TP003: Using RealiMation in Microsoft Windows™ Applications

Introduction

When writing Microsoft Windows based RealiMation applications there are certain aspects of program design and implementation that need to be considered. This paper firstly describes these things, and then concentrates in particular on MFC based applications, including a description of the programs generated using the RealiMation AppWizard.

Basic familiarity with MFC and Windows applications is assumed. For those readers who have little knowledge of such programs, a good starting point is to do the standard Microsoft Visual C++ “Scribble” tutorial. This will show you the basic structure and C++ classes of an MFC application. All the source discussed in this paper can be found on the RealiMation Developer CD in the SDK directory.

Typical Activities

All RealiMation applications need to do some or all of the following as part of their processing:

1. Initialise the RealiMation API.
2. Load RealiBase files.
3. Put a 3D picture into a window.
4. Allow for the window to be resized.
5. Allow different display drivers to be used within a window
6. Allow for a non-windowed display driver (such as a 3dfx card) to be used.
7. Animate a RealiBase.
8. Close down the RealiMation API and free all resources used.

The difficulty of these operations in a Windows environment is that applications are message driven and so where certain operations are done during the life span of a Windows application becomes important.

Readers that just want to see the basic structure of a RealiMation application, without the complications imposed by Windows, should read Technical Paper No 005 - A Simple Command Line Application.

Initialise (and Close) the RealiMation API

The single most important thing in a RealiMation application is that `RTInitialise()` should be called before any other RealiMation calls. When using the debug version of the API, developers will be warned via assertions when this has not happened. Release builds will probably just crash - just one of many reasons why developers should make use of the debug libraries, particularly in the early stages of development.

Depending on your application, there are two good places to call `RTInitialise()`:

In the `InitInstance()` member of the applications `CWinApp` derived object.

Just before a RealiBase is loaded, for example in the code that responds to a File Open.

The first case is normally used for applications that load a single RealiBase used for the whole time that the application is running. The best place for the corresponding `RTShutDown()` call is the `ExitInstance()` member of the `CWinApp` derived object.

The second case works well for applications such as RealiView which can swap the RealiBases used at any time under user control. In this case it is usual to put a call to `RTShutDown()` immediately before the initialisation call. If desired, this can be done depending on the result of calling `RTIsInitialised()`, which is a very useful function if your application wants to make RealiMation API calls but does not always know if the initialisation stage has been carried out due to the messaging mechanism of Windows. The shutdown function will delete all RealiMation objects, and free up all memory used *directly* by those objects. If applications have added their own properties to RealiMation objects, and these properties were generated with dynamic memory, then they must free up these resources at the same time - typically this is done just prior to calling `RTShutDown()`.

There are obviously other places in your code that you could use to initialise and shutdown RealiMation. The places mentioned above are just examples of those most frequently used.

Load a RealiBase File

The function `RTLoadRealiBase()` loads a previously saved scene description into memory so that it can be manipulated through the API for display and further processing. For applications that are fundamentally RealiBase viewers, the call to load the RealiBase typically follows very closely behind a call to `RTInitialise()`.

The majority of application will require that any images (for texture or backgrounds) are loaded when the RealiBase is loaded. This is done by supplying a callback function to the API using `RTSetImageCallback()` prior to calling `RTLoadRealiBase()`. This application defined function manages the memory for all images, giving a great deal of control over how images are used. See the API documentation for more details of this call.

Note that some applications do not need to load in images. For example, some developers write various off-line batch processing programs to do some specific kind of processing on a RealiBase. They typically load a RealiBase, do their processing, and save the RealiBase to disk again. Since image information is not destroyed if an image callback is not present (they simply do not get loaded into memory), time and code can be saved by not defining an image callback.

Getting a 3D Picture in a Window

Once the library has been initialised and a RealiBase loaded in, the next problem is to somehow get the RealiMation API to draw views from the 3D scene.

The key decision the developer has to make is which window to use to contain the 3D imagery (note that you can have more than one 3D window, just like the STE). You will give the `HWND` (Windows handle) to the RealiMation API, along with an indication of expected size of the display surface.

RealiMation uses a *channel* object to abstract a logical 3D display surface from a physical 3D graphics rendering device. In Windows, a window is the channel, and on this channel we can put one or more views - the list of objects that actually get rendered. Most RealiMation applications use a view found in the RealiBase file and add it to a channel object created locally.

In an MFC-based application, the 3D window will be described by a `CWnd` derived class, called `CChannel3D` for example. This will typically include a couple of member variables for holding the ID (RealiMation handle) of the channel and view objects. The channel object will typically be created at some stage during window initialisation.

For a more detailed discussion on this process, see Technical Paper #002 - The RealiMation Scene Graph.

Resizing the Window

Why should an application allow the window displaying the 3D view to be resized? Primarily, most users of Windows expect to be able to resize their applications. Preventing this for a 3D application will not present a very good impression to the user about the flexibility and “Windowness” of the program.

Another reason is to do with memory usage and speed. Generally it can be said that the smaller the window the less main or graphics card memory is being used to display the 3D view, therefore the scene will be animated at a higher frame rate. This can be very important if the scene being displayed is complex with many textures, or if the display driver being used fails when it runs out of memory. Allowing the user to control the size of the window allows them some measure of control over the frame rate.

For example, Direct3D fails to display anything at all when it runs out of memory, and Open GL drivers will often revert to software mode. This could cause the user some anxiety when the view disappears or the frame rate suddenly drops when the window changes size.

Some applications fix some maximum size for the window to reduce the likelihood of this problem occurring. Others, like the STE, warn the user when the window is grown too big, and then wait for the window to be reduced in size again before trying to access the card again.

Using Different Display Drivers

RealiMation is 'renderer independent', i.e. it is not built upon one particular display technology such as OpenGL. It has been designed so that different 3D display mechanisms can be used, and, given underlying operating system support, that these renderers can even be dynamically swapped at run time.

There are several advantages to this approach:

- **Portability** - One particular rendering technology may not be available on all the different combinations of CPU type and operating system that our customers want to use with RealiMation. For example, hardware accelerated Direct3D is not currently available under Window NT. Therefore an application written under Windows 95 using Direct3D would not run under Windows NT.
- **User choice** - Different 3D display technologies have different rendering 'artifacts', normally determined by the tradeoffs that each technology decided to make when being developed. For example, one renderer might decide to sacrifice quality of display for speed. Since every user has a different set of requirements, we decided not to limit our users to one particular set of rendering tradeoffs. The user can decide what rendering artifacts are acceptable in their particular application.
- **Future Proof** - There are new 3D display technologies being announced all the time. By not being tied to any one renderer, RealiMation can expand its capabilities in line with new developments.

RealiMation applications can take advantage of these benefits by allowing their users to choose the display driver to use. Allowing the user to select the display driver means that one program can run on almost any combination of Windows operating system and graphics card.

Using Non-windows Display Drivers

Why would an application want to use non-windows display drivers - such as 3Dfx/Glide?

The most obvious answer is given by the benefits of renderer independence explained in the previous section. There are, however, other reasons to support rendering technologies that do not work within a window:

- Some advanced and very powerful rendering technologies are only available as full screen devices independent of the Windows desktop. Examples are the 3Dfx Voodoo chips, Real3D Pro1000, and the Datapath Merlin card (which can do high resolution anti-aliasing, and live video texturing).
- Improved performance can be gained by use a 3D graphics card to display direct to the screen without going through Windows. One drawback, however, might be that only one view can be displayed at any one time. A great advantage of RealiMation is that applications

can run multiple renderers simultaneously so that if the application wishes to show multiple views, yet still use a full screen driver, then subsequent views can be displayed using a window based display driver such as Direct3D or OpenGL (or both!). To allow for this your application must have the concept of a secondary or alternate display driver. The STE is a good example of a program offering this kind of extreme flexibility in rendering choice.

- Multi-screen applications need to use a non-windows display driver with each screen being driven by a separate graphics card. For example, a 3 screen system can be built by plugging 3 cards into one computer.

Animate a RealiBase

A RealiBase is not just a set of static 3D objects. Objects, cameras, and lights can have automatic motion attached to them using the *path* RealiMation object.

Motion in RealiMation is parameterised by time, so that you can specify that an object is at position *P1* at time *t=0*, and position *P2* at time *t=10*. By convention, time is in seconds. The RealiMation system interpolates the motion between these *P1* and *P2* over the time interval.

By default, RealiMation applications will normally generate the time value to feed into the motion interpolator from the computer's real-time clock. So, no matter how fast your card actually generates each picture, the object will *always* be at P2 after 10 seconds of real time. The only difference between a fast graphics display and a slow one, is that the fast display will show smoother movement.

In code terms, you can see the difference as follows:

<u>Using Real-Time Clock:</u>

```
float tStart = RInqClock ();  // returns current system time in seconds
for (i = 0; i < nFrames; i++)
{
    // Set the time value for the view
    RTSetViewTime (ViewID, RTInqClock() - tStart);
    RTDisplayView (ViewID);      // Generate the polygons
    RTSwapPage     (ChannelID);  // Show the polygons on the display surface
}
```

<u>Using a fixed time delta instead, however, the main loop goes to:</u>

```
float t = 0.0F;
float tDelta = 0.1f;  // Time delta is 1/10th second
for (i = 0; i < nFrames; i++)
{
    RTSetViewTime (ViewID, t);  // Set the time value for the view
    RTDisplayView (ViewID);      // Generate the polygons
    RTSwapPage     (ChannelID);  // Show the polygons on the display surface
    t += tDelta;                 // Increment time delta
}
```

The effect of a fixed time delta is that objects do not move in "real-time", but relative to the performance of the hardware.

This section describes several different ways of letting time pass in a view and making the view update on the screen. Each has their own advantages and disadvantages depending on the application.

Applications should also consider how they want to define "time". As mentioned above, the time value is simply a parameter that controls the evaluation of an equation which varies the position, orientation, and speed of an object. This value can be generated from your computer's real-time clock, so that one second of "program time" is the same as one second of actual time (as seen on your wristwatch). Alternatively, an application may, each time around the program loop, simply add a fixed increment to the time value.

The STE and RealiView allow both these mechanism to be controlled by the end user from the Customise | RealiMate dialog. In most situations, users want the real-time motion. RealiBench, however, only allows a fixed time increment, since it needs to guarantee that each run of the benchmark is being asked to draw exactly the same polygons, in the same location, size, lighting etc., each and every frame.

There are several ways to generate time and animate a scene:

1. When the user or program invokes a "Play" operation, the object controlling the 3D view window can just go into a loop, incrementing the time (using either of the methods above), and displaying the view. The disadvantage is that there is no way for the Windows user interface to regain control of the application.
2. Use a Windows timer to generate a paint request. This means your code will respond to a `WM_TIMER` message, update the time value (as above), and invoke the drawing method of the object representing the 3D window. This mechanism allows other Windows messages to execute. The disadvantage is that a Windows timer has limited resolution, effectively meaning an upper bound on frame rate of around 25Hz on Windows 95 systems.
3. Use a second thread to send a message to the main display thread to update the time and redraw the view. This, while the most complex to implement, provides the best performance while still allowing all other operations in your user interface to continue.

The RealiMation AppWizard

The RealiMation SDK includes an Application Wizard for use with Microsoft Visual C++ 5.0 which generates a basic shell of a RealiMation application. The functionality is quite limited, but it does implement solutions to all the typical activities listed above.

For a more detailed tutorial on using this Wizard, see Technical Paper #004 - Your First RealiMation Windows^{TM} Application.

If the basic application is built and run with no modification it allows the user to open a RealiBase, animate it, resize the window and change the display driver. The application can also swap RealiBase and closes the current one down on exit. We will now look at how this application has implemented the features previously discussed. Please note that this is not the only way to implement a real-time 3D application with RealiMation, but it does provide an excellent starting point with an absolute minimum of effort.

Initialisation and loading a RealiBase

The basic application actually calls `RTInitialise()` in both of the places mentioned. The first is in `CRealiApp::InitInstance()`. This allows the application to change display drivers without a RealiBase being loaded. The second is in `CRealiApp::LoadRealiBase()`.

This calls `RTSetImageCallback()` to assign the image call back function for handling textures. If there is already a RealiBase in memory this routine also calls `RTShutDown()` to close it down, and resets the 3D display.

Once the RealiMation libraries have been initialised the application can call `RTLoadRealiBase()` to bring the scene into memory.

Shutting down the application

This is handled by `CRealiApp::ExitInstance()`. This stops the animation controller (described later) , calls `RTShutdown()` and then calls the base `CWinApp::ExitInstance()` to handle the rest of the shutdown process.

Getting a 3D picture onto the screen

The AppWizard generates a simple applications with a resizeable main frame window. It uses the client area of this window (which, in MFC terms is a separate window) to display the 3D graphics in. This client window is encapsulated in a class of type `CChannel3D`, which represents a RealiMation channel object.

When a RealiBase is loaded, the application looks for a view and gives this to the CChannel3D class, which creates a channel object, sets up the display driver, adds the view, and looks after Windows paint messages to ensure it is drawn.

Rather than go into great detail about how to get a 3D picture onto the screen it is simpler to read the source in `Channel3D.cpp` for how the discussion above has been implemented.

Handling the window resize

The resizing of the 3D view is handled by the `CChannel3D::OnSize()` method. One problem to be avoided, however, is the many unnecessary re-realisations and redraws that can happen while the view is being resized by the user dragging an edge of the main frame window around (especially under Windows NT with its dynamic window operations).

To fix this, the `CMainFrame` class (which encapsulates the main application window) communicates with the `CChannel3D` class whenever such a dynamic resizing is happening. It calls `CChannel3D::SetResizing()` to indicate that a resize is being performed (see `CMainFrame::OnSizing()` and `CMainFrame::OnExitSizeMove()`). The `OnSize` method of `CChannel3D` does not attempt view and channel realisation when in this dynamic resize state.

Changing display drivers

The method `CRealiApp::OnDisplaydriver()` is the entry point to the display driver selection code.

This calls `CChannel3D::SelectDisplayDrivers()` which does all the initialisation of the display driver dialog and acting on the result. This dialog allows the user to set either the primary or secondary (alternate) display driver.

Animating the scene

This is handled by the `CAnimate` class as defined in `animate.h` and `animate.cpp`. These files also contain the `CTicker` class which sets the timer used to control the animations.

The code an be compiled in two ways, depending on a preprocessor directive in `animate.h`. One way just sets up a standard Windows timer, and each call to the timer callback causes the view to update.

With second method, which is the default, the ticker runs in a second thread, which simply sends timer messages to the main application frame. This then updates the views in exactly the same was as the other timer mechanism.

The reason the second thread sends a message is that calling the update method directly breaks MFC’s multithreaded rules (see Microsoft documentation for details). Even though this implementation uses a `WM_TIMER` message, it could just as easily have used a custom `WM_USER` message instead. The second thread has a control on it that prevents tool many messages from swamping the system - see code for details.

The `CAnimate` class contains a list of views that are currently animating. Even though this application has only one view, using a list means that it is easily extendible if required.

When the animation is started the view to be animated is added to the list of currently animating views and the `CTicker::Update()` method is called to start the timer if it is not currently running.

When the timer responds to a `WM_TIMER` message it calls `CAnimate::Tick()`. This loops round all the views on the display list, inquiring the current time and incrementing its current time variable (`m_Time`). This value is passed into `RTSetViewTime()` together with the view ID to set the time value for the view. This value will be used in subsequent calls to `RTDisplayView()`. The method `CChannel3D::DrawView()` is then called to actually display the view. This method basically calls `RTDisplayView()` to render all the items in the view to an off-screen display buffer and `RTSwapPage()` to actually display the off-screen display buffer. Double buffering gives a smoother animation as the user will not see a blank screen between frames.

Summary

RealiMation applications integrate well within Windows 95, 98, and NT. The flexible display driver architecture means that any application can offer an extreme amount of flexibility and choice to the end user in how 3D real-time images are displayed.

The RealiMation AppWizard generates the full source code and project options for a developer to start integrating their code with immediately. It takes care of the most common issues of programming within a message driven environment, allowing a programmer to concentrate on their own task. The code can be completely customised to suit the developers needs and requirements.