Time to First Pixel Performance

The term "time to first pixel" refers to the amount of time it takes RfK and Katana to traverse the scene graph and generate the RIB necessary for RenderMan to start rendering pixels. Contrary to some popular beliefs it **is** possible to send RIB to PRMan from Katana in parallel using RenderMan RiProcedurals. This has the potential to greatly reduce the amount of time it takes to spin up a render however some care must be taken to set up the scene in an optimal way in order to take best advantage of this feature. In a nutshell:

- The global setting **prmanGlobalStatements.plugin.multiThreaded** must be enabled (it is off by default in 21.x).
- The setting **forceExpand** must be disabled at any locations that might request a new thread.
- Your scene must have a valid **bound** attribute at strategic locations.
- You must enable more than one thread in PRMan (see Thread Control in Katana).

The **multiThreaded** setting is used to inform PRMan that the procedurals generated by RfK are reentrant and eligible to be placed in a new thread. With **forceExpand** enabled no procedurals will be generated in the first place; likewise RfK cannot generate procedurals without bounded sections of the scene.

Procedurals

Parallel generation of RIB can only happen through the use of RiProcedurals. (Note: we are not talking about prman rendering, we're talking about the parallel execution of Ri calls which inform the renderer what to render and how to render it). As RfK is traversing the scene graph it will generate a procedural when it hits a bounded group or instance (locations of type **assembly** and **component** are considered to be a group in this context). Specifically, a procedural will be generated and a new thread spawned only if *all* of the below requirements are satisfied:

- location is an **instance source** or is of type **group**, **assembly**, or **component**.
- location has a valid **bound** attribute.
- **forceExpand** = false.

If the setting **forceExpand** is enabled at any location then that location **and all of its children** will be evaluated immediately without the use of procedurals. This will effectively disable parallel RIB gen on that portion of the scene graph. If **forceExpand** is set true at "/root" then the entire scene graph will be traversed in a single thread.

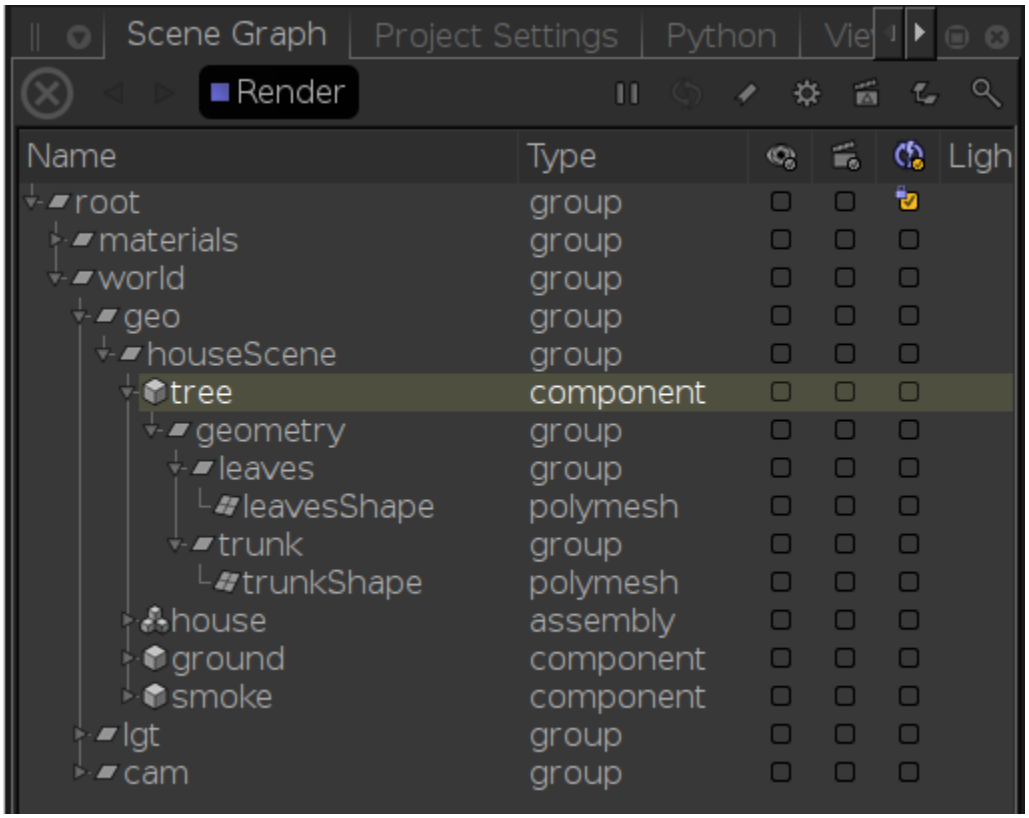
PRMan procedurals are not reentrant by default. Each procedural generated within RfK needs to be explicitly marked as reentrant. With the global setting **multiThreaded** RfK will mark all procedurals reentrant or not. Currently this is not available as a per-object setting.

Relevant Attribute Settings

Below are details regarding the settings used in determining the multi-threaded RIB gen behavior.

Attribute	Default	Effect	Details
prmanGlobalStatements. **plugin.multiThreaded**	0	Turning this on enables multi-threaded procedurals in PRMan.	The value of this setting is directly translated to the value of RiOption "procedural" "reentrant" which is set before calling RiProcedural
forceExpand	0	Turning this on disables procedurals entirely for the location and all of its children.	This setting is not exposed in PrmanGlobalStatements or PrmanObjectStatements. It must be set directly in the Node Graph.
type	n/a	Only locations of type **group**, **assembly**, **component**, and **instance source** are eligible for processing via procedural.	
bound	none	Locations without a valid **bound** attribute cannot be processed via procedural and therefor cannot be evaluated in parallel with other portions of the scene graph.	PRMan requires bounds in order to evaluate when a procedural can be "cracked". The evaluation is based on shader rays; when a ray intersects the bounds of a procedural PRMan knows that the contents must be evaluated.

Using the scene graph from the Katana "houseScene" demo data (`$KATANA_HOME/demos/houseScene_data/houseScene.xml`) we can demonstrate the interactions between the various attributes and how they affect the parallel RIB gen behavior within RfK:



prmanGlobalStatements.plugin.multiThreaded

If this is set to false then none of the other settings matter, RIB gen will be serial.

bound

1. If none of the locations have a valid **bound** attribute then RIB gen will be serial.
2. If only **root** has a valid bound it will be ignored as the top-level ("main") procedural is always opened immediately. RIB gen will be serial.
3. If any single group has bounds then (assuming **forceExpand** is false) a new procedural will be spawned when that location is traversed. When PRMan cracks that procedural a new thread is potentially spawned for that location and its children. RIB gen will happen in two threads: the main thread and the secondary thread from the cracked procedural.
4. If each child of **/root/world/geo/houseScene** has a valid **bound** attribute then they will each spawn a new procedural (again, assuming **forceExp and** is off). Since there are four bounded child locations RIB gen can happen in as many as four threads, however if one procedural finishes before another is cracked then the thread will be reused for the next group.
5. If every group in the scene has bounds then RfK will recursively generate procedurals through the traversal. This is not necessarily a good thing; there is a certain amount of overhead for each thread and if there is not enough work for the threads they have the potential to spin or thrash resulting in performance that might even be worse than single-threaded.

Some amount of experimentation and thoughtful design needs to go into the scene organization for optimal time-to-first-pixel. There is no hard and fast rule, however we do have some hints:

- Due to the bucket rendering nature of PRMan, users should work to create well stratified bounding boxes. Parallel expansion will only happen if there are different buckets hitting different bounding boxes. If all of the buckets are stopped on one bounding box, you will only get one procedural expansion while the threads rendering the other buckets will block until it is done.
- Beware of using the random bucket order. While it may seem like it would improve concurrency it can have unexpected side effects (e.g. on the texturing system) which can actually slow renders down.
- Setting bounds at **every** location is not recommended; as mentioned above this has the potential to overwhelm the render with threading overhead.
- Every scene at every studio is different. What works for one shot or sequence may be a terrible setup for another. The only way to know for sure is to profile and evaluate.

forceExpand

If **forceExpand** is set true at **/root** then RIB gen will be serial regardless of which locations have bounds. If bounds are set up throughout the scene (option 5 above) then forceExpand can be used to disable the bounds at specific groups. Once forceExpand is set at a location it will be passed down through to that location's children, effectively disabling any parallel RIB gen for that scene graph sub-hierarchy.

Tuning a Katana Scene for RenderMan

So the gist of the story is "yes, you can do this in parallel but it takes a bit of effort to do right". So then, how does one tune their scene? By evaluation and experimentation.

1. Evaluate the scene to get a general idea where logical places for decomposition would be. For example, if you're rendering a city then it's better to split it up by blocks so that each block could potentially be rendered independently. Don't put all the windows into one hierarchy as that would be a single procedural and, once cracked, would block all other threads from rendering the buildings in that bound.
2. Ensure that **forceExpand** is off (this could potentially be set anywhere in the scene).
3. Ensure that **multiThreaded** is on (this would only be set in **prmanGlobalStatements**).
4. Ensure that the thread count is greater than 1 (see Thread Control in Katana).
5. Render!
6. Confirm you're using multiple threads: watch the system monitor during the render (and before the first pixel) and/or run a profile using perf or vtune.
    a. if you're still seeing most of your work happening on one thread it's possible you're bounded by some serialized operation. Large OSL network compilation is a common example; check with profiling. Another culprit could be hidden Op costs; Katana versions 2.2 and greater have a profile option which tracks Op processing.
7. Evaluate the time-to-first-pixel. On Linux RfK will approximate this time and print out to the console its value:
   `[RfKProfile]: time-to-first-pixel (approx) = 4.093869s`
8. Alternatively it is often useful to set up the scene to render with a single sample and log the time it takes to finish the render. You'll get the feel of how much time is spent traversing vs how much time is spent rendering the one sample.

Now you've got a baseline. Run it again with a different number of threads. Increase or decrease to get a feel for where it starts falling over. Ideally your time continues to decrease as your threads increase but in reality you may see it get worse with more threads (evaluate if there is enough work in each procedural to warrant a thread or are the threads spending all their time spinning, waiting for more work).

Next start experimenting with placement of bounds to give the renderer more or less work depending on how you see the threads scale.

Lather. Rinse. Repeat.

Performance Profiling Tools

Details of performance profiling are beyond our scope, here are some links for information elsewhere:

- Linux profiling with perf
- Intel VTune
- Katana Op Cook Profiling