (`miniVivisection LogFolder`) and attach `watchviz.log`, or `watchviz.crashed-(time).log` after a crash. For a problem that is hard to see, set *Detailed log* to *Debug* or *Trace* for the area concerned first.