TP005: A Simple Command Line Application

Introduction

The aim of this document is to outline the procedures required to write, build and execute a simple command line application utilising the RealiMation API. This will take the reader through the initial stages of setting up Microsoft Developer Studio™ to produce a Win32 command line application. The final code example presents the minimum code required to load a RealiBase and display a predefined view on a graphics channel. The resulting program makes no attempt to utilise the full RealiMation API but does present a first step to developing more comprehensive and general purpose applications.
In order to keep this example as simple as possible the application uses the 3Dfx display driver. This does not require the creation of a window within which to render. *A code example which uses windows based display drivers can be seen Technical Application Paper 003 - Using RealiMation in Microsoft Windows Applications*.

Setting up Microsoft Developer Studio

To create a new application within Microsoft Developer Studio select File/New. In the “New” dialog, select the “Projects” tab and select “Win32 Console Application”. Set a suitable “Project Name” and specify a “Location” for the new project. Select “OK” to create the application project.

Now select File/New, and select the Files tab. Select the “C++ Source File” type and enter a suitable “File Name”. Note that there is no need to specify an extension for the source file.

So that the necessary header files and libraries are located it is necessary to add directories to the search paths of Developer Studio. Select Tools/Options, and select the “Directories” tab in the “Options” dialog. Select “Show directories for:” “Include files” and add “`C:\RealiMation\SDK\INC`” to the list of directories. Now select “Show directories for:” “Library files” and add “`C:\RealiMation\SDK\LIB\i386`” to the list of directories. These paths should reflect the location of the RealiMation header and library files as installed on your system.

To take advantage of the debugging utilities provided as part of the RealiMation API it is necessary to have `_RMDEBUG` defined at compile time. This is achieved by selecting Project/ Settings and select the “Settings For:” “Win32 Debug” option. Select the “C/C++” tab and add “`_RMDEBUG`” to the list of “Preprocessor definitions:”. This will ensure that the debugging routines can be used for debug versions of the application, but not the release versions.

The Application Explained

Now that a C++ source file has been created the source of the application can be entered.

The first step is to include the RealiMation header files and state which libraries are to be linked in with this application. The `#pragma` directive provides a convenient method for achieving this:

```
#include <stdio.h>
#include <conio.h>

#include “rmation.h”
#include “rfu.h”

#if defined(_RMDEBUG)
#include “rmdebug.h”
#endif
```

```
#if defined(_RMDEBUG)
#pragma comment(lib, "rmation4d.lib")
#pragma comment(lib, "rfu4d.lib")
#pragma comment(lib, "rmm4d.lib")
#pragma comment(lib, "rmu4d.lib")
#pragma comment(lib, "rmdebug.lib")
#else
#pragma comment(lib, "rmation4.lib")
#pragma comment(lib, "rfu4.lib")
#pragma comment(lib, "rmm4.lib")
#pragma comment(lib, "rmu4.lib")
#endif
```

Note that different versions of the RealiMation libraries exist for release and debug versions of the application. Care should be taken not to mix release and debug libraries. Also note that an extra debug library (`rmdebug.lib`) is linked with the debug version of the application.

Some definitions can now be added to define which RealiBase will be loaded, which display driver to use, and the display characteristics:

```
#if defined(_RMDEBUG)
#define DRIVER "rgd3dfx4d.dll"
#else
#define DRIVER "rgd3dfx4.dll"
#endif

#define REALIBASE "helisim.rbs"

#define XRES 640
#define YRES 480
#define PIXELDEPTH 16
#define REFRESH 60

#define XPOS 0
#define YPOS 0
```

For simplicity this example uses defined constant values. A more useful, general purpose, application would probably read user input to set runtime variables. Note that, as above, two display driver libraries are specified, one for each of the release and debug version of the application. Also note that this application will assume that a RealiBase named "`helisim.rbs`" exists in the current directory.

The `main()` function can now be added. This will perform the following basic steps:

1. Initialise the RealiMation libraries.
2. Load the required RealiBase.
3. Create a channel and view.
4. Add a predefined view to the channel.
5. Enter a basic display loop.
6. Close down gracefully.

These steps are outlined below and are indicated in the complete source code example provided at the end of this document.

Step 1: Initialise the RealiMation Libraries

A call to `RTInitialise()` must be made before any other calls to the RealiMation libraries. This ensures that all internal data structures are initialised correctly.

Step 2: Load the Required RealiBase

A RealiBase may contain texture mapped objects for which a texture image is required. These texture images may be stored within the RealiBase file or externally in separate files. In the later case a user defined image callback function must be defined which will take the image file name as stored in the RealiBase and load the required image into memory. The address of this function is specified through the call to `RTSetImageCallback(imageCB)` where `imageCB` is a pointer to an image callback function. Such a function can be obtained from the many source code examples provided with RealiMation. The structure of this function is beyond the scope of this document and shall not be expanded upon here. In the example provided a null image callback has been defined which will not load texture images.

Once the image callback has been specified the required RealiBase can be loaded using the call `RTLoadRealiBase(REALIBASE, 0)`. In this example the filename has been explicitly defined, however any null terminated character string may be passed to the function. The second argument to `RTLoadRealiBase()` specifies a number of flags which affect the loading of the data. These are ignored for the purposes of this simple example.

Step 3: Create a Graphics Channel

To observer the contents of the RealiBase, a graphics channel must be realised and a view of the RealiBase associated with that channel. The first part of this stage is to create a new channel and associated ID using a call to `RTCreateID()`:

```
rtID ChannelID = RTCreateID(RTChannelID);
```

The characteristics of the channel are specified through the fields of an `rtChannel` structure. In the case of a 3Dfx display device, and the definitions provided above, this structure is specified as:

```
rtChannel cInfo;

cInfo.name = DRIVER;
cInfo.res.width = XRES;
cInfo.res.height = YRES;
cInfo.res.depth = PIXELDEPTH;
cInfo.res.refresh = REFRESH;
cInfo.auto_load = 0;
cInfo.n = 0;
cInfo.data = NULL;
```

In the case of the 3Dfx display driver no further information needs to be passed to the libraries though the `cinfo.data` pointer. However, for drivers which require a window within which to render, this field must point to valid data appropriate for the driver (an `HWND` for example). The contents of the `rtChannel` structure is passed to the RealiMation libraries using a call to `RTSetChannel()`:

```
RTSetChannel(ChannelID, &cInfo);
```

Finally the channel is realised using a call to `RTRealise()`. This initialises the graphics device ready for rendering to take place, once a view has been associated with the channel:

```
RTRealise(ChannelID);
```

Step 4: Add a Predefined View to the Channel

A view describes how the scene appears, which for the purposes of this example, should already be defined in the RealiBase that has been loaded. It is distinct from the channel, defined above, in that a channel provides a rendering context whereas a view describes what parts of the scene are to be rendered. Any instances that are associated with the view shall be considered for rendering on the specified channel. Note that a single channel may have multiple views associated with it, and further to this an application may use multiple channels.

The first step in using a predefined view is to obtain the ID from the loaded RealiBase. This is performed using a call to `RTGetNextID()`. In this example the first view encountered within the RealiBase is returned, however, a named view could be obtained explicitly using a call to `RTGetIDFromName()`, or created using `RTCreateID()`:

```
rtID ViewID = RTGetNextID(RTNullID, RTViewID);
```

Next the size and position of the view with respect to the channel is specified. In each case the dimensions given are in units of pixels and relate directly to the channel information provided in the `rtChannel` structure defined above. Note that no validation of the given values is performed since the view may not be realised on a physical display surface:

```
RTSetViewPortSize(ViewID, XRES,YRES);
RTSetViewPortPos(ViewID, XPOS, YPOS);
```

Finally, for this step, the view ID is added to the channel and the view realised:

```
RTAddObject(ChannelID, ViewID);
RTRealise(ViewID);
```

Step 5: Enter a Basic Display Loop

In this example the system clock is used to control the animation of the scene. The current system time can be found using a call to `RTInqClock()`. This determines the time at the start of the display loop. Subsequent calls to `RTInqClock()` within the display loop determine the time for which the next view of the scene is to be evaluated. Setting the view time, using a call to `RTSetViewTime()`, specifies the time at which the position and orientation of objects in the view are evaluated. The time elapsed since the last rendering of the view can be determined using the current system time (`t`) and the time before entering the display loop (`t0`).

The call to `RTDisplayView()` updates the entire scene and renders it using the current view settings. Only once a call to `RTSwapPage()` is made is the rendered image displayed on the display surface. These final two steps are kept separate for the purposes of double buffering and where multiple views and/or channels (which may require synchronisation) are defined.

The display loop is terminated by the user pressing the Escape key, which is detected using the standard C library function `kbhit()` function (declared in `conio.h`).

```
DWord key;
DWord terminate = 0;
Ord t, t0;

t0 = RTInqClock();
while ( !terminate )
{
    t = RTInqClock();
    RTSetViewTime(ViewID, t-t0);
    RTDisplayView(ViewID);
    RTSwapPage(ChannelID);

   while ( kbhit() )
      key = getch();
   if ( key == 0x1B )
      terminate = 1;
}
```

Step 6: Close Down Gracefully

The final stages of this application are to unrealise the view and channel and to close down the RealiMation library to undo the initialisation performed in Step 1:

```
RTUnrealise(ViewID);
RTUnrealise(ChannelID);
RTShutDown();
```

Checking the Error Status

The RealiMation API provides a number of useful utilities to enable the tracing of problems within applications. It is advised that the RealiMation error status is checked to determine the success or otherwise of the last call to the RealiMation library. The error status is determined with a call to `RTInqLastError()`. This returns an error number for which the appropriate error message can be determined using a call to `RTInqErrorString()`. These two function have been used to provide a general purpose function which can be called after each RealiMation API call:

```
void InqError(char *msg)
{
    rtError err = RTInqLastError(0);
#if defined(_RMDEBUG)
    char errmsg[32];
     RTInqErrorString(err, errmsg);
    fprintf(stderr, "%s : %s\n", msg, errmsg);
#endif
    if ( err != RTNoError )
       exit(1);
}
```

Note that the error status messages are only produced in the debug version of the application, however an error will cause the application to terminate for both release and debug versions.

This error handling behavior is simply a feature of this sample application. In actual applications, checking the error status after every call tends to be very useful in the early development stages and when learning the API, but can be largely removed once the developer is sure of the program logic. In this way, error checking only needs to stay in those parts of the code where a genuine error is possible in the final release build.

The Complete Code Example

```
/*———————Start of example code———————————*/

/*— Initial definitions —*/

/* Load standard include files */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

/* Load RealiMation specific include files */
#include "rmation.h"
#include "rfu.h"

#if defined(_RMDEBUG)
#include "rmdebug.h"
#endif

/* Set directives for external library loading */
#if defined(_RMDEBUG)
#pragma comment(lib, "rmation4d.lib")
#pragma comment(lib, "rfu4d.lib")
#pragma comment(lib, "rmm4d.lib")
#pragma comment(lib, "rmu4d.lib")
#pragma comment(lib, "rmdebug.lib")
#else
#pragma comment(lib, "rmation4.lib")
#pragma comment(lib, "rfu4.lib")
#pragma comment(lib, "rmm4.lib")
#pragma comment(lib, "rmu4.lib")
#endif

/* Specify the display driver to use - in this case   */
/* the 3Dfx driver. Note that the driver DLL name     */
/* depends upon whether this is the Debug or Release  */
/* version of the application.                        */
#if defined(_RMDEBUG)
```

```
#define DRIVER "rgd3dfx4d.dll"
#else
#define DRIVER "rgd3dfx4.dll"
#endif

/* Specify the RealiBase to load */
#define REALIBASE "helisim.rbs"

/* Specify the characteristics of the display device */
#define XRES 640
#define YRES 480
#define PIXELDEPTH 16
#define REFRESH 60

/* Specify the position of the view relative to top left */
#define XPOS 0
#define YPOS 0

/* Function prototypes */
void InqError(char *msg);
rtError imageCB (rtImageCBType type, rtID Id,
                 const char *fname, const char *path,
                 rtImage *info);

/* The main function definition */
void main(void)
{
    rtChannel cInfo;
    DWord key;
    DWord terminate = 0;
    Ord t, t0;

     /*————————————————————————*/
    /*— Step 1 : Initialise RealiMation libraries —*/
     /*————————————————————————*/
    RTInitialise();
     InqError("RTInitialise()");

     /*————————————————————————*/
    /*— Step 2 : Load the RealiBase               —*/
     /*————————————————————————*/
     RTSetImageCallback(imageCB);
     InqError("RTSetImageCallback(imageCB)");
    RTLoadRealiBase(REALIBASE, 0);
     InqError("RTLoadRealiBase(REALIBASE, 0)");

     /*————————————————————————*/
    /*— Step 3 : Create a graphics channel       ——*/
     /*————————————————————————*/
    rtID ChannelID = RTCreateID(RTChannelID);
     InqError("RTCreateID(RTChannelID)");

    cInfo.name = DRIVER;
    cInfo.res.width = XRES;
    cInfo.res.height = YRES;
    cInfo.res.depth = PIXELDEPTH;
    cInfo.res.refresh = REFRESH;
    cInfo.auto_load = 0;
    cInfo.n = 0;
    cInfo.data = NULL;

     RTSetChannel(ChannelID, &cInfo);
     InqError("RTSetChannel(ChannelID, &cInfo)");
    RTRealise(ChannelID);
     InqError("RTRealise(ChannelID)");

     /*————————————————————————*/
    /*— Step 4 : Add a predefined view to channel —*/
     /*————————————————————————*/
    rtID ViewID = RTGetNextID(RTNullID, RTViewID);
     InqError("RTGetNextID(RTNullID, RTViewID)");

     RTSetViewPortSize(ViewID, XRES,YRES);
     InqError("RTSetViewPortSize(ViewID, XRES, YRES)");
    RTSetViewPortPos(ViewID, XPOS, YPOS);
```

```
    InqError("RTSetViewPortPos(ViewID, XPOS, YPOS)");

    RTAddObject(ChannelID, ViewID);
    InqError("RTAddObject(ChannelID, ViewID)");
    RTRealise(ViewID);
    InqError("RTRealise(ViewID)");

    /*————————————————————————*/
   /*— Step 5 : Display Loop                    —*/
    /*————————————————————————*/
   t0 = RTInqClock();
   while ( !terminate )
   {
      t = RTInqClock();
       RTSetViewTime(ViewID, t-t0);
       RTDisplayView(ViewID);
       RTSwapPage(ChannelID);

      while ( kbhit() )
         key = getch();
      if ( key == 0x1B )       // Check for Escape key hit
         terminate = 1;
   }

    /*————————————————————————*/
   /*— Step 6 : Close down                      —*/
    /*————————————————————————*/
    RTUnrealise(ViewID);
    InqError("RTUnrealise(ViewID)");
    RTUnrealise(ChannelID);
    InqError("RTUnrealise(ViewID)");
    RTShutDown();
    InqError("RTShutDown()");
}

/*————————————————————————————*/

void InqError(char *msg)
{
   /* Function to enquire the RealiMation error status */
   /* and report an error string based upon the         */
   /* supplied message. This is only enabled for the    */
   /* Debug version of the application.                 */

   rtError err = RTInqLastError(0);
#if defined(_RMDEBUG)
   char errmsg[32];
    RTInqErrorString(err, errmsg);
   fprintf(stderr, "%s : %s\n", msg, errmsg);
#endif
   if ( err != RTNoError )
      exit(1);
}

/*————————————————————————————*/

rtError imageCB (rtImageCBType type, rtID Id,
                 const char *fname, const char *path,
                 rtImage *info)
{
   return RTNoError;
}

/*————————End of example code———————————*/
```

Summary

The document has presented the stages required to produce a simple, command line application using Microsoft Developer Studio and the RealiMation API. Extending this example further to make it more useful could include the following:

- Specify a RealiBase file as a command line argument.
- Add texture support by adding an image callback. See the entry for `RTSetImageCallback()` in the API help file.
- Use a Windows driver instead of a 3Dfx Voodoo one. See the RealiBench source code sample for details on creating a display window.

Even if you are not going to be writing command line applications, it is useful to know the steps required to make a “raw” application that is not encumbered by Windows messages and complex user interfaces.