Compiling Plugins & Linking Applications

RenderMan Plugins

Compiling and using a new plugin requires three steps:

1. Compiling the C++ file that contains your plugin functions.
2. Compiling the shader that uses your functions.
3. Rendering a frame.

Compiling your C++ file is straightforward: just use the standard C++ compiler to generate an object (.o/.obj) file, then generate a shared object (.so/.dll) file from the object file. Remember that, though using C++, you must use C style linkage. You also must ensure that your C++ compiler and libraries are compatible with the compiler and runtime libraries used by PRMan (gcc for Linux and OS-X and Microsoft Visual C for Windows).

Plugin authors can confirm the compiler/library versions for the intended version of PRMan by running `prman -version`.

Here are example commands for building a plugin on several architectures:

Linux

```
g++ -fPIC -I$RMANTREE/include -c myfunc.cpp -o myfunc.o
g++ -shared myfunc.o -o myfunc.so
```

macOS

```
clang++ -std=c++14 -I$RMANTREE/include -c myfunc.cpp -o myfunc.o
clang++ -bundle -undefined dynamic_lookup myfunc.o -o myfunc.so
```

Windows

```
cl -nologo -MD -EHsc -I"%RMANTREE%\include" -c myfunc.cpp
link -nologo -DLL -out:myfunc.dll myfunc.obj "%RMANTREE%\lib\libprman.lib"
```

The resulting file myfunc.so or myfunc.dll is the plugin that implements your new function. It is not important that the filename matches the name of the function.

- On Unix-based platforms, plugins are linked such that symbols that resolve to entry points in libprman.so or libprman.dylib are left unresolved. Note that on Linux the use of -fPIC is important for code that will be used as a plugin and that there is no explicit linkage of l ibprman.so, even if the plugin makes reference to application interfaces, like deeptexture.
- On macOS, the linker must be explicitly told that the unresolved symbols will be resolved at runtime.
- On Windows, the libprman.lib must always be referenced to resolve the unresolved symbols in the plugin and must define PRMANLOADER, this will retrieve the correct version of of the declaration. You must also link libpxrcore.lib which implies the code is dependent on $RMANTREE\bin\libpxrcore.dll. When your code is loaded that directory will need to be in the search path. You may add this path to the PATH environment variable or, if you're authoring a plugin, the following code example would work:

```
BOOL rt = ::SetDefaultDllDirectories(LOAD_LIBRARY_SEARCH_DEFAULT_DIRS);
if (!rt)
{
std::cerr << "Failed to set add directory\n";
return 1;
}

DLL_DIRECTORY_COOKIE cookie = ::AddDllDirectory(L"C:\\Program Files\\Pixar\\RenderManProServer-23.0b1\\bin");

if (!cookie)
{
std::cerr << "Failed to set add directory\n";
return 1;
}
```

Applications and Bridge Tools

For applications, the libprman library will be loading into the the application. Of course, the application will need to be told where to find the library. This can be done at link-time by linking to the libprman library or at runtime using the libloadprman.a static library.

Dynamic Linking

In this case, the libprman library is linked to the application in some platform dependent way (i.e. LD_LIBRARY_PATH, rpath, etc.). When using the RenderMan API, a RixContext pointer may be obtained by calling RixGetContext.

When receiving a RixContext, you must do your due diligence to make sure that the version of the context is the version you are expecting to receive using the GetVersion() API and comparing against the version you are expecting. If the check fails, you should return an error and disable your plugin. Otherwise, a crash is likely.

Here are example commands for building a plugin on several architectures:

Linux

```
g++ -c -fPIC -I$RMANTREE/include myapp.cpp -o myapp.o
g++ myapp.o -L$RMANTREE/lib -lprman -o myapp
```

macOS

```
clang++ -std=c++14 -I$RMANTREE/include -c myfunc.cpp -o myapp.o
clang++ -bundle -undefined dynamic_lookup myfunc.o -o myfunc.so
```

Windows

```
cl -nologo -MD -EHsc -I"%RMANTREE%\include" -c myfunc.cpp
link -nologo -DLL -out:myfunc.dll myfunc.obj "%RMANTREE%\lib\libprman.lib"
```

Dynamic Loading RenderMan

In this case, the libprman library is loaded by the application at runtime from the RMANTREE environment variable. This is made possible by directly linking the libloadprman.a static library the static library into your application. When using the RenderMan API, a RixContext pointer may be obtained by calling RixGetContextViaRMANTREE.

⚠ When receiving an API object from the RixContext, you must do your due diligence to make sure that the version of the API object is the version you are expecting to receive using its GetVersion() method and comparing against the version you are expecting. If the check fails, you should return an error and disable your plugin. Otherwise, a crash is likely.

Here are example commands for building a plugin on several architectures:

Linux

```
g++ -c -fPIC -I$RMANTREE/include myapp.cpp -o myapp.o
g++ myapp.o $RMANTREE/lib/libloadprman.a -o myapp
```

macOS

```
clang++ -std=c++11 -c -I$RMANTREE/include myapp.cpp -o myapp.o
clang++ myapp.o $RMANTREE/lib/libloadprman.a -o myapp
```

Windows

```
cl -nologo -MD -EHsc -I"%RMANTREE%\include" -c myapp.cpp
link -nologo -out:myapp.exe myapp.obj "%RMANTREE%\lib\libloadprman.lib"
```

The following is an example of how to obtain an interface API object, in this case the SceneGraph manager, and applying the version check noted above:

```
#include "RixInterfaces.h"
#include "RixSceneGraph.h"

RixContext* ctx = RixGetContextViaRMANTREE();
if (!ctx)
{
        // cannot proceed, no RenderMan
        return;
}
RixSGManager* sgm = static_cast<RixSGManager*>(ctx->GetRixInterface(k_RixSGManager));
int myCompiledVersion = k_RixSGManagerVersion;
if (sgm->GetVersion() != myCompiledVersion)
{
    // Unrecoverable error, this is the wrong version of scene graph api and
    // we should not continue.
    return;
}
```