Debugging SceneGraph Applications

Table of Contents

Special SceneGraph Debugging Commands

Available since firmware version 7.2

You can use special debugging commands to debug SceneGraph applications. These commands can be invoked in the debug server from telnet port 8080 port. These commands are:

- `sgnodes all`
  Prints every existing node created by the currently running channel.
- `sgnodes roots`
  Prints every existing node without a parent created by the currently running channel. The existence of these unparented nodes means they are being kept alive by direct BrightScript references. These could be in variables local to a function, arrays, or associative arrays, including a component global `m` or an associative array field of a node.
- `sgnodes` *`node_ID`*
  Prints nodes with an `id` field set to `node_ID`, except it bypasses all the hierarchy and rules, and just runs straight down the whole list in the order of node creation. It will list multiple nodes if there are several that match.

> These commands are similar to the **`getAll()`** , **`getRoots()`** , **`getRootsMeta()`,** and **`getAllMeta()`** **ifSGNodeChildren** methods, which can be called on any SceneGraph node.

Available since firmware version 7.5

- `loaded_textures`
  Display the current set of images loaded into texture memory
- sgversion [default | force [<version num>]]
  When the 1st parameter is the default, it sets the default rsg_version of a channel when it is not specified in the manifest. When the 1st parameter is a force, it sets the version despite what's in the manifest. Neither of these overrides survives a reboot.

> Please note that support for the “rsg_version=1.0” manifest flag is deprecated as of Roku OS 8. This deprecation means that the 1.0 features continue to work in Roku OS 8, but will no longer be supported (and thus should not be expected to work) starting with the next major firmware release. All channels will have to adopt the current observer callback model in successive firmware updates.

Troubleshooting Common Development Errors

There are several very common errors that you will encounter when developing SceneGraph applications. Quite often these errors are caused by not spelling component names or variables correctly but may appear as different types of errors on the display screen and in the debugger.

Graphic Image Does Not Appear, Question Mark Appears Instead of Image

The graphic image file was not found in the location specified in the application. Check that graphic image file is in the specified location, either on your development server, or in the application ZIP package, usually in the `pkg:/images` directory. Make sure the path to the file is correct, and the name of the file is spelled correctly. Roku SceneGraph applications, like previous Roku applications, generally follow the convention used in many client display applications, such as web browsers, which is to show a default image if the specified image cannot be found. If a question mark image is shown in a Roku SceneGraph application, check the path and file name spelling to ensure that the correct graphic image appears.

Debugger Message: Type Mismatch

This usually means that a BrightScript variable has been incorrectly spelled after it has been declared, never declared at all, or declared as a local variable in another function. Check the backtrace information supplied by the debugger for the local variables used at the time of the error, and note that the variables listed as `<uninitialized>`. Check for the declaration of the variable or variables listed earlier in the application prior to the error message, and correct the spelling, or declare them correctly, as needed. In many cases, the error occurs because there is an attempt to use a *local* variable declared in one function block in another function. To correct this, you can declare the variable using the `m` object reference which gives the variable file scope.

Debugger Message: 'Dot' Operator attempted with invalid BrightScript Component or interface reference.

This message will often coincide with a blank screen. The line number at which the error is detected will be flagged with an asterisk, and the message will provide the name of the file in which the error was detected:

```
020:*       smallexamplesize = smallexample.localBoundingRect()
...
'Dot' Operator attempted with invalid BrightScript Component or interface reference. (runtime error
&hec) in ...pkg:/components/smallexamplescene.xml(20)
020:       smallexamplesize = smallexample.localBoundingRect()
```

This message will appear if a component by that name has either not been created, or an attempt is made to access a component member using an incorrectly spelled component name. Check the backtrace information supplied by the debugger for the component objects and variables used at the time of the error, and note the component objects listed as `invalid`:

```
Backtrace:
#0  Function init() As Void
   file/line: ...pkg:/components/smallexamplescene.xml(20)
Local Variables:
global           rotINTERFACE:ifGlobal
m                roAssociativeArray refcnt=3 count:2
devinfo          bsc:roDeviceInfo refcnt=1
screenresolution roAssociativeArray refcnt=1 count:3
smallexample     Invalid
smallexamplesize <uninitialized>
centerx          <uninitialized>
centery          <uninitialized>
```

Note also the variables that were assigned values from interface functions on invalid component objects will be listed as `<uninitialized>`. Typically in Roku SceneGraph applications, the problem is caused by attempting to create a component object for a component class name that is not in either the built-in node classes, or extended node classes declared in the application package `components` directory. To fix this error, scroll up in the debugger output to the point at which the component object creation error occurred, which will have the following error message:

```
BRIGHTSCRIPT: ERROR: roSGNode: Failed to create roSGNode with type Rectangleexample:
...pkg:/components/smallexamplescene.xml(16)
```

This shows the file and line number where the actual component object creation error occurred. To fix the error, correct the mismatch between the component name and the component object creation function argument. Quite often this is the result of a case mismatch between the extended component name in a `<component>` element, since these names are case-sensitive. Correct the case-sensitive spelling of the component name

either in the component file, or at the point where you attempted to create the component object.

List or Grid Fails to Appear, or First Item is Blank or Missing Information

This often indicates that the **ContentNode** node assigned to the `content` field of the list or grid either does not exist, or was assigned after focus was set on the list or grid. Ensure that the **ContentNode** node has been created successfully at the time it is assigned to the list or grid content field. Then check that focus was set on the list or grid *after* the `content` field is assigned a valid **ContentNode** node. Since you will generally be generating a **ContentNode** node by parsing data from an XML or JSON file downloaded from your server in a **Task** node (or possibly downloaded as "singleton" at the time the SceneGraph application was created in the `main.brs` file and converted), make sure you set the `content` field and focus on the list or grid in this way:

```
sub showvideolist()
  m.videolist.content = m.readVideoContentTask.videocontent
  m.videolist.setFocus(true)
end sub
```

This is a typical callback function that is triggered by the **ContentNode** node being created in a **Task** node (in this case, the event was a new **ContentNode** node assigned to the `m.readVideoContentTask` **Task** node object **<interface>** element `videocontent` field).

Also, if you are having problems with a callback function not assigning a valid **ContentNode** node, carefully check that the field observers were set *before* the **Task** node was configured and launched (but after the **Task** node object was created). For example, for the above example, the **Task** node object should have been created, had the field observers set, configured, and launched, *in that order*:

```
m.readVideoContentTask = createObject("RoSGNode","MetaDataCR")
m.readVideoContentTask.observeField("videocontent","showvideolist")
m.readVideoContentTask.metadatauri = "pkg:/server/videometadata.xml"
m.readVideoContentTask.control = "RUN"
```

This is the sequence to use for all object field observers. If an event in an observed field triggers a callback function *after* the field observer is set, the callback function is likely not to work as expected.

Debugging SceneGraph Node Memory Usage

In a development channel, you can use the special SceneGraph debugging commands to evaluate node memory usage, including potential memory leaks.

A node leak in a channel is simply any node that is no longer reachable by the channel code, yet still exists. Such a node is holding onto memory and other resources unnecessarily. Due to reference counting of nodes in BrightScript, the only way to create true leaks is to create isolated cycles by severing external references to objects which refer to each other. For node cycles, that means that either a node references one of its ancestors (including itself) in a field or one or more nodes reference each other in fields. Note that field types that can hold references to nodes include node, associative array, and regular array.

Some nodes are not leaks since they are still reachable by the code, but they are nevertheless unneeded if there is no intent for the code to use them ever again.

The following **ifSGNodeChildren** methods can be helpful to find both leaked and unneeded nodes.

The **`getRoots()`** method can be used to find nodes and node cycles that are not referenced by any other nodes, and these roots should be inspected carefully to determine whether any of the root nodes are leaks or unneeded.

If they are leaks, they must be part of cycles or they would have been deleted already. To fix these, the code should either not create the cycle, or break the cycle before removing the last code reference to the node. Automatic garbage collection will recover these leaks after the channel exits, but a long-running channel accumulating leaks may begin to slow down or eventually even crash.

If the nodes are merely unneeded, they might not be in cycles but may be simply stored somewhere the code will no longer access even though it could. If so, the channel should be updated to dereference any nodes that are no longer needed. This can be done by setting the variables or fields that are holding the nodes to invalid.

The **`getAll()`** and **`getAllMeta()`** methods can also be used to find nodes that are still referenced by other nodes, but still may be leaks or

unneeded. The set of top-level nodes directly under the `<All_Nodes>` XML element is similar to those returned by the **`getRoots()`** method, and any leaked cycle will have at least one member in this set. Unneeded or cyclically linked nodes can appear anywhere in the node hierarchy, however, so the complete listing should be reviewed when leaks are suspected. A leaf entry that references any node above it in a tree represents a cycle.

Note that determining whether a node is unneeded is not always a simple task, and is up to the intent of the channel code. Outright cyclical leaks can be detected automatically, but since that can be a time consuming operation, it is only performed on channel exit, and the channel should be coded to avoid creating them.

You can also use the equivalent `sgnodes` debugger commands (see Special SceneGraph Debugging Commands).