SceneGraph Threads

Table of Contents

SceneGraph introduced multi-threaded operations to Roku application programming. The following are the basic threads available to a SceneGraph application programmer:

- **Main BrightScript Thread**
  This is the thread that is launched for all Roku applications from the `main.brs` file. For SceneGraph applications, the thread is used primarily to create the scene component object, which starts the SceneGraph render thread. For other applications, this is the only thread for the entire application.
- **SceneGraph Render Thread**
  The render thread is the main SceneGraph thread that performs all rendering of the application visual elements. Certain BrightScript operations and components that might block or modify the SceneGraph in the render thread cannot be used in this thread. Operations and components that might block the render thread can be used in a **Task** node thread. The thread usage of these operations and components is listed in **BrightScript Support**.
- **Render Thread Execution Timeout**

Production Channels	will terminate after 10 seconds if the render thread blocks.
Side Loaded Channels	will timeout in 3 seconds.

- **Task Node Threads**
  By creating and running a **Task** node, you can launch asynchronous BrightScript threads. These threads can perform most typical BrightScript operations.

Thread Ownership

All SceneGraph node objects have a thread owner, which by default, is the render thread. Only components that extend the **Node** node class and the **ContentNode** node class can be owned by a **Task** node or main BrightScript thread when created by those threads. When a node object is stored in a field, the ownership of the node is changed to the node object that contains the field. When a node object is added as a child of another node object, the ownership is changed to the parent node object.

Operations on node objects are executed on their owning thread. If invoked by another thread, the invoking thread must rendezvous with the owning thread to execute the operation.

Thread Rendezvous

In a thread rendezvous, the invoking thread makes a request of the owning thread, and then waits for a response. The owning thread receives the request, processes it, and then responds. The invoking thread then continues with the response as if it had made the call itself. The response appears synchronous to the invoking thread.

Only the render thread may serve a rendezvous. Since **Task** node threads do not have an implicit event loop (though they may have an explicit event loop), they cannot serve a rendezvous. No node object owned by a **Task** node thread is accessible outside that thread. The **Task** node itself is owned by the render thread, so the **Task** node and its fields can only be accessed by rendezvous from other threads, even from the thread launched by the **Task** node itself.

The entire interface to a node object, including field creation, setting, and getting, uses this rendezvous mechanism to ensure thread safety, without having to use explicit locks in the application, and without the possibility of deadlock. The rendezvous mechanism does add more overhead than simple field getting and setting, so SceneGraph application programmers should use it carefully, taking into account the following concerning **Task** node threads.

BrightScript Operations Without SceneGraph Node Objects

If a BrightScript operation does not involve a SceneGraph node object, such as reading a URL, or parsing XML, the rendezvous mechanism is not used. These types of operations can be used in a **Task** node thread without the overhead of the rendezvous mechanism.

Task Node Thread Fully-Owned Node Objects

Task node threads can create node and **ContentNode** node objects that are fully owned by the **Task** node thread. These node objects cannot be parented to unowned nodes, or be set as fields of unowned nodes. Entire trees of the nodes fully owned by the **Task** node thread may be created.

Renderable Node (Group) Objects

You should generally not create renderable node objects in a **Task** node thread. The rendezvous mechanism will be required to create and operate on those node objects. Every field set or get operation on such nodes will require a full rendezvous, and this could impact the performance of your application.

Excessive Rendezvous Operations

You should avoid as many rendezvous operations as possible to ensure maximum performance of your application. It is better to build an entire tree of nodes or **ContentNode** nodes, then pass the tree to the render thread using one rendezvous, than to repeatedly pass each node in the tree as it is created. For field setting and getting, **ifSGNodeField** methods such as **`getFields()`** and **`setFields()`**, which set and get multiple fields at once, should be used rather than several get and set operations.

Task Node Thread Rendezvous Timeout

> As of firmware version 7.5, thread rendezvous no longer timeout and will wait indefinitely. The rendezvous information below is only valid for firmware version 7.2 and older.

A **Task** node thread rendezvous can time out after a few seconds. This can happen if the render thread is very busy, maybe with other rendezvous operations. If a timeout occurs while getting a field, the response will be invalid, which may crash the application if the script is not prepared for the invalid response. During application development, you should check for these timeouts to increase the performance of your published application.

Also note that the render thread can time out while it is executing long linear searches, long iterative calculations, synchronous file accesses, and the like. These types of operations not only slow down the rendering of the UI, but can lead to a rendezvous timeout generating an invalid response, possibly crashing the application. Push computing not directly related to rendering or reacting to the UI into **Task** node threads.

Task Node Objects Ownership

Since **Task** node objects are owned by the render thread, setting **Task** node object fields is done on the render thread, and all observer callbacks on the fields are executed in the render thread. The only case where observer callbacks are executed in a **Task** node object is if the observed field

is in a node object owned by the **Task** node thread.

Re-Running a Task

If a **Task** is already in a given state as indicated by its state field, including RUN, setting its control field to that same state value has no effect. To rerun a **Task**, it must be in the STOP state, either by returning from its function or being commanded to STOP via its control field. At the time a **Task** transitions to RUN, it will look at its functionName field to determine what function to execute. It will transition to the STOP state automatically when that function returns. To get multiple independent threads running from the same **Task** component class, create multiple **Task** instances. There can be several simultaneously executing objects of the same **Task** component class, and each can be running different functions from the component.

Component Global Associative Array

All components have a global associative array designated as `m`, including **Task** node objects. This associative array is local to the component but global within it. For **Task** node objects, this associative array is not shared between threads. The **Task** node `m` is initially owned by the render thread. The **Task** node **`init()`** function then populates the **Task** node object with references in the associative array. On every setting of the **Task** node control field to `RUN`, a new thread is launched, and the **Task** node object associative array is cloned, with the launched thread receiving the original object associative array, and the render thread receiving the clone. Only basic object types are cloned: integers, Booleans, strings, floats, nodes, and recursively, arrays and associative arrays. These are generally the same object types that are copied through the fields, plus functions and timespans. Because of this cloning mechanism, some object references, such as a message port created in **`init()`**, are only passed to the first thread launched from a **Task** node, and not to subsequent threads launched by the same **Task** node.

Task Node Changes In Firmware Version 7.2

In both firmware versions 7.1 and 7.2, within the same **Task** node component object, the `m` for the render thread is distinct from that of any spawned node **Task** node thread. The initialization of a **Task** node object happens in the render thread, so the **Task** node **`init()`** function operates on the `m` for the render thread.

In firmware version 7.1, on transition of a **Task** node control state to `"RUN"`, the `m` for the **Task** node thread was almost empty save for the references to the top and global nodes.

In firmware version 7.2, on each transition of the **Task** node control state to `"RUN"`, the render thread `m` is cloned, the original is passed to the new **Task** node thread, and the render thread retains the clone. Members of `m` which are basic types like integers, Booleans, strings, and floats are cloned. Associative arrays and arrays are cloned recursively. **RoSGNode**s are also cloned, but not recursively, since they are already thread-safe. This is all much like what would happen when an associative array is set to a field.

Since the non-port observer callbacks can modify the render thread `m`, any state changes they make are seen by new threads spawned after such changes.

This means that previously, in firmware version 7.1, only the render thread `m` had state from **`init()`**. In firmware version 7.2, both the render thread `m` and the **Task** node thread `m` have state from **`init()`**. Since the **Task** node thread gets the original, non-cloneable objects, like message ports, they end up in the **Task** node thread `m` . The render thread `m` would no longer have them, and would have to recreate them in its `m` if it wanted to pass such things to the **Task** node thread for subsequent control state transitions to `"RUN"`.

The general idea is that you should expect **`init()`** to operate on the `m` state visible to both the render thread and the **Task** node thread.

In the port form of the **ifSGNodeField** **`observeField()`** method, it can be helpful to create the port, and observe using it in **`init()`**, so that no subsequent field settings are missed due to race. Transferring the original to the **Task** node thread rather than the clone allows this to work since the port is not cloneable.