# Visual Scripting

## Build adventure logic one clear step at a time.

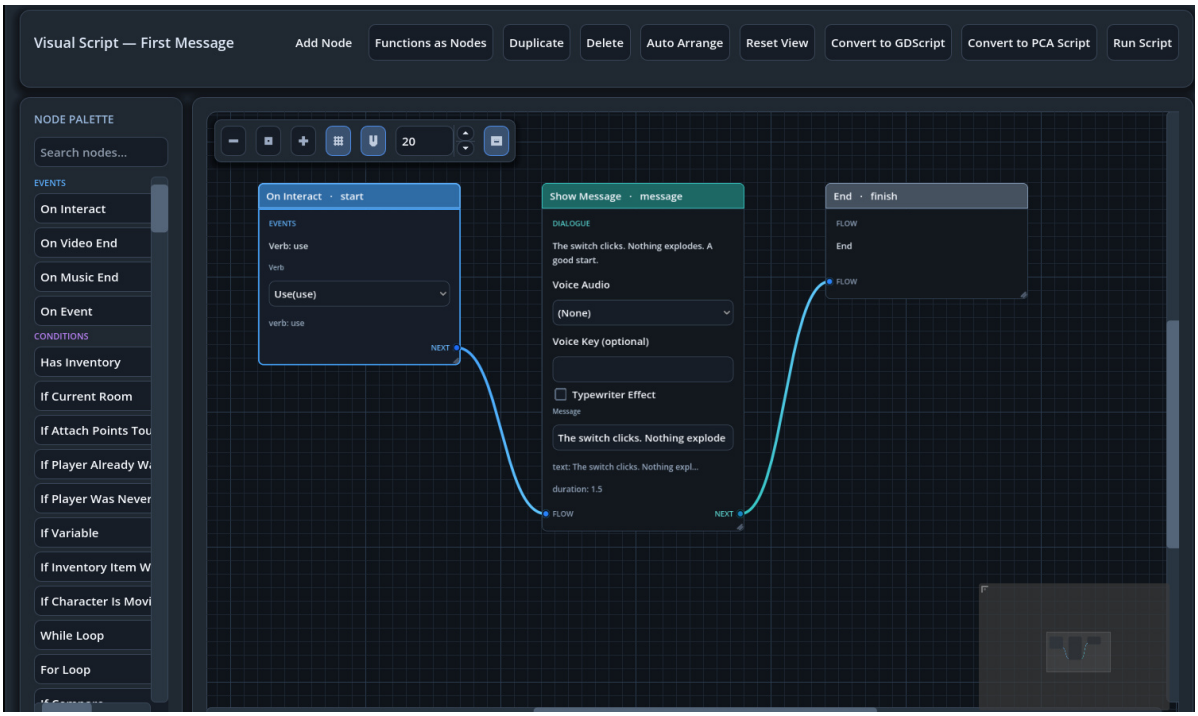


Figure 1. A real Visual Script in the editor: an entry, a message and an ending.

A practical guide to nodes, branches, variables, loops, events, reusable functions and the systems they control.

For PCA 3D Adventure Studio 0.86.3.4

# How to use this guide

Start with the first message. Get it running before building the key-and-door puzzle. Each new idea then has a visible result you can check.

Lessons 1-14 explain the workspace, nodes, connections, event assignments and testing. Lessons 15-28 build the puzzle’s state, branches, pickup and door logic, then introduce caller context and this.

Lessons 29-36 cover waits, loops, random choices, calls, returns and early endings. Lessons 37-47 build reusable visual functions and expose functions written in GDScript, PCA Script, Lua and BASIC.

Lessons 48-64 connect the graphs to events, GUIs, NPC movement, speech, dialogues, cutscenes, cameras, inventories, quests, particles, room changes and game saves. Lessons 65-69 cover testing, troubleshooting, conversion, Raw JSON and project organization.

## A few working habits

Commit text fields with Enter or by moving focus. Save authoring changes. Restart the preview when you need a fresh copy of the project. Test the actual player event as well as a direct script call.

## Names used in the examples

The Workshop Switch is a small demonstration scene made with existing engine assets and simple primitives. The key, door, switches and beacon keep the logic visible. Use your own IDs if you are following along in another project.

Role names in connection tables describe a node’s job. Your node IDs may be n1, n2 and n3; the logic does not depend on reproducing the screenshot’s custom node IDs.

## Screenshots and scope

All editor and gameplay illustrations are actual native captures. Instructions are platform-neutral. Theme, display size and panel layout can differ. No practice files or engine changes are required.

Use the clickable contents and PDF bookmarks to return to a lesson. The appendices contain compact references and examples inside this PDF; they are not separate downloads.

# Contents

1. What a visual script does . . . 5
2. Set up a small working room . . . 6
3. Create and name a Visual Script . . . 7
4. Find your way around the workspace . . . 8
5. Use the searchable node palette . . . 9
6. Read a node and its ports . . . 10
7. Build your first message graph . . . 11
8. Edit values without losing them . . . 12
9. Connect, disconnect and repair a wire . . . 13
10. Understand what Run Script tests . . . 14
11. Assign a graph to a verb interaction . . . 15
12. Move, resize, zoom and arrange . . . 16
13. Duplicate and delete carefully . . . 17
14. Save authoring work and reset test state . . . 18
15. Give the puzzle a memory . . . 19
16. Set, create or change a value . . . 20
17. Ask a simple yes-or-no question . . . 21
18. Compare numbers and live values . . . 22
19. Read values with expressions . . . 23
20. Keep strings, numbers and flags straight . . . 24
21. Join branches without running both . . . 25
22. Build the key pickup . . . 26
23. Make the pickup safe to repeat . . . 27
24. Build a locked-door puzzle . . . 28
25. Change the door’s visible and usable state . . . 29
26. Reuse a graph with “this” . . . 30
27. Read the event context . . . 31
28. Keep caller identity through nested work . . . 32
29. Waits and actions that take time . . . 33
30. Repeat a counted sequence with For Loop . . . 34
31. Repeat while a condition is true . . . 35
32. Keep loops and nested calls bounded . . . 36
33. Choose a random outcome . . . 37
34. Call another script and return a value . . . 38
35. Use a script-owned shared variable . . . 39
36. End a path, return, or stop a caller . . . 40
37. Decide what should become a function . . . 41
38. Build a reusable visual function . . . 42
39. Fill in definitions and use them . . . 43
40. Call the function with named fields . . . 44
41. Pass a live value with Expression . . . 45
42. Use types and return values deliberately . . . 46
43. Keep several visual functions in one record . . . 47
44. Expose a GDScript helper . . . 48
45. Expose the same function in PCA Script . . . 49
46. Call Lua and BASIC functions too . . . 50
47. Fix a missing or invalid function call . . . 51
48. Listen with On Event . . . 52
49. Send a custom event with useful data . . . 53
50. Filter listeners and prevent event loops . . . 54
51. Start work on room entry or game startup . . . 55
52. Respond when video or music ends . . . 56
53. Call a graph from a GUI button . . . 57
54. Update HUD text and use wildcards . . . 58
55. Walk to an NPC, face them and speak . . . 59
56. Add animation, speech, voice and typewriting . . . 60
57. Start a branching conversation . . . 61
58. Start and finish a cutscene from a graph . . . 62

59. Choose and blend cameras . . . 63
60. Handle inventories for different characters . . . 64
61. Tie actions to quests and objectives . . . 65
62. Attach and control a particle effect . . . 66
63. Change rooms and choose a spawn . . . 67
64. Save and load the puzzle . . . 68
65. Test a graph in manageable pieces . . . 69
66. Troubleshoot the symptom you can see . . . 70
67. Convert a copy to a text script . . . 71
68. Use Raw JSON for an omitted field . . . 72
69. Keep a growing project understandable . . . 73
Appendix A. Ports and execution reference . . . 74
Appendix B. Expressions and call data . . . 75
Appendix C. Exposed visual function data . . . 76
Appendix D. A target-aware text helper . . . 77
Appendix E. Event context quick reference . . . 78
Appendix F. A complete small graph . . . 79
Appendix G. Node catalog . . . 80

# 1. What a visual script does

A visual script is a set of instructions joined by wires. An instruction might give the player an item, test a variable, play an animation or open a GUI. The wire tells the game which instruction to visit next.

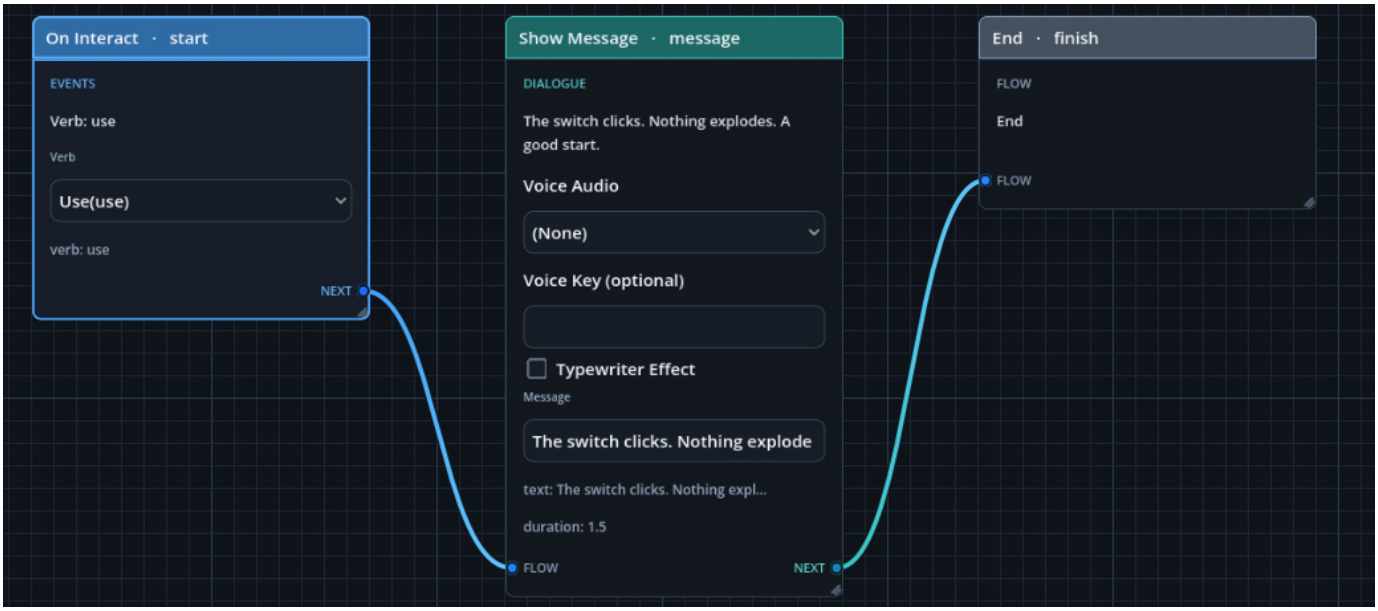


Figure 2. A working three-node graph. Follow the wire from On Interact, through Show Message, to End.

This first graph says: when it is called with the Use verb, show a message, then finish. Nothing in that sentence requires a text programming language. You still need to decide what the game should do, in what order, and under which conditions.

| Term | Meaning in this guide |
|---|---|
| Graph | One Visual Script record, containing nodes and connections. |
| Node | One event entry, action, question or flow instruction. |
| Port | A connection point at the edge of a node. |
| Wire / connection | The route from one output to the next node’s FLOW input. |
| Caller | The object or other record whose event started the work. |

> The wires carry execution order. Values are entered in fields or read through expressions. You do not connect a score-value wire to a separate arithmetic socket in this editor.

# 2. Set up a small working room

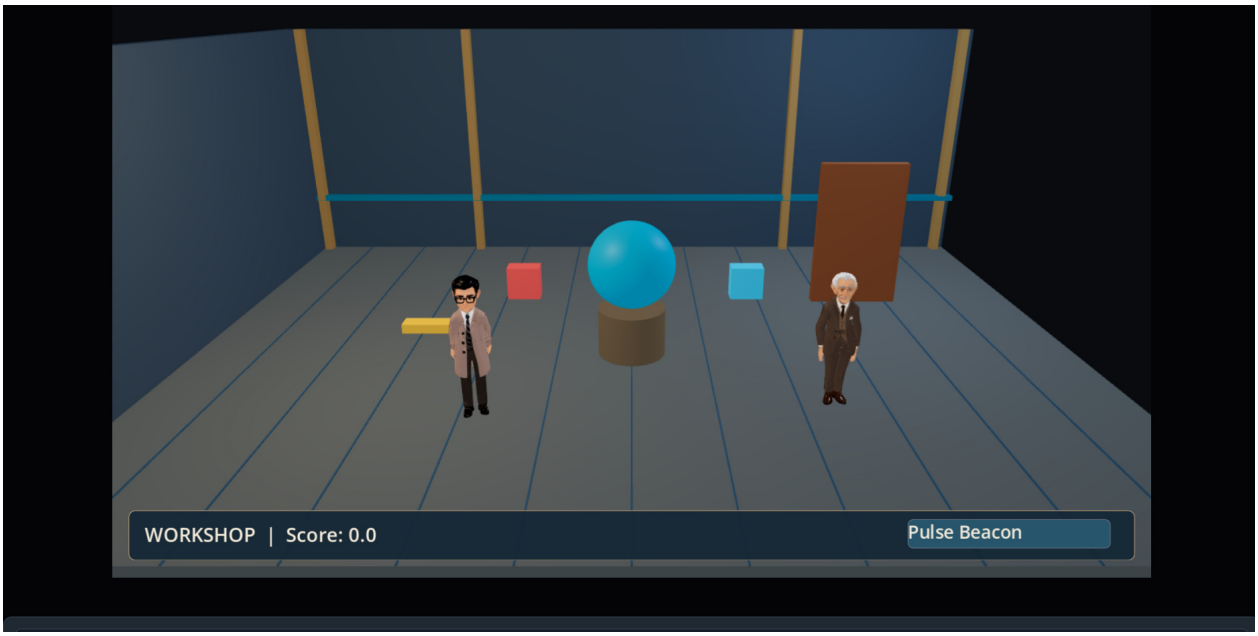


Figure 3. The workshop used throughout the guide. Simple objects make the results of the logic easy to see.

Use a new project or a copy of a working game. You can follow the lessons with primitives, the bundled character models or your own artwork. The logic is the important part; a plain box can stand in for a door.

| Record | Example ID / purpose |
|---|---|
| Room | lab: a floor, navigation, a player and a working default camera. |
| Player / NPC | ada is the live player; player is its runtime alias. elias is the NPC. |
| Scene key | brass_key: the object that can be picked up. |
| Inventory definition | brass_key_item: the item carried after the pickup. |
| Door | workshop_door: a blocking scene object. |
| Beacon / switches | shader_orb, blue_switch and red_switch. |
| GUI | scene_hud, with a label hud_text and button pulse_button. |

If your room uses different IDs, substitute them throughout. Keep the scene key and inventory item distinct. Picking up an object does not automatically create an inventory definition.

> No practice project is needed. These are instructions for building the examples in a project of your own. Save a copy before changing an existing game.

# 3. Create and name a Visual Script

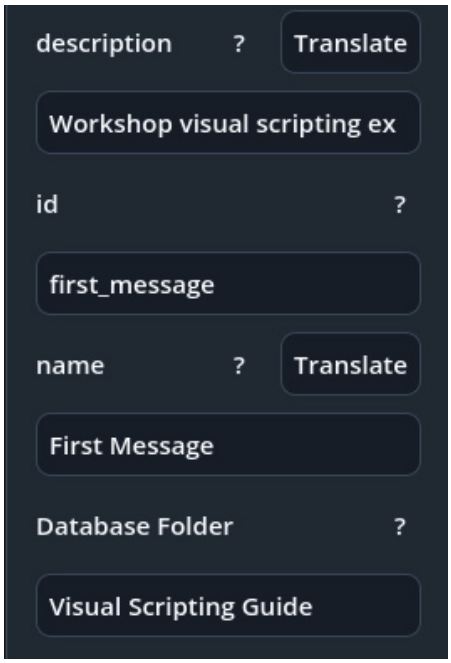


Figure 4. The script record has its own ID, display name, description and database folder.

1. Open Visual Scripts from the workspace toolbar or Adventure Database.
2. Press Add New in the Visual Script Designer. The new record starts with an empty graph. You can also use the database Add command and choose Visual Script.
3. Show the Inspector if it is hidden. In Fields, give the record a useful name, such as First Message.
4. Use a stable ID such as first_message. Apply Changes after editing the record fields.
5. Save the project, then select that record in the designer’s top dropdown.

The name is what people read. The ID is what actions and other records use to find the script. A picker may display First Message(first_message); the stored reference is still first_message.

Node edits and record identity edits use different controls. Edit a node’s fields inside the graph. Edit the whole script’s name, description or folder in the Inspector.

> Use unique script IDs across all languages. Avoid renaming an ID after it has been assigned to many events. A friendly display name can usually be improved without changing the ID.

# 4. Find your way around the workspace

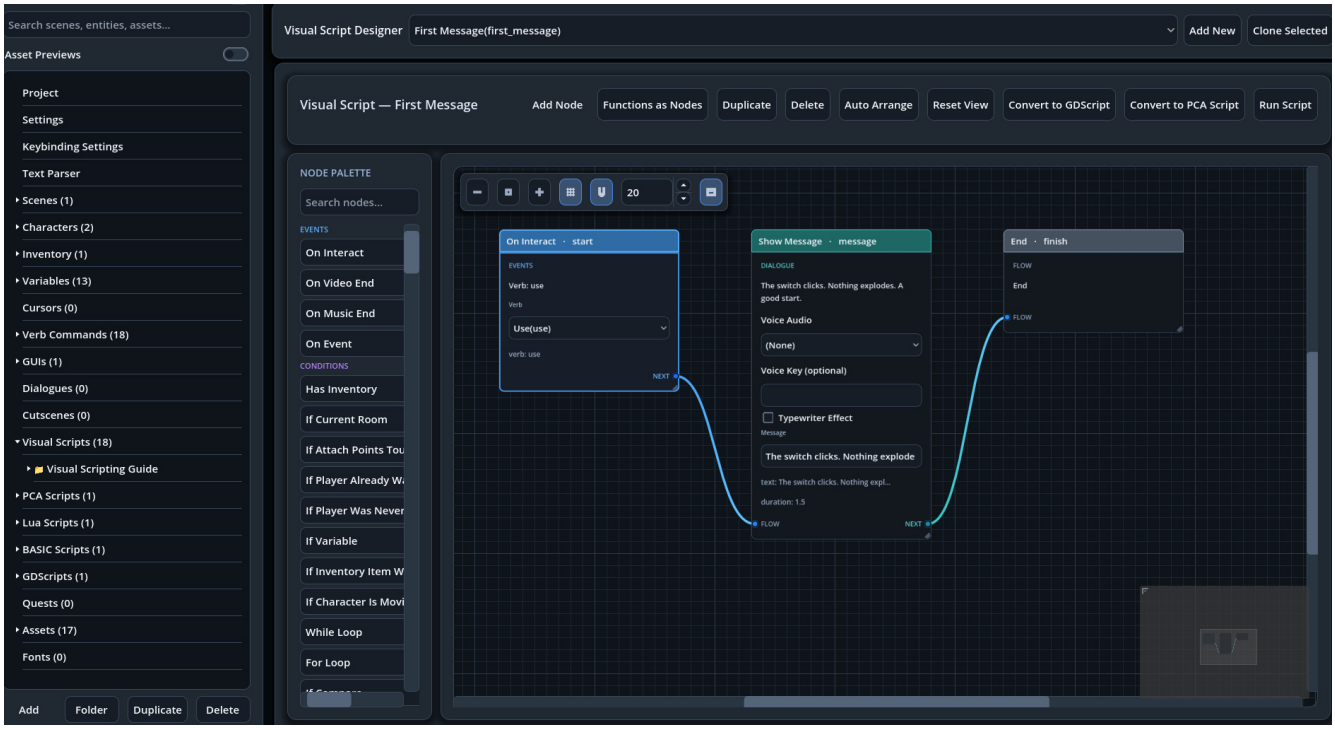


Figure 5. Adventure Database at the left, the script selector above, the node palette beside the graph and the graph toolbar across the top.

| Area | What you use it for |
|---|---|
| Adventure Database | Select and organize scripts alongside the other project records. |
| Designer selector | Choose the Visual Script to edit; Add New creates another record. |
| Node palette | Search for an action, then press its button to add a node. |
| Graph canvas | Move, resize, select and connect nodes. |
| Graph controls | Zoom, grid/snap settings, scroll bars and minimap. |
| Script toolbar | Functions as Nodes, Duplicate, Delete, Auto Arrange, Reset View, conversion and Run Script. |
| Inspector | Record fields and Raw JSON (Advanced). |

A narrow window may put toolbar commands behind a horizontal scroll strip. Expand the workspace, close an unused panel or scroll the toolbar before assuming that a button has disappeared.

> Theme and UI size change the appearance. Follow control names and port labels. The screenshots show one native editor layout, rather than a required screen resolution.

# 5. Use the searchable node palette

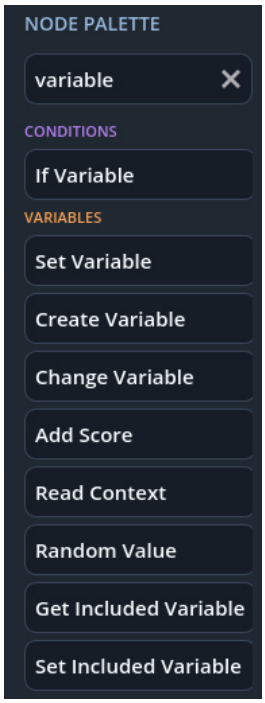


Figure 6. Searching for variable narrows the palette. Search examines action names and their categories.

1. Click Search nodes in the palette.
2. Type a useful word: variable, inventory, camera, sound, attach or GUI.
3. Press the desired node button. The new node appears in the graph and becomes selected.
4. Clear the search to restore the full category list.
5. Use Add Node in the script toolbar when the menu is more convenient.

The current build exposes 231 distinct node types. Many use the same action catalog as object events, cutscenes, GUI events and text scripts. That is why an action learned here often appears elsewhere with the same fields.

The categories are Events, Conditions, Dialogue, Variables, Inventory, Animation, Scene Life, GUI, Media, Characters, Game, Quests, Scene and Flow. Flow also contains newer shared actions, including some camera and attachment controls. Search is often quicker than guessing the category.

> Adding a node does not connect it. An unconnected action can remain visible on the canvas and never run.

# 6. Read a node and its ports

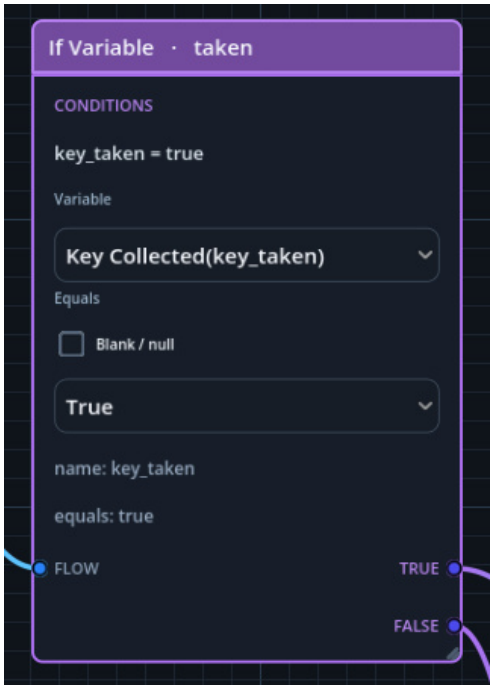


Figure 7. A condition node has two named outputs. TRUE and FALSE are separate paths.

| Part | What it tells you |
|---|---|
| Title | Action name and node ID. n1, n2 and similar IDs distinguish instances of the same action. |
| Category / summary | A quick description of the node’s current settings. |
| Editable controls | The actual values that will be used: a variable, target, text, number or option. |
| FLOW at the left | The execution input for an ordinary action or condition. |
| NEXT at the right | The usual continuation after an action. |
| TRUE / FALSE | The chosen continuation after a question. |
| LOOP / DONE | The two exits of a For Loop. |

Entry nodes such as On Interact do not have a FLOW input. End, End Visual Script and Return have no outgoing path. The runner also stops naturally when the selected output has no connection.

> The role names in the connection tables in this guide, such as Check Key or Give Item, describe the node’s job. Your editor may label the same node n3. You do not need to rename every node ID to follow the instructions.

# 7. Build your first message graph

1. Create the first_message record if you have not already done so.
2. Add On Interact. Set Verb to Use(use).
3. Add Show Message. Enter: The switch clicks. Nothing explodes. A good start. Leave Typewriter Effect off for the first test.
4. Add End. Put the three nodes in reading order.
5. Connect On Interact NEXT to Show Message FLOW. Connect Show Message NEXT to End FLOW.
6. Save the project and press Run Script. You should see the message in the game preview.

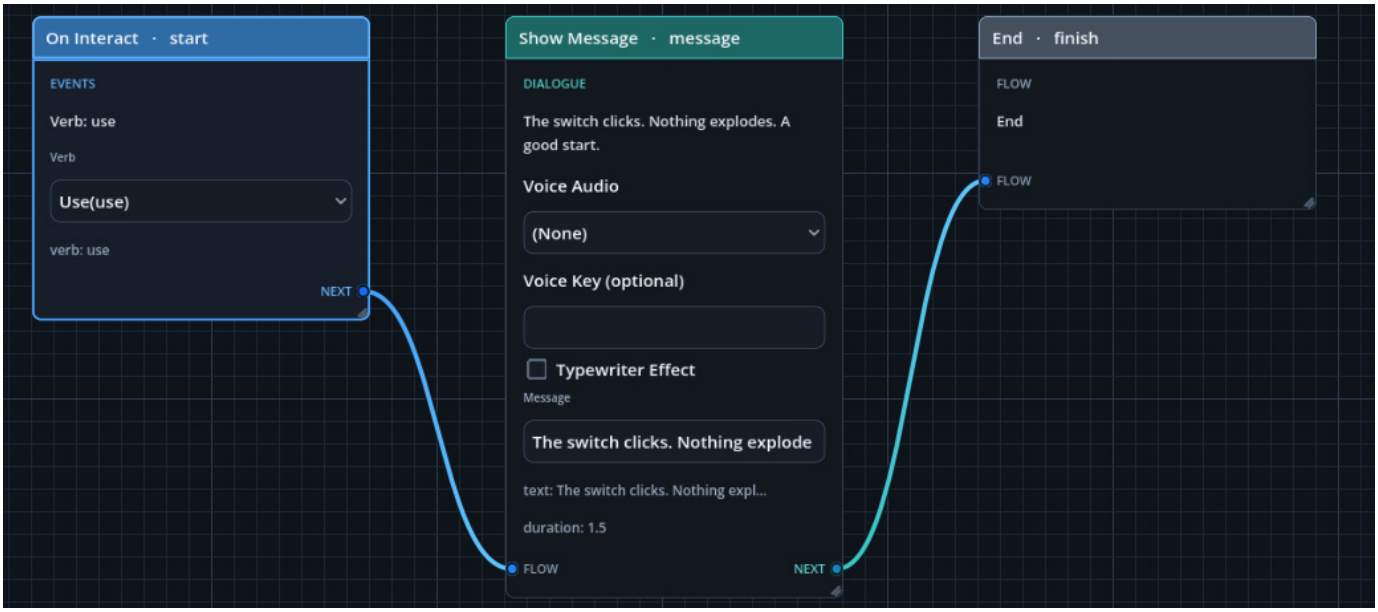


Figure 8. The first complete chain. Node placement makes it easy to read, while the wires determine the real order.

Show Message uses a short timed display. In this build its inline form exposes message text, voice fields and typewriter options; duration may appear only in the summary. The default is about 1.8 seconds. A hidden optional field can be changed through the record’s Raw JSON when necessary; see lesson 68.

> On Interact is an entry and verb check. It does not discover a switch in the room and attach itself to it. The assignment is a separate step in lesson 11.

# 8. Edit values without losing them

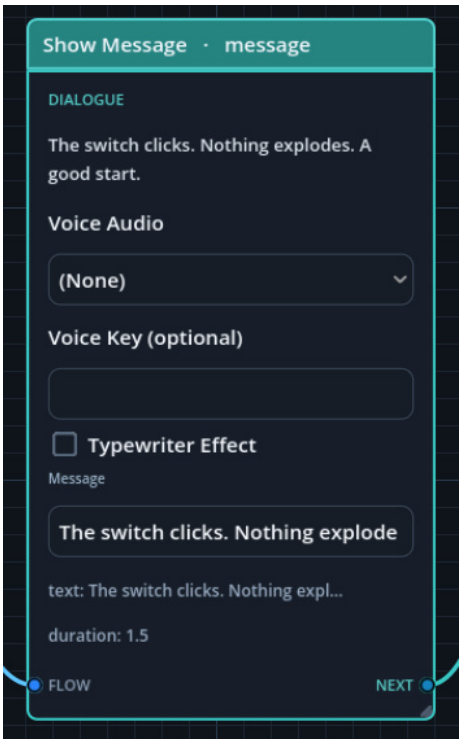


Figure 9. Message text, optional voice and typewriter settings live inside the node.

1. Click the field you want to change.
2. Enter the new value. For a text line, press Enter or click another control to commit it.
3. Check the node’s summary after committing. A graph rebuild may redraw the node.
4. Use a dropdown for record references whenever possible. Check both the friendly name and ID.
5. Save the project after the change.

A text box that still has focus can contain an uncommitted edit. Commit it before changing records or running a test. Dropdown selections commit through their selection control.

Several actions have typed values: Boolean choices, numbers, colors, arrays or dictionaries. A number typed into a JSON-value field is different from the same characters inside quotation marks. Lessons 19 and 20 explain the difference.

> Do not treat every line of a node’s summary as a text box. A displayed duration or property is not editable unless an actual control is provided for it. The advanced record editor remains available for fields omitted from the inline form.

# 9. Connect, disconnect and repair a wire

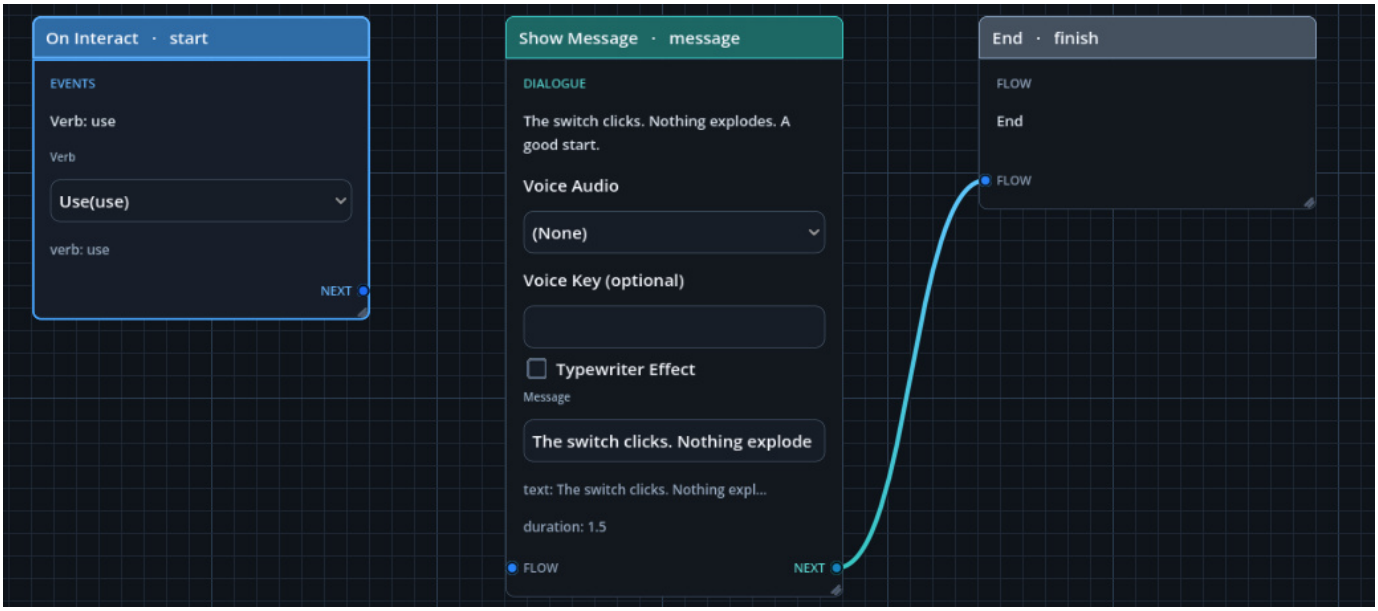


Figure 10. With the first connection removed, the message is no longer reachable from On Interact.

1. Drag from an output dot on the right to the next node’s FLOW dot on the left.
2. For a condition, start from the output whose label you intend: TRUE or FALSE.
3. To disconnect, use the graph’s right-click disconnection gesture at the connection port, then reconnect the intended route.
4. Inspect both ends of the visible wire. A line passing near a node is not a connection to it.
5. Run the graph after the repair and exercise both sides of a condition.

Keep one destination for each named output. The editor can store more than one connection from an output, but the runner follows the first matching edge. Two NEXT wires do not mean “run both actions in parallel.” Chain those actions together instead.

Several earlier paths may enter the same action. That is a merge: whichever path was taken can continue through the shared action. The shared action runs when execution reaches it.

> The graph is not executed from left to right by position. Moving an action above or below another does not change the stored connections.

# 10. Understand what Run Script tests

Run Script is a convenient quick test. It starts the embedded preview if necessary, selects Play and calls the selected script with a Use verb and an editor-test context. It is useful for checking a message or a small calculation.

| Test | What it proves |
| --- | --- |
| Run Script on first_message | The graph can start and reach the message with Use. |
| Use on an actual switch | The switch’s event assignment, selected verb and graph work together. |
| Pick Up on the key | The Take entry and the key’s Take action agree. |
| A real trigger entry | The correct actor entered the correct trigger and supplied its context. |
| A GUI button in Play | The runtime click event, target script and GUI context are correct. |

A Take-only On Interact graph can appear to do nothing under Run Script, because that quick test supplies Use. A script using this can also fail there because the editor call has no owning scene object.

1. Use Run Script first for an owner-independent chain.
2. Assign the graph to the real event.
3. Restart Play or Test Scene after changing authoring data.
4. Perform the actual interaction with the actual verb.
5. Check the visible outcome and the saved state, not only the absence of an error.

> An already-running preview can retain its earlier project snapshot and runtime variables. A fresh restart is the reliable way to test a changed graph against its intended starting state.

# 11. Assign a graph to a verb interaction

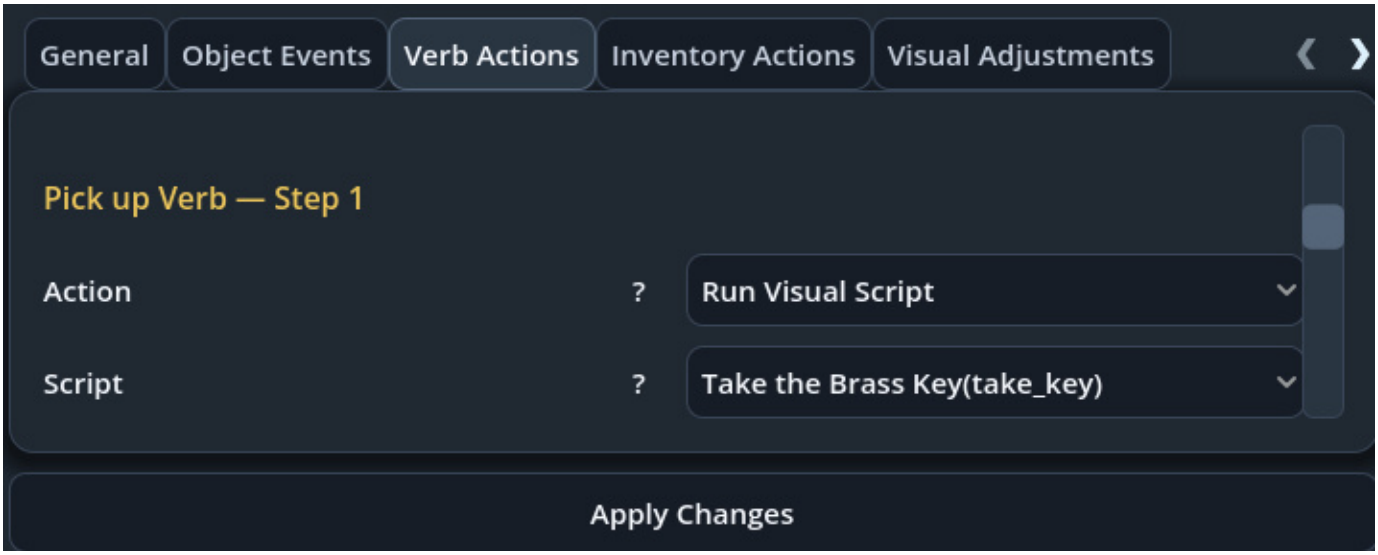


Figure 11. The key's Pick Up verb calls Take the Brass Key through Run Visual Script.

1. Open the room in Scene Designer. Select the actual key object.
2. Open its Verb Actions tab. Find Pick Up / Take.
3. For the first step, choose Run Visual Script.
4. Choose Take the Brass Key(take_key) in Script. Apply Changes and save.
5. In the graph's On Interact node, choose Pick up(take).
6. Start Test Scene, choose Pick Up and click the key.

For a Use switch, make the same assignment under Use and choose use on the entry. You can also call a graph from a general click or trigger event. Choose the event you want the player to perform, rather than assigning the same work everywhere.

An action list can run work before or after the graph call. For example, a short sound can precede the call. Keep one clear owner of the reward logic so it is not awarded once in the list and again in the graph.

> Use Preposition is a separate interaction setting. A simple switch can use a plain Use action; a "Use key with door" puzzle also needs its inventory/preposition response configured.

# 12. Move, resize, zoom and arrange

1. Drag a node by its title bar to move it. Keep branches separated vertically.
2. Drag its lower corner to make room for a long field or description. Saved node sizes are editor layout information.
3. Use the graph’s minus, reset and plus controls to change zoom. The supported range is roughly 35% to 220%.
4. Use the graph scroll bars and minimap to reach a distant branch.
5. Use Auto Arrange for a starting layout, then adjust spacing where a large message or parameter form needs more room.
6. Use Reset View to return zoom and scroll to their defaults.

The minimap helps you find the shape of a large graph. It is not a substitute for reading the actual port labels. Zoom in before deciding whether a branch is connected to TRUE or FALSE.

| Change | Effect on gameplay |
| --- | --- |
| Node position or size | Changes the editor layout. |
| Zoom, grid and snap | Changes how you work on the canvas. |
| Connection source / output / target | Changes the route the runner takes. |
| A field inside data | Changes the instruction’s values. |

Reset View is a view reset, not a reset of the graph or game variables. A graph placed far from the origin may need scrolling or framing again afterward.

# 13. Duplicate and delete carefully

1. Select the node or group of nodes you want to reuse. Use the canvas selection controls appropriate to your mouse or touch setup.
2. Press Duplicate. Copies receive new node IDs and appear with an offset.
3. Connections between nodes inside the selected group are copied. Connections to unselected outside nodes are not copied into the new group.
4. Inspect the copied fields. A duplicated reward still refers to the original item or variable until you change it.
5. Reconnect the copied group deliberately.
6. Use Delete to remove selected nodes. Connections touching a deleted node are removed too.

Cloning a whole script through Clone Selected creates another script record. That is different from duplicating several nodes inside one record. A separate record is useful for a separate puzzle; a callable function is useful for shared behavior.

Deleting a function entry node can leave its exposed definition pointing to a missing entry. Repair the definition, choose the new entry or remove the unused definition before testing.

> Save a working version before a large rearrangement. After duplication or deletion, test every route that used to enter or leave that section.

# 14. Save authoring work and reset test state

There are two kinds of state to keep straight: the project you are editing, and the running game. Saving the project preserves the graph and its defaults. A game save preserves supported runtime state such as inventory and variables.

| Action | Expected result |
| --- | --- |
| Save Project | Preserves the authored graph, assignments and default values. |
| Run a graph again in the same preview | Can see values already changed by its earlier run. |
| Restart the preview | Starts a fresh run from the authored starting configuration. |
| Load a game save | Restores the supported runtime state stored in that save. |
| Change a variable’s default in the editor | Changes the next fresh run; it is not a command to rewrite every existing save. |

1. Save after changing a record or graph.
2. Restart before a clean first-time test.
3. Run the interaction twice to test repeat behavior.
4. Leave and re-enter the room to test persistence.
5. Make a game save, change the puzzle, then load the save and check the result.

> A puzzle that works once can still give duplicate items or points on the second click. Repeated-use tests are part of building the puzzle, not an optional finishing touch.

# 15. Give the puzzle a memory

Variables remember what has happened. A door needs a door_open flag. A collectible needs a key_taken flag. A score is a number. Good variables let the graph ask a clear question instead of trying to infer the entire puzzle from what happens to be visible.

| ID | Type | Starting value | Purpose |
|---|---|---|---|
| key_taken | Boolean | false | Prevents a second pickup reward. |
| door_open | Boolean | false | Records the completed door puzzle. |
| score | Number | 0 | Tracks the points awarded. |
| pulse_count | Number | 0 | Counts a demonstration loop. |
| loop_index | Number | 0 | Receives the For Loop index. |
| last_target | String | empty text | Stores an event entity ID. |
| busy | Boolean | false | Prevents overlapping starts when required. |

1. Open Variable Designer and add the variables your lesson needs.
2. Give each a stable ID, a readable name and the appropriate type.
3. Set the starting value, apply and save.
4. Return to the graph and choose the variable from the relevant dropdown.
5. Restart the preview to test those authored starting values.

Use a Boolean for a yes/no state. A text value containing the word false is not the Boolean false. A missing variable is also not the same as a correctly initialized false flag.

> Declare important puzzle state in Variable Designer even though several runtime nodes can create destinations by name. Declared variables are easier to discover, inspect and maintain.

# 16. Set, create or change a value

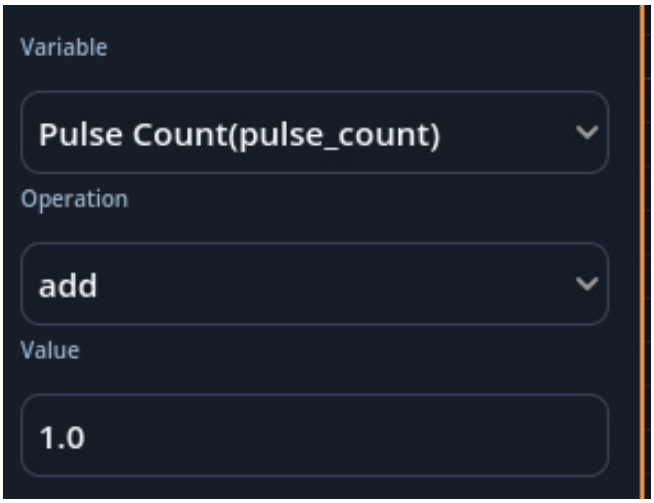


Figure 12. Change Variable can add to a number, toggle a flag or set a live value from an expression.

| Node | Use |
|---|---|
| Set Variable | Assign one value. Set key_taken to Boolean true. |
| Create Variable | Add a runtime variable only if that name does not already exist. It does not reset an existing one. |
| Change Variable: set | Assign a JSON-capable value, including a resolved expression. |
| Change Variable: add / subtract | Increase or decrease a numeric value. |
| multiply / divide | Numeric arithmetic. A zero divisor leaves the existing value unchanged. |
| append | Append one value to an array; it is not text concatenation. |
| toggle | Invert the current truth value. |

1. Add Change Variable. Choose pulse_count.
2. Choose add under Operation.
3. Enter 1 in Value, without quotation marks. Commit the field.
4. Call the graph, then inspect pulse_count. A second call should add one again.

The special Set Variable editor follows the declared variable’s type. For an expression such as {"$param":"amount"}, use the JSON-capable Change Variable node with operation set or add instead of pasting that object into a Boolean dropdown.

# 17. Ask a simple yes-or-no question

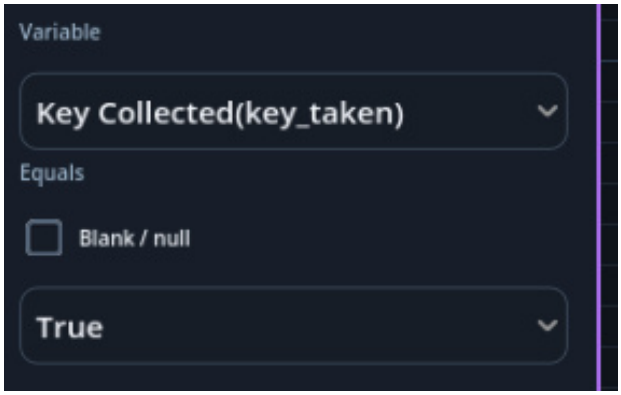


Figure 13. If Variable compares one runtime variable with its Equals value.

1. Add If Variable after the event entry.
2. Select key_taken.
3. For a Boolean variable, clear Blank / null and choose True for Equals.
4. Connect TRUE to the already-collected response.
5. Connect FALSE to the pickup actions.
6. Test once with the flag false and again after the pickup has made it true.

The graph takes exactly one of those outputs. It does not run both responses. An unconnected chosen output ends that route, which can be useful when nothing should happen on the false branch.

| Stored value | Equals | Expected route |
|---|---|---|
| Boolean false | Boolean true | FALSE |
| Boolean true | Boolean true | TRUE |
| Text "true" | Boolean true | Different value types; correct the input. |
| null / missing | Boolean false | A missing flag is not a deliberate false default. |

If the typed Equals editor changes unexpectedly when you choose a different variable, check the new variable’s declared type and default. Do not keep the old value by habit.

# 18. Compare numbers and live values

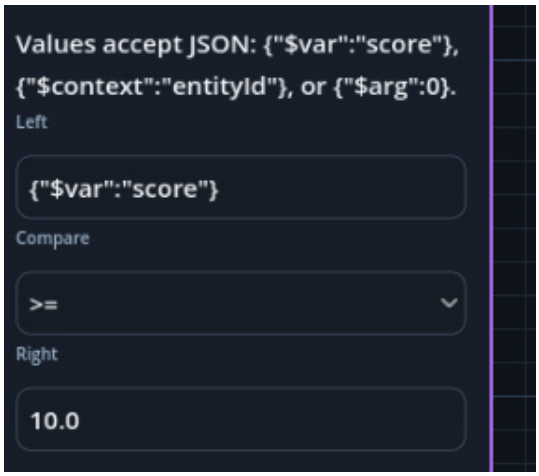


Figure 14. If Compare accepts a live expression on either side and a chosen comparison operator.

1. Add If Compare.
2. In Left, enter {"$var":"score"}. Commit the field.
3. Choose >= under Compare.
4. In Right, enter 10.
5. Connect TRUE to the “enough points” response and FALSE to the “not yet” response.
6. Test with score 9, 10 and 11. Ten belongs on the TRUE path because the operator includes equality.

| Operator | Meaning |
|---|---|
| == | Equal |
| != | Different |
| < / <= | Less than / less than or equal to |
| > / >= | Greater than / greater than or equal to |

Use equality for IDs, text and flags. Use the ordering operators for genuine numeric values. This runner converts operands to numbers for <, <=, > and >=, so those operators are not an alphabetical text-sorting feature.

> Typing score by itself produces a literal string in a JSON-value field. The dictionary {"$var":"score"} is what asks the runner to read the current score.

# 19. Read values with expressions

Expressions are small JSON dictionaries with one special key. The runner resolves them when it reaches the node, so a later node sees values changed by an earlier one.

| Expression | Reads |
|---|---|
| {"$var":"score"} | The live project/runtime variable score. |
| {"$context":"entityId"} | entityId supplied by the event or caller. |
| {"$this":"id"} | The captured caller’s ID. |
| {"$this":"name"} | The captured caller’s name property. |
| {"$param":"amount"} | The named parameter of the current exposed visual function. |
| {"$arg":0} | The first positional call argument; 1 means the second. |

Use these in fields labeled as JSON values, such as If Compare Left/Right, Return Value, Call Script Arguments, Change Variable Value and Emit Event context. Call Script Function supplies its own Expression checkbox for each named parameter.

```
{
  "owner": {"$this":"id"},
  "amount": {"$param":"amount"},
  "items": ["brass_key_item", {"$var":"score"}]
}
```

Values are resolved recursively inside arrays and dictionaries. A dictionary used as a special expression must contain just that special key. The dictionary above is an ordinary result object whose individual values contain expressions.

> These are lookups, not arbitrary formula strings. {"$var":"score + 1"} looks for a variable literally named score + 1. Use Change Variable → add for the arithmetic.

# 20. Keep strings, numbers and flags straight

| Input in a JSON-value field | Meaning |
|---|---|
| 7 | A number. |
| "7" | Text containing a digit. |
| true / false | Booleans, with no quotes. |
| "true" | Text, not a Boolean flag. |
| "brass_key_item" | A literal item ID as text. |
| [1,2,3] | An array. |
| {"x":1,"y":2} | A dictionary/object. |
| null | No value. |
| {"$var":"score"} | An expression that reads a live value. |

A plain string field, such as a message or parameter of type string, normally accepts ordinary unquoted text. A field labeled JSON or variant needs valid JSON to preserve structured values. Follow the field’s type rather than applying one quoting rule to every box.

1. If a comparison fails unexpectedly, inspect the value type before adding more branches.
2. Check whether a number was quoted, a flag was typed as text or an expression was entered in a plain string field.
3. Check capitalization and spelling in IDs and expression keys.
4. Use a small Return or result variable to inspect the actual value.

> Invalid JSON may remain literal text in generic variant fields or fail validation in stricter function fields. Do not assume a field accepted the intended value just because the characters are still visible.

# 21. Join branches without running both

Suppose either response should finish by restoring player input. Both responses can connect to the same restore action. The runner reaches that shared node from whichever branch it actually took.

| From | Output | To |
|---|---|---|
| Check answer | TRUE | Success response |
| Check answer | FALSE | Failure response |
| Success response | NEXT | Enable player input |
| Failure response | NEXT | Enable player input |
| Enable player input | NEXT | End |

That is a merge, not a parallel join. There is no pair of simultaneously running branches for the last node to wait on. One call follows one path at a time.

1. Give the condition one TRUE wire and one FALSE wire.
2. Make each route do only the work intended for that outcome.
3. Connect both routes to a shared cleanup step if they need the same cleanup.
4. Test the two routes separately.
5. Check early Returns and Ends: they stop before any cleanup further down the graph.

> Keep one wire per named output. If you need two actions after a success, connect Success → Action A → Action B. Two wires from Success are not a fan-out feature in this runner.

# 22. Build the key pickup

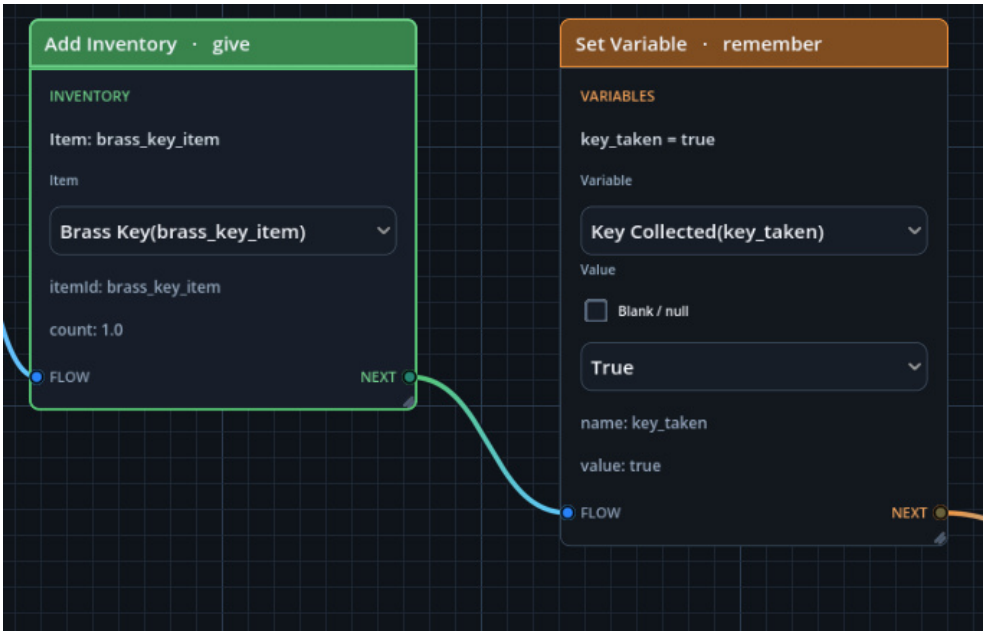


Figure 15. The pickup adds an inventory item, then remembers that the reward was collected.

| Node role | Type and settings |
|---|---|
| Start | On Interact; verb take. |
| Check Taken | If Variable; key_taken equals true. |
| Already Taken | Show Message: I already have the key. |
| Give Item | Add Inventory; brass_key_item; count 1. |
| Remember | Set Variable; key_taken = true. |
| Hide Scene Key | Hide / Unhide Object; room lab; object brass_key or this; Visible false. |
| Award | Add Score; amount 5. |
| Found Message | Show Message: A key. Now, that door... |

Wire Start to Check Taken. TRUE goes to Already Taken. FALSE goes to Give Item, then Remember, Hide Scene Key, Award and Found Message. The item definition must already exist in Inventory Designer.

> The scene object and carried item have different IDs here. Add Inventory uses brass_key_item. Hiding the scene object uses brass_key. Mixing them up is a common reason a pickup appears half-finished.

# 23. Make the pickup safe to repeat

1. Set key_taken to false in Variable Designer for a fresh game.
2. Place the If Variable guard before Give Item and Add Score.
3. Set key_taken true as part of the successful pickup.
4. Assign the graph only to the intended pickup response.
5. Test the successful pickup. You should gain one item and five points.
6. Call it again during testing. The guard should produce the already-collected response without another reward.

Hiding the key usually prevents another ordinary click, but the flag still matters. Another script, an event, a saved room state or a later change could call the same pickup logic again.

| After a fresh run | Expected state |
|---|---|
| Before pickup | key_taken false; key quantity 0; score 0. |
| After first pickup | key_taken true; quantity 1; score 5; scene key hidden. |
| After repeated call | Same quantity and score. |
| After leaving and returning | The supported room/puzzle state remains consistent. |

For a longer asynchronous pickup sequence, consider a busy guard as well. Set busy before the first wait, reject overlapping starts and clear it on every completion path. This prevents two calls from both passing an initial test while the first is suspended.

> A one-time flag and a temporary busy flag solve different problems. Keep the permanent completion flag even after adding a busy guard.

# 24. Build a locked-door puzzle

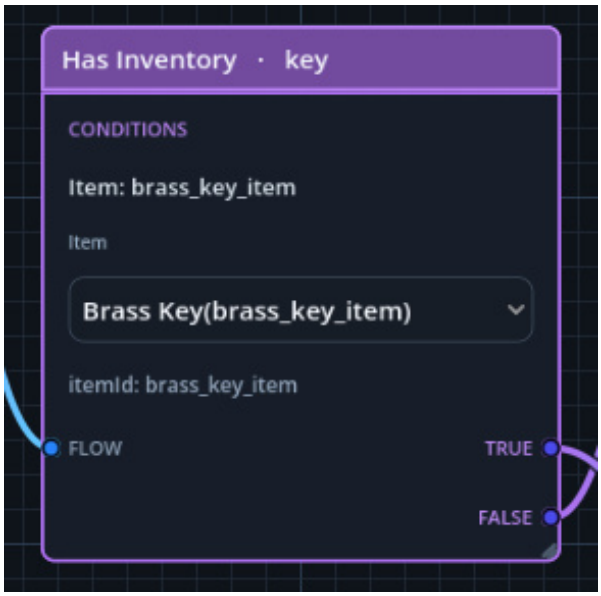


Figure 16. Has Inventory asks whether the active inventory contains the required item.

| From | Output | To |
|---|---|---|
| On Interact: use | NEXT | If Variable: door_open == true |
| Door already open? | TRUE | Already-unlocked message |
| Door already open? | FALSE | Has Inventory: brass_key_item |
| Has key? | FALSE | Locked / need-key message |
| Has key? | TRUE | Set door_open true |
| Set flag | NEXT | Disable door → hide door → add 10 points → success message |

Assign this graph to the door’s Use verb. A plain Use interaction tests possession of the key. It does not require the player to select the key and use it with the door; that more specific input needs the appropriate inventory/preposition response to call the same puzzle logic.

Keep the key unless the story says it should be consumed. To consume it, insert Remove Inventory only on the success path after the possession check. Do not remove it on the locked response.

> The already-open guard comes before the reward. Otherwise a repeat call can award another ten points even if the door is no longer visible.

# 25. Change the door’s visible and usable state

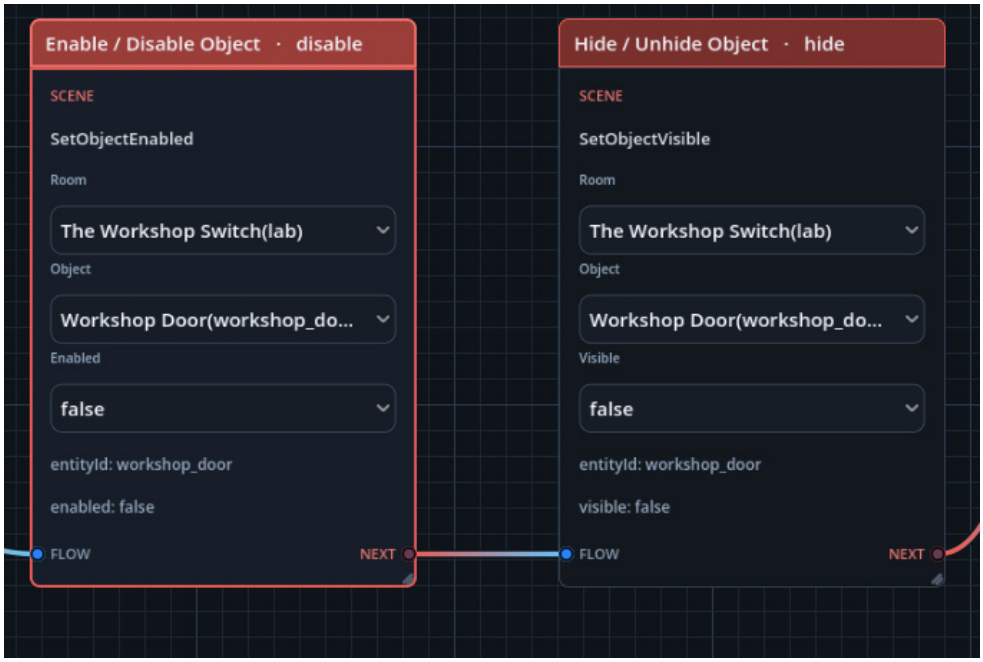


Figure 17. These actions explicitly disable the door and hide it after the puzzle is solved.

1. On the success path, set door_open true.
2. Use Enable / Disable Object with workshop_door and Enabled false when the old blocking object should stop participating.
3. Use Hide / Unhide Object with Visible false when the old door should disappear.
4. For a finished game, replace the disappearance with an opening animation or transform, while updating the collider and interaction state deliberately.
5. Test walking through the opening and clicking the area after the change.

In the current 3D runtime, disabling the entity disables its runtime node, and hiding it also updates its active collision state. The explicit pair makes the intended solved state easy to read. Test the actual object type and room mode you use.

A flag alone does not move a mesh. A mesh animation alone does not necessarily make an authored navigation blocker match the new opening. Keep the logical flag, visible state and collision/navigation behavior in agreement.

> If the door is already hidden or disabled, an ordinary player click will not reach it again. A repeat-call test can still verify that its reward guard is sound.

# 26. Reuse a graph with “this”

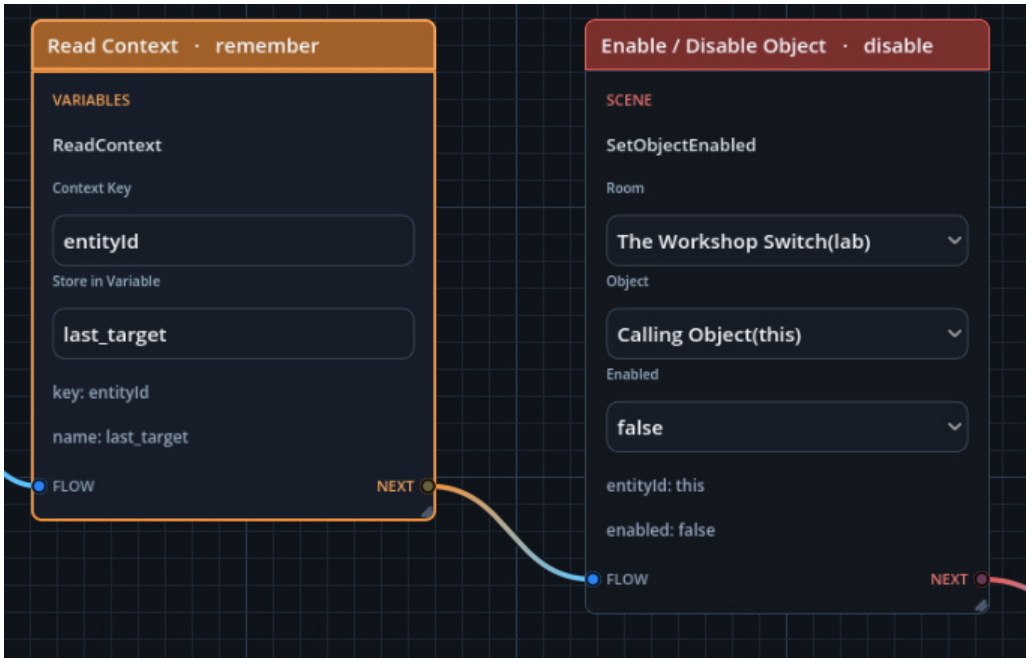


Figure 18. The shared switch graph stores the event target, then disables this rather than a hard-coded switch ID.

1. Create blue_switch and red_switch as separate scene objects.
2. Create one switch_self graph with On Interact: use.
3. Add Enable / Disable Object. Choose this as the object and set Enabled false.
4. Assign the same graph to the Use verb on both switches.
5. Use the blue switch. Only the blue switch should disable.
6. Use the red switch. Only the red switch should disable.

this refers to the captured owner of the event that called the graph. It is not the object selected in the editor. It is also different from player, which resolves to the currently active player character.

A direct editor test may have no owning object. In that case this cannot resolve; the action is skipped with a diagnostic. Test this through a real owner event, or supply an explicit target for an owner-independent utility.

> A GUI event’s owner is the GUI, not a nearby scene object. Use a specific object ID or an entity parameter when a GUI button should change an object in the room.

# 27. Read the event context




Figure 19. Return can assemble a dictionary from the caller and explicit event fields.

1. Add Read Context after the entry.
2. Set Context Key to entityId.
3. Set Store in Variable to last_target.
4. Call the graph through an object event.
5. Inspect last_target; it should contain that object’s live ID.

| Context key | Common source |
|---|---|
| entityId / verb | A scene object interaction. |
| itemId / amount / operation | Inventory-change notifications. |
| guiId / guiElementId / guiEvent | A GUI element event. |
| sceneId / previousSceneId / firstVisit | OnSceneLoaded. |
| triggerId / playerId | Trigger entry and exit. |
| arguments | Arguments supplied by a script call. |

Context fields depend on the event. Read Context copies the exact field; a missing field produces null. It does not infer a target from whatever the mouse happens to point at.

> Choose the key that answers the question. OnInventoryChanged includes the active character and the changed item; read itemId explicitly when you mean the item. The captured caller and an individual context field need not be the same record.

# 28. Keep caller identity through nested work

A graph can call another graph or a text function, wait, then continue. The original caller is captured so that this continues to mean the owner that started that chain. Changing an action’s target does not make that target the new caller.

```
{
  "owner": {"$this":"id"},
  "name": {"$this":"name"},
  "item": {"$context":"itemId"}
}
```

Put that value in Return for a small diagnostic helper. Call it from two different switches and inspect the returned owner and name. Then call it through Call Script from another graph; the original switch identity should remain.

| Reference | Meaning |
| --- | --- |
| this | Captured event owner for the current call chain. |
| player | The active player alias. |
| An explicit ID | The named entity or record. |
| {"$context":"entityId"} | The entityId field in the call context. |
| {"$param":"target"} | A named function argument, if declared and supplied. |

An ownerless startup event has no this. A scene change does not silently retarget an old call to a new room object. Shared helpers should either require an owner or take an explicit entity parameter.

> In GDScript, self continues to mean the script instance. this is PCA’s calling-record reference. They serve different purposes.

# 29. Waits and actions that take time

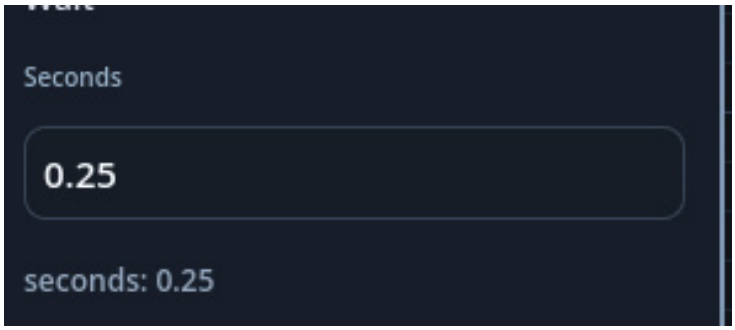


Figure 20. Wait uses seconds. It pauses this call chain before following NEXT.

Most actions complete immediately. Some take time: a timed message, a walk, a cutscene or an action with an explicit Wait for Blend option. The graph awaits each action call, but the action itself decides whether it waits for its full visible effect.

1. Add Wait where the sequence should pause.
2. Enter 0.25 for a quarter-second pause, or 1 for one second.
3. Connect its NEXT output to the following action.
4. Run the sequence and watch the actual timing.
5. For a camera or media action, inspect its own wait options rather than adding a guessed delay automatically.

| Action | Timing to consider |
|---|---|
| Wait | Always waits for its timer. |
| Show Message | Awaits its timed message presentation. |
| Character Walk To | Awaits the movement request’s completion. |
| Play Sound | Starts playback; do not assume it holds the graph until the sound ends. |
| Play Animation | Starts the selected clip; inspect the required completion logic. |
| Activate / Blend to Camera | Has an explicit Wait for Blend setting. |

Wait pauses this graph, not the whole game. Other input and events can still matter. Use a busy guard or input control deliberately if the same interaction must not start again during the pause.

# 30. Repeat a counted sequence with For Loop

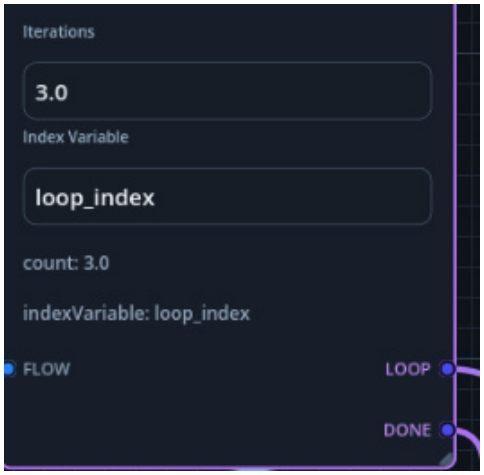


Figure 21. For Loop has LOOP and DONE outputs. The body must return to the loop node.

1. Create pulse_count and loop_index as number variables.
2. Set pulse_count to 0 before entering the loop.
3. Add For Loop. Set Iterations to 3 and Index Variable to loop_index.
4. Connect LOOP to Change Variable: pulse_count add 1.
5. Continue through a visible beacon on/off sequence, with a short Wait after each change.
6. Connect the last body node back to For Loop FLOW. Connect DONE to a final restore or Return node.

| Visit | Index / behavior |
|---|---|
| First body | loop_index = 0 |
| Second body | loop_index = 1 |
| Third body | loop_index = 2 |
| Next visit to For Loop | DONE; the body no longer runs. |

The loop counter belongs to this graph invocation and loop-node ID. The Index Variable is a runtime variable you can inspect or use. At the end of a three-iteration loop, its last written index is 2; it is not automatically changed to 3.

> Without the return wire, the body runs once and the graph follows whatever route you connected after it. The loop node must be revisited to decide whether another iteration is due.

# 31. Repeat while a condition is true

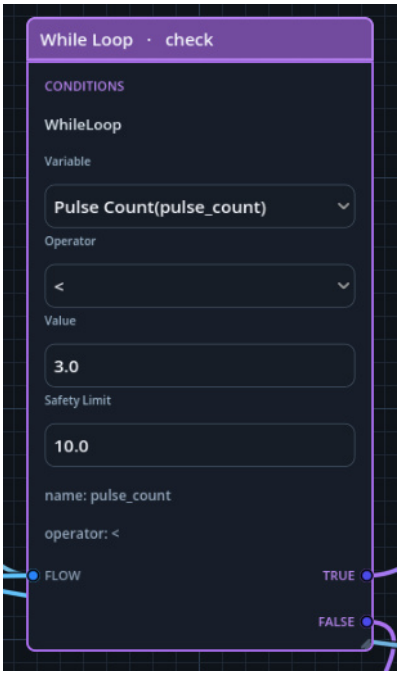


Figure 22. While Loop compares the named variable every time the body returns.

1. Set pulse_count to 0.
2. Add While Loop. Choose pulse_count, operator <, and Value 3.
3. Connect TRUE to Change Variable: pulse_count add 1.
4. Add a short Wait if the effect should be visible over time.
5. Connect the last body node back to While Loop FLOW.
6. Connect FALSE to Return with {"$var":"pulse_count"}. The result should be 3.

A While Loop does not increment the variable for you. Its condition has to become false through something the body or another part of the game changes.

The current inline form includes Safety Limit, but the visual runner does not apply that field as an independent per-loop cap. Build a genuinely bounded condition. The runner’s shared 4096-step guard is the fallback, not the normal exit for your puzzle.

> An endless loop can repeatedly award points, emit events or overwrite state before the shared guard stops it. Use a changing counter or a clear external completion condition, and check the FALSE exit.

# 32. Keep loops and nested calls bounded

The runner stops one graph after 4096 visited nodes. It also yields a frame every 64 visited nodes and rejects graph nesting at depth 24. These limits keep a broken graph from running forever, but they do not make an unbounded design correct.

| Limit / behavior | Practical consequence |
|---|---|
| 4096 node visits | Repeated visits count. A loop body of several nodes consumes several steps per iteration. |
| Yield every 64 visits | The editor/game can get a frame during heavy graph work. This is not a cinematic delay. |
| Nesting depth 24 | A graph calling another graph repeatedly, or calling itself, must have a proper stop condition. |
| Loop body back-edge | A missing return wire prevents repetition. |
| Shared index/result variables | Two overlapping calls can overwrite the same runtime destination. |

1. Use a small count while building a loop.
2. Test zero, one and several iterations.
3. Check the exit path and its cleanup.
4. Keep long-running polling loops slow enough to observe and easy to stop.
5. For large data processing or complex algorithms, consider a text helper with a narrow parameter/result contract.

> If the log says the graph stopped after 4096 steps, repair the graph. Increasing delays or hiding the message does not fix the route that keeps repeating.

# 33. Choose a random outcome

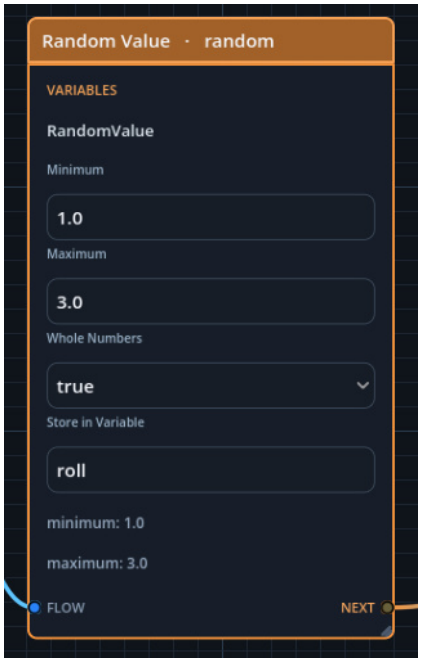


Figure 23. Random Value stores a result. Separate conditions decide what that result means.

1. Add Random Value. Set Minimum 1, Maximum 3 and Whole Numbers true.
2. Set Store in Variable to roll.
3. Add If Compare: {"$var":"roll"} == 1. TRUE goes to the first reply.
4. On FALSE, add another If Compare for roll == 2. TRUE goes to the second reply.
5. The second FALSE route goes to the third reply.
6. Call the graph several times and confirm that every route is valid.

The integer range includes both ends, so 1 through 3 gives three possible values. A floating range produces values between the limits and is better suited to variation in timing or strength than selecting numbered replies.

Random choice does not guarantee that all three results will appear in the first three tests. Use controlled values in the conditions when checking the wiring, then restore the random node.

> If each outcome changes persistent state, test each one deliberately. A rare random branch still needs correct item IDs, cleanup and an exit.

# 34. Call another script and return a value

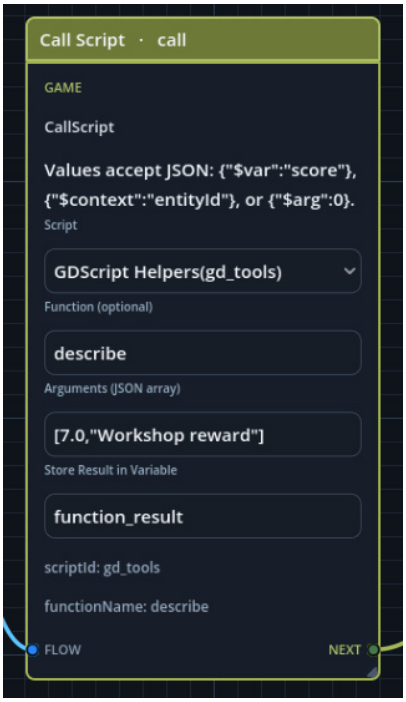


Figure 24. Call Script supplies a function name, a JSON argument array and an optional result variable.

1. Add Call Script. Choose the script ID.
2. Leave Function blank for its default entry, or enter an exact function name such as describe.
3. Enter Arguments as a JSON array, for example [7,"Workshop reward"].
4. Set Store Result in Variable to function_result if the caller needs the answer.
5. Add Return in the called visual function, or return a value from the text function.
6. Use the destination variable after the call completes.

Return ends the current graph invocation and sends one value back. That value can be a number, flag, string, array or dictionary. It does not continue through another wire after returning.

Use Call Script Function when you want a friendly exposed function with named, typed parameter controls. Use Call Script for a lower-level explicit function name and positional argument array. Run Visual Script is convenient for calling a default script without storing its return.

> A direct Call Script does not use an exposed definition to choose a visual function’s entry. For a multi-entry visual function with named parameters, use Call Script Function. Explicitly requesting a nonexistent text function does not silently fall back to run.

# 35. Use a script-owned shared variable

Get Included Variable and Set Included Variable address a separate shared scope by script ID and variable name. They are useful when an included-script workflow needs state under that script’s identity.

| Node | Example fields |
| --- | --- |
| Set Included Variable | Script gd_tools; Shared Variable shared_counter; Value 9. |
| Get Included Variable | Script gd_tools; Shared Variable shared_counter; Store in Variable result_value. |
| Return | {"$var":"result_value"} reads the copied result. |

1. Choose an existing project script in Script.
2. Set the same Shared Variable name on the setter and getter.
3. Run the setter, then the getter.
4. Inspect the destination runtime variable.
5. Use a project variable instead when this is ordinary puzzle state that should be obvious in Variable Designer.

This scope is not an automatic window into every native GDScript member or Lua table field. It is the engine’s included-variable storage. The script ID has to be known; a missing source or never-written name can yield no value.

> Keep the scope clear in your notes. Project variable shared_counter and an included variable called shared_counter are different locations, even though the names match.

# 36. End a path, return, or stop a caller

| Node | Effect |
|---|---|
| End | Stops this graph invocation. |
| Return | Stops this invocation and returns its Value. |
| End Visual Script | Requests that the current visual call stops. |
| End Visual Script + Also End Calling Script | Also aborts the calling visual level, where one exists. |
| Unconnected chosen output | The current route simply runs out. |

Ordinary End in a called helper lets its caller continue after the call. Return does the same while providing a value. Also End Calling Script is a stronger instruction: work after that nested call may be skipped.

1. Use End for a normal completed path.
2. Use Return when the caller needs an answer.
3. Use End Visual Script only when stopping the running visual chain is actually intended.
4. Before using Also End Calling Script, inspect the caller’s cleanup.
5. Test both an ordinary finish and an early cancellation.

> An early end can bypass the node that re-enables input, hides a temporary GUI or stops an effect. Place cleanup before the exit or route every normal outcome through a shared cleanup section.

# 37. Decide what should become a function

A function is a named piece of reusable work. Parameters are the values a caller supplies: how many points, which object, whether an effect is enabled, or which text to use. A return value is the answer sent back.

| Repeated work | Useful parameters / result |
|---|---|
| Award points | amount → amount awarded. |
| Show or hide a target | target entity and visible flag → requested visibility. |
| Choose a response | topic or state → selected reply ID. |
| Configure an effect | target and strength → success/failure. |
| Test a puzzle requirement | item or condition → Boolean result. |

Use a function when several events need the same behavior and should stay consistent. Copying six nodes into ten puzzles makes later fixes tedious. One well-named function can keep that shared work in one place.

Keep a function’s job narrow. Award Workshop Points should award points and return its amount. It should not quietly change the room, replace the GUI and start a dialogue unless that larger sequence is what its name promises.

> Functions as Nodes is a definition system over real code or a real visual entry. The definition describes the work; it does not invent the function body.

# 38. Build a reusable visual function

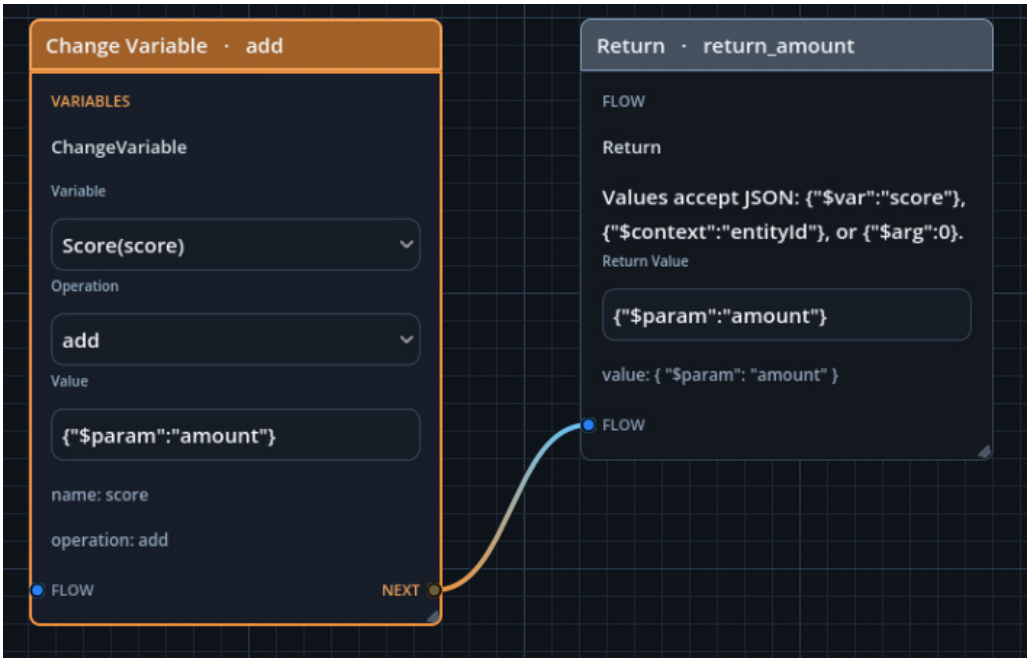


Figure 25. The Award function starts at Change Variable and finishes with Return.

1. Create a Visual Script named Reusable Visual Tools, ID visual_tools.
2. Add Change Variable. Choose score, Operation add, Value {"$param":"amount"}.
3. Add Return with Value {"$param":"amount"}.
4. Connect Change Variable NEXT to Return FLOW.
5. Open Functions as Nodes. Add a function named award, display name Award Workshop Points.
6. Choose the Change Variable node as its entry. Add amount as an int parameter with default 5. Keep Expose checked, then press Use Definitions.

A visual function can begin at an ordinary action node. It does not need an On Interact entry in front of it. The exposed definition supplies the starting node when called.

The amount expression is resolved from the named function parameters. If you run this graph directly without invoking its function definition, that named parameter has not been supplied. Test it through Call Script Function.

> Keep the node ID used by the definition valid. Moving or resizing that node is safe. Deleting it requires choosing a replacement entry in the definition.

# 39. Fill in definitions and use them

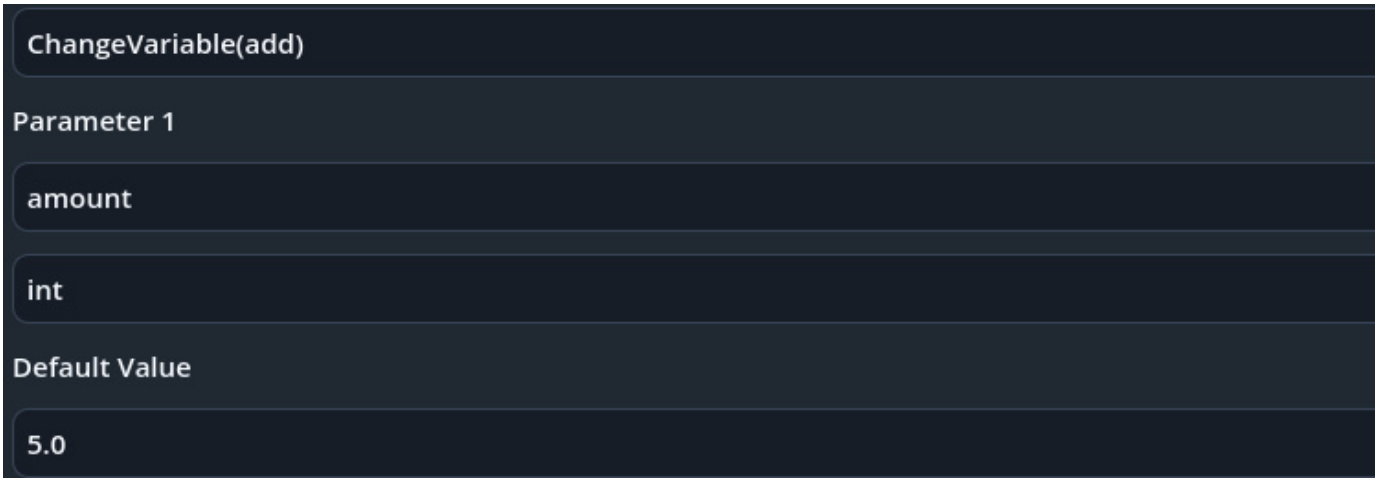


Figure 26. Choose the entry, then declare amount as an int with a default value of five.

| Control | What to enter |
|---|---|
| Function Name | award: a valid identifier, unique within this script. |
| Display Name | Award Workshop Points: the friendly name in pickers and a selected function-call node. |
| Description | A short statement of what the function changes or returns. |
| Entry node | The first action of this visual function. |
| Parameter | amount, type int, default 5. |
| Expose as Action and Visual Node | Checked if authors should be able to choose it. |

Use Definitions accepts the valid definitions and closes the window. It does not run the function and does not place a node on the canvas. In the Visual Script editor the definitions are written to the script record through the normal edit signal; save the project afterward.

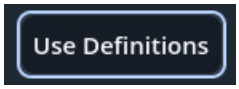


Figure 27. Accept the edited definitions.

If the window stays open, read its validation message and scroll as needed. A missing visual entry, duplicate/invalid function name, repeated parameter name or wrongly typed default prevents acceptance. A text-script editor may also require its own Apply Changes step.

# 40. Call the function with named fields

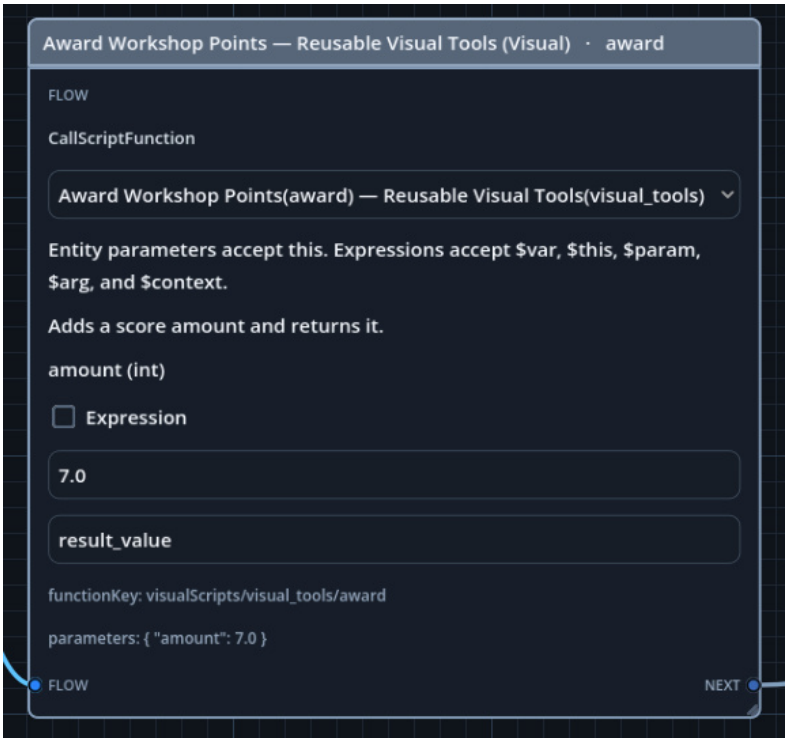


Figure 28. Call Script Function shows the chosen function’s named parameters and optional Result Variable.

1. In a separate caller graph, add On Interact with Use.
2. Add Call Script Function. Choose Award Workshop Points from Reusable Visual Tools.
3. Leave Expression off for amount and enter 7.
4. Enter result_value in Result Variable, then leave the field to commit it.
5. Connect On Interact to the function call, then to If Compare.
6. Compare {"$var":"result_value"} == 7 and show a success response on TRUE.

The selected function’s friendly name becomes the call node’s title. You still add the generic Call Script Function action from the palette; exposure does not create a separate permanent palette button for every project function.

The call runs its body, waits for it to finish, stores the returned value when requested, then follows NEXT. This example adds seven to score and stores 7 in result_value.

> The function picker distinguishes script records and their language tables. Similar display names do not mean the functions are interchangeable. Check the script as well as the function name.

# 41. Pass a live value with Expression




Figure 29. Expression lets a function parameter read a runtime value instead of keeping a fixed literal.

1. Select the Call Script Function node.
2. For amount, turn on Expression.
3. Replace the initial expression with {"$var":"pulse_count"}.
4. Commit the field and save.
5. Set pulse_count to 3 through the loop example, then call the function.
6. The call should add three points and return 3.

The expression must resolve to a value accepted by the declared parameter type. An int parameter needs an integer-valued number. Text containing "three" is not a valid substitute.

| Parameter choice | Example |
|---|---|
| Fixed literal | amount = 7 with Expression off. |
| Project/runtime variable | {"$var":"pulse_count"} |
| Caller name for a string parameter | {"$this":"name"} |
| Outer function argument | {"$param":"amount"} |
| Event field | {"$context":"entityId"} |
| Positional argument | {"$arg":0} |

Turning Expression on initially creates a caller-ID expression. Replace it with the expression your parameter actually needs. Turning it off restores the parameter’s default rather than preserving the expression as a literal.

# 42. Use types and return values deliberately

| Type | Suitable value / default |
|---|---|
| string | Ordinary text, such as Workshop reward. |
| int | A whole number, such as 7. |
| float | A finite number, such as 0.75. |
| bool | Boolean true or false; the caller form provides a checkbox. |
| entity | A record ID as text. The special value this resolves to the captured caller. |
| variant | A JSON-compatible value, including an array, dictionary or null. |

Set the parameter type before entering its default. Changing the type resets that default to the new type’s initial value. In the definition window, numeric, Boolean and variant defaults use JSON values; string and entity defaults use ordinary text.

1. Decide which inputs the function genuinely needs.
2. Give each parameter a valid, unique name.
3. Choose its type, then enter a sensible default.
4. Use Return for a deliberate result.
5. Store the result only when the caller will use it.
6. Check that any later comparison expects the actual return type.

A destination runtime variable can receive a compound result. For example, describe returns a dictionary containing amount and label. Do not compare that whole dictionary directly with the number 7 and expect it to match.

> An entity parameter is an ID, not a preloaded Godot node. Resolve or pass that ID through PCA’s record/action APIs. Its spelling and availability still matter.

# 43. Keep several visual functions in one record

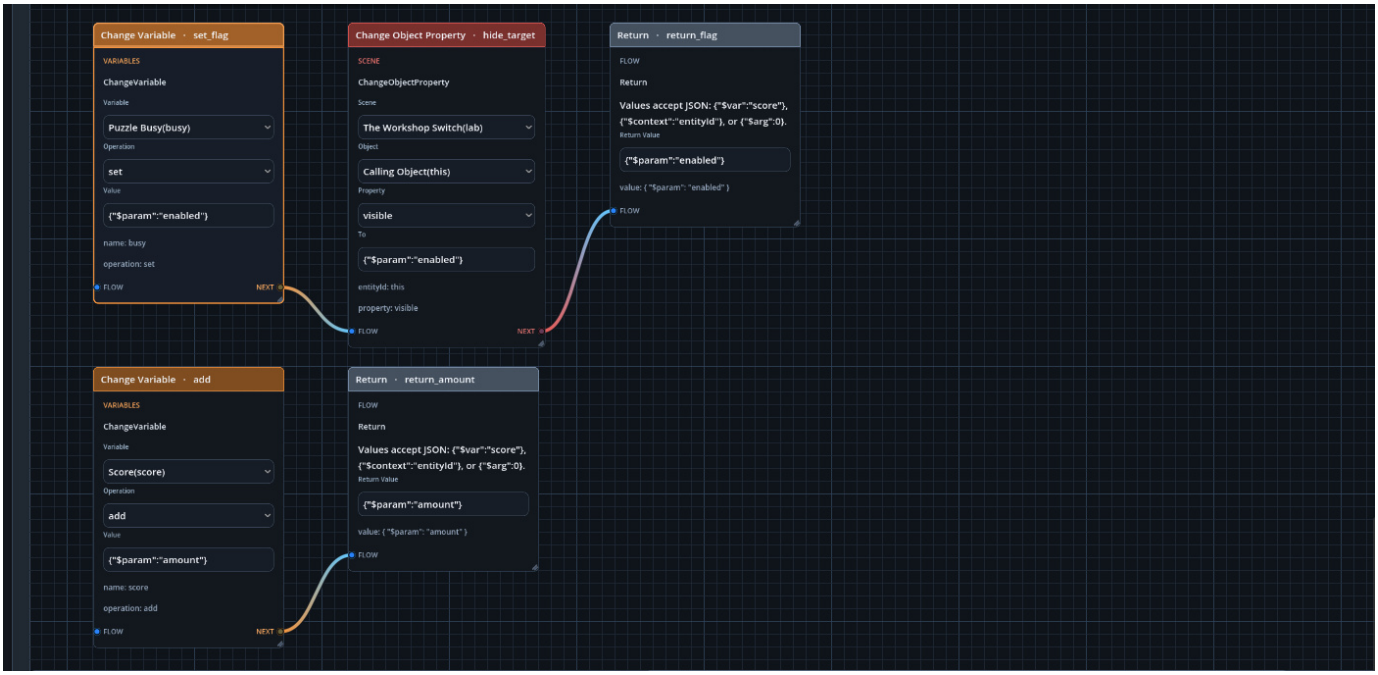


Figure 30. A utility graph can contain separate function bodies with separate entry definitions.

One Visual Script record can contain several disconnected bodies. Each exposed function chooses its own first node. That lets a small utility record hold related operations without making every call start at the same entry.

1. Build the Award body as in lesson 38.
2. Build another body, such as a caller visibility operation using a Boolean enabled parameter.
3. Add a second definition with a different Function Name and the correct entry node.
4. Use {"$param":"enabled"} in a JSON-capable value field in that body.
5. End each body with its intended Return or End.
6. Test each function from a separate Call Script Function node.

Place each function body in its own area of the canvas. Keep the entry and return close enough to follow. A disconnected action in this situation is intentional because a function definition can start there.

A default run chooses the first On-prefixed node, or the first node if none exists. It does not run every disconnected body. For On Event helpers, use an exposed function: a generic default call can inherit the original event entry ID in this build.

# 44. Expose a GDScript helper

A small text helper can perform a calculation or an operation that is awkward to express in a graph. The graph then gives the rest of the author a friendly parameter form.

```
extends PCAScriptBase

func describe(amount, label):
    return {"amount": amount, "label": label}
```

1. Create a GDScript record, for example gd_tools. Enter the code above.
2. Open Functions as Nodes in that script editor. Add describe.
3. Give it a friendly name such as Describe a Reward.
4. Add amount as int with default 7, then label as string with default Workshop reward. Keep the order aligned with the code.
5. Use Definitions, apply the script changes and save.
6. Choose that function in a visual Call Script Function node and store its return.

The function name must match the source exactly. The display name can be friendlier. The definition does not change the code signature when you add or reorder parameters.

> Use PCAScriptBase for PCA helpers and caller context. A custom extends declaration must inherit that base to use those facilities. Native Godot self continues to refer to the script object.

# 45. Expose the same function in PCA Script

```
language 1.0
function describe(amount, label)
    return {"amount": amount, "label": label}
end
```

1. Create a PCA Script record, for example pca_tools.
2. Enter the function above. Keep each dictionary expression on its physical source line as shown.
3. Add a definition named describe. Use amount:int default 7 and label:string default Workshop reward, in that order.
4. Use Definitions, apply the editor’s changes and save.
5. Call the PCA Script version from Call Script Function.
6. Check the returned amount and label against the GDScript version.

The visual caller need not change its design because the helper language changed. It still selects a function and supplies typed fields. The implementation remains in the chosen language’s script record.

PCA Script’s return dictionary is useful for a structured answer. For a simpler function, return a number or Boolean and compare that directly in the graph.

Use distinct script IDs such as gd_tools and pca_tools. Changing the language selection is not a translation operation; each script must contain its own valid implementation.

# 46. Call Lua and BASIC functions too

The same definition and caller workflow is available for Lua and BASIC. These two examples return the same amount/label dictionary.

```
local Script = {}
function Script.describe(amount, label)
    return {amount = amount, label = label}
end
return Script
```

```
FUNCTION describe(amount, label)
    RETURN {"amount": amount, "label": label}
END FUNCTION
```

1. Create a Lua record lua_tools or BASIC record basic_tools and enter the matching source.
2. Expose describe with amount:int and label:string in that order.
3. Accept the definitions, apply the script editor changes and save.
4. Add or select Call Script Function in the visual caller.
5. Choose the intended language’s helper and verify the result.

The Lua example returns its Script table. The BASIC example uses FUNCTION and END FUNCTION. Neither example needs a default run function because it is called by the named function.

> Language syntax still matters inside each helper. Exposure makes the call form consistent; it does not make Lua table syntax interchangeable with JSON or GDScript.

# 47. Fix a missing or invalid function call

| Symptom | Check first |
|---|---|
| Use Definitions closes; no new node appears | Expected behavior. Add Call Script Function and choose the exposed definition. |
| The definition dialog stays open | Read its validation message; check names, types, defaults and visual entry. |
| Function is missing from the picker | Expose is checked, definition accepted, record applied where required, project saved and correct script selected. |
| Missing Function appears | The stored definition key no longer resolves. Check renames, deletion, exposure and script table. |
| Call returns no useful value | The body reaches Return; the right entry/function ran; Result Variable was committed. |
| Type diagnostic | The supplied literal/expression resolves to the declared type. |
| Old calls stop after exposure is disabled | Re-enable the definition or update those calls. Direct code calls are a separate route. |

Selecting a different function resets the caller’s supplied parameter values. Recheck every field after changing the picker. A reused friendly name does not guarantee identical parameter order or meaning.

When a visual entry node has been deleted, choose an existing replacement in Functions as Nodes. Do not leave a stale entry and hope the runner chooses a nearby node.

> Before distributing the project, test each exposed function with its defaults and with at least one non-default value. Test a caller-dependent helper from a real owner event.

# 48. Listen with On Event

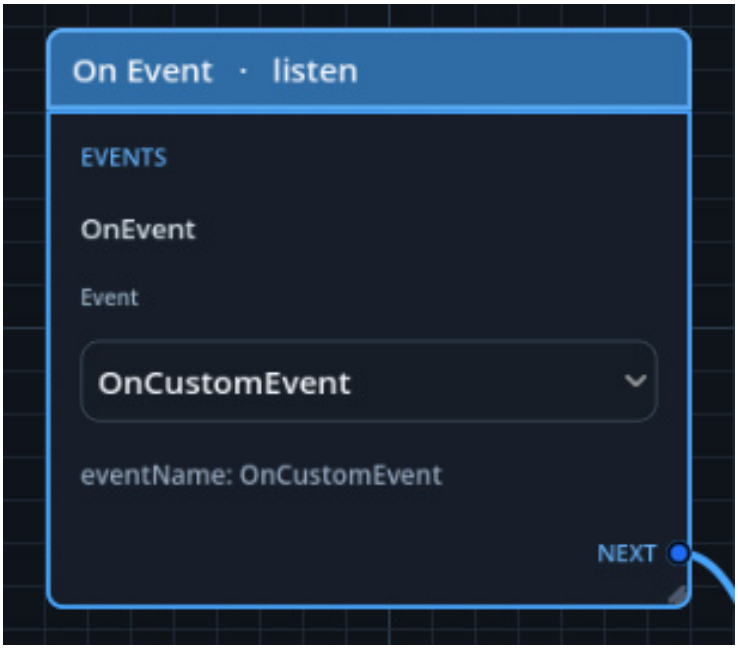


Figure 31. On Event selects an engine event. The dispatcher starts that entry when the event is emitted.

1. Create a separate listener graph, such as custom_listener.
2. Add On Event. Choose OnCustomEvent for this first example.
3. Connect it to a simple condition or variable update.
4. Save and restart the preview so it contains the new listener.
5. Cause the real event, or use Emit Event from another graph.

On Event entries are found across the project’s Visual Scripts. They are different from an On Interact graph that must be assigned to a particular object. A listener can react without being assigned to the object’s own verb action.

Several event entries can live in one graph. The dispatcher supplies the matching entry node ID. A default manual run does not test every entry, and it does not reproduce every event’s context.

> Do not put duplicate reward logic in both a global OnInteraction listener and the object’s own Use action. Both can be reached for one player interaction. Give one system responsibility for the reward.

# 49. Send a custom event with useful data

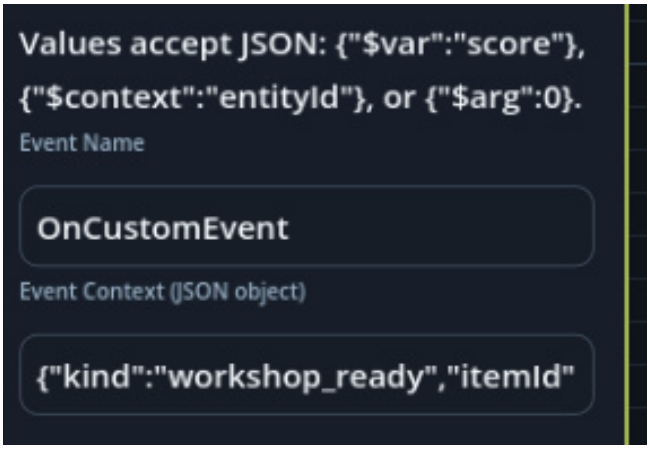


Figure 32. Emit Event sends an event name and a JSON context object.

1. In the sender graph, add Emit Event.
2. Set Event Name to OnCustomEvent.
3. Set Event Context to {"kind":"workshop_ready","itemId":"brass_key_item"}.
4. In the listener, compare {"$context":"kind"} == "workshop_ready".
5. On TRUE, use Read Context to copy itemId into last_item.
6. Continue to a GUI update or other small response.

Using a kind field lets several custom messages share OnCustomEvent while keeping their meanings separate. For example, workshop_ready, door_unlocked and hint_requested can each have a distinct filter.

```
{
  "kind":"workshop_ready",
  "itemId":"brass_key_item",
  "sourceId":{"$this":"id"}
}
```

The sourceId field is optional. Use it only when the sender has a caller or an explicit source. A manually sent custom context does not automatically recreate every field of a normal object interaction.

> The On Event dropdown offers known event families. For an ordinary authoring workflow, use OnCustomEvent plus your own context tag rather than inventing a listener name that the picker cannot select.

# 50. Filter listeners and prevent event loops

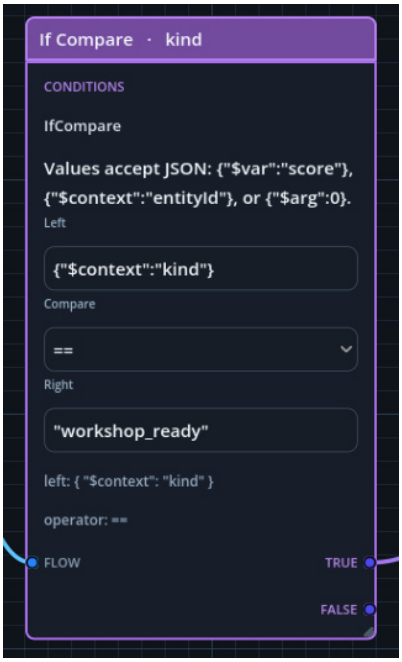


Figure 33. The listener checks its context tag before changing state.

1. Put the event filter immediately after On Event.
2. Leave the irrelevant branch unconnected, or end it explicitly.
3. Check a completion or busy flag before performing one-time work.
4. Set that guard before the first wait or outgoing event.
5. If the listener changes the system it listens to, test whether that change emits another event.
6. Confirm that repeated unrelated events leave the puzzle unchanged.

An OnInventoryChanged listener that adds another item can trigger another inventory notification. An OnCustomEvent listener that emits the same tag can call itself. A short guard is easier to understand than relying on a nesting or step limit to stop the chain.

A listener can use a variable gate, such as effects_enabled or hints_enabled, when its response should be optional. The Functions as Nodes exposure checkbox controls the function catalog; it is not an on/off switch for an On Event entry.

> Event timing can be deferred by the system that emits it. Do not assume every notification has already run immediately after an inventory action. Test the resulting behavior through the normal game flow.

# 51. Start work on room entry or game startup

| Route | Use it for |
| --- | --- |
| Room First Enter action list | One-time introduction to that room. |
| Room Enter action list | Work required whenever that room is entered. |
| On Event: OnSceneLoaded | A listener that can inspect sceneId, previousSceneId and firstVisit. |
| On Event: OnGameStart | A game-wide startup response. |
| Project startup Visual Script | A selected opening flow before normal player control, when that startup mode is intended. |

1. Choose one place for each startup responsibility.
2. For OnSceneLoaded, filter {"$context":"sceneId"} against your room ID.
3. Use {"$context":"firstVisit"} when the distinction matters.
4. For a room action list, add Run Visual Script or Call Script Function as appropriate.
5. Keep initial puzzle values in Variable Designer when they must exist before room-entry work.
6. Test a new game and a later room return separately.

Room loading and game startup are different events. Do not assume an OnGameStart initializer will run before every room-entry action that needs a value. Avoid resetting solved puzzle flags on every room load.

> Room and startup calls may have no object owner. A helper that targets this must be called from an appropriate owner event or be rewritten to take an explicit target.

# 52. Respond when video or music ends

| Entry | Filter / supplied field |
| --- | --- |
| On Video End | Optional video asset ID; compares videoAssetId. |
| On Music End | Optional audio asset ID; compares musicAssetId. |
| Blank asset filter | Accepts completion from any matching media source that reaches this entry. |

1. Create the media playback in the relevant room, GUI or cutscene.
2. Add the appropriate end entry to a Visual Script.
3. Choose the exact asset when only that media should trigger the response.
4. Connect the entry to the small follow-up action: show a GUI, move on or change a flag.
5. Run the media to completion in Play.
6. Test any skip/stop route separately; do not assume it reports the same completion as natural playback.

These entry nodes provide a clean way to keep work after the actual media end instead of guessing its duration with Wait. They can coexist with other entries in a graph, but each route still needs its own intended connection.

> Run Script supplies an editor Use context, not a finished videoAssetId or musicAssetId. A filtered media-end entry can correctly refuse that manual test. Test real playback.

# 53. Call a graph from a GUI button




Figure 34. The button’s Click event calls the selected graph. This screenshot uses the run-visual-script behavior.

1. Open GUI Designer and select the Pulse Beacon button.
2. Open Events. Find the Click event.
3. Choose run-visual-script and select gui_button as its target. The shared Run Visual Script action is another available route.
4. In gui_button, connect an On Interact: use entry to Run Visual Script: pulse_three.
5. Apply/save the GUI and graph, then start Play.
6. Click the runtime button. The beacon should pulse three times.

The event supplies guiId, guiElementId and the GUI event name. The captured owner is the GUI. If a node needs the current element, the special elementId value this can use guiElementId; an object target still needs a real scene object ID.

The visual-only GUI Designer preview is for appearance and control states. Test gameplay event execution in Play. A hover image working in the designer is not proof that a script was called.

> Keep click, press/release and hover responsibilities separate. Repeated hover events should not award inventory or start the same long sequence over and over.

# 54. Update HUD text and use wildcards

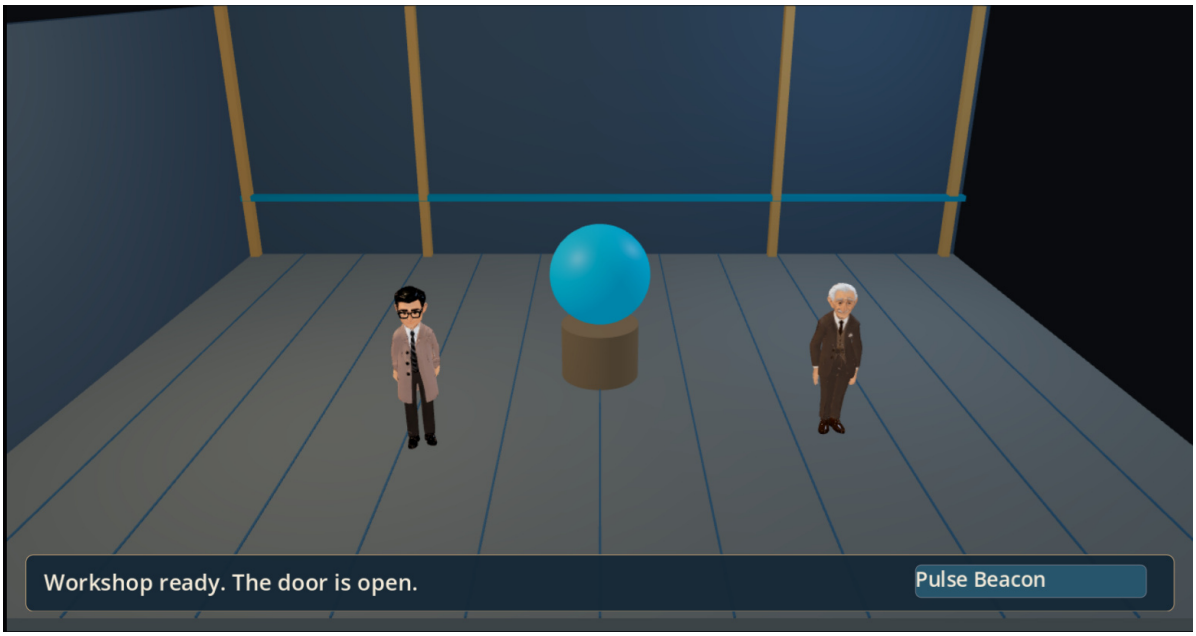


Figure 35. The custom-event listener updates the HUD label while the game is running.

1. Add a label to scene_hud and give it a stable ID such as hud_text.
2. For a score display, enter WORKSHOP | Score: $$score in its text.
3. Use Add Score or Change Variable to change score during play.
4. For an explicit message, add Change GUI Element Property. Choose scene_hud, element hud_text, property text and the new text value.
5. Call it from the filtered custom-event listener.
6. Check the runtime label after the event.

A wildcard in an element’s displayed text is different from a graph expression in a JSON field. $$score asks the text system to display that variable. {"$var":"score"} asks the graph to resolve a value for an action or comparison.

Use the GUI and element dropdowns to avoid addressing the wrong label. An element ID is local to the authored GUI workflow; the containing GUI reference matters when selecting a target.

> Do not paste the literal characters $$score into a numeric action parameter and expect a number. Use the value-expression syntax for that parameter.

# 55. Walk to an NPC, face them and speak

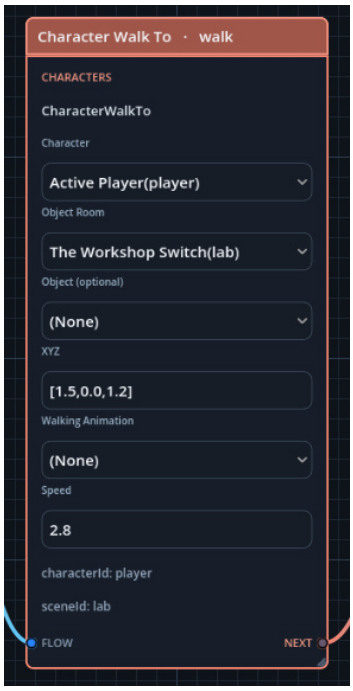


Figure 36. Character Walk To can use an explicit position or a selected target in the room.

| Order | Node / settings |
|---|---|
| 1 | On Interact: talk, assigned to the NPC’s Talk response. |
| 2 | Enable / Disable Player Input: false, if this sequence must control the player. |
| 3 | Character Walk To: player; room lab; XYZ [1.5,0,1.2]; a suitable speed such as 2.8. Leave Object blank for explicit coordinates. |
| 4 | Face Character: Character player; Face Character elias. |
| 5 | Say: Speaker elias; the instruction about the brass key. |
| 6 | Enable / Disable Player Input: true. |

Choose a destination beside the NPC, not the NPC’s exact center. Characters need room for their collision and stopping distance. Confirm that the destination is inside reachable navigation.

The normal interaction system may already manage an input lock for its action chain. Add an explicit input toggle only when your sequence needs that policy, and restore it on every path that uses it.

> A walk that cannot complete can hold the sequence before the speech. Test the movement by itself first, then add facing, dialogue and cleanup.

# 56. Add animation, speech, voice and typewriting

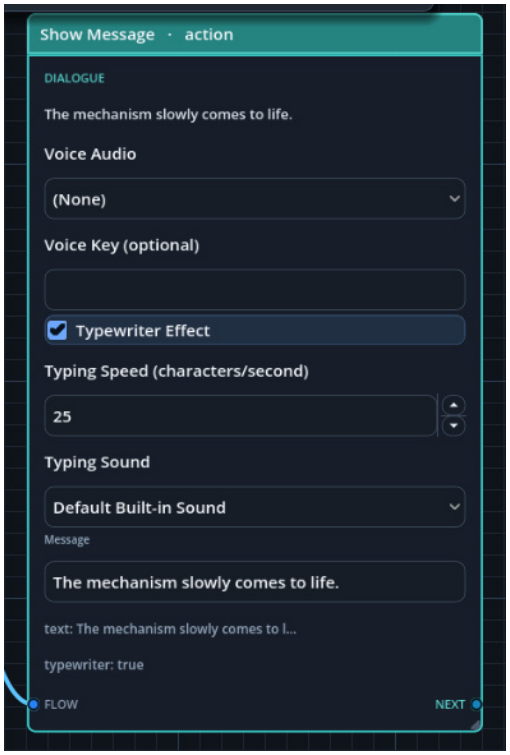


Figure 37. Typewriter Effect exposes speed and sound choices inside a message node.

1. Give the character a working animated model and named animation states in Character Designer.
2. Use Play Animation with the correct Room, Entity and animation/state selection. Test a known state such as talk first.
3. Add Say for character speech, or Show Message for a narrator/system line.
4. Choose Speaker for Say. Enter the text and optional Voice Audio or Voice Key.
5. Enable Typewriter Effect if required. Choose a readable speed, then default sound, no sound or a custom sound asset.
6. Test the line, timing, mouth movement and return to the normal state in Play.

The animation action starts a valid animation; it does not manufacture one in the model. The speech system can use the character’s configured talk animation and lip-sync settings. A voice asset alone does not create mouth shapes or portrait mappings.

A typewriter effect reveals the line over time. Its sound choice controls typing feedback, which is separate from a spoken voice recording. Use a quiet blip so it does not overpower the voice.

> Do not assume Play Animation waits for the entire clip. If later work must happen after a gesture, use a deliberate timing/completion design or a cutscene that coordinates those actions.

# 57. Start a branching conversation

Use Dialogue Designer for lines, choices, revisiting topics and conversation flow. Use the Visual Script graph to decide when that conversation should start and what follows it.

1. Create and test a dialogue record with a valid start node.
2. In the NPC’s Talk graph, do the approach/facing work first.
3. Add Start Dialogue and choose that dialogue ID.
4. Connect the continuation to the response that should happen after the dialogue finishes.
5. Put choice-specific item or variable changes inside the relevant dialogue actions or a called helper.
6. Test each choice, its return path and the end of the conversation.

| Task | Best place |
| --- | --- |
| May this conversation start? | A visual condition before Start Dialogue. |
| Which answer did the player choose? | The dialogue’s choice flow and actions. |
| One shared reward operation | An exposed helper called by the appropriate dialogue branch. |
| Work after the whole conversation | The continuation after Start Dialogue, or a filtered OnDialogueEnded listener when deliberately centralized. |

The current Start Dialogue call awaits the dialogue run. A waiting line or choice can keep it open until the player responds. End Dialogue cancels the active conversation; plan what should happen after cancellation as well as normal completion.

> Avoid awarding the same item both inside the dialogue’s final branch and again after Start Dialogue. Put the reward in one clearly guarded place.

# 58. Start and finish a cutscene from a graph

1. Build and test the cutscene in Cutscene Designer. Check its camera, duration, input lock, HUD policy and skip behavior.
2. Add a completion flag such as machine_intro_seen.
3. Before Start Cutscene, test that flag and any puzzle requirements.
4. Set the flag at the point appropriate to your replay/skip policy.
5. Add Start Cutscene and choose the cutscene record.
6. Continue to the final game state and cleanup after the call finishes.

The current Start Cutscene action awaits the cutscene runner. The runner’s timeline can launch overlapping timed actions, while the visual graph remains a useful way to control the larger before-and-after sequence.

| Question | Decide before wiring |
| --- | --- |
| Can it replay? | Use a completion flag if it should only happen once. |
| What if the player skips? | The puzzle still needs a sensible final state. |
| Who restores the camera/HUD? | Use the cutscene restoration settings and avoid conflicting duplicate cleanup. |
| Does a graph also lock input? | Make sure every extra lock is restored. |

A cutscene that shows a door opening and a graph that records door_open are related, but not automatically the same operation. Make the logical final state explicit and test skipping.

# 59. Choose and blend cameras

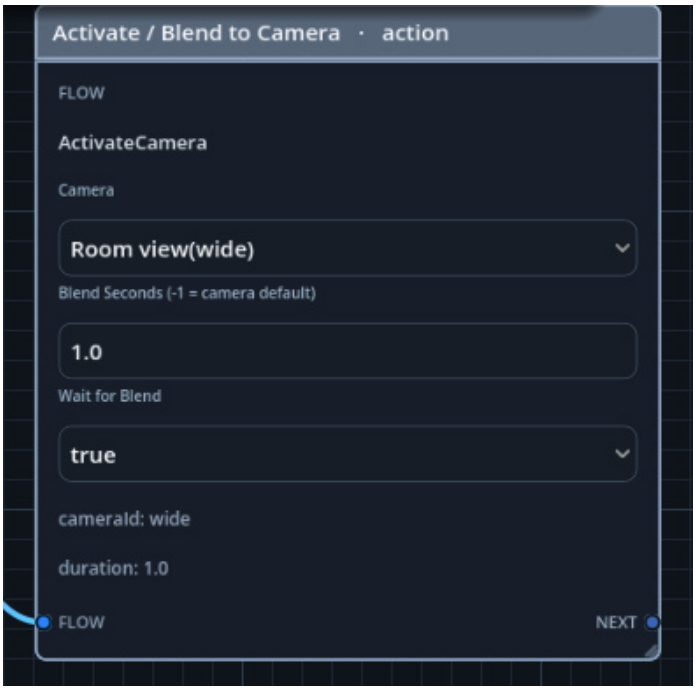


Figure 38. Activate / Blend to Camera has a blend duration and Wait for Blend option.

1. Create the required room cameras and give them readable names and stable IDs.
2. For an ordinary room-view change, add Change Camera and choose its Room and Camera View.
3. For the newer camera system, choose Activate / Blend to Camera. Select the intended camera.
4. Set Blend Seconds, for example 1.0. Use -1 when the camera’s own default is intended.
5. Choose Wait for Blend when the next action should begin after the transition.
6. Test the camera from the same room, player position and aspect ratio as the real interaction.

Other shared actions can set camera behavior, follow targets and priority. Choose a camera first and use its named settings. A following camera and an explicit change to another view can compete if their activation rules are left unclear.

A 2D Stage has a locked stage-camera workflow. Do not assume that an arbitrary 3D camera move is the right way to scroll it. Use the room’s stage camera settings and the action appropriate to that room mode.

> Camera motion is visible work. A node can start a blend and continue immediately if its wait option is off. Add a guessed Wait only when it is genuinely part of the intended timing.

# 60. Handle inventories for different characters

| Node | Purpose |
| --- | --- |
| Add Inventory / Remove Inventory | Change the active inventory used by the ordinary pickup workflow. |
| Has Inventory | Checks the active inventory for a positive quantity of the chosen item. |
| Change Inventory Quantity | Choose the Player / Character, item and signed change. |
| Transfer Inventory | Choose From Player, To Player, item and quantity. |
| Switch Character | Changes the active player, subject to the project’s character setup. |

1. Decide whether the game uses a shared inventory or separate character inventories.
2. For a normal player pickup, use Add Inventory with a known definition.
3. For an explicit recipient, choose the character-targeted quantity/give/transfer action that matches the job.
4. After switching characters, inspect the active inventory and associated HUD.
5. Test a transfer with sufficient and insufficient quantities.
6. Recheck any Has Inventory branch after switching the active player.

A variable saying key_taken is not the inventory itself. Keep the distinction useful: the flag prevents a second reward, while the inventory determines who currently holds the key.

> If the key can be transferred or consumed, testing only key_taken to unlock a door can be wrong. Test actual possession when possession is the requirement.

# 61. Tie actions to quests and objectives

1. Create the quest, its stages and objectives in Quest Designer.
2. Choose Start Quest when the player learns about the task. Select the optional initial stage only when needed.
3. On the successful pickup, use Quest: Complete Objective with the exact quest, stage and objective.
4. Use Quest: Set Stage when the story reaches the next stage.
5. Use Complete Quest only when the intended requirements are satisfied.
6. Check the journal and quest display after each change.

| Situation | Useful action |
| --- | --- |
| The quest should become known | Start Quest / Set Quest Hidden. |
| The player completes one task | Complete Objective. |
| An attempt fails | Fail Objective or another guarded story response. |
| The story advances | Set Quest Stage. |
| The whole task is resolved | Complete Quest. |
| The player wants to follow it | Set Quest Tracked. |

The graph controls the quest actions; the quest definition describes their meaning and display. Pick the quest first, then its stage and objective so the dependent selectors use the correct records.

> Some quests have automatic rules tied to inventory or events. Check those before adding a duplicate manual completion action. The visible quest update should have one clear cause.

# 62. Attach and control a particle effect

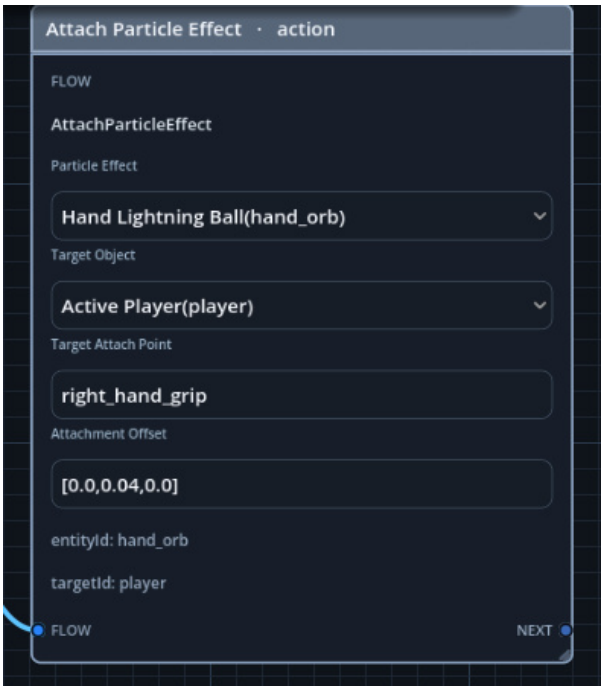


Figure 39. Attach Particle Effect targets an existing emitter and a named point on its parent.

1. Author a Particle object such as hand_orb, initially with emission off.
2. Create and test right_hand_grip on the character; bind it to the actual hand bone if it must follow animation.
3. Add Attach Particle Effect: effect hand_orb, target player, point right_hand_grip, offset [0,0.04,0].
4. Use Change Object Property: hand_orb, particle.enabled, Boolean true.
5. When the effect ends, set particle.enabled false and hide it if immediate disappearance is required.
6. Use Detach Particle Effect when it should stop following the target.

The action links an existing emitter. It does not create a particle record, invent a missing attach point, add damage or play a sound. Put those responsibilities in explicit actions when needed.

Use Change Game Setting or the relevant scene/property action for screen effects. A surface effect can travel with an attached object; a whole-screen post-process is part of the view.

> Detaching preserves the current world pose and does not disable emission. A missing named point should be corrected on the intended target rather than replaced with a guessed offset.

# 63. Change rooms and choose a spawn

1. Create and test the destination room. Add a clearly named spawn such as from_workshop.
2. In the exit graph, test the requirement: door_open true, for example.
3. Add Change Scene. Choose the destination Scene first.
4. Choose Spawn (optional) from that scene. Use XYZ when no spawn only when a coordinate-based arrival is intended.
5. Commit/save, then walk through the actual exit interaction.
6. Return to the old room and check its solved state.

The Scene and Spawn selectors are references. A spawn ID from another room is not a valid arrival simply because the text looks familiar. Dependent dropdowns should be selected after the parent room choice.

Treat the room change as a boundary. Persistent puzzle variables and supported saved state can survive it; the old room’s live nodes may no longer be available for later object operations.

| Before the change | After the change |
| --- | --- |
| Record completed puzzle state. | Use destination room entry logic for its local presentation. |
| Stop a temporary effect if needed. | Choose targets that actually exist in the new room. |
| Avoid duplicate exit calls. | Check that control and the default camera are correct. |

Do not continue targeting an old door through this after changing rooms. Capture only the information you need and let the destination room own its local setup.

# 64. Save and load the puzzle

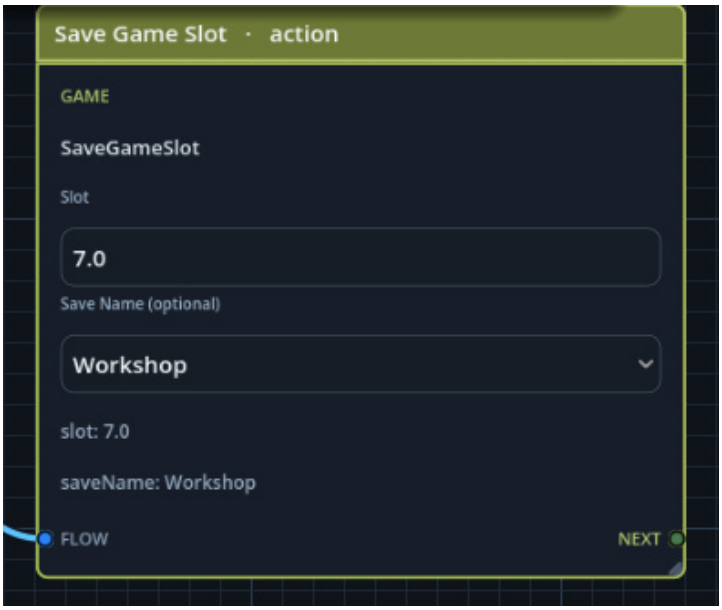


Figure 40. Save Game Slot chooses a numeric slot and optional display name.

1. Reach a clear test state: key collected, door_open true and score 15.
2. Call Save Game Slot with a known test slot and name.
3. Change the puzzle state or begin a different test.
4. Call Load Game Slot for the saved slot.
5. Confirm the variables, inventory and supported room/object state match the saved point.
6. Test the same workflow through the actual save/load GUI used by the game.

A completed graph call can store its resulting game state. A save is not a promise that an arbitrary suspended visual call will resume halfway through its wire path with a live timer and local loop counter.

Use stable state flags so room-entry presentation can reconstruct a solved or unsolved puzzle when appropriate. Avoid saving while a delicate interaction has half-finished unless you have designed and tested that situation.

> Save Project preserves authoring work. Save Game Slot preserves a running game’s supported state. They are separate controls and should be tested separately.

# 65. Test a graph in manageable pieces

1. Start with the smallest visible result: a message, changed label or returned value.
2. Add one condition. Test TRUE and FALSE deliberately.
3. Add the state change or reward. Run the interaction twice.
4. Add timing, animation and presentation after the logic works.
5. Test the real owner event and selected verb, not only Run Script.
6. Restart, leave/re-enter, save/load and test the completed flow again.

| Check | Expected workshop result |
|---|---|
| Use door without key | Locked response; no points; door_open false. |
| Pick up key | One item, five points, hidden scene key. |
| Call pickup again | No additional item or points. |
| Use door with key | door_open true; score 15; opening clear. |
| Run three pulses | pulse_count 3; final loop_index 2. |
| Call Award with amount 7 | Score increases by 7; result variable 7. |
| Use two this switches | Each call changes its own owner. |
| Load the saved solved state | Door flag, score and inventory are restored as intended. |

Use Validate for record IDs, references and structural problems. The current visual validator checks node IDs and edge endpoints, among other project checks. It cannot prove that the right branch awards the right item or that two wires from one output express the intended sequence.

> Use the game’s debug/output information for runtime diagnostics. This Visual Script editor is not a node-by-node breakpoint debugger. Small tests and visible state checks are the practical way to isolate a route.

# 66. Troubleshoot the symptom you can see

| Symptom | First useful check |
|---|---|
| Nothing happens | Right script selected, event assigned, entry verb matched, fresh preview started. |
| It works only under Run Script | The actual object/GUI/trigger event assignment and caller context. |
| Take graph fails under Run Script | The quick test supplies Use; perform a real Take interaction. |
| Only one of two NEXT wires runs | Chain the actions. The runner follows the first matching edge. |
| A branch is backwards | Read the TRUE/FALSE port at the source, then check the comparison value and type. |
| A loop runs once | Reconnect the last body node to the loop’s FLOW input. |
| A loop hits 4096 steps | Fix its changing condition or back-edge; do not rely on the guard as the exit. |
| this has no caller | Run through a real owner event or pass an explicit entity parameter. |
| A function is missing | Definition exposure, script ID/table, function name and visual entry. |
| A value remains old | Commit the focused field and restart the preview after authoring changes. |
| Player input stays disabled | An early exit or failed path skipped the restore action. |
| An object vanishes but puzzle fails | The item, flag, collision and later interaction state need consistent logic. |

Repair the earliest incorrect step. Adding more message nodes at the end does not help if the graph never reaches the condition that leads there.

# 67. Convert a copy to a text script




Figure 41. The generated GDScript copy embeds the graph data and calls RunVisualGraph.

1. Save the working Visual Script.
2. Choose Convert to GDScript or Convert to PCA Script in its toolbar.
3. Inspect the new script record. The original Visual Script is preserved.
4. Read any generated comments or conversion warnings.
5. Test the converted copy under the same caller, verb, arguments and state.
6. Reassign events explicitly only after you decide to use that copy.

The GDScript conversion preserves the JSON graph and runs it through RunVisualGraph. It is an editable native wrapper; it is not a hand-rewritten series of ordinary GDScript if statements and loops.

The PCA Script conversion tries to make readable source. Graph-only operations, complex loops or unusual branching can require manual work. A generated script compiling successfully does not prove that every original route was translated as intended.

> Conversion is a copy operation, not live synchronization. Later changes to the original graph do not automatically update the converted source, and edits to the copy do not redraw the original graph.

# 68. Use Raw JSON for an omitted field

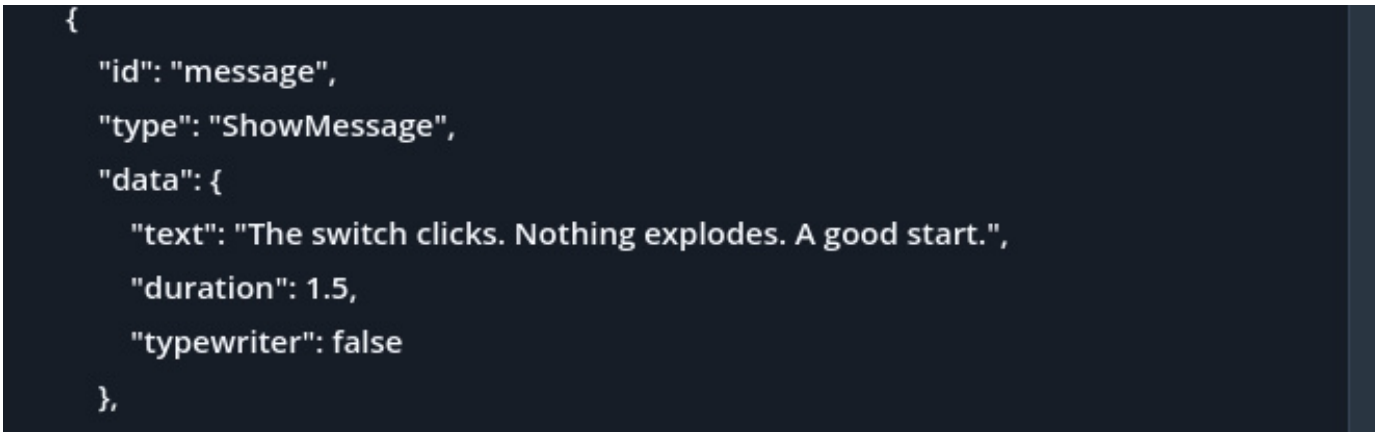


Figure 42. The ShowMessage data sits inside the whole script record’s nodes array.

1. Save a working version before editing structured data.
2. Select the Visual Script record and show Inspector.
3. Open Raw JSON (Advanced). Find the intended object inside nodes by its id and type.
4. Change only the required property in that node’s data dictionary. For example, ShowMessage duration can be 1.8.
5. Preserve the surrounding nodes, edges, IDs and layout fields. Press Validate and Apply JSON (Advanced).
6. Reopen the graph, verify the visible result and test it in a fresh preview.

```
"data": {
  "text":"The switch clicks.",
  "duration":1.8,
  "typewriter":false
}
```

This is an excerpt of a ShowMessage node’s data, not a complete script to paste over the record. Appendix F contains a complete small graph for reference. JSON needs matching braces and brackets, ordinary quotes and commas between entries.

> Clicking a graph node does not turn the whole-record Inspector into a dedicated node JSON editor in this build. Locate the correct node in the record rather than replacing the whole record with just its data object.

# 69. Keep a growing project understandable

Name scripts by the work they own: Take Brass Key, Unlock Workshop Door, Listen for Workshop Events, Reusable Visual Tools. Keep IDs short, stable and distinct across languages.

1. Put related scripts in database folders.
2. Keep one clear place for each puzzle reward and completion flag.
3. Use exposed functions for repeated behavior with changing inputs.
4. Keep listener graphs separate from one-off interaction graphs where that makes their automatic behavior easier to see.
5. Use a small written table of important flags, owners, inputs and results.
6. Before release, test the actual built game as well as the embedded preview.

| Final check | Why it matters |
| --- | --- |
| Fresh game and existing save | Defaults and restored state take different routes. |
| Keyboard, mouse and intended controller/touch input | A working graph still needs the real event to reach it. |
| Every branch and repeat interaction | The second use often reveals reward and cleanup mistakes. |
| Optional systems turned off | A feature gate must leave the game in a usable state. |
| Every exposed function | Defaults, non-default parameters, caller requirements and returns stay aligned. |
| No unused automatic listeners | An old On Event entry can still react when its event is emitted. |

Finish the behavior before adding polish. A simple, correctly guarded interaction is easier to extend than a large graph whose starting assumptions are unclear.

# Appendix A. Ports and execution reference

| Node family | Input / outputs | Behavior |
|---|---|---|
| On Interact | No input; NEXT | Starts a called interaction route and checks the verb. |
| On Event | No input; NEXT | Dispatcher chooses the matching event entry. |
| On Video End / On Music End | No input; NEXT | Asset-filtered completion entry. |
| Ordinary action | FLOW; NEXT | Executes, then follows the continuation. |
| Condition | FLOW; TRUE / FALSE | Chooses one named output. |
| For Loop | FLOW; LOOP / DONE | Revisit it from the body to count another iteration. |
| While Loop | FLOW; TRUE / FALSE | Revisit it after changing the tested state. |
| Return | FLOW; no output | Returns Value and ends this invocation. |
| End / End Visual Script | FLOW; no output | Ends the route or requested visual call level. |

One output should have one destination. Several incoming paths can share one action. A missing chosen connection ends that path. Node positions and sizes do not change these rules.

Default entry selection uses the first On-prefixed node found, or the first node when there is no such entry. Event dispatch and exposed visual function calls can supply an explicit entry. Use explicit function definitions for reusable multi-entry helpers.

> 4096 visited-node limit per graph; a frame yield every 64 visits; nesting capped at 24 graph levels. While Loop’s displayed Safety Limit is not a separately enforced per-loop cap in this build.

# Appendix B. Expressions and call data

| JSON | Meaning |
| --- | --- |
| {"$var":"score"} | Read runtime variable. |
| {"$context":"entityId"} | Read the exact event field. |
| {"$this":"id"} | Read captured caller ID. |
| {"$this":"name"} | Read caller property. |
| {"$param":"amount"} | Read named visual-function parameter. |
| {"$arg":0} | Read positional argument zero. |

These special one-key dictionaries can be nested in arrays or dictionaries. Ordinary strings stay strings. No arithmetic formula parser is implied by the lookup syntax.

```
{
  "type":"CallScriptFunction",
  "functionKey":"visualScripts/visual_tools/award",
  "parameters":{"amount":{"$var":"pulse_count"}},
  "resultVariable":"result_value"
}
```

That object is one action, suitable as action data in the corresponding context. It is not a whole project or graph. In an actual visual node, the type is on the node and the remaining fields belong in its data.

> An entity-typed function parameter containing this resolves to the owner. A string parameter containing the same word remains ordinary text. Type and target context matter.

# Appendix C. Exposed visual function data

Use Functions as Nodes for normal authoring. This reference shows the relationship between the definition and the two-node body described in lesson 38.

```
{
  "name":"award", "label":"Award Workshop Points",
  "enabled":true, "entryNodeId":"add",
  "parameters":[{"name":"amount","type":"int","default":5}]
}
```

| Body node | Data | Connection |
|---|---|---|
| add: ChangeVariable | name score; operation add; value {"$param":"amount"} | NEXT → result |
| result: Return | value {"$param":"amount"} | No output. |

The definition lives in that script record’s exposedFunctions array. entryNodeId must identify the actual first node. The definition name is the callable function name; the label is the friendly name.

A Call Script Function action selects the definition by its language table, script ID and function name. Use the picker instead of hand-constructing this key while learning.

Definitions are metadata. They do not rewrite a text function’s signature or invent a missing visual node. Disabling exposure hides the definition and makes existing exposed calls skip with a diagnostic until repaired.

# Appendix D. A target-aware text helper

This GDScript helper is useful when a GUI or other caller needs to choose an explicit scene target. It also accepts the caller through an entity-typed parameter.

```
extends PCAScriptBase

func set_visible(target, visible):
    await SetObjectVisible({
        "entityId": target, "visible": visible
    })
    return visible
```

| Definition field | Value |
|---|---|
| Function Name / Display Name | set_visible / Set Object Visibility |
| Parameter 1 | target; entity; default this |
| Parameter 2 | visible; bool; default true |
| Return | The requested Boolean visibility value. |

1. Expose the function and apply/save its source.
2. Call it through Call Script Function.
3. For an object event, target can be this.
4. For a GUI event, enter the actual scene object ID instead of the GUI owner.
5. Set visible to false and test that only the intended target changes.

> The target must exist in the applicable live room. This helper makes the target explicit; it does not create a missing entity or load another room’s model.

# Appendix E. Event context quick reference

| Event | Useful context / caution |
|---|---|
| OnGameStart | sceneId, playerId; no object caller is implied. |
| OnSceneLoaded | sceneId, previousSceneId, playerId, firstVisit. |
| OnSceneLeaving | sceneId and nextSceneId. |
| OnPlayerChanged / OnCharacterControl | playerId and displayName; control event has its own caller handling. |
| OnInteraction | entityId, verb, sceneId and inventory/source fields as applicable. |
| OnTriggerEntered / OnTriggerExited | triggerId, entityId, sceneId, playerId; regions can add regionId. |
| OnInventoryChanged | characterId, itemId, amount and operation. |
| OnItemFound | characterId, itemId, amount when first acquired in that inventory context. |
| OnDialogueStarted / OnDialogueEnded | dialogueId plus the preserved context. |
| OnCustomEvent | The explicit JSON context supplied by the sender. |
| GUI element event | guiId, guiElementId, eventName and guiEvent. |
| Media-end entry | videoAssetId or musicAssetId. |

Available fields come from the event source. A missing field reads as null. The stored caller is captured from the context and can differ from a later action target. Do not assume itemId and this always identify the same record.

> Filter a listener before changing state. Never make inventory changes from every inventory notification, or re-emit the same custom tag unconditionally from its own listener.

# Appendix F. A complete small graph

This is a complete Visual Script record for the first message. It is a reference for the structure seen in Raw JSON. It belongs in a Visual Script record, not over the whole .pca3d project file.

```
{
  "id":"first_message", "name":"First Message",
  "nodes":[
    {"id":"start", "type":"OnInteract", "x":40, "y":60,
     "data":{"verb":"use"}},
    {"id":"message", "type":"ShowMessage", "x":460, "y":60,
     "data":{"text":"The switch clicks.", "duration":1.8,
             "typewriter":false}},
    {"id":"finish", "type":"End", "x":880, "y":60,
     "data":{}}
  ],
  "edges":[
    {"from":"start", "out":"next", "to":"message", "in":"flow"},
    {"from":"message", "out":"next", "to":"finish", "in":"flow"}
  ]
}
```

nodes contains the entry, action and ending. edges records each wire’s source, named output and destination. x/y are editor positions. The data dictionary contains the action fields.

If you adapt this for a differently named existing record, preserve its correct record ID and dictionary identity. Normal palette authoring is the safer route while you are learning.

# Appendix G. Node catalog

The current build exposes 231 distinct node types. Search the palette by the visible name or action ID. Flow contains several newer shared action families as well as graph control operations.

| Visible node name | Action ID | Category |
|---|---|---|
| Animation: Pause | PauseAnimation | Animation |
| Animation: Resume | ResumeAnimation | Animation |
| Play Animation | PlayAnimation | Animation |
| Stop Animation | StopAnimation | Animation |
| Add Character To Scene | AddCharacterToScene | Characters |
| Change Character Property | ChangeCharacterProperty | Characters |
| Character Instant Teleport To | CharacterTeleport | Characters |
| Character Walk Through Waypoints | CharacterWalkPath | Characters |
| Character Walk To | CharacterWalkTo | Characters |
| Character Waypoints: Add Waypoint | AddCharacterWaypoint | Characters |
| Character Waypoints: Clear Queue | ClearCharacterWaypoints | Characters |
| Character Waypoints: Walk Queue | CharacterWalkWaypoints | Characters |
| Enable / Disable Character Action | SetCharacterActionEnabled | Characters |
| Face Character | FaceCharacter | Characters |
| Face Coordinates | FaceCoordinates | Characters |
| Face Object | FaceObject | Characters |
| Follow Character | FollowCharacter | Characters |
| Hide Player Character | HidePlayerCharacter | Characters |
| Move NPC to a Different Room | MoveNPCToRoom | Characters |
| Stop Follow Character | StopFollowCharacter | Characters |

# Appendix G. Node catalog (2/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| For Loop | ForLoop | Conditions |
| Has Inventory | HasInventory | Conditions |
| If Attach Points Touching | IfAttachPointsTouching | Conditions |
| If Character Is Moving | IfCharacterMoving | Conditions |
| If Compare | IfCompare | Conditions |
| If Current Room | IfCurrentRoom | Conditions |
| If Inventory Item Was Used | IfInventoryItemWasUsed | Conditions |
| If Player Already Was Here | IfPlayerWasHere | Conditions |
| If Player Was Never Here | IfPlayerWasNeverHere | Conditions |
| If Variable | IfVariable | Conditions |
| While Loop | WhileLoop | Conditions |
| End Dialogue | EndDialogue | Dialogue |
| Say | Say | Dialogue |
| Show Message | ShowMessage | Dialogue |
| Start Dialogue | StartDialogue | Dialogue |
| Unlock Hint Topic / Answer | UnlockHint | Dialogue |
| On Event | OnEvent | Events |
| On Interact | OnInteract | Events |
| On Music End | OnMusicEnd | Events |
| On Video End | OnVideoEnd | Events |

# Appendix G. Node catalog (3/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| Activate / Blend to Camera | ActivateCamera | Flow |
| Add Dialogue Choice | AddDialogueChoice | Flow |
| Add Dialogue Choice Node | AddDialogueChoiceNode | Flow |
| Add Dialogue Line | AddDialogueLine | Flow |
| Align Attach Points | AlignAttachPoints | Flow |
| Attach Object | AttachObject | Flow |
| Attach Particle Effect | AttachParticleEffect | Flow |
| Attach Points | AttachPoints | Flow |
| Attach Points Touching | AttachPointsTouching | Flow |
| Automatic Input Switching | SetAutoInputSwitching | Flow |
| Call Script Function | CallScriptFunction | Flow |
| Camera Orbit | CameraOrbit | Flow |
| Cancel Two-Object Command | CancelPreposition | Flow |
| Carousel: Add Image | CarouselAddImage | Flow |
| Carousel: Clear Images | CarouselClearImages | Flow |
| Carousel: Next Image | CarouselNext | Flow |
| Carousel: Previous Image | CarouselPrevious | Flow |
| Carousel: Remove Image | CarouselRemoveImage | Flow |
| Close Interaction Coin | CloseInteractionCoin | Flow |
| Cycle Verb | CycleInteractionVerb | Flow |

# Appendix G. Node catalog (4/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| Detach Object | DetachObject | Flow |
| Detach Particle Effect | DetachParticleEffect | Flow |
| Enable / Disable Dialogue Choice | SetDialogueChoiceEnabled | Flow |
| Enable Lip Sync | SetLipSyncEnabled | Flow |
| Enable Mouse Clicks | SetMouseClicksEnabled | Flow |
| Enable Mouse Pointing | SetMousePointingEnabled | Flow |
| End | End | Flow |
| End Visual Script | EndVisualScript | Flow |
| Execute Two-Object Command | ExecutePreposition | Flow |
| Get Attach Point | GetAttachPoint | Flow |
| Get Attach Point Contacts | GetAttachPointContacts | Flow |
| Get Control State | GetControlState | Flow |
| Interact with Nearest Hotspot | InteractNearest | Flow |
| Is Current Room | IsCurrentRoom | Flow |
| Jump | Jump | Flow |
| Open Interaction Coin | OpenInteractionCoin | Flow |
| Pause FMV | PauseFMV | Flow |
| Pause Voice | PauseVoice | Flow |
| Play Gameplay Recording | PlayGameplayRecording | Flow |
| Play Voice | PlayVoice | Flow |

# Appendix G. Node catalog (5/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| Proximity Interaction | SetProximityInteractionEnabled | Flow |
| Remove Attach Point | RemoveAttachPoint | Flow |
| Remove Dialogue Choice | RemoveDialogueChoice | Flow |
| Remove Dialogue Node | RemoveDialogueNode | Flow |
| Reset Dialogue History | ResetDialogueHistory | Flow |
| Resume FMV | ResumeFMV | Flow |
| Resume Voice | ResumeVoice | Flow |
| Return | Return | Flow |
| Seek FMV | SeekFMV | Flow |
| Seek Voice | SeekVoice | Flow |
| Set Advanced Camera Behavior | SetCameraBehavior | Flow |
| Set Attach Point | SetAttachPoint | Flow |
| Set Camera Priority | SetCameraPriority | Flow |
| Set Camera Targets | SetCameraTargets | Flow |
| Set Control Setting | SetControlSetting | Flow |
| Set Game Language | SetGameLanguage | Flow |
| Set Input Device | SetInputDevice | Flow |
| Set Interaction Mode | SetInteractionMode | Flow |
| Set Movement Profile | SetMovementProfile | Flow |
| Set Runtime Controls Enabled | SetRuntimeControlsEnabled | Flow |

# Appendix G. Node catalog (6/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| Set Save Slot Image | SetSaveSlotImage | Flow |
| Set Voice Speaker | SetVoiceSpeaker | Flow |
| Shake Screen | ShakeScreen | Flow |
| Show Mouse Pointer | SetMousePointerVisible | Flow |
| Start FMV | StartFMV | Flow |
| Start Gameplay Recording | StartGameplayRecording | Flow |
| Stop Attach Point Watch | StopAttachPointWatch | Flow |
| Stop Camera Orbit | StopCameraOrbit | Flow |
| Stop FMV | StopFMV | Flow |
| Stop Gameplay Playback | StopGameplayPlayback | Flow |
| Stop Gameplay Recording | StopGameplayRecording | Flow |
| Stop Screen Shake | StopScreenShake | Flow |
| Stop Voice | StopVoice | Flow |
| Wait | Wait | Flow |
| Watch Attach Point Contacts | WatchAttachPointContacts | Flow |
| Center GUI Model | CenterGUIModel | GUI |
| Change GUI Element Property | ChangeGUIElementProperty | GUI |
| Change GUI Property | ChangeGUIProperty | GUI |
| Disable GUI | DisableGUI | GUI |
| Disable GUI Element | DisableGUIElement | GUI |

# Appendix G. Node catalog (7/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| Enable GUI | EnableGUI | GUI |
| Enable GUI Element | EnableGUIElement | GUI |
| Hide GUI | HideGUI | GUI |
| Hide GUI Element | HideGUIElement | GUI |
| List Box: Add Item | ListBoxAddItem | GUI |
| List Box: Clear Items | ListBoxClearItems | GUI |
| List Box: Delete Item | ListBoxDeleteItem | GUI |
| List Box: Modify Item | ListBoxModifyItem | GUI |
| Play GUI Model Animation | PlayGUIModelAnimation | GUI |
| Save Game Widget: Command | SaveGameWidgetCommand | GUI |
| Show GUI | ShowGUI | GUI |
| Show GUI Element | ShowGUIElement | GUI |
| Call Script | CallScript | Game |
| Change Current Verb Command | ChangeVerb | Game |
| Change Game Cursor | ChangeGameCursor | Game |
| Change Game Setting | ChangeGameSetting | Game |
| Change Project Property | ChangeProjectProperty | Game |
| Change Subtitle Language | ChangeSubtitleLanguage | Game |
| Change Voice Language | ChangeVoiceLanguage | Game |
| Emit Event | EmitEvent | Game |

# Appendix G. Node catalog (8/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| Enable / Disable Automatic Interaction Approach | SetInteractionApproachEnabled | Game |
| Enable / Disable Player Input | SetPlayerInputEnabled | Game |
| Enable / Disable Verb Command Usage | SetVerbUsageEnabled | Game |
| Load Game | LoadGame | Game |
| Load Game Slot | LoadGameSlot | Game |
| Run Visual Script | RunVisualScript | Game |
| Save Game | SaveGame | Game |
| Save Game Slot | SaveGameSlot | Game |
| Add Inventory | AddInventory | Inventory |
| Change Inventory from Player to Player | TransferInventory | Inventory |
| Change Inventory Quantity | ChangeInventoryQuantity | Inventory |
| Remove Inventory | RemoveInventory | Inventory |
| Music: Fade | FadeMusic | Media |
| Music: Pause | PauseMusic | Media |
| Music: Resume | ResumeMusic | Media |
| Play Music | PlayMusic | Media |
| Play Sound | PlaySound | Media |
| Play Video | PlayVideo | Media |
| Seek Video | SeekVideo | Media |
| Skip Ahead in Soundtrack | SeekMusic | Media |

# Appendix G. Node catalog (9/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| Sound: Fade | FadeSound | Media |
| Sound: Pause | PauseSound | Media |
| Sound: Resume | ResumeSound | Media |
| Stop Music | StopMusic | Media |
| Stop Sound | StopSound | Media |
| Stop Startup Splash Audio | StopStartupSplashAudio | Media |
| Stop Video | StopVideo | Media |
| Quest: Change Objective Property | ChangeQuestObjectiveProperty | Quests |
| Quest: Change Property | ChangeQuestProperty | Quests |
| Quest: Change Stage Property | ChangeQuestStageProperty | Quests |
| Quest: Complete Objective | CompleteObjective | Quests |
| Quest: Complete Quest | CompleteQuest | Quests |
| Quest: Fail Objective | FailObjective | Quests |
| Quest: Fail Quest | FailQuest | Quests |
| Quest: Hide / Unhide | SetQuestHidden | Quests |
| Quest: Reset Objective | ResetObjective | Quests |
| Quest: Reset Quest | ResetQuest | Quests |
| Quest: Set Stage | SetQuestStage | Quests |
| Quest: Track / Untrack | SetQuestTracked | Quests |
| Start Quest | StartQuest | Quests |

# Appendix G. Node catalog (10/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| Add Asset to Scene | AddAssetToScene | Scene |
| Camera Shake | CameraShake | Scene |
| Change Camera | ChangeCamera | Scene |
| Change Character Description | ChangeCharacterDescription | Scene |
| Change Inventory Description | ChangeInventoryDescription | Scene |
| Change Object Description | ChangeObjectDescription | Scene |
| Change Object Property | ChangeObjectProperty | Scene |
| Change Region Description | ChangeRegionDescription | Scene |
| Change Room Camera Bounds | SetRoomCameraBounds | Scene |
| Change Scene | ChangeScene | Scene |
| Change Scene Description | ChangeSceneDescription | Scene |
| Create Primitive | CreatePrimitive | Scene |
| Create Procedural | CreateProcedural | Scene |
| Cutscene: End | EndCutscene | Scene |
| Cutscene: Start | StartCutscene | Scene |
| Despawn Scene Object | DespawnObject | Scene |
| Enable / Disable Flashlight Mode | SetFlashlightMode | Scene |
| Enable / Disable Object | SetObjectEnabled | Scene |
| Enable / Disable Object Action | SetObjectActionEnabled | Scene |
| Enable / Disable Trigger | SetTriggerEnabled | Scene |

# Appendix G. Node catalog (11/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| Hide / Unhide Object | SetObjectVisible | Scene |
| Melt Accumulated Snow | MeltSnow | Scene |
| Screen Fade | FadeScreen | Scene |
| Screen Transition | ScreenTransition | Scene |
| Set 2D Camera | Set2DCamera | Scene |
| Set 2D Entity Depth | Set2DEntityDepth | Scene |
| Set 2D Layer Parallax | Set2DParallax | Scene |
| Set Object Interaction Approach | SetObjectInteractionApproach | Scene |
| Set Retro Rendering Style | SetRetroStyle | Scene |
| Set Room Brightness | SetSceneBrightness | Scene |
| Set Room Dimension | SetRoomDimension | Scene |
| Set Room Tint | SetSceneTint | Scene |
| Set Snow Accumulation | SetSnowAccumulation | Scene |
| Show / Hide 2D Layer | Set2DLayerVisible | Scene |
| Spawn Scene Object | SpawnObject | Scene |
| Stop Zoom Blur | StopZoomBlur | Scene |
| Switch Character | SwitchCharacter | Scene |
| Zoom Blur | ZoomBlur | Scene |
| Set Glow | SetGlow | Scene Life |
| Set Motion | SetMotion | Scene Life |

# Appendix G. Node catalog (12/12)

Continued from the searchable node catalog. Names and categories match the current editor.

| Visible node name | Action ID | Category |
|---|---|---|
| Toggle Glow | ToggleGlow | Scene Life |
| Toggle Motion | ToggleMotion | Scene Life |
| Toggle RGB String | ToggleRGBString | Scene Life |
| Add Score | AddScore | Variables |
| Change Variable | ChangeVariable | Variables |
| Create Variable | CreateVariable | Variables |
| Get Included Variable | GetIncludedVariable | Variables |
| Random Value | RandomValue | Variables |
| Read Context | ReadContext | Variables |
| Set Included Variable | SetIncludedVariable | Variables |
| Set Variable | SetVariable | Variables |

# Version and verification notes

Prepared against PCA 3D Adventure Studio 0.86.3.4. The guide follows the current native Visual Script editor, shared action catalog, function-definition forms and runtime behavior.

The workshop checks include both door outcomes, one-time pickup rewards, repeated calls, caller-aware targets, three-iteration For and While loops, return values, live parameter expressions, custom events, GUI-triggered graph work, included variables, parent-call stopping and a saved puzzle state.

The function workflow was exercised with GDScript, PCA Script, Lua and BASIC. A deliberately invalid integer parameter was rejected. The examples and JSON references in this PDF were checked for usable copy/extraction and structure.

Screenshots were captured in the native editor and its Play preview using the Compatibility renderer. This is not a performance certification for every operating system, device, renderer or imported asset.

The tutorial uses existing engine artwork and bundled character assets for demonstration. PCA 3D Adventure Studio and the supplied third-party assets retain their own credits and licenses. Keep the engine’s bundled credits with any distributed work that requires them.

> No engine source changes or separate practice files are required. Use the PDF contents and bookmarks to return to the relevant lesson.