Checkpointing and Recovery

- Checkpointing
- How to Specify
- Temporary Files While Writing
- Deep Data
- Skipping Deep EXRs or DDC files
- Elapsed Time
- Recovery
- System Commands
- Signals
- Workflow

Checkpoint at 30 seconds
Checkpoint at 60 seconds
Checkpoint at 90 seconds

PRMan has the ability to resume interrupted renders. How this works depends on the mode that it's running in. Non-incremental renders to TIFF and OpenEXR images can always be recovered. Incremental renders to OpenEXR images can also be recovered, but only if the checkpointing option was used.

Checkpointing

Checkpointing is specific to the incremental rendering mode when doing batch rendering. In incremental mode, the renderer makes repeated passes over the image, refining it a bit more with each pass. While the image will be quite noisy during the initial passes, it is usually sufficient to give an impression of how the final image will look.

Checkpoints are snapshots that save the state of the image as the renderer works on it. While view-able as ordinary images, these are also slightly larger than usual because they embed extra state that the renderer needs in order to recover the render. If the render is interrupted or fails for some reason, the renderer can resume the render from the last checkpoint image. If instead, the render finishes then the extra state will be removed when writing the final version of the image. There are two main ways to produce checkpoints:

The first of the these is a periodic checkpoint. This can be specified with an interval measured either in a number of increments (i.e., passes over the image), or by the elapsed wall clock time. For example, you could ask the renderer to write checkpoints with an interval of100i, meaning every 100 increments and it will update the images on disk with the state of the render on the 100th, 200th, 300th increment and so forth.

Alternatively, you could set the interval to 300s and the renderer will update the image approximately every five minutes after it starts work on the frame. This time includes the renderer startup time, such as parsing RIB, cracking procedurals, and building ray tracing acceleration structures. As a result, there may be fewer increments before the first checkpoint than between later checkpoints. For convenience, the time-based interval can also be specified with suffixes of s, m, h, or d for seconds, minutes, hours, and days respectively and these may be combined. For example, intervals of 360s, 6m, 5m60s, and 0.1h are all equivalent. Instead of a suffix, you can also just specify a positive number and time in seconds will be assumed, while a negative number will be interpreted as the number of increments.

There's a balance to be had on the frequency of checkpoints. More frequent checkpoints mean less work lost if the render fails. However, writing checkpoints too frequently can also reduce the efficiency of the renderer. Attempting to write a checkpoint on every increment or every second is generally not a good idea.

The second way to produce a checkpoint is by placing a limit on the total render time (specified with the same notation as for periodic checkpoints). The render will proceed as normal unless it reaches the time limit, at which point it will finish its current increment, write the checkpoint image and then stop rendering the frame. If there are multiple frames to render then it will simply go on to the next at this point; each frame can have its own time limit.

Both methods for generating checkpoints can be used together. For example, it is possible to request a checkpoint be written every 100 increments until 15 minutes has elapsed. At that point, any periodic checkpoints that were written will simply be replaced with the final checkpoint when exiting.

Note that checkpointing is designed for batch rendering to images on disk. Renders to a live framebuffer such as "it" are already updated on-the-fly as the render proceeds. Of the built-in display drivers, currently, only the TIFF and OpenEXR display drivers support checkpointing. You may notice that the files produced include a ~w labeled channel. This is data stored to resume a render from this file.

How to Specify

In RIB, checkpoints can be specified via an Option or by passing the optional `-checkpoint` argument to `prman`.

```
prman -checkpoint t[,t]
```

The above command is seen as the interval given and an optional exit time (the square backets).

You can use several different types of intervals and may be combined:

- 60s - this is 60 seconds
- 60m - this is 60 minutes
- 60h - this is 60 hours

- 60d - this is 60 days
- 60i - this is 60 increments (incremental frame rendering, this is ignored if you're rendering to tiles/buckets)

You can combine these as needed, for example: -checkpoint 1h30m would create a checkpoint every hour and 30 minutes

The options for exitat are the same and can even be provided without needed a checkpoint written, in which case you've just told the renderer you want to stop a render and write out the result at a certain time. Below we write out a final result and exit at 30 minutes

```
prman -checkpoint ,30m
```

It is possible to maintain a checkpointed "final" image using either an .ini setting or a RIB option, with the later overriding the former. If neither is present it defaults to off. When set, this prevents removing the extra channels and the checkpoint tag when writing the final image for the render. The final image will essentially be just another checkpoint, rather than a slimmed down image. This means that once your image has reached the quality you've set and it completes, it can always be restarted by the user later:

```
/prman/checkpoint/asfinal [bool]

Option "checkpoint" "int asfinal" [0|1]
```

Temporary Files While Writing

When writing checkpoint files, we now write them to temporary files with a .part extension added. E.g., foo.exr will be written to foo.exr.part and kittens.ddc will be written to kittens.ddc.part during the checkpointing process.

Only after all of the files to write for a checkpoint have been written will these files be renamed to strip the .part extension and overwrite the previous checkpoint. This step happens immediately before the postcheckpoint command is run, if any.

Though not strictly atomic, this renaming is done in as small of a window of time as the OS permits in order to avoid mixing new checkpoint files with old. If the renderer is killed or dies for some reason while checkpointing, there may be some .part files left over. Deleting these should safely leave the previous checkpoint intact. Note that looking for these .part files is one way of detecting whether the renderer was killed during a checkpoint. *Note too, that this new behavior means that the peak disk space used by checkpointing is now doubled.*

Deep Data



When using checkpointing a deep EXR while using adaptive sampling, all of the AOVs considered by the adaptive sampler (e.g., normally the beauty) must be written as well to a shallow EXR. Otherwise, the checkpoint will not be recoverable.

We support Deep Data Checkpointing (DDC files) when rendering to Deep EXR formats. When using this feature you will not only see a "shallow" EXR written to disk for each checkpoint but *also* the DDC, which uses lossless compression, file for recovering the deep data. Since checkpointing often leaves multiple files on disk and deep data can be expensive to store, there is an option to compress the resulting deep data.

Compression can be set via a rendermn.ini option:

```
/prman/checkpoint/ddccomplvl  7
```

The default level is 7. The typical range is 1 to 20 where higher numbers increase the compression at the cost of speed.

The default level of 7 is chosen as a balance between size and performance. Values above 7 will decrease the performance of writing the files with diminishing returns on compression as values go higher. For this reason we do not recommend going above 7 unless your storage capacity is *severely* limited. A value of 1 will result in large files but the best performance and is ideal when storage for DDC files is plentiful.

To reiterate, note that this is *lossless compression,* unlike the legacy deepshadowerror rendering option. The choice of the ddccomplvl has no effect on image quality after recovery and is purely a question of checkpoint time performance versus disk space.

Skipping Deep EXRs or DDC files

When recovering checkpoints involving deep data, any deep EXRs are ignored and only the DDC files are used. If you do not need to view the deep EXRs from a checkpoint, you can now save both disk disk space and the time spent on processing and I/O to generate these deep EXRs by disabling them. To do this, set the following in your rendermn.ini to anything “truthy” (e.g., true, yes, on, or 1):

```
/prman/checkpoint/skipdeepexr true
```

Even with this option set, we still write the deep EXRs when the render finishes normally (i.e., either hitting max samples or stopped by the adaptive sampler) since you cannot directly view or composite with the DDC files.

Also note that this option only applies to deep EXR files. All shallow EXRs are still written out with checkpoints as normal, albeit with the usual checkpointing channels and metadata.

The counterpart to all this is that the renderer now also normally saves time by skipping the writing of DDC checkpoint files when the render finishes and it is writing the final deep EXR image. You can use the existing asfinal option (either Option ”checkpoint” ”int asfinal” in the RIB or /prman/checkpoint /asfinal 1 in the rendermn.ini) to prevent this and have it write the DDC files anyway.

Finally, note that we continue the prior behavior of not deleting the last DDC files, even when the renderer finishes normally and asfinal is not set. At that point, the DDC files correspond to the checkpoint just before finishing and become stale data. This is something we may change in the future, depending on feedback, but remains the behavior for now.

Elapsed Time

This doesn’t strictly pertain to deep checkpoints, but shallow EXR checkpoint images now contain a new EXR attribute, checkpointElapsed. This represents the elapsed time in seconds since the renderer started until checkpointing the current image begins. It does not include prior rendering time if this process was started from a recovered checkpoint.

Recovery

Recovery of an interrupted render is enabled by passing the `-recover 1` option to `prman` when starting a render. RenderMan will then load the scene as normal but rather than start from scratch and overwrite the existing images it will examine them to determine where it was interrupted. Ifsuccessful, it will continue from close the point where it left off. If instead the images were finished, missing or don't match the current scene or each other for some reason, it will silently start from scratch.

It is not necessary to recover a render on the same render machine that it began on. So long as a RenderMan process can still find the images specified by the scene file and they are in a consistent and recoverable state it can resume the render.

Recovery can also be paired with the checkpointing options described above. In the case of a time limit, this basically serves as an extension to the original time limit. Incremental renders can be broken into an arbitrary number of time slices this way. Note that each recovery requires the scene and all of its assets to be reload, however, so care should be taken with this. Additionally, this feature is only supported for OpenEXR files.

System Commands

RenderMan can call a system command after a checkpoint using `Option "checkpoint" "string command"`; this can also be specified through the `rendermn.ini` file with `/prman/checkpoint/command`. If system calls are enabled, then after a checkpoint has been written, the specified command will be called. This is synchronous; the rendering threads are quiescent while this runs and will not resume again until the process returns, avoiding possible race conditions if the command takes a while.

A limited amount of substitution is available. The token `%i` will be replaced with the current increment, zero-padded to 5 digits. The token `%e` will be replaced with the elapsed time in seconds, zero-padded to 6 digits. The token `%r` will be replaced with the reason for this update to the checkpoint files (either completely `finished`, `exiting` early due to exitat option, or a normal `checkpoint`). Literal % characters may be inserted with `%%`.

Signals

On Linux and OSX, prman will listen for the HUP and USR1 signals. If checkpointing was enabled for the render then when prman receives a HUP signal it will attempt to write a checkpoint and continue rendering. If it receives the USR1 it will write a checkpoint and then begin an orderly shutdown. This is similar to the behavior with the internal checkpoint interval and exitat parameters, but allows these to be externally triggered such as by a pipeline queuing system. The Windows operating system does not support these signals, so this feature is not available on that platform

Workflow

While the most basic use for checkpointing and recovery is simply to be able to resume a render in case of a failure, checkpointing with incremental renders enables some powerful new workflows.

Using the time limit option, for example, on a sequence of frames allows the creation of draft animations. A rough version could be rendered quickly, viewed in a playblast and then continued to final rendering after approval. The render time already expended on the rough version would give a head start on the final version. Carefully balancing the time limit with the workload to render and the available render cycles could ensure that the playblast is available by a given deadline (e.g., rendering overnight for morning viewing).

Another possibility is gentler studio-wide time limits on render farms. Rather than simply killing renders that exceed a time limit such as 24 hours, setting it as the default time to exit with a checkpoint means that renders that go over the time budget could be reviewed and then either accepted in their current state or resumed.