This is a draft of something that may one day end up as a develop article, I hope that you will find it useful. If you have any comments, particularly if you need more information, please email me at nickt@apple.com. This is a first draft that I got together in about an hour to get some developers off the ground, I'll be enhancing this as I go.

3/6 added some information about the fields of the dialog anchor and a para talking about interactive vs. non-interactive.

...

Supporting plug-in renderers

QuickDraw 3D 1.5 supported plug-in renderers, but the first implementations of plug-in renderers from 3rd party developers need 1.5.1 or later to function correctly. It is important when designing your application to consider the difference between renderers that have "interactive" performance (such as Apple's Wireframe and Interactive renderers), and other renderers (such as the ray tracer from LightWork Design - check out http://www.lightwork.com for some more information about their QuickDraw 3D plug-in renderer). It is likely that if the renderer does not offer interactive performance, you'll want to create a new window for the renderer output, whilst allowing the user to manipulate elements of the scene using a window that manages interaction using a renderer that is classed as interactive. You can determine if a renderer supports interactive performance by using the API call

```
/*
 *  Q3Renderer_IsInteractive
 *      Determine if this renderer is intended to be used interactively.
 */
QD3D_EXPORT TQ3Boolean QD3D_CALL Q3Renderer_IsInteractive(
    TQ3RendererObject     renderer);
```

Even if you don't want to write a renderer, you almost certainly want to be able to provide support for plug-ins in your application. There are a number of things you must do in order to properly support plug-in renderers in your application. You need to be able to query QuickDraw 3D to find out which renderers are available, you need to be able to construct a list of available renderers so that your users can select the renderer that they are interested in, and you need to be able to set the renderer in response to a user selection.

Lets look at how to build a menu that contains the list of available plug-in renderers. First you need to get the list of renderers installed in the system. In QD3D.h you'll find the following:

```
/*
 * TQ3SubClassData is used when querying the object system for
 * the subclasses of a particular parent type:
 */
```

```
typedef struct TQ3SubClassData {
    unsigned long     numClasses;    /* the # of subclass types found */
                      /* for a parent class */
    TQ3ObjectType     *classTypes;   /* an array containing the class */
                      /* types                          */
} TQ3SubClassData;
```

This structure gets filled out with the number of classes of a particular type, along with an array of `TQ3ObjectType` which contains the 4 char identifier of the class type. You fill out this data structure with a call to:

```
/*
 *  Given a parent type and an instance of the TQ3SubClassData struct fill
 *  it in with the number and class types of all of the subclasses immediately
 *  below the parent in the class hierarchy.  Return kQ3Success to indicate no
 *  errors occurred, else kQ3Failure.
 *
 *  NOTE:  This function will allocate memory for the classTypes array.  Be
 *  sure to call Q3ObjectClass_EmptySubClassData to free this memory up.
 */
QD3D_EXPORT TQ3Status QD3D_CALL Q3ObjectHierarchy_GetSubClassData(
    TQ3ObjectType            objectClassType,
    TQ3SubClassData          *subClassData);
```

Note that this call can allocate memory, but we also provide a call to dispose of the memory allocated:

```
/*
 *  Given an instance of the TQ3SubClassData struct free all memory allocated
 *  by the Q3ObjectClass_GetSubClassData call.
 *
 *  NOTE: This call MUST be made after a call to Q3ObjectClass_GetSubClassData
 *  to avoid memory leaks.
 */
QD3D_EXPORT TQ3Status QD3D_CALL Q3ObjectHierarchy_EmptySubClassData(
    TQ3SubClassData              *subClassData);
```

Once you have obtained the list of subclasses of type renderer (you'd pass in kQ3SharedTypeRenderer to the first call above) you can loop through the array and process it in order, for each renderer sub-class in the array.

You'll see in the code snippet below that we try to get the renderer nickname. Each class in the QuickDraw 3D system has a unique type and a class name. The notion of a nickname is a new one. When we were developing the plug-in renderer system, it became apparent that the class name could not be used in the application user interface, since the class names are not localizable. We decided to give a renderer the option of providing a localized string to a calling application. With QuickDraw 3D 1.5.1 there is a new call:

```
/*
 *  Q3RendererClass_GetNickNameString
 *      Allows an application to get a renderers name string, the
 *      renderer is responsible for storing this in a localizable format
```

```
 *     for example as a resource.  This string can then be used to provide
 *     a selection mechanism for an application (for example in a menu).
 *
 *     If this call returns nil in the supplied string, then the App may
 *     choose to use the class name for the renderer.  You should always
 *     try to get the name string before using the class name, since the
 *     class name is not localizable.
 */
QD3D_EXPORT TQ3Status QD3D_CALL Q3RendererClass_GetNickNameString(
    TQ3ObjectType        rendererClassType,
    TQ3ObjectClassNameString rendererClassString);
```

This call lets you get the renderer supplied string that describes the renderer. This can then be put into a menu, or some other UI element. That's all the prices we need, lets put it together. Here is a routine that given a menu handle, will add all of the renderers installed to that menu:

```
void SetUpRendererMenu( void )
{
    MenuHandle               theMenu ;
    TQ3SubClassData          subClassData;
    TQ3ObjectType            *classPointer;
    short                    i;
    TQ3ObjectClassNameString objectClassName;
    TQ3RendererObject        tempRendererObject ;
    unsigned char            nameBuffer[256] ;
    unsigned long            actualLength ;
    TQ3ObjectClassNameString objectClassString ;

    theMenu = GetMHandle(mRendererMenu);

    Q3ObjectHierarchy_GetSubClassData(kQ3SharedTypeRenderer, &subClassData);

    classPointer = subClassData.classTypes;

    i = subClassData.numClasses;

    while( i-- > 0 && pRendererCount <= kMaxRendererCount) {

        /*
         * the "generic" renderer is used internally, it can't draw,
         * so don't display it in any user interface item
         */
        if( *classPointer != kQ3RendererTypeGeneric )
        {

            Q3RendererClass_GetNickNameString(*classPointer, objectClassString );

            if( objectClassString[0] == '\0' )
            {
                /* the renderer did not provide the name, just use the class name */
                Q3ObjectHierarchy_GetStringFromType(*classPointer, objectClassName);

                AppendMenu(theMenu,c2pstr((char *)objectClassName));
                pTypes[pRendererCount++] = *classPointer ;
            }
            else
```

```
            {
                AppendMenu(theMenu,c2pstr(objectClassString));
                pTypes[pRendererCount++] = *classPointer ;
            }

        }

        classPointer++ ;
    }

    pTypes[pRendererCount] = NULL ;
    Q3ObjectHierarchy_EmptySubClassData( &subClassData ) ;
}
```

There are three important things to note about the example given above. First, notice that if we cannot get the name from the renderer, we use the class name instead. Second, notice that we ignore a renderer of type `kQ3RendererTypeGeneric` this is a dummy renderer use while picking, and writing to files and since it cannot be used to draw anything, we don't add it to the menu. Third, notice that at the appropriate index into the menu, we store the renderer type in the pTypes array. This will be most useful later when we set the renderer according to the user selection.

Finally notice the call to `Q3ObjectHierarchy_EmptySubClassData` which disposes of the memory allocated when we queried the system for the installed plug-in renderers.

Setting the renderer

Once we have set up the menu with the list of installed renderers it is an easy matter to set the renderer based on the user selection. This example is somewhat more complex than it needs to be, since it illustrates setting the renderer in a QuickDraw 3D viewer application, but the principle will be the same for all kinds of applications:

```
/*
 * handle menu commands in the renderer menu
 */


void HandleRendererMenu( short menuItem )
{
    ViewerDocumentHdl theViewerDocumentHdl ;
    WindowPtr         theWindow ;
    TQ3ViewerObject       theViewer ;
    OSErr             theError ;
    TQ3ViewObject     myView ;
    TQ3Status         myStatus ;
    TQ3RendererObject myRenderer ;

    theWindow = FrontWindow() ;



    if( theWindow != NULL )
    {
```

```
        /*
         * get the reference to our viewer document data structure
         * from the long reference constant for the window.  Cast
         * it to the appropriate type.  If we can't get it (i.e. it's
         * null we want to bail
         */
        theViewerDocumentHdl = (ViewerDocumentHdl)GetWRefCon(theWindow) ;
        if(theViewerDocumentHdl == NULL)
            return ;

        /* get the reference to the viewer object from our data structure  */
        theViewer = (**theViewerDocumentHdl).fViewer ;
        if(theViewer == NULL)
            return ;

        myRenderer = Q3Renderer_NewFromType( pTypes[ menuItem - 1 ] );

        /*
         * set the renderer for the view
         */
        myView = Q3ViewerGetView( theViewer );
        if( myView != NULL && myRenderer != NULL )
        {
            /* set the renderer to the one created in the switch statement above  */
            myStatus = Q3View_SetRenderer(myView, myRenderer) ;

            /* now we have set the renderer we can dispose of the reference to it  */
            myStatus = Q3Object_Dispose( myRenderer ) ;

            /* and redraw the content region of the viewer  */
            theError = Q3ViewerDrawContent( theViewer ) ;
        }
    }
}
```

Supporting renderer preferences.

If you are going to support plug-in renderers, you'll want to have access to the renderer preferences dialog. This is a dialog that allows to to set preferences for how the renderer draws things.

```
            {
                TQ3RendererObject       qd3dRenderer;
                TQ3ViewObject           qd3dView;
                TQ3Status               qd3dStatus;
                TQ3Boolean              qd3dCanceled;
                TQ3DialogAnchor         qd3dAnchor ;


                // Get the renderer
                qd3dView = Q3ViewerGetView(theViewer);
                qd3dStatus = Q3View_GetRenderer(qd3dView, &qd3dRenderer);


                // Put up the configure dialog
                if (Q3Renderer_HasModalConfigure(qd3dRenderer))
```

```
                {
#if 0
                    /* this will be modal */
                    qd3dAnchor.clientEventHandler = NULL;
#else
                    /* this will enable a movable modal pass in event handler */
                    qd3dAnchor.clientEventHandler = HandleEvent ;
#endif
                    qd3dStatus = Q3Renderer_ModalConfigure(
                                                qd3dRenderer,
                                                qd3dAnchor,
                                                &qd3dCanceled);
                }

                // Clean up
                qd3dStatus = Q3Object_Dispose(qd3dRenderer);
            };
```

You can use the API call `Q3Renderer_HasModalConfigure` to determine if the renderer has a modal configure dialog. For 1.5.1 we did not implement this functionality for either the wireframe renderer or for the interactive renderer, but we expect to have this implemented this in 1.6, coming this summer.

An important thing to notice here is the event handler field. This can be set to nil if the application just wants the dialog to be modal. If the dialog is to be movable then you need to pass in a reference to the applications event handler. Also note that you need to disable menu items so that the about item is disabled, but the rest of the apple menu is enabled, and all menus except the edit menu are disabled with all items in the edit menu disabled except for undo, cut, copy, paste, clear which must be enabled. The following code snippet illustrates this:

```
void SpinDialog_ConfigureRenderer(
    void)
{
    TQ3ViewObject     view;
    TQ3RendererObject renderer;
    TQ3DialogAnchor       dialogAnchor;
    TQ3Boolean        canceled;
    MenuHandle        m;

    SpinView_GetView(&view);
    Q3View_GetRenderer(view, &renderer);
    if (Q3Renderer_HasModalConfigure(renderer)) {
        dialogAnchor.clientEventHandler = SpinEventHandlerWrapper;

        m = GetMHandle(kMenu_Apple);
        SpinMenu_EnableItem(m, kAppleMenu_About, false);

        m = GetMHandle(kMenu_File);
        SpinMenu_EnableItem(m, 0, false);

        m = GetMHandle(kMenu_Edit);
        SpinMenu_EnableItem(m, kEditMenu_Undo, true);
```

```
    SpinMenu_EnableItem(m, kEditMenu_Cut, true);
    SpinMenu_EnableItem(m, kEditMenu_Copy, true);
    SpinMenu_EnableItem(m, kEditMenu_Paste, true);
    SpinMenu_EnableItem(m, kEditMenu_Clear, true);
    SpinMenu_EnableItem(m, kEditMenu_Preferences, false);

    m = GetMHandle(kMenu_Commands);
    SpinMenu_EnableItem(m, 0, false);

    m = GetMHandle(kMenu_Dialog);
    SpinMenu_EnableItem(m, 0, false);

    m = GetMHandle(kMenu_View);
    SpinMenu_EnableItem(m, 0, false);

    m = GetMHandle(kMenu_Geometry);
    SpinMenu_EnableItem(m, 0, false);

    m = GetMHandle(kMenu_Complex);
    SpinMenu_EnableItem(m, 0, false);

    m = GetMHandle(kMenu_Demos);
    SpinMenu_EnableItem(m, 0, false);

    DrawMenuBar();

    Q3Renderer_ModalConfigure(renderer, dialogAnchor, &canceled);

    m = GetMHandle(kMenu_Apple);
    SpinMenu_EnableItem(m, kAppleMenu_About, true);

    m = GetMHandle(kMenu_File);
    SpinMenu_EnableItem(m, 0, true);

    m = GetMHandle(kMenu_Commands);
    SpinMenu_EnableItem(m, 0, true);

    m = GetMHandle(kMenu_Dialog);
    SpinMenu_EnableItem(m, 0, true);

    m = GetMHandle(kMenu_View);
    SpinMenu_EnableItem(m, 0, true);

    m = GetMHandle(kMenu_Geometry);
    SpinMenu_EnableItem(m, 0, true);

    m = GetMHandle(kMenu_Complex);
    SpinMenu_EnableItem(m, 0, true);

    m = GetMHandle(kMenu_Demos);
    SpinMenu_EnableItem(m, 0, true);

    SpinMenu_Update();

    DrawMenuBar();
}
else {
    Alert(kNoModalConfigureALRT, NULL);
```

```
    }
}
```

Supporting renderer names in a plug-in renderer.

Most people won't write renderers since they require a great deal of expertise and time to implement. But in case you are interested, here is the modification I made to the sample renderer that Phil and shehryar on the QuickDraw Ed team have been working on.

```
TQ3Status SR_GetNameString(
    unsigned char            *dataBuffer,
    unsigned long            bufferSize,
    unsigned long            *actualDataSize )
{
    TQ3Status  status = kQ3Success ;
    Boolean    wasChanged;
    FSSpec     fileSpec;
    OSErr      macErr;
    Str255     tempBuffer ;

    /*
     * Get at the resource file for this renderer and open the resource
     * file, locate the renderer name string and close the res file
     * returning the string in the buffer we were passed.
     */
    SRgOldResFile = CurResFile();

    macErr = ResolveAlias(NULL, SRgAliasHandle, &fileSpec, &wasChanged);

    if (macErr == noErr) {
        SRgResFile = FSpOpenResFile(&fileSpec, fsRdPerm);

        if (SRgResFile != -1) {

            *actualDataSize = 0L;
            GetIndString( tempBuffer, SR_NAME_RESOURCE, 1 ) ;

            /* trim the buffer if necessary */
            *actualDataSize = (tempBuffer[0] > bufferSize) ? bufferSize : tempBuffer[0] ;

            if (dataBuffer != NULL)
            {
                /* copy from the pascal str returned by the res mgr */
                memcpy((char *)dataBuffer,
                    (char *) &tempBuffer[1],
                    *actualDataSize );

                /* and terminate the string */
                dataBuffer[*actualDataSize] = '\0' ;

            }

            CloseResFile(SRgResFile);
```

```
            UseResFile(SRgOldResFile);
        } else {
            Q3XMacintoshError_Post(ResError());
            status = kQ3Failure;
        }
    } else {
        Q3XMacintoshError_Post(macErr);
        status = kQ3Failure;
    }

    if( status == kQ3Failure )
    {
        *actualDataSize = 0L ;
        dataBuffer = NULL ;
    }


    return (status);
}
```

The key issue here is to save a reference to the renderer's file (in SRgAliasHandle) so that the resource fork can be opened. In this instance I decided to place the resource in the files as a string list 'STR#' type resource. This allows multiple strings to be stored, each of which could be a localized single or two byte string, so the renderer could switch between. Of course, you could just have a single string that gets localized according to the market for the renderer.

You need to make sure this gets called when the renderer is loaded, and to do this you'll need to modify the metahandler for your renderer. The change is pretty simple, just add a case to reference your name routine:

```
        /*
         *  renderer name string
         */
        case kQ3XMethodTypeRendererGetNickNameString: {
            return (TQ3XFunctionPointer) SR_GetNameString;
            break;
        }
```

The changes to Renderer.h explain this:

```
/*
 *  kQ3XMethodTypeRendererGetNickNameString
 *
 *      Allows an application to collect the name of the renderer for
 *      display in a user interface item such as a menu.
 *
 *      If dataBuffer is NULL actualDataSize returns the required size in
 *      bytes of a data buffer large enough to store the name.
 *
 *       bufferSize is the actual size of the memory block pointed to by
 *      dataBuffer
```

```
 *
 *     actualDataSize - on return the actual number of bytes written to the
 *     buffer or if dataBuffer is NULL the required size of dataBuffer
 *
 *  OPTIONAL
 */
#define kQ3XMethodTypeRendererGetNickNameString \
    Q3_METHOD_TYPE('r','d','n','s')
typedef TQ3Status (QD3D_CALLBACK *TQ3XRendererGetNickNameStringMethod)(
    unsigned char             *dataBuffer,
    unsigned long             bufferSize,
    unsigned long             *actualDataSize);
```