Procedural Primitives

Introduction

RenderMan procedural primitives (procprims, for short) are user-provided subroutines that can be called upon to generate geometry (or issue other Ri requests) during the process of rendering. The advantage of procprims over concrete (RIB-based) primitives is that they can represent complex geometry very efficiently. Procprims can produce incredible geometric complexity from a small number of inputs, and we can defer this *data amplification* until the renderer is prepared to handle it. This may make it possible to render much more complex scenes than would be possible with a non-procedural representation. Procprims can also be thought of as units of memory management. The renderer can elect to unload all concrete geometry associated with a procprim knowing that the geometry can be regenerated should there be further need for it.

To describe procedural primitives to RenderMan, use either RiProcedural or RiProcedural2. Both entrypoints have the same underlying rendering characteristics, so developers should choose the interface that best suits their needs. They both accept callback parameters that are invoked at a later point in the rendering process. The primary distinction between the calls relates to the means by which opaque, plugin-specific data is represented. The original call characterizes custom data with an opaque pointer (a string for RIB) and as a consequence must request cleanup services from the plugin. The newer call uses standard RI parameterlists to convey the custom data and thereby needs no additional cleanup support from plugin.

In a direct-linked setting it's possible to provide the required callbacks as arguments to the RiProcedural/RiProcedural2 calls and you're done. In a RIB-based pipeline this can't work since callbacks are characterized by a function pointer, which has no meaning outside of the running process. The requirement is to encapsulate a reference to your procedural primitive and its parameters for *deferred*execution. RenderMan supports several built-in procprims to facilitate this very requirement. These are intended to work well in both deferred and immediate modes of operation.

> *RiProcDelayedReadArchive, RiProc2DelayedReadArchive*
>
> > *A large part of the value of procedural primitives derives from the deferred and reloadable nature of the data amplification. These procprims simply defer the loading of RIB files and require no additional custom plugins. As such Delayed Archives are probably the most common form of procprim.*
>
> *RiProcDynamicLoad, RiProc2DynamicLoad*
>
> > *Procedural primitive subroutines are implemented in a plugin (.dll, .so) that will be loaded by the renderer when it reads the RIB file. The RIB file must include the name of the plugin as well as its parameters. Once the plugin is loaded the routines are bound to internal primitives and these will be invoked later when the plugin's services are required.*
>
> *RiProcRunProgram*
>
> > *External programs can be called upon to generate RIB following the procedural geometry pattern. TheRunProgram interface models the invocation of the standard subroutines as simple message over a basic interprocess communication (IPC) channel. Custom code executes in response to an IPC message and RIB is produced for consumption by the renderer, again via an IPC channel.*

Using Existing Procedural Primitives
Delayed RIB File Reading

The simplest of the procprims is **Delayed Read Archive**. The existing interface for "including" one RIB file into another isRiReadArchive. Delayed Read Archive operates exactly like RiReadArchive, except that the reading is delayed until the procedural primitive bounding box is reached, unlike RiReadArchive which reads RIB files immediately during parsing. The advantage of the new interface is that since the reading is delayed, memory for the read primitives is not used until the bounding box is actually reached. In addition, if the bounding box proves to be off-screen, the parsing time of the entire RIB file is saved. The disadvantage is that an accurate bounding box for the contents of the RIB file is required, which was never needed before.

In RIB, the syntax for the Delayed Read Archive primitive is:

```
Procedural "DelayedReadArchive" [ "yourfilename" ] [ bound ]
```

-or-

```
Procedural2 "DelayedReadArchive" "SimpleBound"
    "string filename" "yourfilename"
    "float[6] __bound" [xmin xmax ymin ymax zmin zmax]
```

In place of "yourfilename" you must provide a pathname for the desired RIB file. If your RIB archive is referred to with a relative path we search the paths given by the archivepath setting. As with all RIB parameters that are bounding boxes, the bound is an array of six floating point numbers measured in the current object space.

In C, the syntax for calling the Delayed Read Archive primitive is:

```
RtString *data;
RtBound bound;
RiProcedural(data, bound, RiProcDelayedReadArchive, freedata);
```

data is a pointer to a single RtString which contains the filename (ie. data is a char **). bound is a pointer to six floats which contain the bounding box. freedata is the user free method which frees data. data could be a global variable, but almost certainly cannot be an automatic (local) variable, as it must persist until the renderer calls the free routine. In practice, data would probably be malloced, and freedata would merely free() data.

And here's the Procedural2 variant:

```
RtBound bbox;
RtString filename[1];
filename[0] = "yourfilename.rib";
RiProcedural2(RiProc2DelayedReadArchive, RiSimpleBound,
              "string filename", filename,
              "float[6] __bound", bbox, RI_NULL);
```

RiSimpleBound is the name of a built-in bound subroutine that searches the parameterlist for conventionally named parameters. This paves the way for more procedural bounding callbacks, but for most purposes RiSimpleBound should suffice.

Procedural Primitive DSOs

A more efficient method for accessing subdivision routines is to write them as dynamic shared objects (DSOs), and *dynamically load* them into the renderer executable at runtime. In this case, developers write a subset of the subdivision, bound and free routines exactly as required in the direct-linked setting. They are compiled with special compiler options to make them runtime loadable, and you specify the name of the shared .so, .dll file in the RIB file. The renderer loads the DSO the first time it is needed to subdivide a primitive, and from then on it is called as if (and executes as fast as if) it were statically linked. When referring to a DSO keep the following in mind:

- Relative references are resolved via the procedural searchpath setting.
- File extensions are optional and, when omitted, may result in more portable Ri stream.

In RIB, the syntax for specifying a dynamically loadable procedural primitive is:

```
Procedural "DynamicLoad" [ "dsoname" "initial" ] [ bound ]
```

dsoname is the name of the .so file that contains the required entry-points, and has been compiled and prelinked as described below. The current implementation intentionally subverts the LD_LIBRARY_PATH mechanism for searching for DSOs to load, so full pathnames for the DSO archive should be used.

initial is the ASCII printable string that represents the initial data to be sent to the ConvertParameters routine. The bound is an array of six floating point numbers which is *xmin, xmax, ymin, ymax, zmin, zmax* in the current object space.

And here's the Procedural2 variant:

```
Procedural2 "DynamicLoad" "SimpleBound"
    "string __dsoname" "yourdsoname"
    "float[6] __bound" [xmin xmax ymin ymax zmin zmax]
    //  additional meta-arguments to renderer ...
    //  arguments to plugin ...
```

Here, we employ the RiSimpleBound (and the associated bound parameter) to deliver our bounding box to the renderer. An alternate built-in bounding procedure, RiDSOBound, is provided to vector the bound request to a subroutine exported from the given DSO. This eliminates the requirement that the bound be emitted into the RIB file. RiDSOBound will only work if target DSO has implemented and exported the Bound subroutine.

> Please note that the Procedural2 implementation of DynamicLoad supports meta-parameters to control when and in which context the Subdivide2 function will execute. These meta-parameters may be interspersed with the arguments intended for the plugin. Since all meta-parameters are prefixed with '__', plugins can easily discriminate meta parameters from standard plugin parameters. For backwards compatibility, the meta-parameters "dsoname" and "bound" are supported but deprecated in favor of "__dsoname" and "__bound".

Here are two variations of the RIB represention for a trivial DynamicLoad procedural primitive:

```
Procedural "DynamicLoad" ["a_sphere" "3.0" ] [-3 3 -3 3 -3 3]
Procedural2 "DynamicLoad" "SimpleBound" "string __dsoname" "a_sphere"
            "float[6] __bound" [-3 3 -3 3 -3 3]
            "float radius" [3]
```

More complex initialization requirements require either more structure in the initial string data for Procedural or more parameters for Procedural2. As the initial data requirements grow, the benefits of Procedural2 become more apparent.

In C, the syntax for calling the dynamic load primitive is:

```
RtString *data;
RtBound bound;
RiProcedural(data, bound, RiProcDynamicLoad, freedata)

RtInt nargs;
RtToken toks[...];
RtPointer vals[...];
toks[0] = (RtToken) "float[6] __bound"; // RI_BOUND
vals[0] = (RtPointer) bound;
nargs = 1;
// .. plugin data here ..
RiProcedural2(RiProc2DynamicLoad, RiSimpleBound, nargs, toks, vals);
```

data is a pointer to an array of two RtStrings (i.e. data is a char **) that contains the DSO archive name in data[0], and the initial primitive data set in ASCII in data[1] (this will be sent to the ConvertParameters routine). bound is a pointer to six floats that contain the bounding box. freedata is the user free method that frees data. data could be a global variable, but almost certainly cannot be an automatic (local) variable, as it must persist until the renderer calls the free routine. In practice, data and its strings would be allocated with malloc and freedata would call free on data[0], data[1], and data.

Note that when the Subdivide routine wants to create a child procedural primitive of the same type as itself, it should call

```
RiProcedural(data, bound, Subdivide, Free)
```

passing its private notions of data and free, **and not**

```
RiProcedural(asciidata,bound,RiProcDynamicLoad,RiProcDLFree)
```

whose primary role is to simply load the DSO and to be representable in a RIB file. In the case of RiProcedural2 it may be advantageous for recursive procedural primitives to revert to the use of RiProcedural since it supports an unmarshalled representation of the plugin's state. In other words, the value of Procedural2 interface is primarily in the automatic serialization and deserialization to and from RIB. Once a plugin has been loaded and initialized, it's more convenient to convey data to sub-procedurals directly in the most natural (and private) struct or object representation.

RIB Generating Program

Another tool in our procprim toolkit is the RIB Generating Program. Here, the idea is that a separate program is harnessed to generate RIB at the behest of the renderer. As will be seen below, each generated procedural primitive is described by a request to the helper program, in the form of an ASCII datablock that describes the primitive to be generated. This datablock can be anything that is meaningful and adequate to the helper program, such as a sequence of a few floating point numbers, a filename, or a snippet of code in an interpreted modeling language. In addition, as above, the renderer supplies the *detail* of the primitive's bounding box, so that the generating program can make decisions on what to generate based on how large the object will appear on-screen.

In RIB, the syntax for specifying a RIB-generating program procedural primitive is:

```
Procedural "RunProgram" [ "program" "datablock" ] [ bound ]
```

program is the name of the helper program to execute, and may include command line options. datablock is the generation request datablock. It is an ASCII string which is meaningful to program, and adequately describes the children that are to be generated. Notice that program is a quoted string in the RIB file, so if it contains quote marks or other special characters these must be escaped in the standard way. The bound is an array of six floating point numbers - *xmin, xmax, ymin, ymax, zmin, zmax* - in the current object space.

In C, the syntax for calling the RIB-generating program procedural primitive is:

```
RtString *data;
RtBound bound;
RiProcedural(data, bound, RiProcRunProgram, freedata)
```

data is a pointer to an array of two RtStrings (i.e. data is a char **) that contain the program name in data[0], and the generation request data block in ASCII in data[1]. (The program name may include command line options.) bound is a pointer to six floats which contain the bounding box. freedata is the user free method that frees data. data could be a global variable, but almost certainly cannot be an automatic (local) variable, as it must persist until the renderer calls the free routine. In practice, data and its strings would allocated with malloc, and freedata would call free on data[0], data[1], and data.

Procedural Primitive Pathnames

PRMan provides a mechanism for selecting among several compiled versions of a procedural program or plug-in, depending on which platform the RIB file is being rendered on. The renderer will expand the special strings $ARCH or %ARCH in pathnames for procedural primitive programs and plug-ins, substituting a string that identifies the basic platform for which the running prman was built. For example:

```
    Procedural "DynamicLoad" ["/net/prmanDSOs/$ARCH/a_sphere" ...]
```

The default architecture names are:

Default $ARCH name	PRMan build
linux-x86-64	Linux, 64-bit, for x86-64 systems
windows-x86-64	Windows 64-bit, for x86-64 systems
osx-x86-64	Mac OS X, 64-bit, for x86-64 systems

These names can be remapped into site-specific abstract names by placing a line in the rendermn.ini file, for example:

```
/architecture/default_name        new_name

/architecture/linux-x86           gcc41glibc24
```

The ARCH value can also be set directly, using the shell environment variable RMAN_ARCHITECTURE, which overrides both the default andrendermn.ini values.

Authoring Custom Procedural Primitives

The renderer's requirements of any procedural primitive are simple. First and foremost, a procprim must implement a **Subdivide**subroutine. Prior to calling Subdivide, the renderer computes the location in the scene of the primitive and delays invoking theSubdivide subroutine until the last-most moment.

When the renderer reaches the bounding box of a particular procedural primitive instance, it calls the Subdivide method, passing in the blind data pointer and a floating point number that indicates the number of pixels that the bounding box covers (called the *detail*). The subdivide method splits the primitive into smaller primitives. It can generate standard RenderMan primitives, or it can generate more instances of procedural primitives with their own blind data pointers and (presumably smaller) bounding boxes, or some combination of the two.

At some point, the renderer knows that it will not need a particular blind pointer ever again. It then calls the free routine to destroy that blind pointer. It is never safe for the Subdivide routine to assume renderer behavior with respect to freeing data pointers. In particular, the renderer may free a blind pointer without ever subdividing it, or may subdivide a particular pointer multiple times.

The **RiProcedural2** interface requires two different entry points: *Subdivide (v2)* and *Bound*. This subdivide interface supports the transport of blind data through standard RI parameterlists and thereby eliminates the requirement that each procedural primitive serialize internal state into strings for recursive invocation. The downside is that, for highly recursive procprims, additional data management and parsing costs will be incurred. For this reason it may be wise to implement a heterogeneous procprim that uses RiProcedural2 representation for the RIB representation and RiProcedural for recursion.

Because standard RI parameterlists are employed, memory management is the responsibility of the renderer. This means that no freeroutine needs to be specified by the procedural primitive. The bound interface allows for deferred specification of the primitive bounding box and also supports the possibility of providing temporal bounding via two bounding boxes representing the primitive state at shutter-open and shutter-close times. Built-in bound routines introspect either the parameterlist or the DSO's custom subroutine to transmit bounding box information.

Writing a Dynamically Loaded Procprim (RiProcedural)

When writing a procedural primitive DSO for use with RiProcedural, you must create three specific public subroutine entry points, named Subdivide, Free and ConvertParameters.

- Subdivide is a standard RI procedural primitive subdivision routine. It takes a blind data pointer to be subdivided and a floating-point detail to estimate screen size.
- Free is a standard RI procedural primitive free routine, taking a blind data pointer that is to be released.
- ConvertParameters is a special routine that takes a string and returns a blind data pointer. It will be called for each Dynamic Load procedural primitive in the RIB file, and its job is to convert a printable string version of the progenitor's blind data (which must be in ASCII in the RIB file), into something that the Subdivide routine will accept.

The ANSI-C prototypes for these three required entry points are as follows:

```
RtPointer ConvertParameters(RtString paramstr);
RtVoid Subdivide(RtPointer data, RtFloat detail);
RtVoid Free(RtPointer data);
```

As an example, consider the following trivial procedural primitive, which merely emits a sphere of a specified radius into the renderer. This example ignores the subdivision detail, since the primitive is so simple.

```
RtPointer
ConvertParameters(RtString paramstr)
{
    RtFloat *radius_p;

    /* convert the string to a float */
    radius_p = (RtFloat *)malloc(sizeof(RtFloat));
    *radius_p = atof(paramstr);

    /* return the blind data pointer */
    return (RtPointer)radius_p;
}

RtVoid
Subdivide(RtPointer data, RtFloat detail)
{
    RtFloat *radius_p = (RtFloat *)data;

    /* output a sphere with the given radius */
    RiSphere( *radius_p, -*radius_p, *radius_p, 360.0, RI_NULL );
}

RtVoid
Free(RtPointer data)
{
    free(data);
}
```

Writing a Dynamically Loaded Procprim (RiProcedural2)

The RiProcedural2 interface for a procedural primitive DSO recognizes two public subroutine entry points: **Subdivide2**, which is required, and **Bound**, which is optional.

Subdivide2 is a standard RtProc2SubdivFunc procedural primitive subdivision routine. It takes a RtContextHandle, a floating-point*detail*, and a standard RI *parameterlist*.

Bound is a special routine that takes the same RI *parameterlist* as the Subdivide2 subroutine and fills in the bounding information that may depend on the *p arameterlist*.

The ANSI-C prototypes for these two entry points are:

```
RtVoid Subdivide2(RtContextHandle, RtFloat detail,
                 RtInt n, RtToken toks[], RtPointer vals[]);
RtVoid Bound(RtInt n, RtToken const toks[], RtPointer const vals[],
             RtBound result[2]);
```

As an example, consider the following trivial procedural primitive, which emits a sphere of a specified radius into the renderer. This example ignores the subdivision detail, since the primitive is so simple.

```
RtVoid
Subdivide2(RtContextHandle ctx, RtFloat detail,
          RtInt n, RtToken const toks[], RtPointer const vals[])
{
    RtFloat radius = 1;
    while(--n >= 0)
    {
        if(strstr(toks[n], "radius")) // poor man's parser to cope with
                                      // inline declarations
        {
            radius = *(RtFloat *) vals[i];
            break;
        }
    }
    /* output a sphere with the given radius */
    RiSphere( radius, -radius, radius, 360.0, RI_NULL);
}
```

Implementing a Bound method is only necessary if it offers advantages over SimpleBound. The purpose of the Bound routine is to determine a bound through means outside of the Ri stream. An Alembic plugin could read an associated .abc file and deliver the bounds at the first moment that the renderer needs it. Another potential value of the Bound method is that it can deliver two bounding boxes representing the changing bounds across the current shutter interval. This is why Bound's result supports two output bounds instead of one. If you implement a bound routine it's fine to only produce a single bounding box but you must copy the same values into both outputs.

Your C source code can call any RenderMan Interface routine exported by the SDK, as well as functions from other libraries that you may need. Next, the code must be **compiled and linked** to create the plugin itself; see the Compiling & Linking page for details.

Writing a RunProgram-Compatible Procprim

The generation program reads requests on its standard input stream, and emits RIB streams on its standard output stream. These RIB streams are read into the renderer as though they were read from a file (as with ReadArchive above), and may include any standard RenderMan attributes and primitives (including procedural primitive calls to itself or other helper programs). As long as any procedural primitives that require the identical helper program for processing exist in the rendering database the socket connection to the program will remain open. This means that the program should be written with a loop that accepts any number of requests and generates a RIB "snippet" for each one.

The specific syntax of the request from the renderer to the helper program is extremely simple, as follows:

```
fprintf(socket, "%g %s\n", detail, datablock);
```

The detail is provided first, as a single floating-point number, followed by a space, followed by the datablock, and finally a newline. The datablock is completely uninterpreted by the renderer or by the socket write, so any newlines or special characters should be preserved (but the quotation marks that make it a string in the RIB file will, of course, be missing).

The helper program's response should be to create a RIB file on stdout, presumably using the RIB client library, like so:

```
RiBegin(RI_NULL);
RiAttributeBegin();
    /* various attributes */
    /* various primitives */
RiAttributeEnd();
RiArchiveRecord(RI_COMMENT,"\377");
RiEnd();
```

Notice, in particular, the special trick that the helper program must use to stay in synchronized communication with the renderer. Stdout should not be closed when the RIB snippet is complete, but rather a single '\377' character should be emitted and the stdout stream flushed. This will signal the renderer that this particular snippet is complete, yet leave the stream open in order to write the next snippet. The use of RiArchiveRecord and RiEnd as above will accomplish this when the RIB client library is used.

If the "\377" character is not emitted or is accidentally not flushed through the stdout pipe to the renderer the render will hang.

When the renderer is done with the helper program, it will close its end of the IPC socket, so reading an EOF on stdin is the signal for the helper program to exit.

Details and Notes

Prior to subdividing a procedural primitive, the renderer restores the graphics state to the state that existed when the procedural primitive was itself created. The subdivision routine can call any RI routine that is legal within the world scope, so it may modify any attribute in the graphics state before generating new primitives (procedural or otherwise). In other words, procedural primitives are free to manipulate the color, shaders, or any other attribute of their children.

Motion blur of procedural primitives is tricky. Procedural primitives, per se, cannot exist within motion blocks, because this would force the renderer to do some sort of correspondence-matching between the children of the time 0.0 procedural primitive and the children of the time 1.0 procedural primitive. However, once the procedural primitive has decided to subdivide itself into standard RenderMan primitives, it is completely legal for those primitives to be emitted with full motion blocks. Thus, it is the responsibility of the procedural primitive methods themselves to understand "vertex motion" and generate the appropriate moving primitives at the leaf level. Note, however, that normal transformation matrix motion blur is perfectly legal on procedural primitives.

In situations where the renderer is unable to calculate the correct *detail* to pass into the subdivide method (for example, if the procedural primitive crosses the camera plane), the subdivision detail will be set to some extremely large number (quite possibly infinity). Subdivision methods must be able handle this case without generating infinite amounts of leaf data.

The RenderMan Interface has a command for modifying the *detail* that is sent to a procedural primitive. RiRelativeDetail is an*attribute* that provides a scaling factor for the details sent to the subdivision routine. This is a separate control from the RiShadingRate, which will determine the shading rate of the final "standard" RenderMan primitives that are generated. RiProcedurals can employRxAttribute to obtain access to the current shading rate.

If the bounding box of a child of a procedural primitive extends outside the bounding box of the parent, the resulting image will very likely have dropouts. However, the renderer does not generate any warning messages in this case. Be sure that your bounding boxes do in fact contain all the geometry that will be generated at all lower levels. If primitives with multi-segment motion blur (more than two motion samples in their motion blocks) are generated by a procedural plugin that implements a Bound routine, the path between the shutter-open and shutter-close bounds must contain all primitive motion samples.