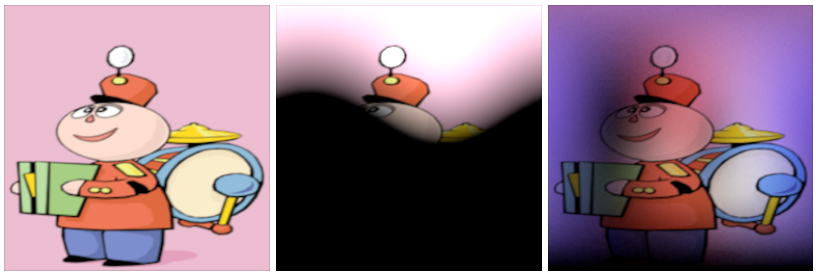
Baking in RenderMan RIS

RenderMan Application Note
Pixar Animation Studios

February, 2020



Abstract

In this application note, we describe two techniques for baking in RenderMan RIS. The first technique has been available for several years: baking of pattern network evaluation results. The second technique is new: baking integrator results such as direct and global illumination. The baking output can be 2D images or 3D point cloud files. We describe the syntax and provide examples. Common uses of baking is precomputation of pattern graphs and generation of lightmaps for render optimization, interactive viewing, and games.

1 Introduction

The term *baking* is common technical jargon; we use it to describe saving of intermediate rendering computations to a file for later use in e.g. final rendering.

RenderMan had a built-in baking feature all the way back in 2004: RenderMan 12.0 introduced a bake3d() function in RSL (RenderMan Shading Language) where one could specify a pointcloud filename, the position and normal of the point, and the names and values of the data to be baked. The baked data could be rather arbitrary: color results from evaluating a texture, ambient occlusion, ray-traced reflections, global illumination, etc.

Some of this ability was lost in the transition from the traditional Reyes pipeline to the modern RIS path-tracing framework. In RenderMan RIS, it has only been possible to bake values in the pattern evaluation network (using the PxrBakeTexture and PxrBakePointCloud pattern nodes). But up until now, it has not been possible to bake illumination values.

In this application note, we present a new, extended implementation of baking that allows baking of integrator results. The main examples are direct illumination from the PxrDirectLighting integrator and global illumination from the PxrPathTracer integrator. It is also possible to bake results from simpler integrators, for example PxrVisualizer and PxrOcclusion.

The following two sections describe baking of pattern node values and integrator results. The baking output can be 2D images or 3D point clouds.

2 Pattern baking

Pattern baking is useful for precomputing pattern graph results so they can be read as texture lookups. This can significantly speed up final rendering if pattern evaluation is a large portion of the render time. These savings become even more significant when the baked results are reused over many frames in a sequence. As such, pattern baking can be a useful pipeline tool for reducing render times.

To put RenderMan in pattern baking mode, select the bake hider with bakemode "pattern":

```
Hider "bake" "string bakemode" "pattern"
```

Bakemode can be "pattern", "integrator", or "all"; the default is "pattern".

2.1 2D pattern baking: images

Baking of 2D pattern results can be done with the PxrBakeTexture pattern node. During ray-trace rendering, this pattern behaves as a pass-through shader. During bake rendering, this pattern operates as the location that defines all aspects of a bake output such as the display driver, output filename, and image resolution. Additionally, this pattern specifies the texture coordinates that are used to unwrap the geometry source into a 2D manifold. The most common texture coordinates are "st", but other varying and facevarying float primvars can be used.

Here is an example of baking a Worley noise pattern to the 2D image pattern2d.png:

```
Pattern "PxrWorley" "worley"
Pattern "PxrMix" "mix"
  "color color1" [1 0 0]
  "color color2" [0 1 0]
  "reference float mix" ["worley:resultF"]
Pattern "PxrBakeTexture" "pattern2d"
  "reference color inputRGB" ["mix:resultRGB"]
  "string filename" ["pattern2d.png"]
  "string display" ["png"]
  "string primVar" ["st"]
  "int resolutionX" [512]
  "int resolutionY" [512]
Bxdf "PxrDiffuse" "diffuse"
  "reference color diffuseColor" ["pattern2d:resultRGB"]
Sphere 1 -1 1 360
  "varying float[2] st" [0 0 1 0 0 1 1 1]
```

Figure 1(left) shows the resulting image.

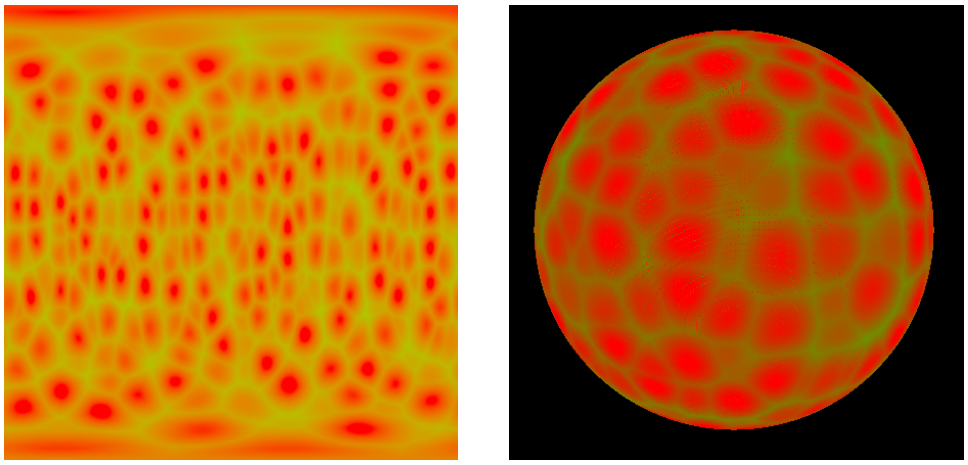


Figure 1: Baked 2D and 3D pattern outputs from the example scenes. (3D view generated with the interactive point cloud viewer 'ptviewer'.)

2.2 3D pattern baking: point cloud files

Baking 3D pattern results can be done with the PxrBakePointCloud pattern node. Similar to PxrBakeTexture, this pattern behaves as a pass-through shader during raytrace rendering and as an output location during bake rendering. 3D outputs do not need to be unwrapped into a 2D manifold and do not require any special texture coordinates. Instead, a shading density and output coordinate space are provided.

Here is an example of baking the same Worley noise pattern, but this time to the 3D point cloud file pattern3d.ptc:

```
  Pattern "PxrWorley" "worley"
  Pattern "PxrMix" "mix"
    "color color1" [1 0 0]
    "color color2" [0 1 0]
    "reference float mix" ["worley:resultF"]
  Pattern "PxrBakePointCloud" "pattern3d"
    "reference color inputRGB" ["mix:resultRGB"]
    "string filename" ["pattern3d.ptc"]
    "string coordsys" ["object"]
    "float density" [100]
Bxdf "PxrDiffuse" "default"
    "reference color diffuseColor" ["pattern3d:resultRGB"]
Sphere 1 -1 1 360
```

Figure 1(right) shows the resulting point cloud.

2.3 Ptex

RenderMan also supports Ptex outputs. Ptex textures do not require explicit texture coordinates. To generate a Ptex output, the PxrBakePointCloud pattern can be used with the coordsys parameter set to "ptex". The bake geometry must contain the same uniform "__faceindex"

primvar typically required by Ptex. This will generate a point cloud (.ptc) output that can then be converted to Ptex (.ptx) with the 'ptxmake' stand-alone utility.

3 Illumination baking

To put RenderMan in integrator baking mode, select the bake hider with bakemode "integrator":

```
Hider "bake" "string bakemode" "integrator"
```

Specify an integrator such as e.g. PxrDirectLighting or PxrPathTracer the usual way, for example:

```
Integrator "PxrDirectLighting" "direct"
  "int numLightSamples" 16 "int numBxdfSamples" 16
```

or

```
Integrator "PxrPathTracer" "spt" "int maxIndirectBounces" 4
  "int numLightSamples" 16 "int numBxdfSamples" 16
  "int numIndirectSamples" 256
```

Other integrators such as PxrOcclusion or PxrVisualizer can also be used.

3.1 2D illumination baking: images

The name(s) of the baked image file(s) is/are specified with a Display line. Baking uses the same display drivers as raytrace rendering, including openexr, tiff, png, and many more. Usually one will want to bake to a separate image for each object. This can be done by encoding the Display filename with string "wildcards" that get substituted with actual filenames depending on Attributes on the objects. For example:

```
Display "<user:filename>.exr" "openexr" "Ci"
```

In this case, the wildcard <user:filename> is replaced by the corresponding user attribute specified for each object. For example:

```
Attribute "user" "string filename" "box1_global"
```

which then gets substituted into the Display filename. Alternatively, the filename can be specified with other attributes such as <identifier:name>. No image will be baked for an object if the wildcard substitution fails or if the filename for that object is the empty string.

The resolution of the baked 2D images is determined by the standard Format description. For example:

```
Format 256 256 1
```

As an example, we'll bake 2D textures of global illumination in a Cornell box with two spheres. The box consists of five squares, each with a separate texture. One of the two spheres is purely specular so we do not bake on that. Figure 2(left) shows the baked images for the five faces of the box and for the diffuse sphere.

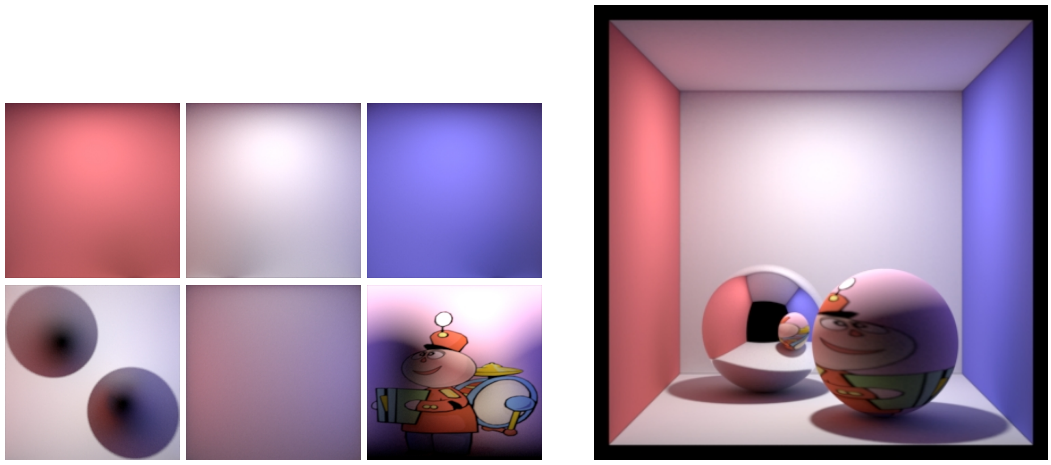


Figure 2: Left: six baked 2D global illumination textures (left, back, right, bottom, and top of box; diffuse sphere). Right: image with global illumination rendered using the baked 2D textures.

After baking is done, the next step is to run 'txmake' on each image to create a MIP-mapped cache-friendly texture in .tex format. These textures can then be read during rendering using e.g. OSL's texture() function. Figure 2(right) shows the Cornell box and diffuse sphere shaded using the precomputed illumination textures (while the reflection in the specular sphere is rendered with ray tracing).

Similar to pattern baking, you may specify which texture coordinates are used for the output manifold. The default texture coordinates are "st". Additionally you may choose to invert the "t" direction. The invert default is true.

```
Hider "bake"
  "string bakemode" ["integrator"]
  "string[2] primvar" ["foo" "bar"]
  "int invert" [0]
```

3.2 Atlas images

Multiple outputs can be generated if the texture coordinates for the selected output extend outside the [0,1) range. In this case, the filename should contain a special atlas wildcard to generate a unique output name for each image tile.

```
Attribute "user" "string filename" "box_direct.<UDIM>.tif"
```

In this case, the (s,t) texture coordinate of each baked point gets mapped to a UV tile such as 1001. The supported atlas wildcards are:

- <udim> or <UDIM> (i.e. Udim tile: 1001, 1002, 1003, ...)
- <u> and <v> (i.e. tile indices starting at zero: 0, 1, 2, ...)
- <U> and <V> (i.e. tile indices starting at one: 1, 2, 3, ...)

3.3 3D illumination baking: point clouds

To bake illumination in the entire scene to a single point cloud file, simply specify the file name in the Display line:

```
Display "box_direct.ptc" "pointcloud" "Ci"
```

Or to specify a separate point cloud file for each object:

```
Display "<user:filename>.ptc" "pointcloud" "Ci"
```

The dicing and hence point cloud density can be controlled using existing dicing controls such as "worlddistancelength":

```
Attribute "dice" "float worlddistancelength" 0.02
```

As two examples, we'll bake 3D point clouds of direct and global illumination in a Cornell box with two teapots. Figure 3 shows two views of a single point cloud with direct illumination on the box and the two teapots (baked with the PxrDirectLighting integrator).

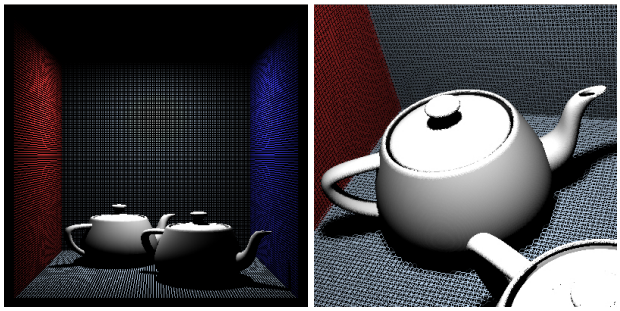


Figure 3: Two views of a direct illumination point cloud for Cornell box with teapots. (Views generated with the interactive point cloud viewer 'ptviewer'.)

Figure 4(left) shows two point clouds with global illumination (baked using the PxrPathTracer integrator): one for the box and one for a teapot. Finally, we can render an image using these global illumination point clouds as textures. The 3D point clouds can be read in an OSL shader using the texture3d() function. Figure 4(right) shows such an image, where the box faces and the diffuse teapot are rendered using the baked illumination data in the point cloud files. (For a more cache-friendly 3D texture format, the point clouds can be converted into brick maps using the 'brickmake' utility program.)

3.4 Multiple outputs, arbitrary output variables, LPEs

Similar to raytrace rendering, *light path expressions* (LPEs) may be used to isolate different components of global illumination while baking. The data that can be baked is very general: if an integrator can splat it, then the illumination baker can write it to a display driver. As a simple example, here we declare direct diffuse and indirect diffuse outputs:

```
DisplayChannel "color directDiffuse" "string source" ["color lpe:C<RD>[LO]"]
DisplayChannel "color indirectDiffuse" "string source" ["color lpe:C<RD>.+[LO]"]
Display "<identifier:object>DirectDiffuse.tif" "tiff" "directDiffuse"
Display "+<identifier:object>IndirectDiffuse.tif" "tiff" "indirectDiffuse"
```

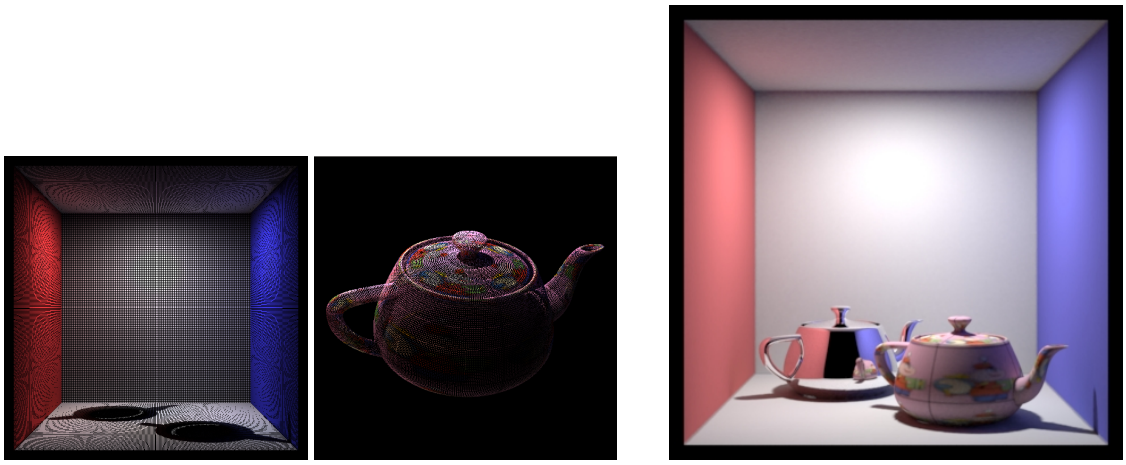


Figure 4: Left: baked global illumination point clouds (box and diffuse teapot). Right: image rendered using baked 3D point clouds with global illumination data.

Similarly, the albedo user lobe may be rendered with an albedo LPE:

```
Option "lpe" "string user2" ["Albedo,DiffuseAlbedo,SubsurfaceAlbedo,HairAlbedo"]
DisplayChannel "color albedo" "string source" ["color lpe:overwrite;C(U2L)|O"]
Display "+<identifier:object>Albedo.tif" "tiff" "albedo"
```

Figure 5 shows albedo, direct diffuse, and indirect diffuse for the Cornell box with two spheres.

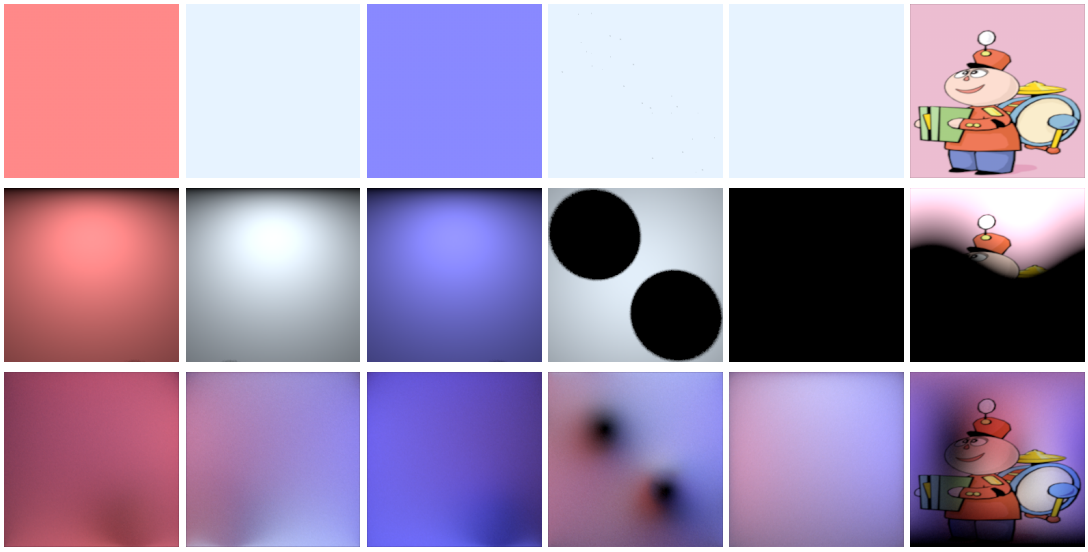


Figure 5: Baked LPEs. Top row: albedo. Middle row: direct diffuse. Bottom row: indirect diffuse.

4 Known limitations

Current limitations:

- Cannot bake curves, points, or volume geometry. RenderMan no longer dices these in the traditional way, and so cannot easily bake point on them. Additionally, these geometry types are not parameterized suitably for 2D baking.
- Cannot bake results of the PxrVCM integrator. The PxrVCM integrator is progressive (incremental) and that does not fit in with the current baking framework.
- Bake outputs do not evaluate sample or display filters. Baking does not support shadow outputs which rely on those filters.
- Only single-channel outputs per display (not compatible with packed displays). The bake renderer can only send single channel RGBA data to individual display drivers. Use multiple display drivers to output multiple signals.
- No breakpointing / stop-and-resume during baking.
- There is no separate irradiance output. As a workaround, we suggest baking illumination and albedo; dividing the illumination by the albedo gives the irradiance.

5 Conclusion

RenderMan has been extended with the ability to bake integrator results, in addition to the previously supported pattern results. The outputs can be 2D images (including udim and ptex) or 3D point clouds. We have shown examples of syntax and uses of these baking features.