RENDITION
1675 North Shoreline Blvd.
Mountain View, CA 94043
650-335-5900
http://www.rendition.com

Rendition

RRedline™ Programming Guide

Guide to the RRedline HAL for Vérité™ Graphics Rendering

RRedline Programming Guide

Getting Started

The RRedline HAL provides an interface for applications to directly access Rendition Vérité hardware for optimized 2D and 3D graphics rendering. This guide introduces RRedline and explains important concepts needed to use it.

What is the Vérité?

Rendition's Vérité V1000 is a single chip 2D/3D graphics engine, GUI accelerator, digital video accelerator, and VGA engine. The Vérité V2000 series of chips increase the functionality of the V1000 by adding per vertex specular, video in/out operations, and dramatically improved 2D and 3D performance. In this document, we'll refer to both architectures as just 'Vérité'.

The Vérité employs a unique architecture, combining a fully programmable RISC core with a hard-wired pixel pipeline (known as the "Pixel Engine"). This design provides very high performance while maintaining a great deal of flexibility through the use of custom microcode.

For 2D, 3D, and video operations, work is divided between the RISC and Pixel Engine. The RISC engine performs setup operations and passes drawing commands to the Pixel Engine. The Pixel Engine supports texture mapping with correct perspective, bilinear filtering, Z buffering, alpha blending, fog, and other advanced rendering features, such as color space conversion for digital video.

Utilizing the Vérité RISC core for setup operations reduces the load on the CPU for 3D drawing operations such as gradient calculations. The CPU is therefore freed to perform application tasks such as simulation and other non-graphics processes.

Additionally, the Vérité is a DMA busmaster, which allows it to transfer commands and data asynchronously from host memory to the Vérité command FIFO. This asynchronous operation allows both the CPU and Vérité to run at close to peak rates simultaneously.

What is RRedline?

RRedline was developed with the input of many game and application developers to provide a powerful, flexible interface. Only one version of a RRedline application is required to run on any of the Vérité chipsets.

It is an immediate-mode, device coordinate, rasterization interface to the Vérité. The application must perform all transformation, lighting, clipping, and screen projection of vertices before sending the vertices to RRedline. RRedline is currently a fixed-point data interface to the Vérité.

RRedline includes functions for memory management, command and data (texture-

map) transfer, setting drawing state, and drawing primitives.

Supported Platforms

RRedline is a Windows 95 interface for Pentium-class PCs. Those familiar with programming the Vérité in DOS will see that RRedline bears a striking resemblance to the original Rendition Speedy3D interface. Rendition is investigating NT support for RRedline.

RRedline currently supports the following graphics cards. Note that RRedline requires at least the 2.0 series of display drivers. The latest drivers have been included in the SDK for development purposes only. For final testing of products, you will want to get, at least, each vendor's 2.0 display drivers. V2000 series cards will initially support RRedline.

- **Stealth II from Diamond (V2100)**
- **Thriller 3D from Hercules (V2200)**
- **Bonnie & Clyde by Jazz (V2200)**
- **Outlaw 3D by Jazz (V2200)**
- **GLadiator by DSystems (V2200)**
- **Vision 1 from QDI (V2200)**
- **V-Raptor 3D from Genoa Systems (V2200)**
- **Magic Video 3D from I/O Magic (V1000)**
- **Tornado 3D from Max 2 Tech (V1000)**
- **Total 3D from Canopus (V1000)**
- **3D Blaster PCI from Creative Labs (V1000)**
- **Intense 3D 100 from Intergraph (V1000)**
- **miro CRYSTAL VRX from Miro (V1000)**
- **Screamin' 3D from Sierra On-Line (V1000)**
- **Rendition development boards (V2200/V1000)**

Supported Compilers

RRedline currently supports Microsoft Visual C++ (version 4.0 and later), Watcom C/C++ (version 10.5 and later), and Borland C++ (version 5.0 and later).

Verite vs. Redline

RRedline is comprised of two layers. The lowest layer, `verite.lib`, is the platform dependent Hardware Abstraction Layer (HAL). Specifically, the function categories which fall into `verite.lib` deal with host and video memory allocation, command transfer, and display modes. All function calls beginning with *V_* are part of `verite.lib`.

Above `verite.lib` sits `redline.lib`. This higher level library provides helper functions to ease the development process. It includes wrappers around functions dealing with the Vérité handle, memory allocation, etc. It also describes and creates the commands for the Vérité to execute. All function calls beginning with *VL_* are part of `redline.lib`.

In general, the most successful approach will be to use `redline.lib` extensively to develop the application, then optimize the most time-critical sections of the

to develop the application, then optimize the most time critical sections of the application by talking directly to `verite.lib`.

The Build Environment

Setting up your compiler

Refer to the documentation for your compiler if you need instruction regarding this process.

Includes

All the necessary header files are in the `include` directory in the SDK. Add this directory to your include search path. If you are using any of the source code from the `utils` directory, the associated headers are found there; add it, too.

Though your application need only include `verite.h` and `redline.h`, these files reference other files in the `include` directory.

Libraries

The RRedline import libraries for the Microsoft and Watcom compilers are in the `lib` directory. Libraries for the Borland compiler are located in the `lib/Borland` sub-directory. You can either add this directory to your library search path or specify the libraries by their full paths.

Bug Reporting

If you are not a member of our developer's program, you can report bugs via Rendition's web site. You will find a bug reporting form on the Developer page on the Rendition Web Site. If you cannot use the web-based form, fill out the form provided in `doc\bugrpt.txt` and email it to rredline_bugs@rendition.com. Use the words "RRedline Bug" in the subject line.

Trademarks

Windows 95 and Windows NT are trademarks of Microsoft. All other product names are trademarks of their respective owners.

Hello Triangle

This section contains a quick introduction to RRedline basics. By explaining each line of a simple example application along with the data structures used you'll become acquainted with the minimum requirements necessary to get your application to run with RRedline. The code here is also quite reusable requiring little or no modification to get it to work in your own applications.

The code we're going to describe here and in later examples is complete and runs well on the Vérité. However, for real applications you'll want to modify your routines according to the methods discussed in the chapter titled *Optimizations*. This will result in the best performance on our hardware.

The example we are going to use draws a flat shaded red triangle in the middle of the entire screen. After working with this example, you should know how to do the following tasks:

- Write error handlers
- Initialize the Vérité chip
- Create the command buffers needed for Vérité commands
- Create the surfaces required for the front and back buffer
- Draw a simple triangle into the back buffer
- Flip the front and back buffers to display the triangle

• Shut down the Vérité chip

This first example will be quite thorough, explaining in detail each of the RRedline functions used. Later in this document, other examples will be provided showing how you can modify the starting Hello Triangle and provide additional features. All the code for the Hello Triangle examples can be found in the *htri* sub-directory of the RRedline SDK's example code section. The initial version of this example can be found in *htri1* while modified versions are found in successive htri directories.

Some bookkeeping

Before we start using RRedline functions, we must prepare all the information required by RRedline. Since RRedline is a Windows interface, it requires that a window be created and passed to the initialization routine. Even if the application is drawing to the entire screen and not drawing to an individual window, RRedline still requires one. The window for this application will be created within the main routine. The code for this part of the example is not included here but can be found in the *main.c* file. However, in order that we can discuss the general structure of this function, the pseudo-code follows:

• Create a 640x480 window

• Initialize Vérité with call to *redline_Init()*

• Run main application loop:

  • Render scene with *render_scene()*

  • Exit loop when 'Q' character is hit

• Close Vérité with *redline_Close()*

All the application is doing is creating a window of size 640 by 480, passing a handle of the window to the application's initialization routine *redline_Init()* and then starting up the process loop. Within the process loop the program renders the flat shaded triangle with *render_scene()*. Once the key 'Q' has been hit, the application exits out of the main loop and calls *redline_Close()* to close down the Vérité. We'll be focusing on explaining each of these functions in turn along with all the necessary RRedline calls. However, before we start calling RRedline we need to know how errors are returned from each function and how to handle them.

Writing Error Handlers

Many RRedline functions have the possibility of returning an error code during their execution such as when the board runs out of memory or when improper arguments are passed to a function. When this occurs the program has the possibility of handling it in two ways. They can either handle the return code from each function, displaying error messages and exiting cleanly when an error is detected, or they could install error handlers that will handle all possible error conditions.

When you install error handlers, every time RRedline runs into an error condition, a handler will be called. If an error occurs within `verite.lib` the handler installed with *V_RegisterErrorHandler()* will be called. If an error occurs within `redline.lib` the handler installed with *VL_RegisterErrorHandler*() will be called. Note that some `redline.lib` function calls implement their functions with `verite.lib` calls. In other words, it is possible that the error handler installed for `verite.lib` will be called when a `redline.lib` function is called by your application. It is also possible that functions in `redline.lib` are implemented by other functions in `redline.lib`. Functions in `verite.lib` never call functions in `redline.lib`.

In each of these error routines you can display an appropriate error message, clean up

In each of these error routines you can display an appropriate error message, clean up the Vérité states, return back to the calling routine, or exit from the application. In the case of Hello Triangle, we won't return back to the calling routine. Instead we'll just display the error message, clean up the Vérité state, and exit from the program.

To do this, we'll need three functions. The first simply displays an appropriate error message to a Windows message box. This routine will be called whenever we need to display an error.

```
void redline_ErrorBox
       ( LPCTSTR     lpText )
{
       // display error message
       MessageBox(NULL, lpText, "RRedline Error",
       MB_ICONERROR | MB_OK);
}
```

The next two routines required are the error handlers themselves. The routine *verite_ErrorHandler()* will be installed as the error handler for `verite.lib` calls and the routine *redline_ErrorHandler()* will be installed as the error handler for `redline.lib` calls. They are extremely similar to each other. Similar enough that we'll just discuss *verite_ErrorHandler()* and tell you what function calls and constant values differ for *redline_ErrorHandler()*.

The first thing to notice is the declaration of the handler. Any error handler you write must have an identical declaration. The *V_CDECL* is especially important since it asserts that the arguments are passed with a very specific calling convention irrespective of the compiler you're using.

```
void V_CDECL verite_ErrorHandler
       ( v_handle    verite,
         v_routine    routine,
         v_error     error,
         int         ext )
```

The first piece of code in the routine is meant to insure that while you're in the error handler that calls made to other RRedline routines, like during the cleanup process, don't result in another error and another call to the error handler. If they do, the program will detect that recursive errors were found and will exit with a generic error message.

```
{
       static unsigned int entry = 0;
       char   func_name[64];
       char   err_text[128];
       char   err_message[1024];

       if (entry++)
       {
              // called recursively (during closing)
              redline_ErrorBox("Recursive Verite errors.");
              ExitProcess(1);
       }
```

The next piece of code from our error handler makes some RRedline calls to try and extract the name of the function that caused the error and extract the error text describing it. For *redline_ErrorHandler()*, the functions *VL_GetFunctionName()* and *VL_GetErrorText()* are used instead of *V_GetFunctionName()* and *V_GetErrorText()*. Also the returned *VL_SUCCESS* is used for *redline_ErrorHandler()* and not *V_SUCCESS*. In order to prevent recursive errors from occurring, RRedline insures that none of those error extracting functions will ever cause the error handlers to be called. Instead they return errors only through returned error codes.

```
	// get name of function that caused the error
	if (V_GetFunctionName(routine, func_name, 64) != V_SUCCESS)
		sprintf(err_message, "Error retrieving error function
name.");

	// get text of error
	else if (V_GetErrorText(error, err_text, 128) != V_SUCCESS)
		sprintf(err_message, "Error retreiving error text");

	// store error message
	else
		sprintf(err_message, "Error in routine:\n%s\n%s\n0x%x",
			func_name, err_text, ext);
```

The last section of code displays the resulting error message using the previously described routine *redline_ErrorBox()*, frees resources and shuts down the Vérité with a call to *redline_Close()*, and exits the program. Don't worry too much about the *V_SetDisplayType()* call. All we're doing here is just making sure the Vérité is in windows mode so the message box is displayed correctly. We'll talk more about windows mode in *Modification One: Switching Between Full Screen Mode and Windowed Mode*. The function *redline_Close()* will be discussed at the end of this example.

```
	// change to windowed application if verite active
	if (verite)
		V_SetDisplayType(verite, V_WINDOWED_APP);

	// display error message
	redline_ErrorBox(err_message);

	// close down verite and exit
	redline_Close();
	ExitProcess(1);
}
```

Initializing RRedline

Now that we've got our framework and error handler written it's time to do some actual RRedline calls. We'll start with the routine *redline_Init()* called just before the main process loop. Versions of this routine should exist in every RRedline application. The purpose for this routine is to initialize the Vérité and all of the global variables used to process drawing commands. Definitions of globals used can be found in the file *htri.h*.

The first thing to do before making any other RRedline calls is to register our error handlers. Note that if either of these calls fail, they will not cause error handlers to be run. Because of this, we need to explicitly check the returned error code.

```
void redline_Init
	( HWND hWindow )
{
	int		i;

	// register error handlers
	if ((V_RegisterErrorHandler(verite_ErrorHandler) != V_SUCCESS)
||
	    (VL_RegisterErrorHandler(redline_ErrorHandler) !=
VL_SUCCESS))
```

```
    {
        // display error message
        redline_ErrorBox("Unable to register error handlers");
        ExitProcess(1);
    }
```

The first global to initialize is the handle to the Vérité device. This next call opens the Vérité device and stores a handle to it in the global *verite*. In turn this handle will allow us to make other RRedline function calls. This is the only RRedline call that requires a window handle.

```
    // use first board found
    VL_OpenVerite(hWindow, &verite);
```

The next piece of code shows us how the global *verite* can be used.

```
// set display type
V_SetDisplayType(verite, V_FULLSCREEN_APP);
V_SetDisplayMode(verite, 640, 480, 16, 60);
```

In the call to *V_SetDisplayType()* we are telling the chip that the application will run in full-screen mode using the entire screen for the application and not restrict drawing to a window. The following call to *V_SetDisplayMode()* tells the chip that the full-screen window will be 640 by 480 pixels, have 16 bits of color, and run at a refresh rate of 60 Hz.

Later in this example you'll see other calls that use the *verite* global. In fact, most RRedline calls will use the *verite* global as one of its arguments or a pointer to a command buffer created in the next section.

Creating the Command Buffers

The next topic covered is probably one of the most unique features of the Vérité architecture and so requires a little more explanation.

When drawing with the Vérité you may have just expected to call functions like "draw_triangle" and have it immediately result in a drawn triangle. However, passing commands and getting results synchronously with the CPU does not result in the best Vérité performance. The best way is to work with the chip asynchronously through command buffers. That instead of passing information directly to the chip, the commands are stored in a command buffer created by the application. Later, when the program explicitly chooses to, the command buffer is issued. Issuing simply means that the chip is notified that commands are ready on a specific buffer. Then, when the Vérité next becomes available, it will pull these new commands over from system memory via DMA and start executing them. The CPU is not involved during the transfer of these commands and could be processing code in other parts of your application.

This then brings us to our next problem. When the command buffer is being issued the Vérité gains control over that buffer. It would then be a bad idea to try and add commands to that same buffer or you might be overwriting data that the Vérité was actively reading. The way to prevent this is to have a ring of buffers. The idea being that while you are adding commands to one buffer the Vérité is executing another previous buffer.

The next section of code from *redline_Init()* shows how this ring of command buffers is created.

```
    // create command buffers
    for (i=0; i<REDLINE_NUMCMDBUFS; i++)
    {
```

```
            // create a command buffer with the specified sizes
            cmdbufs[i] = V_CreateCmdBuffer(verite,
REDLINE_NUMDMAENTRIES,
                  REDLINE_CMDSIZE);


            // install callback to run when a buffer runs out of
space
            V_SetCmdBufferCallBack(cmdbufs[i],
redline_IssueCommands);
      }
```

This bit of code required depends on some constants we've defined in our include file *htri.h*.

```
// command buffer sizes
#define REDLINE_NUMCMDBUFS          8
#define REDLINE_NUMDMAENTRIES 1024
#define REDLINE_CMDSIZE             4096
```

All were doing is creating eight command buffers, each with a specific DMA entry list size and command list size. For now, think of DMA entries as being the part of the command buffer that stores information about memory addresses being passed to the Vérité and the command list entries as being the part that actually contains commands and some actual data passed to the Vérité. For this simple application, we could have easily gotten away with two much smaller command buffers. For your own application you'll end up playing around to determine the best constant values. Making the buffers large enough to process the commands you require but making them small enough to prevent too much lag from occurring.

After creating the command buffer you'll notice that we're making a call to *V_SetCmdBufferCallBack()*. The purpose of this set up an overflow function. What this does is prevent the program from placing too many commands on any given command buffer. That when a buffer doesn't have enough room to contain a command the callback routine is automatically called so that the command can be placed on another buffer returned by the callback.

This callback function will need to do several things. First, it will need to issue the current command buffer that caused the overflow. Next, it will need to move to the next available entry in the ring of command buffers. Lastly, it will need to return a pointer to the current command buffer. Later on we'll see that our function meant to simply issue command buffers, *redline_IssueCommands()*, does just that.

The next section of code in *redline_Init()* sets up the globals required to traverse the command buffer ring and add keep track of the active buffer.

```
      // set initial command buffer state
      cmdbuf_index = 0;
      cmdbuffer = cmdbufs[0];
```

The global *cmdbuf_index* is simply used to tell us which command buffer in the ring is currently being used and the global *cmdbuffer* is a pointer to the currently active buffer in the ring. The global *cmdbuffer* will be used throughout the application to add commands to the buffer while *cmdbuf_index* will simply be used to move from one ring buffer to the next.

Surfaces

The next structure we'll discuss from the RRedline API is the surface structure *v_surface*. This structure is very important since display buffers and textures are all implemented as surfaces.

Within a surface structure are a number of buffers along with their description. A buffer is simply a section of memory on the board that can be used as a front buffer, a back buffer, an additional display buffer, a Z-buffer, or even a texture. How many they are and how they are used is entirely up to you as long as you have enough memory left on the board.

Within the surface structure are the common attributes of the contained buffers. This includes width and height in pixels, pixel format used, chromakey value, color padding flag, whether or not the values are ordered BGR or RGB, the clamping mode of the texture, and the 4-bit palette (if any).

```
typedef struct _v_surface {
   v_u32 width;
   v_u32 height;
   v_u32 pixel_fmt;
   v_u32 chroma_color;
   v_u32 chroma_mask;
   v_u32 chromakey;
   v_u32 color_pad;
   v_u32 bgr;
   v_u32 clamp;
   v_u32 *palette;
   v_u32 start_index;
   v_u32 num_entries;
   v_u32 memsize;
   v_u32 num_buffers;
   v_u32 buffer_mask;
   v_buffer_group buffer_group;
} v_surface;
```

Most of the attributes of this structure you won't need to worry about. However, for this example and for the examples to follow you will need to know about the following elements of the structure:

width Width of each contained buffer in pixels. No surface may have a width greater than the width of the primary surface. No width may exceed 5,120 for the V1000 series or 5,248 for the V2000 series.

height Height of each contained buffer in pixels.

pixel_fmt Format of pixels contained in buffers. Pixel size tells how many bits/pixel. Description tells how many bits are used for each of the color components. First bits specified in description are the high-order bits in the pixel.

Pixel	**Pixel size (bits)**	**Description (R for red, G for green, B for blue, A for alpha)**
V_PIXFMT_332	8	3 R bits, 3 G bits, 2 B bits, A = 255
V_PIXFMT_8I	8	R = G = B = 8 intensity bits, A = 255
V_PIXFMT_8A	8	R = G = B = 255, 8 A bits
V_PIXFMT_565	16	5 R bits, 6 G bits, 5 B bits, A = 255
V_PIXFMT_4444	16	4 A bits, 4 R bits, 4 G bits, 4 B bits
V_PIXFMT_1555	16	1 A bit, 5 R bits, 5 G bits, 5 B bits
V_PIXFMT_4I_565[2]	4	Palette contains 5 R bits, 6 G bits, 5 B bits

V_PIXFMT_4I_4444[2]	4	Palette contains 4 A bits, 4 R bits, 4 G bits, 4 B bits
V_PIXFMT_4I_1555	4	Palette contains 1 A bit, 5 R bits, 5 G bits, 5 B bits
V_PIXFMT_8888	32	8 A bits, 8 R bits, 8 G bits, 8 B bits
V_PIXFMT_Y0CRY1CB	32/2 pixels	Standard 4:2:2 format: (Y_0, C_r, Y_1, C_b)

clamp Contains flag bits *V_SURFACE_UCLAMP* and *V_SURFACE_VCLAMP*. See *Modification Two: Drawing a Texture Mapped Triangle* for more details on clamping.

buffer_mask Contains flag bits telling what kind of buffers are included in the surface structure. Bits used are *V_SURFACE_PRIMARY*, *V_SURFACE_Z_BUFFER*, and *V_SURFACE_INTERLEAVED*.

The only buffer that isn't affected by these common attributes is the Z-buffer. This kind of buffer is created if the buffer mask contains *V_SURFACE_Z_BUFFER*. It is only bounded by the width and height values contained in the structure. It always contains 16 bit values and there can only be one of them in each surface. If a surface contains a Z-buffer it is always the last buffer in the structure.

Another special kind of buffer is the primary buffer. This kind of buffer is created if the buffer mask contains *V_SURFACE_PRIMARY*. Only one of these buffers can exist per application. This buffer is made to point to the memory that exists on the screen. In other words, this is your front buffer. One more constraint having to do with the primary surface is that no other surface can have a width greater than the primary buffer. Other than that, any width and height can be used – a multiple of two is not required. In addition, the primary surface can only have a pixel format of *V_PIXFMT_565*.

You may have noticed that four texture formats have additional restrictions. The pixel format *V_PIXFMT_332* requires that the extension *PixFmt_332* exists and the formats *V_PIXFMT_4I_565*, *V_PIXFMT_4I_4444*, *V_PIXFMT_4I_1555* require that the extension *PixFmt_4I* exists. Extensions are simply a way to determine if a specific functionality exists on a given card. While these pixel formats are available on both the V1000 and V2000 series of chips they may not be available on future generations of our architecture. Therefore, you need to check for an extension before using them. For more information on extensions, refer to the function *VL_GetExtensions* in the reference guide or look at *Modification Five: Specular Extensions* for an example that uses extensions.

In our first example we don't have Z-buffers, textures, or additional display buffers to worry about. All we need to do is create a front buffer as a primary surface and a back buffer to draw to. The next piece of code from *redline_Init()* is used to create these two buffers within the board memory.

```
// create front and back buffers
    VL_CreateSurface(verite, &redline_Display,
           V_SURFACE_PRIMARY, 2, V_PIXFMT_565, 640, 480);
```

Both buffers are created with one call to *VL_CreateSurface()* with the result being stored in the global surface structure *redline_Display*. Here we are creating a surface that contains the primary buffer of the application, creating two buffers total, using a *V_PIXFMT_565* pixel format, and using a size of 640 by 480. Note that the width

and height given is the same passed to the *V_SetDisplayMode()* call and the number of pixels specified is also the same (*V_PIXFMT_565* uses 16 pixels). This matching is a requirement for full-screen applications. In addition, since this surface contains the primary buffer, the front buffer of this surface will contain the pixels displayed to the screen.

Once we have created these buffers we need to tell RRedline that all drawing commands should go to the back buffer of this surface. This is done with one call.

```
    // install destination of drawing commands
    VL_InstallDstBuffer(&cmdbuffer, redline_Display);
```

This call tells the Vérité to send drawing commands to the back buffer of that surface. If a back buffer doesn't exist, they'll go to the front buffer instead. The interesting thing about this call is that the used buffer doesn't have to be a special type but can be any kind of allocated buffer. For example, you could draw to a buffer that was later used as a texture.

Note that creating the surfaces requires the global *verite* as an argument and that the call to install this surface as the destination requires the *cmdbuffer* global. This is a good way to know how a command is being implemented. If it requires *verite* then the operation may cause the Vérité to run synchronously with the CPU and should not be done within the speed critical portions of the game. This is the case with *VL_CreateSurface()*. On the other hand, the call to *VL_InstallDstBuffer()* goes on the current command buffer. Any command placed in these buffers can occur asynchronously with the CPU and can occur during the speed critical portions of the application.

Other Data Structures

In addition to the *v_surface* structure you may notice other structures located in RRedline such as *v_memory*, *v_handle*, *v_cmdbuffer*, and *v_buffer_group*. You may also notice that the description of those structure's elements does not exist in any of the supplied header files. In order to keep control over our internal data structures, values of the above types are implemented as opaque pointers. In other words, you cannot directly reference an element of those structures. Instead, you must call one of the supplied functions to extract any required information (search the reference guide for functions starting with "VL_Get" and "V_Get"). This allows us to change the contents of those structures without requiring any modifications to your application's code.

Specifying the Source Function

Now that we have a buffer to render to, it now necessary to tell how. This is done with the source function. The purpose of this function is to tell the Vérité whether or not to use a texture map as the source and, if using a texture, how it should be used. This function could tell us to use colors from a texture map only, combine colors from a texture map with vertex information, or to use just the color data stored for the vertices. In this case, we'll specify that we don't need a texture and that information stored within vertices only should be used.

```
   // when drawing, use vertex colors or global colors
      VL_SetSrcFunc(&cmdbuffer, V_SRCFUNC_NOTEXTURE);
```

Later in *Modification Four:Alpha Blending* we'll go into further detail about the other source function modes since they affect alpha blending as well as rendering modes.

The last two calls in *redline_Init()* simply clear the back buffer and flip the buffers so that the start up screen is clear. These functions will be talked about in more detail in

the next section on utilities.

```
   // clear front and back buffers
      redline_ClearDisplay();
      redline_PageFlip();

      // no errors
     return TRUE;
}
```

Utilities

This section deals with utility functions that should appear in some form in all RRedline applications. These functions are extremely important since they are executed many times within the main loop of the application.

To issue the command buffer asynchronously we have the utility function *redline_IssueCommands()*. To flip the back buffer and the front buffer we have the function *redline_PageFlip()*. Lastly we have *redline_ClearDisplay()* to clear the back buffer. The code for all of these functions can be found in the file *utils.c* in the *htri1* sub-directory.

Issuing the Command Buffer

We'll start with a function to issue the command buffer. This piece of code is pretty important since commands placed in the buffer won't be executed by the Vérité until the command buffer is issued. However, issuing should only take place at specific times or the Vérité may behave more synchronously than asynchronously.

One appropriate time to issue commands would be when the current command buffer is full. If you can't add any more commands to it you might as well hand it to the Vérité. In this case, calling the issue function is automatic since we've already set *redline_IssueCommands()* to be our buffer overflow callback when we created the command buffers. Since it is used as a callback it must have a very specific argument list as well as returning a *v_cmdbuffer* type and be declared as *V_CDECL*. Some compilers pass arguments differently from others and we need to make sure with the *V_CDECL* that they are passed only in one manner.

```
v_cmdbuffer V_CDECL redline_IssueCommands
      ( v_handle    v,
        v_cmdbuffer  c )
```

Another appropriate time to issue commands is when a scene is complete. In other words during the page flip. Later when we give the details on page flipping you'll see that our page flip routine calls *redline_IssueCommands()* as its last step.

The last and least used reason to issue the command buffer is during specific points in your program. If you've just added a call that draws a large triangle that takes up the entire screen you may want to issue the command buffer to insure that that command is processed as soon as possible. You may also want to issue the command buffer right before doing some large portion of CPU processing. In that way you can insure that the Vérité is processing while the CPU is doing transformations or adding commands to another buffer. Placing these sort of issue commands in your code should be done very carefully and rarely. In fact, most applications won't require them at all. The example we are working on is one of them that won't.

With these three things in mind, lets take a look at the body of *redline_IssueCommands()* to see what this function needs to do to work for all the above reasons. We first issue the current command asynchronously with a call to *V_IssueCmdBufferAsync()*. This marks the buffer as being in use by the Vérité and allows it to take control of that structure and run it asynchronously with the CPU.

```
{
      // issue the current command buffer asyncronously
      V_IssueCmdBufferAsync(v, c);
```

The next two lines simply advance the command buffer ring index and the current command buffer pointer to point to the next entry in the ring. This is important since you don't want to add commands to a command buffer that you've just issued.

```
   // advance command index to the next command buffer
      cmdbuf_index = (cmdbuf_index + 1) % REDLINE_NUMCMDBUFS;

      // store new command buffer
      cmdbuffer = cmdbufs[cmdbuf_index];
```

A *while* loop is then used to prevent another problem from occurring. It is very possible that an application can quickly load up all the allocated command buffers in the ring and run into a buffer that was issued a while ago but is not yet finished. In this case you must stall the CPU and wait for the buffer to be complete. As long as the Vérité is kept busy, this stall should be avoided as much as possible to get the best performance out of your application.

To prevent stalling, you could simply increase the size of the command buffers or increase the number of command buffers. If this doesn't add to program lag, this is a good method to follow and means that the Vérité didn't have enough commands to keep it busy. You could go a more difficult path by moving code around so that the CPU is busier before the issue command is evoked. That by the time the CPU is ready to add commands the next buffer may be finished and ready to receive them. Note that it is very possible that there is simply a lot of commands for the Vérité to process and that the CPU may need to be stalled.

```
   // insure new buffer is not currently in-use (don't want
      // to add commands to an active command buffer)
      while (V_QueryCmdBuffer(v, cmdbuffer) == V_CMDBUFFER_INUSE)
      {
      // currently in-use (may want to consider creating more
      // buffers)
      ;
      }
```

The last bit of code for this function returns a pointer to the newly available command buffer. While this information may not be needed by the page flip routine, it is needed by the overflow callback function to tell it where the overflowed command needs to be placed (the next buffer in the ring).

```
      // return current (in case it was a callback)
      return cmdbuffer;
}
```

Page Flipping

The next utility we'll discuss is the page flip function *redline_PageFlip()*. This function will be called once the scene has been completely rendered to the back buffer and needs to be displayed to the screen.

All this function does is add page flipping commands to the command buffer and then issue them along with other commands on the buffer. The first set of these commands is added with *VL_SwapDisplaySurface()*. This function tells the Vérité to switch the pointers of the front and back buffers making the back buffer visible and the front buffer our new drawing destination. Note that this command has a nice built in feature preventing the old back buffer from becoming the new front buffer until the next vertical refresh of the screen occurs so that tearing is prevented. The old front buffer becomes the new back buffer immediately after the commands are executed

without the wait.

```
void redline_PageFlip
       ( void )
{
       // flip front and back buffers
       VL_SwapDisplaySurface(&cmdbuffer, redline_Display);
```

The next command needs some thorough explanation. The situation is that even though we've told the Vérité to swap the back and front buffers and even though the Vérité will wait to set the new front buffer it will still continue to process commands while waiting. Since the old front buffer becomes the new back buffer without waiting it is possible that drawing commands that follow immediately could go to the new back buffer while it is still the displayed front buffer. To prevent this from happening we've added a call to *VL_WaitForDisplaySwitch()*. This command tells the Vérité to stall processing commands until the new front buffer has been set. This insures that any command that follows the wait will display to the new back buffer when it is not the displayed front buffer.

```
       // add wait for vertical re-trace to command buffer
       VL_WaitForDisplaySwitch(&cmdbuffer);
```

One interesting technique to use is to remove the call to *VL_WaitForDisplaySwitch()* while running some sort of frame counter. This will show you the true frame rate of the application without waiting for the vertical refresh. You can then see places where frame rate fluctuates and make decisions on what refresh rate to use for your application and see where the application slows down.

The last command uses the utility function redline_ClearDisplay() to clear the new back buffer so that we have a clean slate to work from. After this call is made the commands are issued with *redline_IssueCommands()*.

```
       // clear screen after pageflip
       redline_ClearDisplay();

       // issue new commands immediately
       redline_IssueCommands(verite, cmdbuffer);
}
```

Clearing the Back Buffer

The last and simplest function in *utils.c* is the function to clear the back buffer. This function requires only one RRedline function to work.

```
void redline_ClearDisplay
       ( void )
{
   // clear the back buffer of the display (fill with zeros)
  VL_FillBuffer(c, redline_Display, 1,
       0, 0, redline_Display->width, redline_Display->height, 0);
}
```

The function *VL_FillBuffer()* fills any surface's buffer with a given pixel value. In this case, we're filling the back buffer (buffer index one) with black pixels (zeros).

Rendering the Triangle

Before we get into the commands to draw triangles we'll need to talk a bit about the data these commands require. How these vertex structures store information and how, if any, default values are set.

Choosing a Vertex Type

In RRedline every vertex is assigned a value for each one of these attributes:

- X, Y 2-D screen coordinate
- RGB Unpacked RGB
- K Packed RGB
- I Intensity RGB
- A Alpha value
- S Specular color (specular extensions with V2000 series only)
- F Fog value
- Z Z-Value for Z-buffering
- U,V Texture coordinate
- Q Scaled reciprocal of homogenous W (1/Z is commonly used)

How a value is assigned is determined by its vertex type. In RRedline there isn't just one vertex type – there are over twenty each with their own data structure and list of contained attributes. If a vertex type does contain a specific attribute, then it's value will be taken from the passed data structure. If a vertex type does not contain a specific attribute, a setable global default value is used.

The reason for having many types of vertices is to cut down the amount of work that the chip needs to do. For example, if you don't have alpha to store for your vertex there is no need to choose a vertex type that contains alpha. That will prevent the alpha value from being passed to the chip, saving setup time and preventing the chip from interpolating the alpha value.

After determining which of the attributes are required for your vertex, you must then choose an appropriate vertex type. Some of the time you may not be able to find a perfect fit for your needs. However, each one of the supplied vertex types has been implemented to be as fast as possible. Having a few extra attributes won't slow things down that much. Just remember to store default values in the structure for the attributes that weren't used and also remember that most of the attributes have default values when not specified in the vertex.

The list of all possible vertex types along with the set of supported attributes follow. Note that X,Y was not contained in this table since those two attributes are included in every vertex type.

Vertex Type	Vertex Structure	Contained Attributes								
		K	I	RGB	A	S	F	Z	UV	Q
V_FIFO_XY	v_xy									
V_FIFO_XYUV	v_xyuv								X	
V_FIFO_XYUVQ	v_xyuvq								X	X
V_FIFO_IXYUVQ	v_ixyuvq		X						X	X
V_FIFO_FXYUVQ	v_fxyuvq						X		X	X

V_FIFO_XYZUVQ	v_xyzuvq							X	X	X
V_FIFO_RGBFXY	v_rgbfxy			X			X			
V_FIFO_RGBXYZ	v_rgbxyz			X				X		
V_FIFO_IFXYUVQ	v_ifxyuvq		X				X		X	X
V_FIFO_IXYZUVQ	v_ixyzuvq		X					X	X	X
V_FIFO_RGBAFXYUVQ	v_rgbafxyuvq			X	X		X		X	X
V_FIFO_RGBAXYZUVQ	v_rgbaxyzuvq			X	X			X	X	X
V_FIFO_KFXY	v_kfxy	X					X			
V_FIFO_KXYZ	v_kxyz	X						X		
V_FIFO_KAFXYUVQ	v_kafxyuvq	X			X		X		X	X
V_FIFO_KAXYZUVQ	v_kaxyzuvq	X			X			X	X	X
V_FIFO_IXYZUV	v_ixyzuv		X					X	X	
V_FIFO_RGBAFXYZUVQ	v_rgbafxyzuvq			X	X		X	X	X	X
V_FIFO_KAFXYZUVQ	v_kafxyzuvq	X			X		X	X	X	X
V_FIFO_KXYUVQ	v_kxyuvq	X							X	X
V_FIFO_AXY	v_axy				X					
V_FIFO_KXY	v_kxy	X								
V_FIFO_KSXYUVQ	v_ksxyuvq	X				X			X	X
V_FIFO_KSXYZUVQ[1]	v_ksxyzuvq	X				X		X	X	X
V_FIFO_KaSFXYZUVQ[1,]	v_kasfxyzuvq	X			X	X	X	X	X	X

For our example we're only concerned with a few of these attributes. Specifically the 2-D coordinate pair and the color attributes. Other attributes will be discussed later in other examples.

X,Y

The X,Y coordinate pair can be found in all RRedline vertex types. The values are not stored as floating point values. Instead they are stored as 16.16 signed values. The values are in screen coordinates ranging from [0, width) for X and [0, height) for Y. If drawing in full-screen mode the origin of the coordinate system is in the upper left hand corner of the screen and the height and width are the pixel height and width of the screen. If drawing in windowed mode the origin of the coordinate system is the upper left hand corner of the window and the width and height is the width and height of the window. X always increases from left to

and height is the width and height of the window. X always increases from left to right and Y always increases top to bottom.

X,Y coordinates

msb 31	lsb 0
16.16 fixed point X coordinate	
16.16 fixed point Y coordinate	

RGB, K, I

These three mutually exclusive attributes are used to specify the vertex color in three different ways:

Unpacked RGB, referred to simply as *RGB*, contains three 16.16 fixed point values each representing either the red, green, and blue color value. Each individual RGB value must range from 0 to 255.

Unpacked R, G, B

msb 31	lsb 0
16.16 fixed point R color channel	
16.16 fixed point G color channel	
16.16 fixed point B color channel	

K is a packed version of RGB where you only have 8 bits to store the each of the individual RGB values. Packed values must range from 0 to 255 and are stored as 8.0 fixed point values.

Packed RGB (K)

msb 31			lsb 0
-	R	G	B

Intensity or just *I* is a special case of RGB where the red, green, and blue channels are equal to each other. The value ranges from 0 to 255 and is stored as a 16.16 fixed point value.

Monochrome Intensity (I)

msb 31	lsb 0
16.16 fixed point I monochrome intensity	

If a vertex does not contain any of the color attributes the vertex will use the default color. Each individual value of the default color can be set with VL_SetR(), VL_SetG(), or VL_SetB() or all of them can be set along with alpha with VL_SetFGColorARGB() or VL_SetFGColorABGR().

In the case of our hello triangle example, we are simply drawing a flat shaded triangle. For that case, we'll pick the vertex type V_FIFO_XY and set the color to be used with *VL_SetFGColorARGB()*. We could have picked a vertex type that contains

used with *VL_SetFGColorARGB()*. We could have picked a vertex type that contains color, like V_FIFO_KXY, but that would require the Vérité to unnecessarily interpolate the color value.

Drawing Triangles

Now that we've chosen our vertex type, it's time to draw our triangle. We'll start out by setting the coordinate values in each of the vertex structures.

```
void render_scene
       ( void )
{
       static v_xy   A = {
              FLTOIFIX(100.0),      FLTOIFIX(100.0)       };
       static v_xy   B = {
              FLTOIFIX(400.0),      FLTOIFIX(100.0)       };
       static v_xy   C = {
              FLTOIFIX(250.0),      FLTOIFIX(300.0)       };
```

In order to convert the X and Y coordinates to the 16.16 fixed point value, we're using the macro *FLTOIFIX()* defined in *htri.h*. This macro simply converts a float value to it's 16.16 fixed point equivalent. These and other conversion macros have been written to be readable and easy to understand. Other more efficient and faster algorithms should be considered for real applications and can be found in the section titled *Optimizations*.

```
#define FLTOIFIX(a) \
       ((v_u32)(((float)(a))*((float)(1<<16))))
```

The next thing to do is to set the default color value. We'll need to do this since our vertex type does not contain color. If we were also doing Z-buffering or alpha blending we would have also thought about their default values as well. Since we're not, we only have to set the default color. Here we're setting the default color within the rendering routine to be more readable. However, it would have been more efficient to have set it once within the initializations and not have set it again.

```
       // set color to draw with
       VL_SetFGColorARGB(&cmdbuffer,
       FLTOARGB(255.0, 255.0, 0.0, 0.0));
```

This bit of code uses another macro we've defined in *htri.h*. *FLTOARGB()* takes separate ARGB float values and returns a packed 32 bit format. Conversion macros may differ from application to application depending on the input data ranges.

```
#define FLTOARGB(a, r, g, b) \
              (((v_u32)((float)(a)) << 24)  \
             + ((v_u32)((float)(r)) << 16)\
          + ((v_u32)((float)(g)) << 8)  \
          +  (v_u32)((float)(b)))
```

After all this setup, we're finally ready to draw the triangle. The next command *VL_Triangle()* puts the drawing commands on the command buffer. The arguments of this function simply tell the what command buffer to add to, what kind of vertex we're using (V_FIFO_XY), and what the vertex data is.

```
       // draw triangle
       VL_Triangle(&cmdbuffer, V_FIFO_XY,
       (v_u32 *)&A, (v_u32 *)&B, (v_u32 *)&C);
```

Here the vertex information is being supplied through pointers to the filled data structures. Since we have many different kinds of vertex structures to choose from, we're casting the pointer to the generic RRedline pointer (*v_u32 *)*. This pointer type is used throughout RRedline to represent that different kinds of data can be passed to those functions that use the pointer.

At this point we're finished rendering the scene and would like to display the results to the front buffer. That is achieved with *redline_PageFlip()*, our utility to flip the front and back buffers.

```
	// display back buffer
	redline_PageFlip();

}
```

Shutting down the Vérité

When the application terminates or when an error occurs we need to shut down the Vérité cleanly. This involves destroying the command buffers, destroying the front and back buffer, and closing the handle to the Vérité itself. All this is done within the function *redline_Close()* found in the file *close.c*. This function is called after the main loop exits in *main.c* or by the error handler when an error occurs.

To destroy each of the command buffers in the ring we'll use the function *V_DestroyCmdBuffer()*. This function will wait until all issued commands on the buffer are complete and then free the memory associated with the buffers.

```
void redline_Close
	( void )
{
	int	i;

	// free command buffers if they exist
	if (cmdbuffer)
	for (i=0; i<REDLINE_NUMCMDBUFS; i++)
		V_DestroyCmdBuffer(cmdbufs[i]);
```

The last piece of the code will destroy the front and back buffers with a call to *VL_DestroySurface()* and shut down the Vérité with a call to *VL_CloseVerite()*. At this point, no further commands may be passed to the chip unless the application starts over with a call to *VL_OpenVerite()*.

```
	// free drawing surface if it exists
	if (redline_Display)
	VL_DestroySurface(verite, redline_Display);

	// close verite if exists
	if (verite)
	VL_CloseVerite(verite);
}
```

Modifications to Hello Triangle

At this point you should know the basics explained in hello triangle. You should know about command buffers, surfaces, vertex types, and page flipping. Now we're going to extend this example to show you how to do some other operations. Included in this section are the following modifications:

- Switching between full screen mode and windowed mode (*htri2*)

- Switching between full-screen mode and windowed mode (*htri2*)

- Drawing a texture mapped triangle (*htri3*)

- Z buffering and perspective correction (*htri4*)

- Alpha blending (*htri5*)

- Specular extensions (*htri6*)

Modification One: Switching Between Full Screen Mode and Windowed Mode

The first modification to our program is one of the most useful modifications you can make. By allowing your application run in a window, you can debug it a lot easier than you could in full-screen mode. Your program's specifications may also require that your application run in a window making the modification not just useful but necessary.

What we're going to implement here is a modification to the initial Hello Triangle that allows the user to switch between full-screen mode and windowed mode by hitting the Alt-Enter key combination. Switching between modes may not be what you require (most applications will be either full-screen or windowed but not both) but we'll structure the code so that you'll be able to copy most of the routines without modification. We'll also need to re-organize some of the code from the original Hello Triangle so that this will work efficiently. The code for this example can be found in the sub-directory *htri2*.

Page Flipping in a Window

The main difference between a full-screen application and a windowed application is how the page flipping routines work. To handle this difference we'll rename the old *redline_PageFlip()* to *redline_PageFlipFullscreen()* and create a new function *redline_PageFlipWindowed()*. We'll keep a new global function pointer *redline_PageFlip* pointed to the one function we need given the current display state. In that way, all our drawing applications need to do is continue to call our function pointer *redline_PageFlip()* and not have to worry about the current mode.

Differences between these page flip routines explain the main difference between full-screen and windowed applications. In the case of a full screen application all you need to do is tell the Vérité that you need to swap front and back buffer pointers with *VL_SwapDisplaySurface()*. In the case of a windowed application you are sharing the front buffer (the desktop) with other applications and can't just swap memory pointers. Instead, you'll need to blit the entire back buffer to the window's location on the front buffer.

The position of the window, stored in the static variable *redline_WindowPos*, is set within *redline_SetWindowPos()*. This routine is called whenever a window move message (WM_MOVE) or window resize message (WM_SIZE) is received in the window procedure in *main.c*. Its also called when the window is first created.

```
void redline_SetWindowPos
       ( HWND hWindow )
{
       // get screen coordinates of window rectangle and store
       GetClientRect(hWindow, (RECT*)&redline_WindowPos);
       ClientToScreen(hWindow, (LPPOINT)&redline_WindowPos);
       ClientToScreen(hWindow, (LPPOINT)&redline_WindowPos+1);
}
```

The blit itself is done within the function *redline_PageFlipWindowed()* with a call to *V_BltDisplayBuffer()*.

```
void redline_PageFlipWindowed
       ( void )
{
       // issue current commands (before blitting)
       redline_IssueCommands(verite, cmdbuffer);

       // blit display to window (waits for commands to complete)
       V_BltDisplayBuffer(verite, redline_Display->buffer_group, 0,
       &redline_WindowPos, redline_Display->buffer_group, 1, NULL);
```

Notice that we are issuing the commands before doing the blit. The reason for this is that the *V_BltDisplayBuffer()* does not go on the command buffer and requires that all the required commands are issued before blitting. The blit function has a wait built into it to insure that all previously issued commands are complete (you wouldn't want to blit an incomplete scene). *V_BltDisplayBuffer()* also handles all the clipping problems that one might have with partially obscured windows and also stretches or contracts the scene when windows are resized.

The last thing this function needs to do is to add a clear back buffer command to the command buffer. This will clear the blitted back buffer preparing it for the next scene. Sometimes it's efficient to place a command buffer issue after the clear since some programs will start CPU processing at the beginning of the scene and the clear command deals with a lot of pixels. Issuing it as soon as possible insures that it gets processed quickly.

```
       // clear the display buffer
       redline_ClearDisplay();
}
```

Initializing Buffers for a Windowed Application

The next modification we'll have to make is to *redline_Init()*. There's some code contained in that function that we'll need to remove and put into another function. The purpose of this new function, *redline_SetDisplay()*, will be to set up the modes and buffers required for a given display type. We've turned this into a separate function so that it can be called each time the display type changes.

Within *redline_Init()*, you'll notice that all the code that sets the display mode, sets the display type, creates the front and back buffers, and sets the destination buffer have been replaced with one call.

```
       // set display type
       redline_SetDisplay(V_WINDOWED_APP, 640, 480, 16, 60,
V_PIXFMT_565);
```

This new function can be found in the file *utils.c*. The first thing that needs to happen in this function is to insure that if a back buffer already exists that all commands going to that buffer are complete. This is in preparation of the destroying the buffer and then re-creating it for the new mode. Otherwise, previous commands may be written to a back-buffer that will be destroyed

```
void redline_SetDisplay
       ( v_u32 display_type,
         v_u32 width,
         v_u32       height,
         v_u32 bpp,
         v_u32 refresh_rate,
```

```
          v_u32 pixel_fmt )
{
       vl_error      error;

       // clear previous commands from buffer
       redline_FlushAndComplete(verite, cmdbuffer);
```

This new function, *redline_FlushAndComplete()*, can be found in the file *utils.c*. It's a simple routine that issues all the remaining commands and insures they are complete by locking the buffer. Locking the buffer causes the CPU to stall until the Vérité is finished with all issued commands.

```
void redline_FlushAndComplete
       ( v_handle           v,
         v_cmdbuffer         c )
{
       // issue the current command buffer
       redline_IssueCommands(v, c);

       // insure all commands have completed by locking surface
       if (redline_Display)
       {
              V_LockBuffer(verite, redline_Display->buffer_group, 0);
              V_UnlockBuffer(verite, redline_Display->buffer_group,
0);
       }
}
```

The next line in *redline_SetDisplay()* tells the program which page flip routine to use depending on the display type. That instead of calling *redline_PageFlipFullscreen()* or *redline_PageFlipWindowed()* directly, routines will just call *redline_PageFlip()*.

```
       // set page flip function
       redline_PageFlip = (display_type == V_FULLSCREEN_APP) ?
              redline_PageFlipFullscreen : redline_PageFlipWindowed;
```

The next portion of code is useful only because this routine is being called multiple times. If the front and back buffers have already been created then they need to be destroyed and re-created when the display type changes simply because the front buffer changes. This next section of code destroys the old surface if it exists. Later, we'll re-create the front and back buffers.

```
       // free old surface?
       if (redline_Display)
              VL_DestroySurface(verite, redline_Display);
```

The next piece of code was taken directly from the original initialization routine of the first Hello Triangle. The only difference is that the width, height, and refresh rate are only being set if the display type is full-screen. If the program is running in a windowed mode, the application will be sharing the front buffer with other applications and cannot set those parameters.

```
       // set display type
       V_SetDisplayType(verite, display_type);
       if (display_type == V_FULLSCREEN_APP)
              V_SetDisplayMode(verite, width, height, bpp,
refresh_rate);
```

The last bit of code for *redline_SetDisplay()* creates the front and back buffers and installs the back buffer as the drawing destination. This requires no changes from our original code from the starting Hello Triangle.

```
      // create new front and back buffer
      VL_CreateSurface(verite, &redline_Display, V_SURFACE_PRIMARY,
            2,pixel_fmt, width, height);

      // install new buffer as draw destination
      VL_InstallDstBuffer(&cmdbuffer, redline_Display);
}
```

Display switching routine

At this point all we need to do is provide the function that determines which display type is currently in use and makes the appropriate calls to switch to the other one. This is the function called when the Alt-Enter key combination is hit.

Detecting the current display type will be determined by looking at value returned from *V_GetDisplayType()*. After storing the display type to change to, a call to our function *redline_SetDisplay()* will be made.

```
void redline_SwitchDisplay
      ( HWND  hWindow,
        v_u32 width,
        v_u32 height,
        v_u32 bpp,
        v_u32 refresh_rate,
        v_u32 pixel_fmt )
{
      v_u32  display_type;
      v_u32  error;

      // calculate new type
      display_type = (V_GetDisplayType(verite) == V_FULLSCREEN_APP) ?
        V_WINDOWED_APP : V_FULLSCREEN_APP;

      // initialize new values
      redline_SetDisplay(display_type, width, height, bpp,
            refresh_rate, pixel_fmt);
```

At this point the only special case we need to handle is when we are changing from full-screen to windowed mode. If that's the case we'll need to insure that the window we're displaying to is still visible. We'll do this though a Windows call that places the window into the upper left hand corner of the screen. After moving it we'll also need to update the stored window position with *redline_SetWindowPos()* so that the page flipping blit will go to the proper location.

```
      // need to position window?
      if (display_type == V_WINDOWED_APP)
      {
            // make sure window is displayed
            if (!SetWindowPos(hWindow, HWND_TOPMOST, 0, 0,
              width, height, SWP_NOMOVE|SWP_NOOWNERZORDER))
            {
                  // error setting position
                  redline_ErrorBox("Unable to set window
position");
                  ExitProcess(1);
            }

            // store position of new window
```

```
                redline_SetWindowPos(hWindow);
        }
}
```

De-Activating and Re-Activating the Application

When an application is running in a window, it ends up having to share many resources with other windowed applications. It shares CPU cycles and system memory and it also shares the Vérité and video memory. While the operating system handles the first two, it is up to your application to deal with other applications sharing the Vérité and its video memory.

This isn't as difficult as it might seem. You won't need to write a memory allocator or try to figure out how to deal with multiple sets of command buffers. Instead, all you need to do is detect when the application is de-activated, either through another application taking over or having your application minimized, and detect when the application is re-activated. When de-activated you just have to prevent command buffers from being issued and when re-activated, you'll need to restore your video memory and pixel engine state.

In order to detect when the application is about to be deactivated we've added the following code to our Windows message procedure.

```
        case WM_ACTIVATEAPP:
                // is application going to be de-activated?
                if (!wParam)
                {
                        // disallow drawing and reset command buffer
                        app_Active = FALSE;
                        redline_ResetCommandBuffer();
                }
                break;
```

If this message tells us that our application is going to be de-activated we do two things. We first store *FALSE* for *app_Active*. This will prevent the *render_scene()* function from being called in the main loop which effectively prevents the command buffer from being issued while the application isn't active. The second thing we do is call the function *redline_ResetCommandBuffer()* to clear out any unissued commands from the current buffer so that when the application becomes active, the command buffer will start from scratch. All issued commands will be automatically completed before the application allows a switch to occur.

```
void redline_ResetCommandBuffer
        ( void )
{
        // reset current command buffer
        if (cmdbuffer)
        V_ResetCmdBuffer(cmdbuffer);
}
```

The second message we look for in our Windows procedure is *WM_ACTIVATE*. This message will tell us if our application has become active and whether or not it is minimized or not. We aren't using the message *WM_ACTIVATEAPP* for this part since it would tell us that the application is about to become active not that the application has become active.

```
        case WM_ACTIVATE:
                // has application been activated and not minimized?
                if ((LOWORD(wParam) != WA_INACTIVE) && !HIWORD(wParam))
                {
```

```
                {
                        // allow drawing and restore surfaces
                        app_Active = TRUE;
                        redline_RestoreSurfaces();
                }
                break;
```

If this message tells us that the application has become active and that the application isn't minimized it will do two things to re-activate the application. First, it will set the *app_Active* flag to *TRUE*, allowing the *render_scene()* routine to run again. Second, it will call the routine *redline_RestoreSurfaces()* used to restore the allocated surfaces and restore the pixel engine state. This routine is found in *utils.c*.

```
void redline_RestoreSurfaces
        ( void )
{
        // command buffers exist?
        if (cmdbuffer)
        {
        // initialize and restore state
        VL_ContextInit(&cmdbuffer);
        redline_InitState();
        }

        // restore surfaces
        if (redline_Display)
        {
        VL_RestoreSurface(verite, redline_Display);
        VL_InstallDstBuffer(&cmdbuffer, redline_Display);
        }
}
```

When an application regains focus, none of the Vérité's pixel engine states or video memory allocations are guaranteed to be the same as when the application lost focus. This is especially true when the application loses focus to a full-screen MS-DOS box (alt-enter a MS-DOS window). Because of this we need to restore the pixel engine state and the video memory.

The pixel engine state consists of many things. It contains the source function, the default values for the attributes, the foreground color, etc. If your application sets any of these states at the beginning of the app and assumes that they won't change then you'll need to restore them whenever the application regains focus. This is done first with a call to *VL_ContextInit()* that restores all RRedline pixel engine states to their default values and then calls *redline_InitState()* in *init.c* used to restore our application's state.

```
void redline_InitState
        ( void )
{
        // when drawing, use vertex colors or global colors
        VL_SetSrcFunc(&cmdbuffer, V_SRCFUNC_NOTEXTURE);
}
```

You'll notice that for this application we just need to restore the source function state. Other examples that follow will be a little more complicated. In addition, the *VL_SetSrcFunc()* has been removed from *redline_Init()* and replaced with a call to this new function *redline_InitState()*. This is to limit the state setting calls to one function in our application.

The last thing that is done within *redline_RestoreSurfaces()* is to restore the video memory. Specifically, restore the front and back buffers. This is done with a call to *VL_RestoreSurface()*. All this call does is insure that the previously allocated surface

VL_RestoreSurface(). All this call does is insure that the previously allocated surface exists in video memory and if not re-allocates it. Since a re-allocation is possible any states that use the surface as a function argument must be re-set. This explains the call to *VL_InstallDstBuffer()* that follows the restore surface call.

At this point all states should be restored and all video memory should be available for use. Always remember that the active application should have complete control of the Vérité and its memory and no control when not active.

Modification Two: Drawing a Texture Mapped Triangle

The next modification we'll implement will start making our example a little more interesting. Most applications won't just draw flat shaded triangles like in our first two examples. Instead some will add to the detail by mapping a texture to the triangle.

What we're going to do for this example is to start with the windowed version of Hello Triangle (htri2) and modify it to draw two texture mapped triangles one on top of the other. We'll create the textures, load them into memory, install each of them as the texture we'd like to use, and then draw them by mapping vertices to texture U,V coordinates. In addition one of the triangles will be wrapped (tiled) and the other will be clamped. We'll also add a feature to the program so that every time the 'f' key is hit, the textures will toggle between point sampled and bilinear filtered. The code for this example can be found in the sub-directory *htri3*.

Creating and Loading the Textures

The first thing we need to do is create the textures we'd like to use. In your own applications you'd probably read these in from a file of some kind. Here we're just going to create some checkerboard type textures where the "checker" section is of different sizes and colors. This will be done by the routine *locked_checkerboard()* found in the file *texture.c*.

```
static v_memory locked_checkerboard
       ( int         width,
         int         height,
         int         section_width,
         int         section_height,
         v_u16 color1,
         v_u16 color2 )
{
       v_memory      result;

       int           i, j;
       v_u16         *ptr;

       // allocate locked memory (16 bit texture)
       result = V_AllocLockedMem(verite, width * height * 2);

       // traverse through memory setting colors
       ptr = (v_u16 *)V_GetMemoryObjectAddress(result);
       for (i=height; --i>=0; )
       for (j=width; --j>=0; )
              if ((i/section_height)%2 == (j/section_width)%2)
                     *(ptr++) = color1;
              else
                     *(ptr++) = color2;
```

```
        // return locked memory
        return result;
}
```

Since we're going to transfer the data to the Vérité and since we'd like this transfer to happen asynchronously we're going to have to insure that the memory exists in the physical memory of the CPU and not virtual memory that may be swapped out to disk. We'll do this by placing the texture in locked memory that cannot be swapped. This is done by allocating memory with *V_AllocLockedMemory()*. To get a pointer to this memory object, we'll use *V_GetMemoryObjectAddress()* and store the texture data.

The next thing that needs to be done is to create a section of memory on the board and load the locked texture memory into it. This will be done by passing the locked memory to the function *create_texture()* also found in the file *texture.c*.

```
static v_surface *create_texture
        ( v_memory    data,
          int         width,
          int         height )
{
        v_surface     *result;

        // create surface
        VL_CreateSurface(verite, &result, 0, 1,
        V_PIXFMT_565, width, height);

        // load data into texture
    VL_LoadBuffer(&cmdbuffer, result, 0, width * 2,
        width, height, data, NULL);

        // return loaded texture
        return result;
}
```

The first RRedline call should be pretty familiar to you. It's the exact same function used to create the front and back buffers. In the same manner that we created that surface we'll just create a surface of a given width and height that contains one buffer and uses a pixel format of *V_PIXFMT_565*. Later we'll just use that surface as if it was a texture. Its that simple.

The next call is used to load the buffer with our initialized locked memory that we created with *locked_checkerboard()*. You just simply pass the memory pointer, width, height, and number of bytes per texture line (16 bit format = 2 bytes) and this function places the load command into the command buffer. Commands to use this texture can then immediately follow even if they occur on the same command buffer.

However, it is important that you don't free that locked memory right away, if you did you may end up freeing it before the command buffer was issued or during the actual memory access by the Vérité. To prevent that from occurring we'll just free the memory at the end of the application within our *redline_Close()* routine. Other more complex applications could keep track of that memory along with the command buffer pointers. As long as the command buffer that contains the load buffer command is finished (*V_QueryCmdBuffer()* is not *V_CMDBUFFER_INUSE)* then the locked memory can be freed or re-used.

Note that the call to *VL_LoadBuffer()* goes on the command buffer while the call to *VL_CreateSurface()* does not. In this case this is a very important distinction. In fact the call to *VL_CreateSurface()* locks the Vérité and forces it to catch up with the CPU. This is one of the largest hindrances to Vérité performance. Ways around this include creating all the surfaces required at the beginning of the program and re-using them or allocating a large chunk of memory on the board and keeping track of

them of allocating a large chunk of memory on the board and keeping track of memory management within the application itself. More details about these techniques will be discussed in the *Optimizations* chapter.

The next function we'll describe is called by *redline_Init()* in order to set up the textures. It calls our routines to allocate, create, and initialize the two textures along with setting up the surface clamping.

```
void redline_CreateTextures
       ( void )
{
       // place two checkerboard patterns in locked memory
       locked1 = locked_checkerboard(128, 256, 16, 32, 0x07E0,
0x001F);
       locked2 = locked_checkerboard(200, 100, 10, 10, 0xF81F,
0x07FF);

       // create texture surfaces for each of the patterns
       checker1 = create_texture(locked1, 128, 256);
       checker2 = create_texture(locked2, 200, 100);

       // disable clamping for the first, enable for second
       VL_SetSurfaceClamp(checker1, FALSE, FALSE);
       VL_SetSurfaceClamp(checker2, TRUE, TRUE);
}
```

Within RRedline, the default behavior of the texture coordinates is that they fully cover the texture when they range from [0, 1) with the U,V coordinate (0,0) representing the first pixel in the texture. Coordinates outside that range have different behaviors according to the texture's clamping flag. If a texture is not clamped the texture is wrapped (tiled) with the value before the decimal point effectively being ignored. In other words, the texture pixel (texel) found at texture coordinate (0.5, 0.5) would be the same at (1.5, 1.5) or (3.5, 6.5). When a texture is clamped then the coordinate is forced to be within the ranges [0, 1) by clamping it with maximum and minimum functions. For example, the coordinate (2.5, 0.5) would result in the last texel in the column V=0.5 and the coordinate (-1.0, -1.0) would result in the texel found at coordinate (0,0). Wrapping is not available for texture dimensions that aren't a power of two. Clamping is available for all valid texture dimensions.

We'll set up clamping and wrapping by modifying the flag found in the surface structure with *VL_SetSurfaceClamp()*. Later when the texture gets installed the clamp and wrap flag will be passed to the Vérité via the command buffer. Instead of calling a function, the application could have modified the structure directly adding the necessary flags to the *clamp* element.

Filtering the Textures

One last issue we'll deal with is the choice of filtering. Here we've written a function *redline_SwitchFilter()* that is called whenever the user hits a 'f' key. This function will tell the Vérité to either display all textures as with point filtering or to pass them through a bi-linear filter before displaying.

```
void redline_SwitchFilter
       ( void )
{
       // switch active filter
       if (redline_Filter == V_SRCFILTER_POINT)
              redline_SetFilter(V_SRCFILTER_BILINEAR);
       else
              redline_SetFilter(V_SRCFILTER_POINT);
```

```
                redline_SetFilter(V_SRCFILTER_POINT);
}
```

This function uses the function *redline_SetFilter()* to do the actual work.

```
void redline_SetFilter
       ( v_u32 filter )
{
       // which filter to use?
       redline_Filter = filter;
       VL_SetSrcFilter(&cmdbuffer, redline_Filter);

       // set filter offsets
       if (redline_Filter == V_SRCFILTER_POINT)
       {
       // point offsets
       VL_SetSOffset(&cmdbuffer, FLTOIFIX(0.0));
       VL_SetTOffset(&cmdbuffer, FLTOIFIX(0.0));
       }
       else
       {
       // bilinear offsets
       VL_SetSOffset(&cmdbuffer, FLTOIFIX(-0.5));
       VL_SetTOffset(&cmdbuffer, FLTOIFIX(-0.5));
       }
}
```

Here we're just passing the filter state to the chip with a call to *VL_SetSrcFilter()*. In order for this call to work properly we'll need to change the origin offset. By default, the coordinate (0,0) refers to the exact *center* of the first texel. While this might work fine for point filtered applications, it won't work as well for bi-linear filtered textures. For those, the origin needs to be set to the upper left corner of the texel for the filter function to operate properly. This shifting of origins is done with a call to *VL_SetSOffset()* and *VL_SetTOffset()*. The S and T coordinates refer to texel coordinates within the texture. Therefore, an offset of (-0.5, -0.5) refers to a shift of a half a texel in both S and T.

Setting the Source Function

In our first Hello Triangle example we were displaying a flat shaded triangle. In order to tell the Vérité that we were using the color information stored in the vertex, we had set the source function to *V_SRCFUNC_NOTEXTURE*. However in this example we'd like to ignore any color information stored at the vertex and display the color contained in the texture map instead. We'll do this by modifying the *VL_SetSrcFunc()* found in *redline_InitState()*.

```
       // when drawing, use texture colors
       VL_SetSrcFunc(&cmdbuffer, V_SRCFUNC_REPLACE);
```

By passing *V_SRCFUNC_REPLACE* as our source function we'll accomplish our goal. This mode tells the Vérité to display according to the texture's colors instead of the vertex colors. A complete discussion of the source function can be found in *Modification Four: Alpha Blending*.

Rendering The Triangle

Now that we've got the texture set up all we need to do is draw with it. That involves picking an appropriate vertex type, installing the texture we'd like to use, and then drawing the triangle. Before we pick a vertex type, we'll need to discuss the new

drawing the triangle. Before we pick a vertex type, we'll need to discuss the new attributes we're going to use in further detail, namely the texture coordinates U and V.

U, V

U and V are the texture coordinates used to determine the mapping between a texture pixel (texel) and a specific vertex. U and V coordinates are 16.16 signed fixed point values. Coordinates range from [0,1) for both U and V with the coordinate (0,0) referring to the first pixel of the texture. Coordinates outside that range either result in wrapping (repeating) of the texture or result in clamping. Wrapping and clamping can be enabled in either the U or V direction independently.

All the possibilities of texture coordinates can be explained by looking at the equations used to determine what texture pixel is used for a given U,V coordinate. Typically the UMultiplier, VMultiplier, UMask, and VMask are automatically set when a texture is installed with *VL_InstallTextureMap()*. With this function, both multipliers are set to the width and height of the texture while the masks are set two different ways depending on whether or not clamping is set. If clamping isn't set, the masks will be set to the highest power of two minus one that is greater than or equal to the width and height of the texture. If clamping is set, the masks are set to the texture width and height minus one. The resulting S and T coordinate from these equations is a texture coordinate that ranges from 0 to texture width minus one for S and 0 to texture height minus one for T. This identifies the mapped texture pixel for that coordinate.

Wrapping functions (use only with a width and height of a power of two)

```
S = ( U * UMultiplier + SOffset ) & UMask )
T = ( V * VMultiplier + TOffset ) & VMask )
```

Clamping functions:

```
S = MAX ( MIN( U * UMultiplier + SOffset, UMask ), 0)
T = MAX ( MIN( V * VMultiplier + TOffset, VMask ), 0)
```

With this information you can then change the behavior of U and V. For example, if the UMultiplier were set to 1 with *VL_SetUMuliplier()* and VMultiplier were set to 1 with *VL_SetVMultiplier()*, the input U and V coordinates would no longer range from 0 to 1 but become texel coordinates instead.

Since U and V values are stored as 16.16 fixed point values and since the S and T texel calculations are also done with 16.16 fixed point values, there is a limit where the precision of the calculation is overcome. To remove this possibility, the product of the maximum U and V value by the width and height of the texture should not be greater than 2^{11}. For example, the maximum U and V for a 256 x 128 texture is 8 for U and 16 for V. We highly recommend that an application keep their U and V texture coordinates as small as possible.

U,V coordinates

msb 31	lsb 0
16.16 fixed point X coordinate	
16.16 fixed point Y coordinate	

Since we're obviously going to require these two attributes along with X and Y we'll pick the least capable vertex type from our vertex table. It just so happens that the

second type listed in the table is the most appropriate: *V_FIFO_XYUV*. Now we just have to store the coordinates. This will be done in our re-write of *render_scene()* found in the file *render.c*.

```
void render_scene
       ( void )
{
       static v_xyuv tri1[] =
       {
              {
                     FLTOIFIX(100.0),      FLTOIFIX(100.0),
                     FLTOIFIX(0.0), FLTOIFIX(0.0)
              },
              {
                     FLTOIFIX(500.0),      FLTOIFIX(100.0),
                     FLTOIFIX(2.0), FLTOIFIX(0.0)
              },
              {
                     FLTOIFIX(100.0),      FLTOIFIX(300.0),
                     FLTOIFIX(0.0), FLTOIFIX(2.0)
              }
       };
       static v_xyuv tri2[] =
       {
              {
                     FLTOIFIX(200.0),      FLTOIFIX(150.0),
                     FLTOIFIX(0.0), FLTOIFIX(0.0)
              },
              {
                     FLTOIFIX(600.0),      FLTOIFIX(150.0),
                     FLTOIFIX(2.0), FLTOIFIX(0.0)
              },
              {
                     FLTOIFIX(200.0),      FLTOIFIX(350.0),
                     FLTOIFIX(0.0), FLTOIFIX(2.0)
              }
       };
```

The only differences between this initialization and the previous Hello Triangle example are that we're storing two more pieces of information (U and V), that we're storing into a different structure (v_xyuv), and that we're creating data for two triangles. Notice that we're using the same macro *FLTOIFIX()* to store the U,V 16.16 fixed point coordinates. Also note that some of the texture coordinates are outside the [0,1) texture coordinate range. This is to show the differences between clamping and wrapping.

With the coordinates prepared, its time to tell the Vérité what texture we'd like to use when drawing our triangle. After that, all we need to do is draw it.

```
       // use first checkerboard
       VL_InstallTextureMap(&cmdbuffer, checker1);

       // draw triangle
       VL_Triangle(&cmdbuffer, V_FIFO_XYUV,
       (v_u32 *)&(tri1[0]), (v_u32 *)&(tri1[1]),
       (v_u32 *)&(tri1[2]));

       // use second checkerboard
       VL_InstallTextureMap(&cmdbuffer, checker2);
```

```
        VL_InstallTextureMap(&cmdbuffer, checker2);

        // draw triangle
        VL_Triangle(&cmdbuffer, V_FIFO_XYUV,
        (v_u32 *)&(tri2[0]), (v_u32 *)&(tri2[1]),
        (v_u32 *)&(tri2[2]));

        // display back buffer
        redline_PageFlip();
}
```

As you can see, installing a texture map only requires passing the texture surface to the function *VL_InstallTextureMap()*. In turn that call passes information like clamping modes, source buffer, source width, source pixel format, etc. to the Vérité through the command buffer. Making these calls immediately is also possible and can be more efficient for your own applications. For example, if you can guarantee that the pixel format doesn't change it isn't necessary to keep telling the Vérité what it is. Reducing the amount of information passed to the chip will decrease the time required for texture state setup and increase chip performance. More about this topic can be found in the chapter titled *Optimizations*.

With the surface installed we call the same routine used to draw a triangle in the first example *VL_Triangle()*. When both are drawn, you should see a clamped red and white triangle drawn on top of a wrapped blue and green checkerboard triangle. Remember to hit the 'f' key to see the difference between point filtered and bi-linear filtering modes.

Freeing the Textures

The last modification we'll require is to free up the locked memory and destroy the texture surfaces we've created. We'll do this in our existing *redline_Close()* routine in *close.c*.

```
        // free texture memory and surfaces
        if (locked1)
                V_FreeLockedMem(verite, locked1);
        if (locked2)
                V_FreeLockedMem(verite, locked2);
        if (checker1)
                VL_DestroySurface(verite, checker1);
        if (checker2)
                VL_DestroySurface(verite, checker2);
```

The checks being made are to insure that if an error occurs within the application, this routine will free whatever locked memory and surfaces that were allocated at the time the error occurred.

Restoring the Textures and Pixel Engine State

Just like we needed to restore the front and back buffer surface when the application looses and re-gains focus we'll also need to restore the texture surfaces. This will be done by modifying the existing routine *redline_RestoreSurfaces()* in *utils.c*.

```
    // restore and re-load textures
        if (checker1)
        {
                VL_RestoreSurface(verite, checker1);
                VL_LoadBuffer(&cmdbuffer, checker1, 0, checker1->width *
2,
```

```
                    checker1->width, checker1->height, locked1,
NULL);
      }
      if (checker2)
      {
            VL_RestoreSurface(verite, checker2);
            VL_LoadBuffer(&cmdbuffer, checker2, 0, checker2->width *
2,
                  checker2->width, checker2->height, locked2,
NULL);
      }
```

Since textures are implemented as surfaces, we can use the same routine *VL_RestoreSurface()* to insure that the memory still exists on the video board. In addition, we'll need to re-load the contents of the texture with *VL_LoadBuffer()*. Remember that *locked1* and *locked2* still contain the locked memory contents of the texture. If they didn't, we might have had to re-lock some memory and initialize it before calling *VL_LoadBuffer()*.

As far as the pixel engine state is concerned, all we need to do is insure that the correct filtering options are specified by having a call to *redline_SetFilter()* within the initialization routine for the pixel engine state *redline_InitState()*

```
void redline_InitState
      ( void )
{
      // initially point sampled filter
      redline_SetFilter(redline_Filter);

      // when drawing, use texture colors
      VL_SetSrcFunc(&cmdbuffer, V_SRCFUNC_REPLACE);
}
```

Modification Three: Z buffering and Perspective Correction

One of the problems first faced when drawing a three dimensional scene is how to insure that triangles in the back of the scene are drawn behind triangles in the front of the scene. If this isn't handled properly, you could have objects appear in front of others when they should be behind them.

One solution to this is to sort your triangles and draw them according to the sorted order. This solution requires a bit of programming along with some extra CPU cycles. One simpler solution we're going to describe here is how to use the Z-buffer to determine the visibility of the triangles. By telling our chip to use a Z buffer with a specific Z-comparison function, each triangle will appear correctly even though polygons may be rendered in a random order by the application.

In addition to determining polygon order, the Z-value should also be used to correct texture mapping so that it has the correct perspective. This is not directly done with the Z attribute. Instead, this will involve adding another attribute Q to our vertex so that the texture appears with perspective correction. Q will simply be a function of our Z value.

This example will just use the previous texture mapped triangle example and add the described functionality. The code for this example can be found in the sub-directory *htri4*.

Creating and Using a Z-buffer

In order to create the Z-buffer all we need is to modify the arguments passed when creating the front and back buffers. This is done in *redline_SetDisplay()* contained in

the file *utils.c*.

```
      // create new front, back, and Z buffer
      VL_CreateSurface(verite, &redline_Display,
      V_SURFACE_PRIMARY | V_SURFACE_Z_BUFFER,
      3, pixel_fmt, width, height);
```

The only changes we've made is to add the flag *V_SURFACE_Z_BUFFER* to the surface's buffer mask and to tell the routine to create three buffers instead of two. The Z-buffer will automatically become the last buffer in that surface.

In the same way that we had to install the destination buffer after creating the back buffer, we now have to install the Z-buffer after we've created it. This code is found in *redline_InitState()* since these values are part of the pixel engine state.

```
      // install and use Z-buffer
      VL_InstallZBuffer(&cmdbuffer, redline_Display);
```

We also need to specify the Z-buffer modes the application plans to use. This code is found in *redline_InitState()* since these values are part of the pixel engine state.

```
   // Z buffer modes
      VL_SetZBufWrMode(&cmdbuffer, V_ZBUFWRMODE_ENABLE);
      VL_SetZBufMode(&cmdbuffer, V_ZBUFMODE_LT);
```

These commands enable writing to this buffer and set up the comparison function used to determine if a pixel is going to be drawn. In this example we'll use *V_ZBUFMODE_LT* as our comparison function. This states that if the drawn pixel contains a Z-value that is less than the current value found in the Z-buffer then draw the new pixel. In other words, Z increases the further you go into the screen.

Clearing the Z-buffer

Before we render to the scene we should insure that the Z-values are initialized. We'll do this with the function *redline_ClearZBuffer()* found in the file *utils.c*.

```
void redline_ClearZBuffer
      ( void )
{
      // clear the Z buffer of the display (fill with farthest value)
   VL_FillBuffer(&cmdbuffer, redline_Display, 2,
      0, 0, redline_Display->width, redline_Display->height,
                0xFFFF);
}
```

In the same way that we used *VL_FillBuffer()* to clear the back buffer we'll use it now to place a value into the Z-buffer. Note that the fill value used is no longer a zero. If we were to store that value then any triangle that had a Z-value greater than zero would not be drawn. In this example, this would result in neither one of the triangles being drawn. To solve this problem we're storing the largest value possible in our Z-buffer *0xFFFF*. This is the largest value possible when using unsigned 16.0 values.

Rendering the Triangle

Rendering the triangles will only require some slight changes to the rendering routine. First, we've got to pick another vertex type that contains the attributes we require.

```
Z
```

Z is the depth attribute for the vertex. It is used only for Z-buffering and not for perspective correct texture mapping. In a vertex structure, Z is a 16.16 fixed point value that ranges from 0 to 65535. Note that Z is interpolated internally as 21.11 so the low order 5 bits of the supplied Z are discarded. It is then stored in our 16 bit Z buffer as 16.0. Note that some applications store 1/Z in the buffer in order to get more resolution at smaller values of Z. If this is the case, comparison functions should be modified to scale the Z to the 0 to 65535 range.

Depth Value (Z)

msb 31	lsb 0
16.16 fixed point Z value	

If the vertex does not contain a Z value the default value from VL_SetZ() will be used.

Q

Q is unsigned 0.24, and is in the range [0.0, 1.0). Q is the scaled reciprocal of the homogeneous w, or the reciprocal of Z_{eye}. Note that for multi-vertex primitives (e.g., lines, triangles) you can scale all of the Q's by the same amount (say, a power of 2) without affecting the perspective correction. To maximize accuracy, the largest Q of a multi-vertex primitive should be in the range [0.5, 1.0).

Scaled Reciprocal of Homogenous W (Q)

msb 31	lsb 0
0.24 unsigned fixed point Q coordinate	

For this example we'll require the new Z and Q attribute in addition to the X,Y screen coordinate and the U,V texture coordinate. Referring back to the table of vertices we find the obvious choice for our vertex type V_FIFO_XYZUVQ.

```
void render_scene
       ( void )
{
       static v_xyzuvq      tri1[] =
       {
              {
                     FLTOIFIX(100.0), FLTOIFIX(100.0), FLTOIFIX(2.0),
                     FLTOIFIX(0.0),   FLTOIFIX(0.0),
FLTOQFIX(1/2.0),
              },
              {
                     FLTOIFIX(500.0), FLTOIFIX(100.0), FLTOIFIX(4.0),
                     FLTOIFIX(2.0),   FLTOIFIX(0.0),
FLTOQFIX(1/4.0),
              },
              {
                     FLTOIFIX(100.0), FLTOIFIX(300.0), FLTOIFIX(2.0),
                     FLTOIFIX(0.0),   FLTOIFIX(2.0),
FLTOQFIX(1/2.0),
              }
       };
```

```
	static v_xyzuvq	tri2[] =
	{
		{
			FLTOIFIX(200.0), FLTOIFIX(150.0), FLTOIFIX(6.0),
			FLTOIFIX(0.0),   FLTOIFIX(0.0),
FLTOQFIX(1/3.0),
		},
		{
			FLTOIFIX(600.0), FLTOIFIX(150.0), FLTOIFIX(8.0),
			FLTOIFIX(2.0),   FLTOIFIX(0.0),
FLTOQFIX(1/5.0),
		},
		{
			FLTOIFIX(200.0), FLTOIFIX(350.0), FLTOIFIX(6.0),
			FLTOIFIX(0.0),   FLTOIFIX(2.0),
FLTOQFIX(1/3.0),
		}
	};
```

All we've done here is add Z and Q values to the vertices and use a new vertex structure. Since the first triangle contains Z values that are less than the second triangle, it should be displayed in front of it even though it is drawn first. In addition we're using a different macro to store the value for Q

```
#define FLTOQFIX(a)    ((v_u32)(((float)(a))*((float)((1<<24) - 1))))
```

A new macro is required since Q is not a 16.16 fixed point number but a 0.24 fixed point number. This macro successfully stores a zero as the smallest 0.24 number (0x00000000) and a one as the largest 0.24 fixed point value (0x00FFFFFF). Note that we're simply using the reciprocal of Z in order to calculate Q. Any similar calculation should work fine for your applications as long as it falls within the proper range.

The code to render the triangle remains unchanged except for the vertex type being passed to *VL_Triangle()*.

```
	// use first checkerboard
	VL_InstallTextureMap(&cmdbuffer, checker1);

	// draw triangle
	VL_Triangle(&cmdbuffer, V_FIFO_XYZUVQ,
	(v_u32 *)&(tri1[0]), (v_u32 *)&(tri1[1]),
	(v_u32 *)&(tri1[2]));

	// use second checkerboard
	VL_InstallTextureMap(&cmdbuffer, checker2);

	// draw triangle
	VL_Triangle(&cmdbuffer, V_FIFO_XYZUVQ,
	(v_u32 *)&(tri2[0]), (v_u32 *)&(tri2[1]),
	(v_u32 *)&(tri2[2]));

	// display back buffer
	redline_PageFlip();
}
```

Modification Four: Alpha Blending

Some applications may require some more effects other than just drawing flat shaded,

gouraud shaded, or texture mapped triangles. They may wish to have polygons blended together in order to get transparency effects or in order to use texture mapped lighting. Either way, there are a lot of effects that can be accomplished with alpha blending.

For this example we're going to start with an older example, *Modification Two: Drawing a Texture Mapped Triangle* (htri3) and add alpha blending. The textures we create will contain alpha values that when rendered will give the effect that the red and white texture mapped triangle is slightly transparent.

The code for this example will be included in the *htri5* sub-directory.

Alpha Blending Basics

Before we start talking about the calls needed to enable alpha blending we need to explain a little more about how the Vérité uses state settings to determine the final pixel stored in into the destination buffer.

In the case where there is no alpha blending, the equations used to calculate the new pixel values are simple:

```
new pixel color =     src color
new pixel alpha =     src alpha
```

The *new pixel alpha* value is only stored if the destination pixel format contains alpha, like when drawing to a *V_PIXFMT_4444* pixel format. The *src color* and *src alpha* are determined by the source function. The actual source values can be determined by checking the following table of *VL_SetSrcFunc()* modes:

Source Function Mode	Values Used
V_SRCFUNC_NOTEXTURE	`src color = color` `src alpha = alpha`
V_SRCFUNC_REPLACE	`src color = texture color` `src alpha = texture alpha`

V_SRCFUNC_DECAL	`src color = (1 - texture alpha) * color +`
	`texture alpha * texture color`
	`src alpha = alpha`

V_SRCFUNC_MODULATE	`src color = color * texture color` `src alpha = alpha * texture alpha`

The *color* and *alpha* values come from the vertex structure. If the vertex does not contain these attributes then the default color and the default alpha will be used. The *texture color* and *texture alpha* values always come from the installed texture. If the texture's format does not include alpha, a maximum alpha value of 255 will be used. If the texture format does not include color, RGB color values of 255 (white) will be used. Note that the standard color and alpha ranges of 0 to 255 are scaled to be from 0 to 1 for purposes of these equations.

Things start to become trickier when alpha blending is enabled. If it is, the new pixel value equations become:

```
new pixel color =     blend src function * src color +
                    blend dst funciton * dst color
new pixel alpha =     blend src function * src alpha +
                    blend dst function * dst alpha
```

The values for *src color* and *src alpha* are still defined by *VL_SetSrcFunc()*. The *dst color* and *dst alpha* are the existing values of the pixel being drawn to (i.e. *old pixel color* and *old pixel alpha*). The new values *blend src function* and *blend dst function*

come from *VL_SetBlendDstFunc()* and *VL_SetBlendSrcFunc()*:

VL_SetBlendDstFunc() Argument	blend dst function
V_BLENDSRCCOLOR	`src color`
V_BLENDSRCCOLORINV	`1 - src color`
V_BLENDSRCALPHA	`src alpha`
V_BLENDSRCALPHAINV	`1 - src alpha`
V_BLENDDSTALPHA	`dst alpha`
V_BLENDDSTALPHAINV	`1 - dst alpha`
V_BLEND0	`0`
V_BLEND1	`1`

VL_SetBlendSrcFunc() Argument	blend src function
V_BLENDDSTCOLOR	`dst color`
V_BLENDDSTCOLORINV	`1 - dst color`
V_BLENDSRCALPHA	`src alpha`
V_BLENDSRCALPHAINV	`1 - src alpha`
V_BLENDDSTALPHA	`dst alpha`
V_BLENDDSTALPHAINV	`1 - dst alpha`
V_BLEND0	`0`
V_BLEND1	`1`
V_BLENDSRCALPHASAT	`MIN(src alpha, 1 - dst alpha)`
V_BLENDSRCALPHASATINV	`1 - MIN(src alpha,` `  1 - dst alpha)`

In addition to these settings, the only other thing you can set for alpha blending is the destination pixel. If you need to, you can ignore the current destination pixel value and substitute one of your own. This is done by disabling destination pixel read with *VL_SetDstRdDisable()* and by setting the destination pixel attributes with *VL_SetDstColorARGB()* or *VL_SetDstColorABGR()*. This may be done in order to incorporate a constant value into these blend equations such as specular color.

One thing to remember is how these equations interact with the Z-buffer. While the Z-buffer determines when a pixel is rendered it does not determine what pixel it is blended with. If the correct destination pixel hasn't been drawn yet the blending results will come out incorrect. Therefore, even if doing Z-buffering, you must first render the base scene and then add alpha blended effects afterwards (like lighting, etc.)

In the case of our example, all we're going to do is enable alpha blending and blend the pixel of the texture with the existing destination pixel according to the texture's alpha value. This will result in a slightly transparent texture mapped triangle for the second triangle. The first triangle will remain unaffected since it contains the maximum alpha value. These alpha blend settings are done in *redline_Init()* in the file *init.c*.

```
   // turn on alpha blending and set blend states
      VL_SetBlendEnable(&cmdbuffer, V_BLEND_ENABLE);
      VL_SetBlendSrcFunc(&cmdbuffer, V_BLENDSRCALPHA);
      VL_SetBlendDstFunc(&cmdbuffer, V_BLENDSRCALPHAINV);
```

Textures with Alpha

The source function is still set to *V_SRCFUNC_REPLACE* telling use that our source values come from the texture so we'll need to store our alpha values in the texture itself. To do this, we'll modify the call *create_texture()* from *texture.c* to create the texture so that the pixel format is 4444 (4 bits each of alpha, red, green, and blue)

```
   // create surface
      VL_CreateSurface(verite, &result, 0, 1,
      V_PIXFMT_4444, width, height);
```

The colors stored in the checkerboard textures need to be different as well. That will mean a change to the last argument of the *locked_checkerboard()* calls in *redline_CreateTextures()* in the file *texture.c*.

```
   // place two checkerboard patterns in locked memory
      locked1 = locked_checkerboard(128, 256, 16, 32, 0xF0F0,
0xF00F);
      locked2 = locked_checkerboard(200, 100, 10, 10, 0x7F00,
0x7FFF);
```

The first checkerboard is created with the maximum alpha value. This means that it won't be transparent at all. However, the next texture has approximately half alpha resulting in the second triangle being about half transparent.

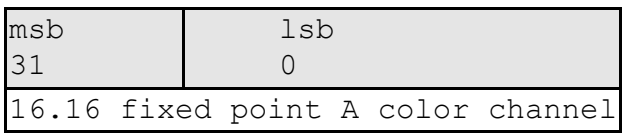
Rendering the Triangle

When rendering the triangle there is no change from the original code. We've already set the necessary alpha modes and stored the alpha values in the texture itself. If we wanted to store the alpha value in the vertex instead of the alpha we would have had to choose a new vertex type that contains the alpha attribute.

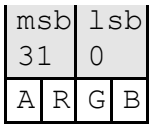
A

A is the alpha attribute for each vertex. For all but one vertex type, alpha is stored as a 16.16 fixed point value that ranges from 0 to 255. The only exception to this rule is with the vertex type V_FIFO_KaSFXYZUVQ. It stores alpha in the eight available bits of the compressed RGB color value K. In that case, alpha is an 8.0 fixed point value ranging from 0 to 255.

Alpha (A)



Packed RGB (K) with Alpha



If a vertex does not contain an alpha attribute the default value will be used. Setting the default alpha can be done directly with VL_SetA() or along with the color value with either VL_SetFGColorARGB() or VL_SetFGColorABGR().

Modification Five: Specular Extensions

When adding new functionality to RRedline, either through new functions or new vertex types, we don't just add them to the library and act like they've always been there. RRedline dlls already exist and are available to users. To modify the library at this point would result in programs failing with unidentifiable errors when they link with older versions of the dlls that don't contain the functions they require. To solve this problem we provide a mechanism for programs to acquire function pointers if certain extensions are found. This allows new features to be supported within RRedline and still be compatible with older architectures.

One of the new features of the V2000 series of cards is that they support per-vertex specular highlights. This feature, however, is not directly supported within the hardware of the V1000 series of chips. What we're going to show in this example is how to detect what functionality exists within the current RRedline/chip combination and how to implement specular highlights when the extension doesn't exist on the V1000.

This example is based on the Z-buffered texture mapped example *Modification Three: Z buffering and Perspective Correction* (htri4). The files for this example are stored in the sub-directory *htri6*.

Specular Extensions for the V2000 series

On the V2000 series of cards specular is supported as a vertex attribute. It is therefore possible to choose a vertex type that contains specular and store the specular color values within each vertex. However, this extension is not supported on the V1000 series of cards. It is therefore important for the program to determine what chip is being used and process the effect accordingly.

This detection is done through the use of extensions. By asking the card what extensions are available, you'll be able to determine what additional capabilities are available or missing. You can then determine what needs to be done in order to get specular functioning properly. This detection is being done within *redline_Init()* in the file *init.c*.

```
char   *exts;

   ….

   // get and check extensions for specular
   exts = VL_GetExtensions(verite);
   specular_available = (v_u16)strstr(exts, " Specular ");
```

Each extension supported by the current RRedline/chip combination is contained in the string returned from *VL_GetExtensions()*. Each extension is always begins and ends with a space. Note that we're looking for these spaces in our string search.

Otherwise our *strstr()* call would result in a match for any future extensions that contain the string “Specular” like “SpecularX” or “Specular_Adaptations”.

Once we’ve determined that the specular extension does exist, we can make calls to enable specular highlights. However, the function itself isn’t part of the library that you link with when compiling your program. If it did then your program would fail if it ran into a RRedline dll that didn’t contain the specular function. Instead, you need to gain access to it via a function pointer returned by *VL_GetExtensionFunction()*.

```
// get specular function
   if (specular_available)
   VL_SetSpecularEnable = (VL_SetSpecularEnable_Type)
         VL_GetExtensionFunction("VL_SetSpecularEnable");
```

The global function pointer *VL_SetSpecularEnable()* is declared at the top of *init.c*. Its type *VL_SetSpecularEnable_Type* is declared in the include file *rlex.h*. This include file contains all the necessary declarations for all the available extensions. This file should be included whenever extensions are used

```
VL_SetSpecularEnable_Type    VL_SetSpecularEnable = NULL;
```

Now that we’ve got a pointer to this function, it’s time to call it so that specular is enabled. The calls setting up the specular state are included in the function *redline_InitState()* found in *init.c* since they are part of the pixel engine state and need to be set if the application ever loses focus to another application.

```
// specular extensions available?
   if (specular_available)
   {
         // enable specular on V2000 series
         VL_SetSrcFunc(&cmdbuffer, V_SRCFUNC_REPLACE);
         VL_SetSpecularEnable(&cmdbuffer, V_SPECULAR_ENABLE);
   } else
   {
         …
   }
```

In the same manner as alpha blending, specular must be enabled before the specular attribute is used. Note that we’ve included a call to set the source function as well since the V1000 method will need to store its specular value within the vertex requiring a change to the source function mode.

Specular Alpha Settings for the V1000

On the V1000 series of cards, specular extensions are not available since that hardware does not support per vertex specular types. However, if you wish to still be able to do specular highlights on that chipset, it is possible by using the alpha value.

In order to determine what alpha modes are required, we should first take a look at the input data and then determine how to get the proper equations. Our input consists of three items: our texture map, per vertex alpha value containing the specular value, and the specular color. The first thing to determine is the source function to be used. Since we’re requiring that data come from the vertex and from the texture, we’ll need to use either V_SRCFUNC_DECAL or V_SRCFUNC_MODULATE. The result being that we want to have

```
src color =    texture color
src alpha =    vertex alpha
```

It turns out that we can get this result from either of the two listed source function

modes. For V_SRCFUNC_DECAL the equations are:

```
   src color =    (1 - texture alpha) * color +
                    texture alpha * texture color
   src alpha =    alpha
```

Since the texture contains no alpha value, the default of 255 will be used for *texture alpha*. Remembering that 255 is translated to 1 for the purpose of the source and blending equations, the V_SRCFUNC_DECAL functions automatically become:

```
      src color =    0 * color + 1 * texture color = texture color
      src alpha =    alpha
```

Which is exactly what we needed. If we wanted to use V_SRCFUNC_MODULATE instead we would have started with these equations:

```
   src color =    color * texture color
   src alpha =    alpha * texture alpha
```

With the *texture alpha* still being 255, if the *color* value were white (translated to 1 for the purpose of the source and blending equations) then the modulate equations would become:

```
      src color =    1 * texture color =   texture color
      src alpha =    alpha * 1 =    alpha
```

Which is again the proper result. The only difference being that for this mode we need to have each vertex have a white color or set the default color to white when the vertex contains no color attribute. Since specular is normally implemented with a color value, we'll use this V_SRCFUNC_MODULATE version.

Now that we have our source worked out, we'll need to set up the blend functions. What we need is to have our specular value multiplied by the specular color and added to our texture map pixel to get the final result. Knowing that our specular value is stored as *src alpha* and the texture pixels are stored as *src color*, we can then write the equation we need as:

```
      new pixel color =     src color + src alpha * constant specular
color
```

We'll need to derive this function from the standard blend function. With a destination pixel format of *V_PIXFMT_565* we aren't interested in the blend function for *new pixel alpha* since it isn't being stored.

```
      new pixel color =     blend src function * src color +
                           blend dst funciton * dst color
```

We'll use V_BLEND1 for the blend source function so that we'll end up with just *src color* for that part of the equation. We'll also use V_BLENDSRCALPHA for the blend destination function so that the *src alpha* is introduced in the second part of the equation. The problem then becomes how to store the specular color as the *dst color* value.

One of the tricks you can use for alpha blending is to tell the Vérité to ignore the existing destination pixel and replace it with a value of your own. This is a good way to introduce constants into your blend equations. In this case, we'll do this by disabling the destination read and storing our constant specular color as the destination color. The blend equations then become:

```
      new pixel color =     1 * src color + src alpha * specular
color
```

Which is again precisely what we need.

All these settings are done within *redline_InitState()* if specular extensions don't exist.

```
   // specular extensions available?
      if (specular_available)
      {
            …
      } else
      {
      // use specular alpha blending on V1000 series
      VL_SetSrcFunc(&cmdbuffer, V_SRCFUNC_MODULATE);
      VL_SetBlendEnable(&cmdbuffer, V_BLEND_ENABLE);
      VL_SetBlendSrcFunc(&cmdbuffer, V_BLEND1);
      VL_SetBlendDstFunc(&cmdbuffer, V_BLENDSRCALPHA);
      VL_SetDstRdDisable(&cmdbuffer, V_DSTRD_DISABLE);
      }
```

Rendering with Specular

With the appropriate V1000 and V2000 states set all we have to do now is render the triangles. Each version will render triangles in a slightly different manner. The V1000 will implement specular by using a vertex type that contains an alpha attribute. The V2000 will implement the effect by using a specular vertex type.

We'll start first with the vertices of the triangles for the V1000 version. For that version we'll need a vertex type that contains alpha along with K, X, Y, Z, U, V, and Q. The smallest vertex type fitting that description is the type V_FIFO_KAXYZUVQ which contains exactly what we need.

```
void render_scene
      ( void )
{
      static v_kaxyzuvq    tri1[] =
      {
      {
            FLTOARGB(255.0, 255.0, 255.0, 255.0),
            FLTOIFIX(255.0),
            FLTOIFIX(100.0),  FLTOIFIX(100.0),  FLTOIFIX(2.0),
            FLTOIFIX(0.0),    FLTOIFIX(0.0),    FLTOQFIX(1/2.0),
      },
      {
            FLTOARGB(255.0, 255.0, 255.0, 255.0),
            FLTOIFIX(0.0),
            FLTOIFIX(500.0),  FLTOIFIX(100.0),  FLTOIFIX(4.0),
            FLTOIFIX(2.0),    FLTOIFIX(0.0),    FLTOQFIX(1/4.0),
      },
      {
            FLTOARGB(255.0, 255.0, 255.0, 255.0),
            FLTOIFIX(200.0),
            FLTOIFIX(100.0),  FLTOIFIX(300.0),  FLTOIFIX(2.0),
            FLTOIFIX(0.0),    FLTOIFIX(2.0),    FLTOQFIX(1/2.0),
      }
      };
      static v_kaxyzuvq    tri2[] =
      {
      {
```

```
        {
              FLTOARGB(255.0, 255.0, 255.0, 255.0),
              FLTOIFIX(255.0),
              FLTOIFIX(200.0),  FLTOIFIX(150.0),  FLTOIFIX(6.0),
              FLTOIFIX(0.0),    FLTOIFIX(0.0),    FLTOQFIX(1/3.0),
        },
        {
              FLTOARGB(255.0, 255.0, 255.0, 255.0),
              FLTOIFIX(0.0),
              FLTOIFIX(600.0),  FLTOIFIX(150.0),  FLTOIFIX(8.0),
              FLTOIFIX(2.0),    FLTOIFIX(0.0),    FLTOQFIX(1/5.0),
        },
        {
              FLTOARGB(255.0, 255.0, 255.0, 255.0),
              FLTOIFIX(200.0),
              FLTOIFIX(200.0),  FLTOIFIX(350.0),  FLTOIFIX(6.0),
              FLTOIFIX(0.0),    FLTOIFIX(2.0),    FLTOQFIX(1/3.0),
        }
        };
```

For the V2000 series we'll need a vertex type that contains the new specular attribute

S

S is a special vertex attribute that stands for specular color. This attribute is only supported on those chips that allow specular extensions. The V1000 series of chips does not support those extensions. Specular color is stored in the same format as packed RGB with each 8.0 fixed point RGB value stored in a specific location of a 32 bit word.

Specular Color (S)

msb 31			lsb 0
-	R	G	B

If a vertex does not contain a specular value the default specular color, set with *VL_SetSpecularColorRGB()* or *VL_SetSpecularColorBGR()* will be used. Note that both of these functions are extension functions for specular and do not exist on the V1000 series of cards.

In addition to the specular attribute we'll need a vertex type that contains X, Y, Z, U, V, and Q. We find that the smallest vertex type containing these attributes contains an extra K attribute which will be initialized to white. The vertex type used is V_FIFO_KSXYZUVQ.

```
static v_ksxyzuvq    tri1b[] =
    {
    {
          FLTOARGB(255.0, 255.0, 255.0, 255.0),
          FLTOARGB(0.0, 255.0, 0.0, 0.0),
          FLTOIFIX(100.0),  FLTOIFIX(100.0),  FLTOIFIX(2.0),
          FLTOIFIX(0.0),    FLTOIFIX(0.0),    FLTOQFIX(1/2.0),
    },
    {
```

```
        FLTOARGB(255.0, 255.0, 255.0, 255.0),
        FLTOARGB(0.0, 0.0, 0.0, 0.0),
        FLTOIFIX(500.0),  FLTOIFIX(100.0),  FLTOIFIX(4.0),
        FLTOIFIX(2.0),    FLTOIFIX(0.0),    FLTOQFIX(1/4.0),
    },
    {
        FLTOARGB(255.0, 255.0, 255.0, 255.0),
        FLTOARGB(0.0, 200.0, 0.0, 0.0),
        FLTOIFIX(100.0),  FLTOIFIX(300.0),  FLTOIFIX(2.0),
        FLTOIFIX(0.0),    FLTOIFIX(2.0),    FLTOQFIX(1/2.0),
    }
    };
    static v_ksxyzuvq    tri2b[] =
    {
    {
        FLTOARGB(255.0, 255.0, 255.0, 255.0),
        FLTOARGB(0.0, 0.0, 0.0, 255.0),
        FLTOIFIX(200.0),  FLTOIFIX(150.0),  FLTOIFIX(6.0),
        FLTOIFIX(0.0),    FLTOIFIX(0.0),    FLTOQFIX(1/3.0),
    },
    {
        FLTOARGB(255.0, 255.0, 255.0, 255.0),
        FLTOARGB(0.0, 0.0, 0.0, 0.0),
        FLTOIFIX(600.0),  FLTOIFIX(150.0),  FLTOIFIX(8.0),
        FLTOIFIX(2.0),    FLTOIFIX(0.0),    FLTOQFIX(1/5.0),
    },
    {
        FLTOARGB(255.0, 255.0, 255.0, 255.0),
        FLTOARGB(0.0, 0.0, 0.0, 200.0),
        FLTOIFIX(200.0),  FLTOIFIX(350.0),  FLTOIFIX(6.0),
        FLTOIFIX(0.0),    FLTOIFIX(2.0),    FLTOQFIX(1/3.0),
    }
    };
```

All we have to do now is install the texture we're going to use and then render the triangle. When specular extensions are available, the triangle will simply be rendered with a call to *VL_Triangle()*. When they aren't available, the specular color is set with *VL_DstColorARGB()* and the triangle is then drawn with *VL_Triangle()*.

```
    // use first checkerboard
    VL_InstallTextureMap(&cmdbuffer, checker1);

    // specular not available?
    if (!specular_available)
    {
    // choose specular color and draw triangle
    VL_SetDstColorARGB(&cmdbuffer,
    FLTOARGB(255.0, 255.0, 0.0, 0.0));
    VL_Triangle(&cmdbuffer, V_FIFO_KAXYZUVQ,
           (v_u32 *)&(tri1[0]), (v_u32 *)&(tri1[1]),
       (v_u32 *)&(tri1[2]));
    } else
    {
    // draw triangle with specular vertex type
    VL_Triangle(&cmdbuffer, V_FIFO_KSXYZUVQ,
           (v_u32 *)&(tri1b[0]), (v_u32 *)&(tri1b[1]),
    (v_u32 *)&(tri1b[2]));
```

```
	}

	// use second checkerboard
	VL_InstallTextureMap(&cmdbuffer, checker2);

	// specular vertex not available?
	if (!specular_available)
	{
	// choose specular color and draw triangle
	VL_SetDstColorARGB(&cmdbuffer,
		FLTOARGB(255.0, 0.0, 0.0, 255.0));
	VL_Triangle(&cmdbuffer, V_FIFO_KAXYZUVQ,
		(v_u32 *)&(tri2[0]), (v_u32 *)&(tri2[1]),
		(v_u32 *)&(tri2[2]));
	} else
	{
	// draw triangle with specular vertex type
	VL_Triangle(&cmdbuffer, V_FIFO_KSXYZUVQ,
		(v_u32 *)&(tri2b[0]), (v_u32 *)&(tri2b[1]),
		(v_u32 *)&(tri2b[2]));
	}

	// display back buffer
	redline_PageFlip();
}
```

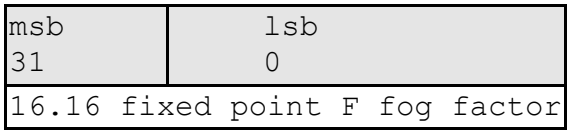
Attributes not covered in examples

In our current examples the following attributes were not covered

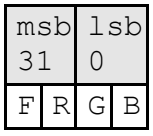
F

F is the fog attribute for the vertex. Most of the time it is a 16.16 fixed point value ranging from 0 (full fog) to 255 (no fog). In the vertex type V_FIFO_KaSFXYZUVQ the fog value is stored in the free 8 bits of the specular attribute as an 8.0 fixed point value ranging from 0 to 255.

Fog Alpha (F)



Specular Color (S) with Fog Alpha (F)



If the vertex does not contain a fog value the default value from VL_SetF() will be used.

Optimizations

Now that you know how to program RRedline so that it works it is know important for you to know how to get it to work fast. Following are a list of several

for you to know how to get it to work fast. Following are a list of several enhancements that you may be able to use for you own application. Some require that you only change the functions called while others require more effort. In either case, I'd recommend that you add these optimizations after you get your RRedline application running. The reason being that some of these optimizations, if not done correctly, can result in the Vérité locking up in such a way that it will be difficult to track down. However, if applied one at a time these optimizations can result in greater performance and a higher frame rate for your application. One nice thing to note is that some of these optimizations can be implemented within other APIs as well including Direct3D and OpenGL.

Note that some of the optimizations discussed in this document are most useful on the V1000 and may have only minor impact on the V2000.

Add drawing primitive data to the command list directly

You learned in the first part of the programming guide that the RRedline method for drawing a primitive is to convert your application's data format to the RRedline data format as you fill up an array of RRedline vertex structures. You then call a *VL_* function (like *VL_Triangle*) providing the vertex data through the function's arguments. However, we've found that that isn't the most efficient method for handling this procedure. In fact, by creating your own version of *VL_Triangle()* you can get some quick performance increases.

In this example we'll do this for *VL_Triangle()*. However, all the other primitive rendering functions such as *VL_Trifan(), VL_Tristrip(),* etc. will have similar versions as well. Details on the command buffer structure for each of those primitives can be found in the reference guide and used to modify the described source code. Here is the source for VL_Triangle (minus error checking).

```
vl_error VL_Triangle
       ( v_cmdbuffer        *cmdbuffer,
         v_u32       v_type,
         v_u32       *v1,
         v_u32       *v2,
         v_u32       *v3 )
{
       /* Compute number of bytes in vertex list */
       v_u32       lcnt = ((VL_VertexSize(v_type)) + 3) >> 2;
       v_u32       *cmdlist_ptr;

       /* Allocate space on the command buffer for the command and
data */
       cmdlist_ptr = V_AddToCmdList(cmdbuffer, (3*lcnt+1));

       /* Write vertex count & command */
       *(cmdlist_ptr++) = ((v_type << 16) | V_FIFO_TRIANGLE);

       /* Copy each vertex into the command buffer */
       memcpy(cmdlist_ptr, v1, (lcnt << 2));
       cmdlist_ptr += lcnt;
       memcpy(cmdlist_ptr, v2, (lcnt << 2));
       cmdlist_ptr += lcnt;
       memcpy(cmdlist_ptr, v3, (lcnt << 2));
}
```

As you can see, all this function needs to do is determine how much space is required on the command buffer (in 32 bit words), place the FIFO primitive drawing command along with the vertex type in the command buffer, and then copy each vertex into the command buffer. For your application, there are a few key places

where this function can be modified to work better with you application.

You can avoid putting your data in RRedline vertex types and extra *memcpy()* calls by calling *V_AddToCmdList()* within your own *VL_Triangle()* routine and placing your data directly into the command list. This procedure also does not require that any RRedline vertex structure be created. In the following example, an imaginary application's data is converted to the IXYUVQ vertex type at the same time it is placed in the command buffer.

```
vl_error VL_AppTriangle
       ( v_cmdbuffer        *cmdbuffer,
         app_vertex         *v1,
         app_vertex         *v2,
         app_vertex         *v3 )
{
       /* Compute number of bytes in vertex list */
       v_u32  lcnt = ((VL_VertexSize(V_FIFO_IXYUVQ)) + 3) >> 2;
       v_u32  *cmdlist_ptr;

       /* Allocate space on the command buffer for the command and
data */
       cmdlist_ptr = V_AddToCmdList(cmdbuffer, (3*lcnt+1));

       /* Write vertex count & command */
       *(cmdlist_ptr++) = (V_FIFO_IXYUVQ << 16) | V_FIFO_TRIANGLE;

       /* Copy each vertex into the command buffer */
       cmdlist_ptr[0] = v1->light << 16;                  /* I */
       cmdlist_ptr[1] = FLTOIFIX(v1->screen_x);    /* X */
       cmdlist_ptr[2] = FLTOIFIX(v1->screen_y);    /* Y */
       cmdlist_ptr[3] = (v1->tex & 0xFF00) << 8;   /* U */
       cmdlist_ptr[4] = (v1->tex & 0xFF) << 16;    /* V */
       cmdlist_ptr[5] = FLTOQFIX(1.0 / v1->z);            /* Q */
       cmdlist_ptr += 6;

       cmdlist_ptr[0] = v2->light << 16;
       cmdlist_ptr[1] = FLTOIFIX(v2->screen_x);
       cmdlist_ptr[2] = FLTOIFIX(v2->screen_y);
       cmdlist_ptr[3] = (v2->tex & 0xFF00) << 8;
       cmdlist_ptr[4] = (v2->tex & 0xFF) << 16;
       cmdlist_ptr[5] = FLTOQFIX(1.0 / v2->z);
       cmdlist_ptr += 6;

       cmdlist_ptr[0] = v3->light << 16;
       cmdlist_ptr[1] = FLTOIFIX(v3->screen_x);
       cmdlist_ptr[2] = FLTOIFIX(v3->screen_y);
       cmdlist_ptr[3] = (v3->tex & 0xFF00) << 8;
       cmdlist_ptr[4] = (v3->tex & 0xFF) << 16;
       cmdlist_ptr[5] = FLTOQFIX(1.0 / v3->z);
}
```

For other primitives, the *FIFO* primitive command will change and there may be another word needed in the command buffer in order to store a count such as with triangle fans and strips. When storing other vertex types you just need to remember that the attributes are added to the command list in the order listed in the vertex type (order for V_FIFO_IXYUVQ is I, X, Y, U, V, then Q).

Note that we recommend doing this optimization (and all other command list management) later in the optimization process, as it is easy to generate errors that will crash the graphics system.

will crash the graphics system.

Change the 3D engine's internal data formats to RRedline's formats

To avoid conversion steps when storing vertex information, consider changing your application's internal data format to match RRedline's. This may not be possible for X and Y coordinates that change a lot but may be possible for attributes such as texture coordinates that don't change during the course of a program.

Use fast float to integer

Some C compilers generate bad code for something as simple as

```
int_var = (int)float_var;
```

So it is recommended that when doing the float-to-fixed point conversion of your data that you use some alternate method. One possible method is this, which uses the bit arrangement of double precision floating point numbers to do quick conversions:

```
typedef union {
       unsigned int i[2];
       double f;
} ftoi;

ftoi converter_16_16 =       { 0, 0x42380000 };
ftoi converter_8_24 = { 0, 0x41B80000 };
ftoi converter_0_32 = { 0, 0x41380000 };
ftoi result;

#define FLTO16_16(n)  (result.f = (n) + converter_16_16.f, \
              result.i[0])
#define FLTO8_24(n)   (result.f = (n) + converter_8_24.f, \
                     result.i[0])
#define FLTO0_32(n)   (result.f = (n) + converter_0_32.f, \
                     result.i[0])

#define FLTOIFIX(i) (FLTO16_16(i))
#define FLTOQFIX(q) (FLTO8_24(q) - 1)
#define FLTOZFIX(z) (FLTO0_32(z-0.000000001))
```

FLTOIFIX() creates a 16.16 fixed point number for most Vérité attributes.

FLTOQFIX() creates a 0.24 fixed point number (input must be <= 1) for Q.

FLTOZFIX() creates a 16.16 fixed point number taking 0 <= Z <= 1 and mapping it to 0 <= Z < 65536 for the Vérité Z buffer.

Note that this technique won't work if you have set the floating point control word to perform low-precision math.

Maintain application state records

To avoid calling *VL_Set* functions unnecessarily, you can wrap them in your own "smart" state function. For example, this function could replace *VL_SetSrcFunc()* :

```
void VL_AppSetSrcFunc(v_u32 srcfunc)
{
       static v_u32 lastfunc = -1;
       if (srcfunc == lastfunc)
       return;
```

```
        VL_SetSrcFunc(&cmdbuffer, srcfunc);
        lastfunc = srcfunc;
}
```

All *VL_Set* functions call *V_AddToCmdList()*. While fast, calling the function hundreds of extra times per frame can add up to a performance decrease. Note that the extra bus traffic for the state commands or the time the Vérité spent changing state values are secondary effects on performance— in fact, they are probably hard to measure at all.

Another approach is to track drawing states in a higher level "context" sense. For example, if your application draws four different kinds of things, consider a method like:

```
typedef enum {
        STATE_FIRSTTHING,
        STATE_SECONDTHING,
        STATE_THIRDTHING,
        STATE_FOURTHTHING
} statetype;

void SetState(statetype state)
{
        static statetype laststate;
        if (state == laststate)
        return;

        switch (state)
        {
        case STATE_FIRSTTHING:
                /* set first thing's state */
                break;
        case STATE_SECONDTHING:
                /* set second thing's state */
                break;
        case STATE_THIRDTHING:
                /* set third thing's state */
                break;
        case STATE_FOURTHTHING:
                /* set fourth thing's state */
                break;
        }

        laststate = state;
}
```

Now one check can determine if any *VL_Set* functions need be called at all.

Implement your own texture caching algorithm

One of the best ways to increase performance is to optimize your texture caching algorithm for your application. This can be done by changing your RRedline surface creation calls to lower level implementations, managing the video memory yourself, and preventing locks between the Vérité and the CPU. Preventing locks caused by *VL_CreateSurface()* is especially important since it can eat up a lot of the available performance.

However, this technique is only useful if the amount of textures required for your application exceeds the amount of memory available on the board. Currently Vérité boards contain 4 or 8 megabytes of general purpose memory that is used for

boards contain 4 or 8 megabytes of general purpose memory that is used for microcode, display buffers, Z-buffers, and texture memory. In order to determine if this optimization can help your application you should calculate how much texture memory you have to work with and see if your textures will fit.

To calculate the amount of video memory available, you must sum all the other pieces and subtract from the memory available on the board. For example, lets say that I have an application running on a 4 megabyte board in full-screen mode at 640x480 with 16 bits of color, front and back buffers, and no Z-buffer. To calculate the amount of memory available for textures you would use the following calculation:

```
      4 megs memory on board
-       640 x 480 x 2 x 2 bytes memory for front and back buffers (16
bit)
-       128 K  memory for microcode (fixed)
=      2,768 K available memory (enough for about 86 128 x 128 16 bit
    textures)
```

In windowed mode you don't have a front buffer but you do have the original screen setting to account for. Let's say that your screen is set to 1024 x 768 with 16 bits of color, your application is running at 640 x 480 with 16 bits of color, you have a back buffer, and you have a Z-buffer. The calculation would then become:

```
      4 megs  memory on board
-       1024 x 768 x 2 bytes  memory for screen setting (16 bit)
-       640 x 480 x 2 bytes   memory for back buffer (16 bit)
-       640 x 480 x 2 bytes   memory for Z-buffer (16 bit values)
-       128 K         memory for microcode (fixed)
=      1,232 K available memory
       (enough for about 38 128 x 128 16 bit textures)
```

Even though these formulas are accurate, it is a lot better to use a general purpose algorithm within your application to determine the amount of memory available. Future chipsets may have a lot more memory available that you'll want to use. The algorithm we recommend involves iteratively allocating texture memory on the board. When the algorithm succeeds in allocating, the memory is freed and the amount is increased. When the algorithm fails in allocating, the amount is decreased. This is continued until the amount of available memory is determined.

Following is some code taken from our framework example (*frame* example directory). This example covers a lot of advanced topics such as command buffer management, locked memory management, and texture caching. In this section we'll only talk about its texture caching mechanism.

```
#define CACHE_BUFFER_WIDTH 256

…

v_u32 min_height, max_height, mid_height;
v_buffer_group buffer;
v_u32 size;

…

VeriteErrorHandler = V_GetErrorHandler();

if (V_RegisterErrorHandler(CreateSurfaceCache_ErrorHandler)
       != V_SUCCESS)
{
       DoError(VF_ERROR_V_REGISTER_ERROR_HANDLER);
       return;
```

```
}

min_height = 1;
max_height = 4 * 1024 * 1024 / (CACHE_BUFFER_WIDTH * 2);

while (V_CreateBufferGroup(frame->verite, &buffer, &size,
       0, 1, V_PIXFMT_565, CACHE_BUFFER_WIDTH, max_height) ==
V_SUCCESS)
{
       V_DestroyBufferGroup(frame->verite, buffer);
       min_height = max_height;
       max_height *= 2;
}

while (max_height - min_height > 1)
{
       mid_height = (min_height + max_height) / 2;

       if (V_CreateBufferGroup(frame->verite, &buffer, &size,
              0, 1, V_PIXFMT_565, CACHE_BUFFER_WIDTH,
              mid_height) == V_SUCCESS)
       {
              V_DestroyBufferGroup(frame->verite, buffer);
              min_height = mid_height;
       } else
       {
              max_height = mid_height;
       }
}

if (V_RegisterErrorHandler(VeriteErrorHandler) != V_SUCCESS) {
       DoError(VF_ERROR_V_REGISTER_ERROR_HANDLER);
       return;
}
```

Note that the first thing we've done is installed a new error handler. This is to prevent the *V_CreateBufferGroup()* calls from calling your real error handler and cause a program termination when they're just trying to test memory allocations. All this new error handler needs to do is return the error status given to it (i.e the error returned when *V_CreateBufferGroup()* fails to allocate). At the end of the algorithm, the previous error handler is restored.

Also notice that we used a fixed width of 256 when allocating the buffer. The reason for this is to insure that the width of our text surface never exceeds the width of the primary surface (one of the surface restrictions). This will mean that the final size determined (*max_height)* will be need to be multiplied by 256 to get the number of bytes available. Also note that this routine should be run after the front, back, and Z-buffer are allocated so that you can determine the amount of texture memory only.

If you find that your textures will fit then all you need to do is allocate all of them at the very beginning using *VL_CreateSurface()* and operate normally. However, if you find that they cannot all fit at once it is best that you implement your own texture caching algorithm. This algorithm will need to allocate one large chunk of memory from the board and then break it up according to the application's requirements. The code found in the example *frame* directory works by allocating a huge chunk and starting to allocate textures at the beginning of that heap. When the end of the heap is reached, the memory at the beginning of the heap is freed and re-used. Textures that are booted out of the heap are marked as such and re-loaded back into the heap when needed.

To do your own allocation scheme you need to use a few RRedline functions. First you should use *V_CreateBufferGroup()* to allocate your heap of memory. You then can use *V_GetBufferAddress()* to get a video memory pointer to that allocated buffer

can use *V_GetBufferAddress()* to get a video memory pointer to that allocated buffer. To use that heap, you first need to determine how much room the texture will use. Normally, you would think that the memory you need to store a texture would be pixel width * pixel size * pixel height. However, the Vérité may need to pad the texture with some extra memory. To determine the actual size, you need to make the following calls (code taken from *frame)*:

```
pixel_size = V_PIXEL_SIZE(pixel_fmt);
V_ASSERT(pixel_size != -1);

if (!pixel_size)
       linebytes = (width + 1) / 2;
else
       linebytes = width * pixel_size;
linebytes = (linebytes + 0x3) & ~0x3;


…


surface->vid_stride =
   V_Stride(frame->verite, linebytes);
surface->vid_linebytes =
   V_Linebytes(frame->verite, surface->vid_stride);
surface->vid_size = surface->vid_linebytes * surface->height;
```

The first piece of code computes how many bytes per line the texture takes on the CPU side (*linebytes)*. Note that this value is rounded up to be 32 bit aligned. Next that *linebytes* value is passes to *V_Stride()* to get a stride index capable of storing that texture width. However, the value returned isn't that useful so we'll need to convert it back into a linebytes value. We'll do that by passing the stride to *V_Linebytes()*. That will determine how many bytes per line are required for the texture in video memory. That final linebytes value multiplied by the texture height is how much memory that texture needs to take in your allocated heap.

Once you've determined where that texture can go in your heap, you can then load the texture asynchronously (without locking the Vérité) with a call to *VL_MemWriteRect()*.

Use *VL_Lookup()* on slower CPUs

If your application uses 8 bit texture maps, it will have to convert them to 16 bit (A)RGB for use on the Vérité. On fast CPUs, this table lookup process is quite fast. If the CPU is bogged down with other tasks or is slow enough that it make sense to have the Vérité do some extra work, the *VL_Lookup()* function may be better.

Use triangle fans and strips when possible

In general, using triangle fans and strips instead of just triangles results in better performance on our chip. Unlike other chips that end up breaking triangle fans and strips into triangles before rendering, the Vérité chip actually understands those formats. By using triangle fans and strips less information is passed to into the command buffer and thorough the bus and less work is required by the chip itself since calculations from the previous triangle are used for the next one.

Avoid surface locks

Surface locks require that all DMA transfer and rendering operations be complete. In other words, when you lock a surface, you stall the CPU until the Vérité completes all issued command buffers (command buffers with commands in them that haven't been issued will not be automatically issued when you lock). While all right for occasional operations like screenshots, this is not something you want to do every frame, as it will slow your application's framerate. If you must lock do as much work as possible beforehand.

Change textures in system memory and redownload rather than modifying video memory directly

Reading and writing from the Vérité board memory requires a surface lock to be performed (see above for the reasons not to do that…) A better idea is to modify the texture in system memory and re-download it to the same spot in video memory.

Choose the best vertex type (especially for the V1000 series)

The right vertex type can improve performance, sometimes dramatically. On the V1000, this is especially true, as the vertex type will impact triangle setup, scanline setup, and pixel drawing. On the V2000, only triangle setup is affected, so overall performance will probably be degraded only if you are drawing small triangles (since setup is a major part of small triangles, but a small part of large triangles).

A simple version of how to calculate rendering times:

Chip	Triangle Setup	Scanline Setup	Pixel Drawing
V1000	~20 cycles / attribute	~2 cycles / attribute	1 cycle / 2 attributes
V2000	~10 cycles / attribute	1 cycle	1 cycle

Note that texture minification, Z buffering, and alpha blending will increase memory bandwidth requirements and slow performance on both chips. The pixel rate numbers above are at peak rates.

Check attribute value consistency and remove them from vertex type

One optimization to try is to check for attribute consistency before choosing a vertex type and sending the primitive to the Vérité. Check to see if the value for a given attribute is the same for all of the primitives vertices. If the values are consistent, you can use a *VL_Set* call for that attribute and remove it from the vertex type. For example, if you are using vertex attribute fog, after determining that all of a primitive's fog values are the same, you would call:

```
VL_SetF(&cmdbuffer, fog_value);
```

and choose a vertex type without F in it. On the V1000 this saves you setup time and 0-1 cycles per pixel (i.e., one cycle if including F gave you an odd number of vertex attributes and zero cycles if it gave you an even number of vertex attributes). On the V2000, you save setup time.

One possible further step. If your application generally does colored vertex lighting (which would require RGB or K to be in the vertex type), you can check to see if the R, G, and B values are equal to each other. If they are, you can use the attribute I (intensity) instead. On the V1000 this will save you setup time and 1 cycle per pixel. On the V2000, you will save setup time.

Avoid blending and Z buffering unless absolutely necessary

Z buffering and alpha blending increase memory bandwidth requirements and slow performance on the Vérité. Therefore, enable alpha blending and Z buffering only when necessary.

Note that leaving blending on and setting alpha to 1.0 (255) is not the same as disabling blending. Disabling alpha blending must be done by calling

VL_SetBlendEnable() with *V_BLEND_DISABLE*.

If you are not Z buffering at all, disable both the Z compare and Z write with:

```
VL_SetZBufMode(&cmdbuffer, V_ZBUFMODE_ALWAYS);
VL_SetZBufWrMode(&cmdbuffer, V_ZBUFWRMODE_DISABLE);
```

If you can, disable one or the other. For example, if you are rendering a light map on top of a textured polygon you have just drawn, you do not need Z buffer writing to be on (the textured polygon took care of updating the Z buffer). So disable the Z write. Another example: If you are drawing large polygons in the background (a sky, for example), and you know you're drawing it first, you do not need the Z buffer. Clear the Z buffer, disable Z buffering, and draw the background polygons with the Z buffer write enabled.

Z Buffer tricks

Clearing the Z buffer is faster than drawing Z buffered polygons. So if you are drawing distant polygons that contain a constant Z value (like a sky background, for example), clear the Z buffer with *VL_FillBuffer()*, then draw the polygons without Z buffering.

If you have sorted the world polygons, but require the Z buffer for drawing more complex objects within the world, draw the world polygons with the Z write enabled, but Z compare disabled. Then enable the Z compare when you draw the objects in the world.

If your application is fully Z buffered, it may be worthwhile to do a coarse, object level sort and draw the scene front to back. This will bias the Z compare towards failure and reduce memory bandwidth requirements.

On the V1000, you can improve performance by interleaving the scanlines of the display and Z buffers. When you call *VL_CreateSurface() to* create the display, add *V_SURFACE_INTERLEAVED* to the buffer mask parameter. If the total size of one scanline from each buffer will fit within 2K or 4K (the Vérité uses 4K memory pages), the buffers will be interleaved. You can check to see if the buffers were successfully interleaved by checking if the *V_SURFACE_INTERLEAVED* bit is set in your v_surface structure's buffer_mask. If you have interleaved buffers, make sure to set the CRT controller to properly display the surface:

```
      V_SetDisplayLinebytes(verite,
           V_GetBufferLinebytes(display->buffer_group, 0));
```

Note that you cannot interleave buffers in a windowed display. Also note that interleaving may require extra video memory for alignment purposes.

Don't lock lots of system memory

Although memory must be locked in order to transfer to the Vérité, locking too much memory can be a tremendous drag on overall system performance, since it will interfere with Windows' Virtual Memory Manager. Lock only what you need to. Also, note that command buffers use locked memory, so allocating too much command buffer space can adversely affect performance.

Some applications store locked memory pointers with the command buffer itself. In other words, when a command requires a piece of locked memory (such as a texture download) the locked memory is allocated, the command is added to the buffer, and the pointer to locked memory is associated with that buffer (ring of locked buffers). When that buffer becomes available (*V_QueryCmdBuffer()* does not return V_CMDBUFFER_INUSE) then the locked memory can be freed. This code is typically implemented within the command buffer callback.

typically implemented within the command buffer callback.

Use 2D blits rather than polygons if that's what's necessary

The Vérité has a fast blitter with transparency capability. So for 2D overlay graphics (a cockpit, for example), use blit operations rather than textured polygons.

The *VL_MemWriteSprite()* function is a compressed, host-to-video memory blit. This can be very useful for cockpit overlays, or any 2D blit operation that has a lot of transparent space in it. It is very fast, and you keep the compressed data in host memory, freeing up precious texture space.

Refresh rate

Once you get your application running quickly, changes in performance will be harder to measure because of the framerate interacting with the screen refresh rate. If your application is capable of rendering at 24 frames per second and is running the screen refresh at 60Hz, you will only get 20fps (1 of every three screen refreshes). If you then increase your engine's speed to be able to render 29fps, you will still get only get 20fps on screen (since you haven't made it to 1 or every two screen refreshes).

When doing optimizations, therefore, it is useful to remove the call to *VL_WaitForDisplaySwitch()* from your full screen page flipping routine allowing your application to avoid synching with the monitor refresh. While tearing may occur, you will be able to measure the real rendering capability of your engine, without interference from the screen refresh rate. This new rate will tell you how close you are to getting to the next interval.

You can also try different screen refresh rates when you call *V_SetDisplayMode()*. Sometimes, increasing the refresh rate will increase your framerate by changing the interference pattern. In other cases, however, you might lose performance, since some small memory bandwidth is lost to the CRT controller refresh. You can get the refresh rate set my the user by sending -1 to *V_SetDisplayMode()* for the refresh rate.

Try copy double buffering. The simplest way to do this is:

```
// set source state
VL_InstallTextureMap(&cmdbuffer, display);

// InstallTex uses front buffer as source, though, so set
// to the back buffer
VL_SetSrcBase(&cmdbuffer,
   V_GetBufferAddress(display->buffer_group, 1));

// destination state is probably set to display's back buffer, so
chat
// change to front buffer
VL_SetDstBase(&cmdbuffer,
 V_GetBufferAddress(display->buffer_group, 0));
VL_BitBlt(&cmdbuffer,
   0, 0,                                          // source position
   (v_u16)display->width, (v_u16)display->height,     // source size
      0, 0);                                         // destination
pos

// restore destination to back buffer
VL_SetDstBase(&cmdbuffer,
 V_GetBufferAddress(display->buffer_group, 1));
```

Note that this will produce some image tearing. It might be wise to offer it as a user setting.

If you have enough video memory, try triple buffering. You can then page flip without interacting with the screen refresh rate. Note that when triple buffering you

without interacting with the screen refresh rate. Note that when triple buffering you will want to change the order of the swap and wait functions. The wait will be for the previous swap, ensuring that you can safely render to the next buffer. In other words, when double buffering, the order is as so:

```
      render frame 0
      display frame 0   (VL_SwapDisplaySurface)
  wait for frame 0  (VL_WaitForDisplaySwitch)
```

Now you know that frame 0 is being displayed, so it is safe to operate on frame 1.

```
      render frame 1
      display frame 1   (VL_SwapDisplaySurface)
  wait for frame 1  (VL_WaitForDisplaySwitch)
```

When triple buffering however, it is:

```
      render frame 0
      wait for frame 2  (VL_WaitForDisplaySwitch)
display frame 0   (VL_SwapDisplaySurface)
```

Frame 2 has been displayed, so frame 1 is safe to render.

```
      wait for frame 0  (VL_WaitForDisplaySwitch)
display frame 1   (VL_SwapDisplaySurface)
```

Frame 0 has been displayed, so frame 2 is safe to render.

```
   wait for frame 1  (VL_WaitForDisplaySwitch)
display frame 2   (VL_SwapDisplaySurface)
```

VL_SwapDisplaySurface() does not currently support triple buffering, so your page flip function will look something like:

```
      // When you call VL_InstallDstBuffer, the destination will
first be
    // set to buffer 1 (that function assumes double buffering)
     int destbuffer = 1;

 VL_WaitForDisplaySwitch(&cmdbuffer);
      VL_SetDisplayBase(&cmdbuffer,
            V_GetBufferAddress(display->buffer_group, destbuffer));

  destbuffer = (destbuffer + 1) % 3;

 VL_SetDstBase(&cmdbuffer,
        V_GetBufferAddress(display->buffer_group, destbuffer));
```

Note that unlike the standard double-buffered system, in which the back buffer is always buffer number 1 and the front buffer is number 0, when triple buffering, the “back” buffer number is now held in the variable “destbuffer” (in this example case).

Use time between swap and wait

When double buffering, you cannot perform any RRedline drawing operations until after the *VL_WaitForDisplaySwitch()* call. You can however, use the time before vertical retrace to do non-screen related operations. For example, you could clear the Z-buffer, set any of the drawing states, or even download textures.

For arbitrary host-to-verite blits 4-byte-aligned destination addresses go way faster

If you are using *VL_MemWriteRect()* to draw 2D things to the screen, note that if the destination address is not 32-bit aligned, performance will drop. Try to ensure that all *VL_MemWriteRect()* calls have 32-bit aligned destination addresses. Note that *VL_CreateSurface()* and *V_CreateBufferGroup()* allocate buffers at 32-bit aligned addresses. Therefore *VL_MemWriteRect()* calls to these buffers' base addresses and VL_LoadBuffer, which calls *VL_MemWriteRect()*, will run at full performance.

Keep the Vérité busy

To maximize performance, try to ensure that the Vérité is always rendering. One way to get it started at the beginning of a frame is to issue a command buffer with some big operation in it immediately. For example, if you have background polygons to draw, put them in a command buffer and issue the buffer explicitly. Then go ahead and process the more complex part of your scene. This will get the Vérité working on drawing a lot of pixels, and give the CPU time to prepare more polygons and texture maps.

Note also that very large command buffers will take a long time to fill. It is therefore better, generally, to have more smaller command buffers than a few very large ones.

Avoid *VL_InstallTextureMap()* (calls ~12 other functions!)

VL_InstallTextureMap() is a very useful function that will set all necessary state for drawing with the specified texture map. The downside is that is sets all necessary state, even if certain states aren't changing from one texture map to another. Below is the source for *VL_InstallTextureMap()* including two functions required to calculate the U and V masks. You may be able to call only a subset of these functions, or create some other optimized texture map installer. A complete description of what each function does can be found in the reference guide.

```
/* convert width/height to mask; if power of 2, subtract 1. Else
  subtract 1 from next higher power of 2 */
static v_u16 get_tile_mask
    ( v_u32 wh )
{
   v_u32 i;

   if (! (wh & (wh-1))) return (v_u16)wh-1;
   for (i = 0; wh >>= 1; ++i) ;
   return (v_u16)((1<<(i+1)) - 1);
}

/* when clamping, just set mask to width/height-1 */
static v_u16 get_clamp_mask
    ( v_u32 wh )
{
   return (v_u16)(wh-1);
}

vl_error V_DLLEXPORT VL_InstallTextureMap
    ( v_cmdbuffer  *cmdbuffer,
     v_surface    *texture_surf )
{

  v_u16 umask, vmask;

  V_ASSERT(texture_surf != NULL);

  /* Get u/v mask */
  if (texture_surf->clamp & V_SURFACE_UCLAMP) {
     umask = get_clamp_mask(texture_surf->width);
     VL_SetUClamp(cmdbuffer, !0);
  } else {
```

```
    umask = get_tile_mask(texture_surf->width);
    VL_SetUClamp(cmdbuffer, 0);
  }
  if (texture_surf->clamp & V_SURFACE_VCLAMP) {
    vmask = get_clamp_mask(texture_surf->height);
    VL_SetVClamp(cmdbuffer, !0);
  } else {
    vmask = get_tile_mask(texture_surf->height);
    VL_SetVClamp(cmdbuffer, 0);
  }

  /* Set SrcBase, SrcStride, Vmask, Umask, SrcWidth, SrcHeight */
  VL_SetCurrentTexture(cmdbuffer,
       (v_u32)V_GetBufferAddress(texture_surf->buffer_group,0),
       V_GetBufferStride(texture_surf->buffer_group,0),
       vmask, umask,
       texture_surf->width<<16,
       texture_surf->height<<16);

  if (texture_surf->palette)
       VL_SetTexturePalette(cmdbuffer, texture_surf->start_index,
            texture_surf->num_entries,
       texture_surf->palette,
            texture_surf->pixel_fmt);

  VL_SetSrcFmt(cmdbuffer, texture_surf->pixel_fmt);

  VL_SetSrcColorNoPad(cmdbuffer, (texture_surf->color_pad == 0));

  VL_SetChromaKey(cmdbuffer, (texture_surf->chromakey != 0));

  /* Set the YUV2RGB bit to match the texture */
  if(texture_surf->pixel_fmt == V_PIXFMT_Y0CRY1CB)
      VL_SetYUV2RGB(cmdbuffer, V_YUV2RGB_ENABLE);
  else
      VL_SetYUV2RGB(cmdbuffer, V_YUV2RGB_DISABLE);

  VL_SetSrcBGR(cmdbuffer, (texture_surf->bgr != 0));

  /* Chromakey color */
  VL_SetChromaColor(cmdbuffer, texture_surf->chroma_color,
                    texture_surf->pixel_fmt);

  /* Chromakey Mask */
  VL_SetChromaMask(cmdbuffer, texture_surf->chroma_mask,
                   texture_surf->pixel_fmt);

  return VL_SUCCESS;
}
```

Use *VL_TriangleFill()* for flat shaded triangles

On the V1000, *VL_TriangleFill()* is the fastest way to draw flat shaded triangles, using an optimized path through the pixel engine. On the V2000, setup is performed a little faster.

Use *VL_Particles()* for dots/stars

The *VL_Particles()* primitive is the fastest way to draw solid colored dots and rectangles.

Different V1K/V2K techniques

You may want to consider different optimizations depending on which Vérité you are

You may want to consider different optimizations depending on which Verite you are using. The high pixel rate of the V2000 means that Z buffering is very useful. The V1000, however, can struggle with Z buffering, so polygon sorting might make more sense. With a Z buffer, however, you lose some texture memory, so on the V2000 you might want to explore ways to speed up your texture caching.

Determining which system you are on can be done by looking for a specific extension returned from *VL_GetExtensions()*. If that returned string contains "V1K_Verite" then you are on a V1000 series card. If it contains "V2K_Verite" then you are on a V2000 series card. It may be a good idea to just check for just one of these (like for the V1000 extension) and do something else if that string isn't found. Then you'll be able to have optimized paths for future chip generations as well and not just for these two.

Try an infinitely fast renderer to measure the application's "speed of light"

When benchmarking your application, try replacing *V_IssueCmdBufferAsync()* with *V_ResetCmdBuffer()*. This will simply ignore the command buffer, which simulates an "infinitely fast" bus and renderer. In this way, you can see how close to "perfect" asynchronous rendering you are. If your frame rate doesn't increase at all, the Vérité is probably not very busy, and you should further optimize your host-side code (or somehow give the Vérité more to do). If the frame rate only slightly increases, your engine is probably achieving very good rendering overlap. If the frame rate does increase dramatically, however, you should consider ways to optimize the work load on the Vérité or move more work towards the CPU.

Note that the speed of the CPU will significantly impact the results of this test. Try it on a few different speed processors to get a good sense of what is actually happening.

Use *VL_Rectangle()* for sprites

If sprites require only scaling (no rotation), the *VL_Rectangle()* or *VL_Square()* primitives are the fastest way to draw them. Both these primitives map a texture exactly to their extents, requiring only one vertex to describe them. In fact, all vertex attributes are ignored; only X and Y are used. Any drawing attributes that are needed must be set explicitly. So, for example, if you are Z buffering the sprite, call *VL_SetZ()* with the appropriate value before calling *VL_Rectangle()*.

Use *VL_Lookup()* for 2D stuff (Quake console/menus)

The *VL_Lookup()* function will convert 8 bit data to 16 bit data by using a texture map that contains the palette. Normally, this routine is used to expand texture maps from 8 to 16 bits as they are transferred from host to video memory. *VL_Lookup()* can draw to any destination, however, even the display. So, for example, if you have 8 bit deep fonts or other graphics and need to draw text to the screen, use *VL_Lookup()*. They will draw reasonably fast (currently about 3 cycles per pixel) and will not require any texture memory. Note that no scaling or rotation is possible with *VL_Lookup()*.

Use mipmapping

Mipmapping can help improve performance in two ways. First, less texture space is required, since only very close objects will be using the largest mipmaps. Second, the Vérité gets closest to its peak rendering rate when it is magnifying texture maps. You will therefore increase the pixel rendering rate if you are able to choose a mipmap that will be at least slightly magnified when drawn.

Collect things into locked memory and use *V_AddToDMAList()*

It is sometimes useful to collect commands into locked memory that is not part of a command buffer, and point the command buffer at it when necessary through the funcion *V_AddToDMAList()*. For example, in VHexen2 and Quake 2, when antialiasing is enabled, the *VL_AAEdge()* commands are not placed on the command buffer when the edges are calculated since they need to be run at the end of the frame. Instead they are stored in locked memory as the polygons are being drawn. When the entire frame is done and ready to be antialiased, the game simply calls *V_AddToDMAList()*. This technique can both save the extra memory copy from an edge cache into the command buffer, as well as saving cycles in the polygon loops by storing the vertices to the edge list at the same time as to the command buffer.

This technique might apply any time you have a large group of “pre-cooked” commands or data, such as state sets, *VL_MemWriteSprite()* calls, and so on.