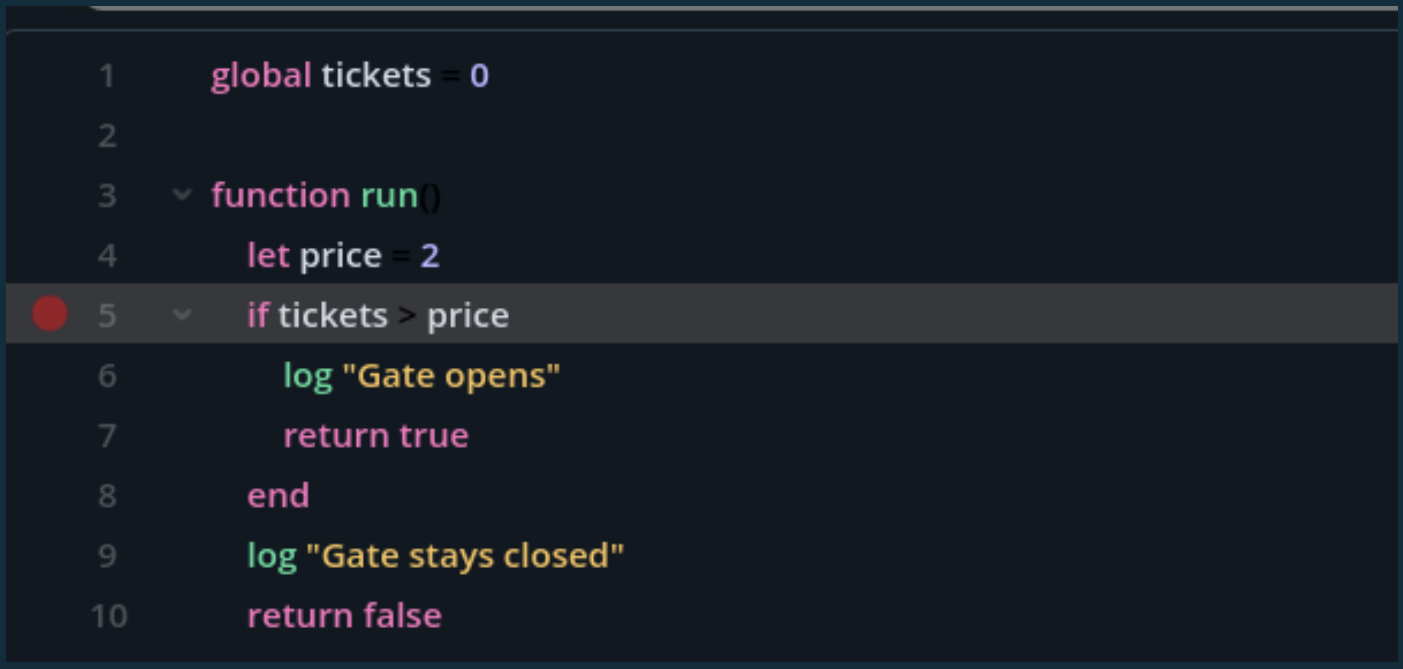
PCA 3D ADVENTURE STUDIO

Error handling and debugging

How to find out why your adventure behaves the way it does.



The gate costs two tickets. The player has two.

Why does it stay shut? Follow the evidence from the player's click to the exact comparison that sends them down the wrong path.

Illustrated with PCA3D 0.86.3.4 on Linux

CONTENTS

Find your starting point

If something is broken right now, start with the symptom on the left. Select a row to jump there. The PDF bookmarks list every chapter.

I can reproduce a bug, but I'm unsure where to look.	3
Validate reports a missing or incorrect record.	4
My script won't validate.	6
The game runs, but I can't see what happened.	7
I need to look at a value while the game is running.	8
The numbers look right, but the wrong branch runs.	9
An event, visual graph or button seems silent.	11
My game needs to recover when something goes wrong.	14
I need syntax for PCA Script, BASIC, Lua or GDScript.	15
It only fails after a script call, a wait or a reload.	19
The preview and the exported game behave differently.	22
I need to confirm the fix or report the bug.	24
A native warning printed a stack of file names.	25
An interrupted interaction charged the player twice.	26

> The steps use control names so you can follow them across operating systems. Screenshots and observed results come from the Linux editor. A different release, theme, language or screen size can change where a control appears.

FINDING THE CAUSE

Turn "it's broken" into a test

"The gate is broken" leaves a lot of room for guesswork. "Press Use gate with two tickets and the gate stays closed" tells you exactly where to start.

01 **Write down the rule.** The fare is two tickets. A player with one ticket should be refused. A player with two or more should get through.

02 **Keep the starting conditions steady.** Use the same room, the same button, the same character and a known ticket count. Record whether you started a new game, tested the scene directly or loaded a save.

03 **Make the problem happen twice.** If it only fails the second time, that detail matters. A value may survive a room change, a GUI may open twice or a one-time event may already have fired.

04 **Change one thing.** Keep the button and room in place. Change only the ticket count. A failure at exactly two points toward the comparison. Failure at every count points farther up the chain.

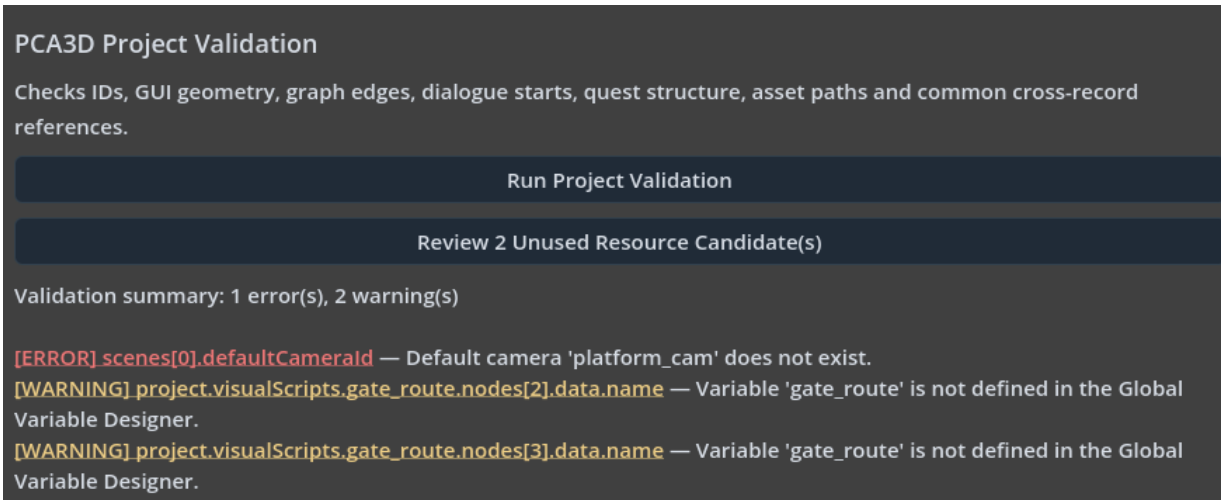
Choose the first place to look

The editor gives a line number	Open that script and inspect the reported line and the one before it.
The report names a room, camera or asset	Open Project Validation and read the record path.
The player can act, but the result is wrong	Check which action ran, what values it received and which branch it chose.
The application disappears	Record the last reliable action and whether the editor or the built game closed. Start with page 23.

PROJECT VALIDATION

Read a project validation error

Project Validation checks relationships across your game. A room can contain valid script code and still point to a camera that no longer exists.



The validator found one error and two warnings. The red camera entry identifies both the record path and the missing ID.

01 Select **Validate** in the main editor toolbar. In the Project Validation window, **Run Project Validation** repeats the check. Read the first error and keep the full text.

02 Here the path is **scenes[0].defaultCameraId**. That means "the default camera on the first room record." The message names **platform_cam** as the missing camera.

03 Select the issue to navigate when the editor can resolve its target. Open that room's camera choices. If an existing camera should be used, select its ID. If the missing camera was intentional, restore it and its settings.

04 Apply the room edit and run the check again. In this game, choosing the existing camera **wide** removed the error. Confirm the framing by starting in that room.

PROJECT VALIDATION

Decide what a warning needs

After fixing the camera, the report still showed two warnings. They looked almost identical because two different visual nodes wrote to the same undeclared variable.

```
Validation summary: 0 error(s), 2 warning(s)

[WARNING] project.visualScripts.gate_route.nodes[2].data.name — Variable 'gate_route' is not defined in the Global Variable Designer.
[WARNING] project.visualScripts.gate_route.nodes[3].data.name — Variable 'gate_route' is not defined in the Global Variable Designer.
```

The error count fell to zero. Both remaining paths end in data.name and identify nodes in the same gate_route graph.

Both nodes intended to record the route in a string called **gate_route**. Defining that name as a string in the Global Variable Designer cleared both warnings. If a node had simply misspelled an existing name, correcting the node would have been the right edit.

```
Validation summary: 0 error(s), 0 warning(s)

No structural or reference problems were found.
```

After declaring the intended variable, the report shows zero errors and zero warnings.

When an unfinished room blocks a preview

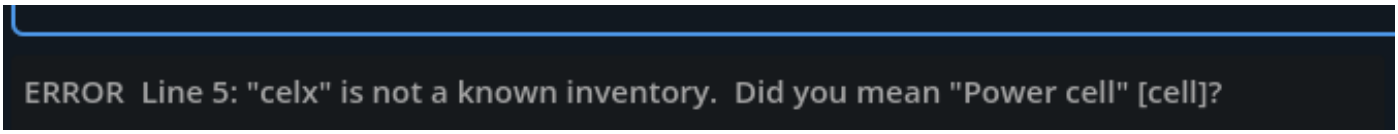
If a playtest is stopped by eligible project errors, the validation dialog can offer **Preview with errors**. Use it to inspect a separate part of an unfinished game. The broken reference still needs fixing before Build Game. Some requests, such as testing a room that does not exist, cannot continue.

An unused-resource candidate deserves a look before removal. A script might choose the resource by combining strings at runtime. Confirm what uses it in your game before changing a record.

SCRIPT VALIDATION

Follow a script error to its source

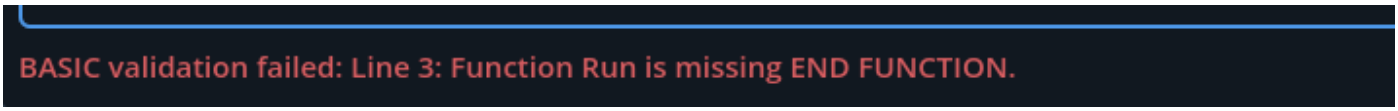
A script validator reads what you wrote and checks what it can recognize. Start with the earliest relevant message. A missing quote or block ending can make several later lines look wrong.



PCA Script identifies line 5, the unrecognized inventory ID celx, and Power cell [cell] as a possible match.

```
give celx to player
```

The author intended to give the player the Power cell. Opening the inventory record confirms the stable ID is **cell**. Change the statement to **give cell to player** and validate the script again. A suggestion is worth checking against the intended item, especially when several items share a similar name.



The BASIC Designer reports a missing END FUNCTION. The opening FUNCTION can be correct while the mistake is at the bottom of the block.

PCA Script	Use the script toolbar's Validate button. Activate a diagnostic entry to jump to the source line. Go to Record and autocomplete can help confirm an authored ID.
BASIC, Lua or GDScript	Use Validate in that language's designer. Fix the reported syntax, validate again, apply the edit and save your project. Only then repeat the gameplay action.

> A clean report means the checks that ran passed. A perfectly legal comparison can still produce the wrong story outcome. The two-ticket case on page 10 shows exactly how.

OUTPUT & DIAGNOSTICS

Make the output tell a story

Give the log one short line at each point where the answer could change. You're trying to learn whether the button ran, what it received and what it decided.



Output & Diagnostics collects messages as the preview runs. Open it again from Panels if it is hidden.

```
gate | entered
gate | tickets=2 price=2
gate | branch=closed
```

These are example log messages. Keeping the same prefix makes it easier to pick out one interaction among room-load, sound and inventory messages. Write the values before the choice and the chosen result inside each branch.

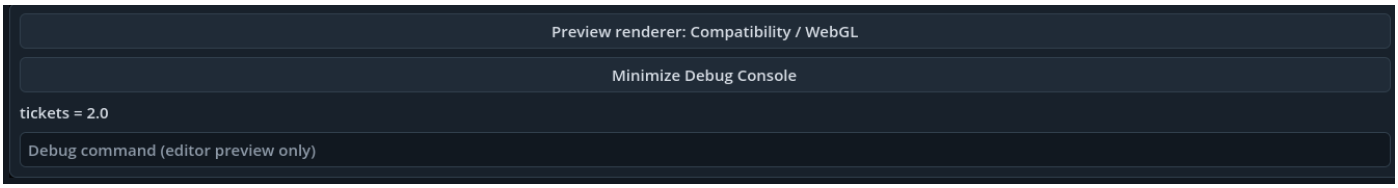
No entry line	Trace the button, verb or event that should start the script. Make sure this is the record the game actually called.
Entry appears, no values	Inspect the statements between the two messages. Did a return, wait or failed call stop the path?
Values are right, branch is wrong	Look closely at the operator and the data types. A number and text that looks like a number can behave differently.
Correct branch, wrong screen	Trace the action that changes the visible room, object or label. Another handler may immediately overwrite it.

Use log in PCA Script, PRINT in BASIC, and Log(...) in a PCA3D Lua or native GDScript record. Remove high-frequency tracing after the fix so important messages stay visible.

PREVIEW CONSOLE

Ask the live preview for a value

There are two different places to read. Output & Diagnostics holds a running history. The small console inside Runtime Preview answers a command about the current game.



The expanded test console under the running game. Its result line shows tickets = 2.0.

01 Start a game preview or use **Test Scene**. In Runtime Preview select **Show Debug Console**. The command field appears beneath its result line.

02 Type each line below separately and submit it. **get** reads a shared game variable. **set** changes it in the current preview.

```
get tickets
set tickets 2
get tickets
```

This run started at **0.0** and ended at **2.0**. Both 2 and 2.0 represent a numeric fare. A missing name displays **<undefined>**. The console reads shared runtime variables. For a function's local price, use the PCA Script watch panel.

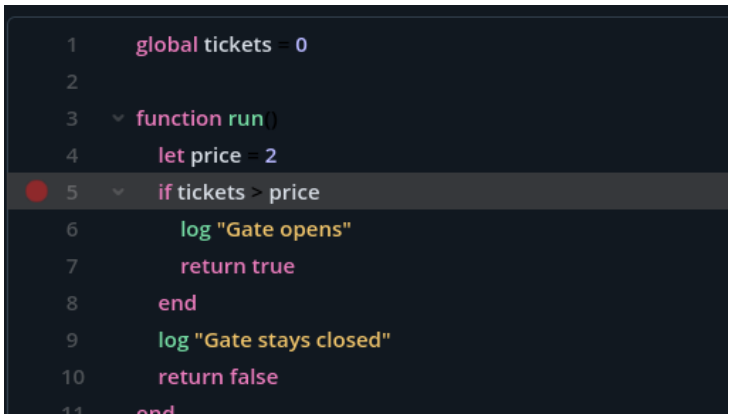
03 Select **Minimize Debug Console** before testing keyboard movement or pressing a game button. It releases keyboard focus from the command field.

> Setting a value is a shortcut for diagnosing the next step. After the comparison works, earn the tickets through normal gameplay and repeat the click. This checks the part the shortcut bypassed.

PCA SCRIPT DEBUGGER

Pause before the decision

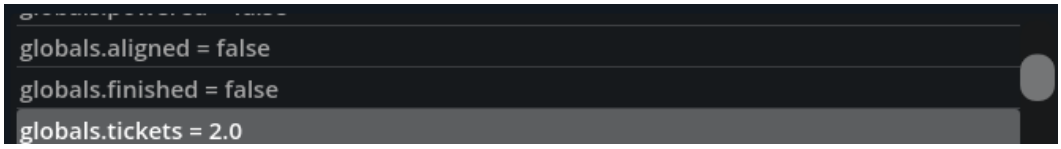
A breakpoint tells the editor to pause a running PCA Script so you can inspect the values in that call. Put it near the choice you want to understand.



The red dot marks line 5 in gate_check. The highlighted comparison asks whether tickets is strictly greater than price.

01 Open the record in the **PCA Script** workspace. Mark an executable line using the breakpoint gutter or the code editor's **Toggle Breakpoint** command. Run the record from that workspace with **Run**.

02 Run can switch you to Runtime Preview. Return to **PCA Script** to see the pause status, caret and watches. If Continue and Step are disabled, check that this record ran and execution reached that line.



Scroll the watch list to the shared value you need. At this pause, globals.tickets reads 2.0.

03 **Step** advances to the next debug stop. **Continue** lets execution proceed until another stop or the end. Read the output after continuing to confirm which branch actually ran.

> The watch list is a snapshot grouped by locals, constants, globals and context. Scroll if necessary. These controls belong to the PCA Script workspace. A released game does not wait for the editor's Continue button.

A LOGIC ERROR, TRACED

Why exactly two tickets failed

The script passed validation. When a player pressed the gate button with two tickets, the output still said Gate stays closed. Here is the decision that produced it.

```
function run()
    let price = 2
    if tickets > price
        log "Gate opens"
        return true
    end
    log "Gate stays closed"
    return false
end
```

The project supplies the shared number **tickets**. Read the condition aloud. “More than two” excludes two. The design calls for “at least two,” written **tickets >= price**. Only the comparison needs to change.

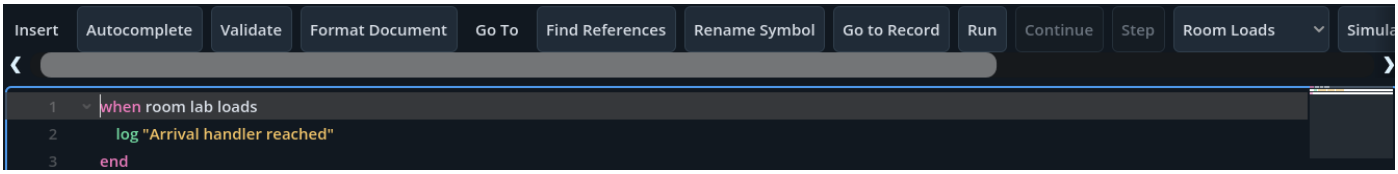
Player holds 1	Before the change: closed. After the change: closed.
Player holds 2	Before: closed. After: open. This is the case that exposed the mistake.
Player holds 3	Before: open. After: open.

> The three counts were run through both versions. The gate button was also checked at the exact fare. For a real gate, verify the character can actually pass, and that a second click behaves as your design intends.

EVENTS

Tell an event problem from a handler problem

A handler is the code that responds to something happening. When a room-entry message never appears, test the handler and the real doorway separately.



The PCA Script workspace offers a handler selector and Simulate Event. This record listens for the room lab to load.

01 Put a short log message at the start of the handler. Check its event name and filters. A handler for **lab** will ignore other rooms. For a BASIC or Lua record, also check the matching event subscription in its Designer.

02 For PCA Script, validate the record, select a handler in the toolbar and use **Simulate Event**. A simulated call carries editor test information. A real room, trigger or inventory event can supply additional fields that a filter or a branch needs.

03 Repeat the actual event. Enter the room through the usual door or start a scene test. In the captured game, entering lab produced **Arrival handler reached**. If an isolated test succeeds but the doorway fails, inspect the destination and the route into the room.

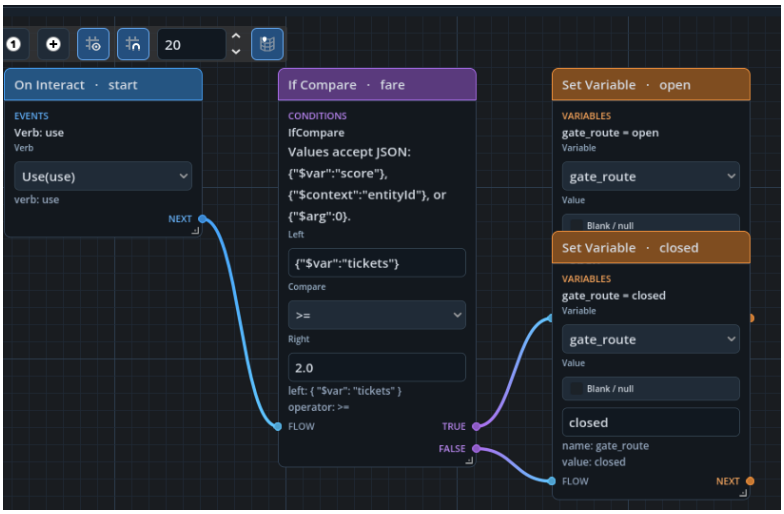
```
when room lab loads
    log "Arrival handler reached"
end
```

Check how often the message appears too. A message printed twice for one entry can mean two subscribed records, two event bindings, or a second scene load. Log the record ID in each handler to tell them apart.

VISUAL SCRIPT

Follow a visual graph one wire at a time

The quickest way to understand a silent graph is to identify its entry, the comparison, and the first action on each possible exit.



The gate graph has an interaction entry, an IfCompare node, and two SetVariable destinations. Each destination records the path taken.

01 Open the Visual Script record used by the interaction. Confirm the entry and any verb filter. An **OnInteract** node filtered to **use** needs the corresponding interaction context.

02 Inspect **IfCompare**. Here the left field reads the shared tickets variable, the operator is **>=**, and the right field is the number 2. A plain string "tickets" would supply the word itself. The JSON value **{"$var":"tickets"}** reads the variable.

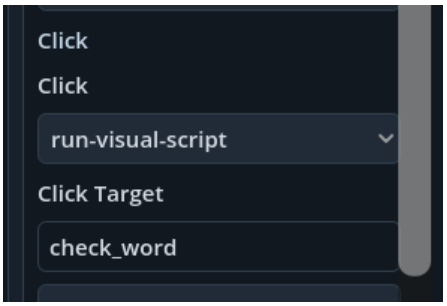
03 Follow both output wires. The true branch stores **open** in gate_route. The false branch stores **closed**. After triggering the graph, enter **get gate_route** in the preview console. The observed results at 1, 2 and 3 tickets were closed, open and open.

> An unwired branch can end quietly. If a variable kept its last value, reset it to a distinct marker before retesting. Otherwise yesterday's "open" can look like evidence from today's click.

GUI AND INPUT

Find where a button click disappears

A button can look correct while pointing to the wrong script. A script can run correctly while writing its answer to a label the player cannot see.



The Events inspector for a GUI button. Click chooses a behavior. Click Target identifies the record that should receive it.

1. Did the button receive input?	Confirm the GUI is open, the element is visible and enabled, and another overlay is not taking the click. Minimize the preview console before retesting.
2. Did the expected record start?	Select the button in GUI Designer and check its Events settings. Compare the Click Target with the script ID. Add a distinctive first log line to that script.
3. What did the script read?	Log the GUI ID, element ID and raw text. Distinguish an empty string from a missing control that returned your chosen fallback.
4. Where did the answer go?	Read back the reply label's text after setting it. If the value is correct but the player sees old text, inspect the visible GUI, label and any later update.

In the pictured direction prompt, blank input produced "Type a direction." The wrong word produced "Read the sign again." Input with spaces and mixed case still produced "The latch clicks." Those three answers let you check the whole route from a click to a visible result.

ERROR HANDLING

Choose how the game should recover

A player typing the wrong word is a normal part of an adventure. A missing destination room is an authoring mistake. Players should get a sensible response in either case, but your code should handle them differently.

Expected input	An empty name, a wrong answer or too few tickets calls for an ordinary condition. Explain what the player can do next. Leave the puzzle in a playable state.
An optional resource	If a hint preference is missing, start with a documented default. If a voice clip is optional, keep the subtitle available. Record the missing ID if it is something the author needs to fix.
An impossible internal value	A ticket balance of −1 violates the rules. Use an assertion while diagnosing where the bad value came from. Before changing inventory or opening the gate, stop that action and log the record and value.
An operation that can fail	If a save or write reports failure, preserve the player’s current state. Show a retry or a return path. Display “Saved” only after the operation reports success.

Read the answer a helper actually returns

An empty string can be a valid player answer. Zero can be a valid fare. FALSE can be a perfectly good preference value. When failure and success can both look like zero or false, return a separate status such as **{"ok": false, "value": 0}**. Let the caller decide what to show.

> Validate the prerequisites before charging the player. If an action can run twice, check the completion flag on entry, and decide how to restore input on both success and failure.

LANGUAGE GUIDE · PCA SCRIPT

Catch a failed operation in PCA Script

Use try around the operation that can fail. Catch receives an error record. Keep the block small enough that the message points to one job.

```
function run()
    let count = -1
    try
        assert count >= 0
        return "Ready"
    catch problem
        log problem.message
        return "Check the count"
    end
end
```

In this listing the count is deliberately fixed at −1 so the failure is visible. The assertion fails. The catch block logs **Assertion failed.** and returns **Check the count**. A nonnegative count reaches the earlier return.

problem.message	A readable description of the failure.
problem.line	The source line reported for that failure.
problem.scriptId	The ID of the record that reported the error.

A syntax error prevents this routine from starting. Fix it with Validate. Use catch for a runtime failure reported through the shared script engine. If an operation returns a failure value normally, inspect that value explicitly.

LANGUAGE GUIDE · BASIC

Handle a bad value in BASIC

PCA3D BASIC spells the same idea TRY, CATCH and END TRY. A guard lets a function report the problem before it changes anything in the game.

```
FUNCTION Run()
    DIM count = -1
    TRY
        ASSERT count >= 0
        RETURN "Ready"
    CATCH problem
        RETURN "Check the count"
    END TRY
END FUNCTION
```

This complete BASIC record returned **Check the count**. The assertion interrupts the normal path. In a catch block, **PRINT problem.message** can give the author the original reason while the caller receives a shorter reply.

When a helper reports failure itself

```
IF NOT WriteTextFile("user://notes.txt", note_text) THEN
    PRINT "journal | write failed"
    RETURN FALSE
END IF
```

This fragment belongs inside a function where **note_text** already contains the intended text. A file write returns a Boolean. Checking it handles an unsuccessful write even when no catchable error occurs. Tell the player the journal was saved only on the success path.

> For a BASIC record, use Validate and PRINT to trace the path. The interactive Step and watch controls shown on page 9 are in the PCA Script workspace.

LANGUAGE GUIDE · LUA

Protect one operation in Lua

The Lua function pcall runs a function and gives you two answers. The first says whether it completed. The second holds its result or an explanation of the error.

```
local Script = {}

function Script.run(context)
    local ok, value = pcall(function()
        local n = tonumber(context.raw)
        assert(n and n >= 0, "Expected a nonnegative fare")
        return n
    end)
    if not ok then
        Log("gate | " .. tostring(value))
        return 0
    end
    return value
end

return Script
```

With raw set to **"3"**, tonumber produces 3 and the record returns 3. With **"bad"**, the assertion fails. The console identifies lua_guard and its failing line, and the routine returns the chosen fallback, 0.

This fallback works for a display that should show zero. For a purchase, returning zero could accidentally grant a free ticket. Return an explicit failure status and refuse the transaction if a real zero would be meaningful.

> This example protects synchronous Lua code. A helper that completes normally and returns false still leaves pcall’s first answer true. Check the helper’s returned value as well.

LANGUAGE GUIDE · GDSCRIPT

Return a clear status in GDScript

Native GDScript can call Godot APIs. For a routine that other PCA3D languages may call, return a simple answer they can inspect without knowing anything about the native object.

```
func run(context: Dictionary = {}) -> Dictionary:
    var text := str(context.get("raw", "")).strip_edges()
    if not text.is_valid_int() or text.to_int() < 0:
        Warn("gate | invalid fare")
        return {"ok": false, "value": 0}
    return {"ok": true, "value": text.to_int()}
```

Input "bad"	Warn writes a warning. The return value contains ok = false and value = 0.
Input "3"	The return value contains ok = true and value = 3.
Input "0"	Zero passes the rule. The ok field lets the caller distinguish a legitimate zero from an invalid entry.

What TryCall checks

```
func run() -> Dictionary:
    return TryCall(Callable(), [], "no fare")
```

This intentionally empty Callable produced **ok = false**, the fallback **no fare** and the message **Callable is invalid.** TryCall tests that a callable exists before calling it. It cannot catch an arbitrary engine error inside a valid method. Check inputs and inspect a method's own return contract.

When working in an editable Godot project, use the Godot debugger for native breakpoints and its call stack. Use the PCA3D preview to confirm the actual button, event and runtime context that reach the native code.

FUNCTIONS, NODES AND THE CALLER

Debug a call that crosses languages

A visual action can call BASIC, and a BASIC record can ask GDScript for a value. The call can reach the right function while the caller still does the wrong thing with the answer.

01 **At the caller**, log the target record ID, function name and argument values, in order. Make sure the script ID is unique across languages. A missing optional argument and a supplied empty string may mean different things.

02 **At the callee**, log the values it received and its return value. If this line never appears, check that the record is enabled and that the function name is still current.

03 **Back at the caller**, inspect the return type. With the GDScript example on page 18, read the **ok** field first. Reading only value would turn bad input into a legitimate-looking zero.

A visual action can lose a good result

In **Functions as Nodes**, check the exposed function name and parameter definitions against the actual function. In **Call Script Function**, check the selected function, the argument values and the result variable. Query that variable immediately after the action.

```
BASIC function: Fare(count, rate)
Inputs: 3, 4
Function result: 12
Stored result: 12
```

Those were the values in a tested visual-to-BASIC call. If the function logged 12 but the next node saw an older value, the next two places to inspect would be the selected result variable and any intervening action that writes it.

SAVE AND LOAD

Find state that vanishes after loading

When a puzzle resets after loading, trace one fact through time. A single Boolean such as signal_seen is easier to follow than a whole room full of props.

1. Before saving	Trigger the story event the normal way. Use get signal_seen and record the result. If it is already wrong, debug the event before investigating the save.
2. During the save	Use a disposable test slot. Record the slot number and wait for the save to finish. Preserve any player save you still need.
3. Before loading	Change the live flag to the opposite value. Reading it now makes the difference observable. Leave the saved slot alone.
4. After loading	Load that same slot and query the flag again. Then inspect the actual door, label or character that should reflect it.

In the verified slot-3 case, the shared BASIC flag read **true** before saving, **false** after clearing it, and **true** after loading. If the value comes back correctly but the door still looks locked, inspect the room’s startup or refresh logic next.

> Function locals last for a call. A BASIC module-level DIM can survive repeated calls without becoming saved progress. Use a shared game variable for a story flag that belongs to a save slot. A settings file at a single user:// path can apply across several slots.

TIMING AND UNFINISHED ACTIONS

Find the wait that never finishes

A character walks to a locked gate and never gives control back. The last visible animation tells you where the story appears stuck. It may be several statements away from the actual cause.

The log stops before movement	Check the actor, target and room IDs. Verify that the destination lies on reachable navigation and your action is using the intended character.
Movement begins but the next line never runs	Check what the action waits for. A looping animation, unreachable target, or condition that no other handler changes can leave the next line waiting.
A second click doubles the reward	Trigger the same input twice quickly. Log the interaction ID on entry and before the reward. Check whether the action can start again while its first call is waiting.
The reward appears but input stays locked	Trace every return, error and cancellation path after the input lock. Check the matching unlock or cleanup is reached. Another active interaction can have its own reason for locking input.

Use a before-and-after pair

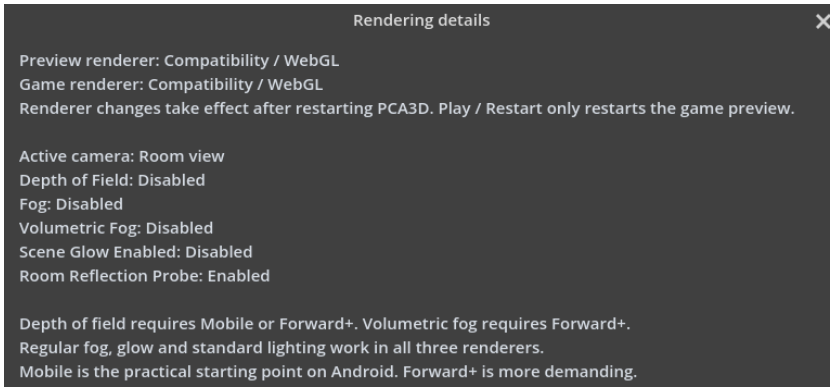
```
gate | before walk: target=station_door
gate | after walk
gate | before open animation
gate | after open animation
```

These are suggested trace messages. If "after walk" appears and "after open animation" does not, the next test belongs at the animation. Avoid adding a longer delay until you know what the code is waiting for. Test a second interaction after the first finishes.

RENDERING AND PERFORMANCE

When an effect is missing or the game is slow

The room can use valid settings and still look different on another device. Check what is actually rendering this frame before adjusting the fog, camera or material.



Rendering details lists the preview renderer, game renderer, active camera and the state of effects such as depth of field and fog.

01 In Runtime Preview, select **Preview renderer** to open **Rendering details**. Record the running renderer, selected game renderer and active camera. Play / Restart restarts the preview. Changing the editor's startup renderer takes effect after restarting PCA3D.

02 Check the effect's status. In this build depth of field needs Mobile or Forward+, and volumetric fog needs Forward+. For distance fog, check the Begin and End distances against the size of the room. For a camera effect, first confirm which camera is active.

03 For slow frames, repeat the same camera move at the same window size. Change one expensive feature at a time, such as a full-screen effect, shadows or particles. Compare the same view and timing on the device where players see the slowdown.

> A renderer label or a slow preview does not identify the cause of a crash. Keep the observed symptom, the exact device and the repeatable trigger together when reporting it.

BUILDS, DEVICES AND CRASHES

When only the exported game fails

First separate four failures that can look alike. Does the build stop, does the finished game fail to start, does one room fail to load, or does the running application close during a particular action?

Build stops	Run Project Validation, fix blocking errors and rebuild. Read the first relevant build error. A preview allowed to run with errors still leaves those errors in the project.
The game starts but an asset is missing	Check the stored asset ID and path, including capitalization. Confirm it is included in that build. Test the built copy outside the authoring environment, where a file on your own machine cannot hide a missing packaged asset.
Only one device fails	Record OS, device, renderer, build version and input method. Repeat the same room and action in a fresh session, then with the affected save, if available. On touch devices also test focus and the on-screen keyboard.
The application closes	Record which application closed, the last visible event, and the time. Retain the first relevant error and nearby log lines if available. In an editable Godot project, inspect the debugger stack and locate the first frame in your authored code.

Capture what you know. "The Windows build closes after the first voice line in this room" gives the next person a starting point. "The voice file crashed it" is a hypothesis until repeating that action with one controlled change isolates the cause.

> The Linux editor supplied the screenshots and runtime checks in this guide. Verify a platform-specific fix in an actual build on the affected target. A trial or older editor may offer a different set of build and debugging controls.

REGRESSION CHECKS AND BUG REPORTS

Prove the fix and write it down

A fix is convincing when it removes the failure and preserves the cases that already worked. For the gate, that means checking one ticket, exactly two, and more than two.

01 Repeat the original sequence from the same starting state. Use the real input at least once. A direct script run can skip the very event or button that caused the problem.

02 Check the cases immediately on either side. Restart a fresh session if startup matters. If you changed saved state, repeat the same test after loading. If the problem was device-specific, run the corrected build on that device.

03 Remove temporary debug messages or reduce them to the ones that explain genuine failures. Clear diagnostic breakpoints and confirm the game can finish the sequence again.

A bug report someone else can act on

Where	PCA3D 0.86.3.4, Linux editor preview. gate_check, triggered by the Use gate button.
Steps and actual result	Put exactly 2 tickets in the shared counter. Press Use gate once. Output says Gate stays closed. With 3 tickets, it says Gate opens.
Expected result	The fare is 2. The gate should admit a player holding 2 or more.
Cause and verification	The condition used >. Changed it to >=. At ticket counts 1, 2 and 3, the corrected routine returned false, true and true. Pressing the button at 2 logged Gate opens.

For an unresolved bug, leave the cause open. Attach the smallest relevant screenshot or log excerpt and include anything that makes it stop happening. "Only after loading slot 3" can be the most important sentence in the report.

TROUBLESHOOTING REFERENCE

Read a stack trace from the top

A stack trace is a list of calls that led to a message. Six lines in the stack can describe one warning. Read the message first, then look for the first frame that belongs to your game.

```
WARNING: gate | invalid fare
[0] Warn (...)
[1] run (...)
[2] _run_script_unlocked (...)
[3] run_script (...)
```

This is a shortened excerpt from the native GDScript case on page 18. The log included internal paths and line numbers. The frame names and order are preserved here. Frame zero is the helper that wrote the warning. Frame one is the authored run function that called it. The following frames are how PCA3D reached the script.

01 **Identify the first authored decision.** Search your record for the distinctive text **gate | invalid fare**. At that line, inspect the immediately preceding test and the argument it used.

02 **Compare one failing call with one good call.** The bad text took the warning path and returned a failure status. The text “3” took the normal path and returned a value of 3. This warning was a deliberate response to input, and the preview kept running.

03 **Preserve the original trace when reporting a fault.** A native record may appear under a generated script path. Include its PCA3D record ID and function name as well. An internal engine line alone can leave another author guessing which record to open.

> Look back immediately before a flurry of messages. A single failed lookup can leave several later actions without their target. Retest after fixing the earliest cause you can reproduce.

TROUBLESHOOTING REFERENCE

Keep a failed interaction recoverable

Suppose a ride charges two tickets and then takes the player to another room. What should happen if the player clicks twice, the destination is missing, or a save is loaded between the charge and arrival?

Before changing state	Check that the ride is available, the player can pay, and the destination belongs to this journey. Decide whether a ticket covers one trip or all future trips.
When the first click starts	Mark the interaction as in progress before the first wait. A second click can give a short response or do nothing. It should never charge a second fare for the same trip.
When a step fails	Record which step failed. If your design already charged the fare, refund that specific charge once or leave a saved "paid, awaiting travel" state that lets the player resume. Release any input lock owned by this interaction.
When travel succeeds	Mark the journey complete at the point your design considers arrival. Refresh the interface from that state so it shows what really happened.
When a save is loaded	Rebuild the room and controls from the stored state. Avoid replaying a charge simply because a room-entry handler ran a second time.

Four questions for the next playtest

How many tickets remain after a normal ride? What about a rapid double-click? Can a failed departure be retried without a second charge? After loading the resulting save, do the ticket count, the current room and the available controls all tell the same story?

> Use this pattern when an action has lasting consequences. For something harmless like examining a sign, a simple condition and a readable message are usually enough.

RUNTIME WALKTHROUGH

Read what the Runner actually did

Use a checked call when the next line of your script depends on the outcome. First check ok. On success, the property actions covered here return a result dictionary with a status. On failure, read error and message on the reply.

result.status	What to do next
applied	The active component accepted its new setting. When supplied, effective gives the value that was applied.
unchanged	The current value already matches. Repeating a brightness edit need not rebuild the light. A first persistent pin can still change its lifetime.
resource_refreshed	A resource such as a texture or GUI style was refreshed. Check that surrounding state, such as the text caret, survived.
component_rebuilt	A component had to be replaced. Check its continuity: for a model, watch the actor; for a particle layer, expect the simulation to be able to restart.
stored_for_load	Visit or reload the target room. Its scene or setting has been recorded for that later point.
stored_for_activation	Activate the named shot or applicable component, then inspect the change.
definition_only	The record changed. Check how the owning feature reads its definition before expecting a visual change.
subsystem_refresh	The widget or other owning system handled its refresh. Inspect that particular control.

An error takes a different route

If ok is false, log reply.error and reply.message. A missing resource and an unchanged value need different responses. Only the successful property result has the status you are looking for.

RUNTIME WALKTHROUGH

Follow a committed entity change

Sometimes the switch fires twice and the light moves only once. Check what was requested, what committed, and what actually changed. Those are three different observations.

```
func turn_up_lamp():
    var reply = await ActionResult("ChangeObjectProperty", {
        "entityId": "desk_lamp", "property": "light.intensity",
        "value": 2.5, "persistence": "room"
    })
    if reply.ok:
        print(reply.result.status)
    else:
        print(reply.error, " ", reply.message)
```

1 Trigger the switch twice

Start the lamp below 2.5. Substitute its real entity ID in the function above. Run it twice in the same room. The first successful change should print applied, the second unchanged. A failure should send you to the error branch.

2 Read the entity counters

In a native GDScript with Runtime available, inspect Runtime.property_write_counts. It groups successful entity writes by status, including unchanged. Compare before and after your two calls. A rejected write adds no successful count.

3 Watch the event when it matters

The Runner emits property_committed(room_id, entity_id, path, outcome). The last argument is the outcome itself, so read outcome.status in the listener. An ordinary repeat is quiet. A first equal-value pin can emit once because its save lifetime changed. Restore suppresses the ordinary notification.

> **If an older request finishes late**
> Visibility or enabled changes issued during physics can wait a frame. A newer write or a replaced target can make the earlier call fail with stale_target. Inspect the newest intent before trying again. GUI, camera, room and layer results should be checked at their own call sites.

RUNTIME WALKTHROUGH

Listen for one gameplay change

A native script can subscribe to the Runner when several interactions can change the same entity. A listener sees the room, object ID and exact path that committed.

```
func listen_for_lamp():
    if not Runtime.property_committed.is_connected(_on_commit):
        Runtime.property_committed.connect(_on_commit)

func _on_commit(room_id, entity_id, path, outcome):
    if room_id == "workshop" and entity_id == "desk_lamp":
        print(path, " ", outcome.status)
```

1 **Connect once**

Put both functions in the same native GDScript. Call listen_for_lamp when your script has a live Runtime, before the switch can fire. Use your real room and lamp IDs. The connection guard lets you call the setup again without adding the same listener twice.

2 **Compare the observations**

Apply a new intensity, repeat it, then try an invalid value. The counter can include the unchanged repeat. The listener usually reports only the first change. The invalid call should return a failure and leave both the lamp and the successful-write counters alone.

3 **Handle loading separately**

If your code must react when a checkpoint restores the room, use the room’s state-restored action list or OnRoomStateRestored. Ordinary entity commit notifications are suppressed during restore. The room event is the place to refresh a display that depends on restored state.

> **A useful bug report**
> Record the Studio version, renderer, room ID, entity ID, property, requested value and scope. Include ok and the full error or result. For a visual issue, hold the camera still and capture the same view before and after.

For a performance check, keep the same room and camera. In Godot’s Profiler and Monitors, record frame time, nodes, resources and memory before repeated writes, after them, and again after leaving the room. The outcome counters tell you how the calls finished. Use the measured trends to investigate slowdowns or growth.