Warp3D_Devel

COLLABORATORS

	TITLE : Warp3D_Devel	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 31, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Warp3D_Devel

1.1 Warp3D_Devel

This is the Programmer's Manual for Warp3D. It should serve as an introduction to Warp3D programming, and not as a reference for the functions (those are explained in the AutoDocs).

1.2 Warp3D_Devel/Introduction

Introduction
****

Overview
========

Warp3D is a library for direct access to 3D Hardware in such a way that it is independent of both the graphics system as well as the hardware installed (if any). Some of the concepts are similar to those found in OpenGL, however, Warp3D is much more low-level, since it does not for example provide functions for 3D rotation or projection (1). Rather, Warp3D requires that coordinates are specified in a coordinate system that is often called Camera- or Eye- Coordinate System: The X- and Y-Coordinates are screen coordinates, and the Z-Coordinate specifies the "depth" of a vertex. There are special constraints about coordinates, which will be explained later.

Warp3D is a low-level API, meaning that it should serve as a direct interface between an application and the 3D hardware. Other then OpenGL, it has no 3D functionality, but rather is a rasterizer. A rasterizer takes descriptions of primitives and renders those to a drawing canvas, in the case of Warp3D this means a BitMap.

How to use this manual
======================

Examples in this manual will be given in the C programming language. This is done for clarity, and since C is my programming language of choice (actually it's C++, but ...). Most examples will be more or less complete with error checking, but it might be left out in some places for brevity. In the real world, however, you should always check the return values, and act accordingly.

Function descriptions may appear a bit short in some places. In general, you should always refer to the AutoDocs for more in-depth discussion of peculiarities. In any case, references to function parameters will be named exactly as in the AutoDoc description for this function, and will be written in this style. For example, the third parameter of the W3D_UpdateTexImage is called teximage.

To understand Warp3D programming, you should read this manual and cross-check with the AutoDocs as much as possible when Warp3D functions are described. The AutoDocs will always have the latest, up-to-date information, and if an AutoDoc entry contradicts this guide, the AutoDocs take precedence. This is because the AutoDocs are usually updated more frequently than this guide, although we are trying hard to keep it up-to-date.

Commonly used terms
===================

When we say Warp3D, usually this means the Warp3D.library or Warp3D as an entity.

The term 'the driver' usually refers to the active driver library, for example, on a system with a CyberVision64/3D card, this means the W3D_Virge.library. This information is not usually necessary to understand, but in some cases it might be important to make this distinction, since even if Warp3D supports a certain feature, 'the driver' might not.

---------- Footnotes ----------

(1) This might change when hardware with geometry engines become available

1.3 Warp3D_Devel/Getting Started

Getting Started
****

1.4 Warp3D_Devel/Opening the library

```
Opening the library
===================

  The first step will almost always be to open the required libraries.
Besides the normal system libraries you need, you must open the Warp3D
library. The following code fragment shows how this is done

         struct Library *Warp3DBase;
         // ...

         Warp3DBase = OpenLibrary(``Warp3D.library'', 0L);
         if (!Warp3DBase) {
             fprintf(stderr, ``Error: Could not open Warp3D.library\n'');
             exit(0L);
         }

  Note for PPC/StormC: You must link with w3d.lib for PPC code.

  Make sure to always check the returned base. If everything goes OK,
then Warp3D is ready for use.  There are numerous reasons beside memory
shortage why the open could go wrong, for example, Warp3D might be
unable to initialize an appropriate driver, or there might be an
unsupported graphics system installed (1) and so on.

  After opening the library, you might want to check what kind of
driver you got. The parameterless function W3D_CheckDriver will
determine this, as is illustrated in this example code:

         ULONG flags = W3D_CheckDriver();
         if (flags & W3D_DRIVER_3DHW) printf("Hardware driver available\n");
         if (flags & W3D_DRIVER_CPU)  printf("Software driver available\n");
         if (flags == 0) {
             printf("PANIC: no driver available!!!\n");
             doPanic();
             exit(0);
         }

  ---------- Footnotes ----------

  (1) In which case you might want to write your own driver for it. If
you want to, please contact us so we can provide you with the details.
```

1.5 Warp3D_Devel/Querying Capabilities

```
Querying Capabilities
=====================

  It is possible that a system has more than one 3D graphics card
installed, and it might be useful to find out which of those matches
your requirements. Furthermore, there might be a couple of different
CPU drivers that also have different capabilities, or support different
destination formats.

  For this purpose, Warp3D provides a library call, W3D_GetDrivers.
This call returns an array of pointers to W3D_Driver structures. You
may examine these read-only stuctures to find out the destination
formats supported by this driver, and if the driver is a hardware- or
CPU-Driver. Furthermore, you can obtain the name of the driver, in case
there are alternatives that you want to present to the user.

  If you need more information about a specific driver, there is a
function W3D_QueryDriver that works exactly like the W3D_Query
function, only that in stead of a context, it accepts a W3D_Driver
structure. Otherwise, the functions work identical.

  For further details about W3D_Query, see the chapter about
Context Queries.
```

1.6 Warp3D_Devel/Opening the display

```
Opening the display
===================

  The next step is to open the display. This might depend on the
graphics system software you are using. Using the Intuition standard
functions OpenScreen and OpenWindow should work on all systems, but
might not support special features of your software.

  First, you need to find out what modes are supported. The function
W3D_GetDestFmt can be used to query this:

         ULONG format;

         format = W3D_GetDestFmt();
         if (format & W3D_FMT_CLUT)      printf("Driver supports 8 bit chunky  ↩
            modes\n");
         if (format & W3D_FMT_R5G5B5)    printf("Driver supports 15 bit RGB modes\ ↩
            n");

  Note that this function doesn't take parameters and can be used
directly after you are sure you have a driver.
```

The following code will try to open a screen suitable for a Warp3D display, but does use CyberGraphX for obtaining a ModeID:

```
        ModeID = BestCModeIDTags(
            CYBRBIDTG_Depth,            15L,
            CYBRBIDTG_NominalWidth,     640,
            CYBRBIDTG_NominalHeight,    480,
        TAG_DONE);

        if (ModeID == INVALID_ID) {
            printf("Error: No ModeID found\n");
            goto panic;
        }

        // Open Screen
        screen = OpenScreenTags(NULL,
            SA_Height,    960,
            SA_DisplayID, ModeID,
            SA_ErrorCode, &OpenErr,
            SA_ShowTitle, FALSE,
            SA_Draggable, FALSE,
        TAG_DONE);

        if (!screen) {
            printf("Unable to open screen. Reason: Error code %d\n", OpenErr);
            goto panic;
        }

        // Open window
        // While this is not strictly necessary, we use it because
        // we want to get IDCMP messages. You can also use the screen's
        // bitmap to render
        window = OpenWindowTags(NULL,
            WA_CustomScreen,    screen,
            WA_Width,           screen->Width,
            WA_Height,          screen->Height,
            WA_Left,            0,
            WA_Top,             0,
            WA_Title,           NULL,
            WA_CloseGadget,     FALSE,
            WA_Backdrop,        TRUE,
            WA_Borderless,      TRUE,
            WA_IDCMP,           IDCMP_CLOSEWINDOW|IDCMP_VANILLAKEY|IDCMP_RAWKEY| ↩
                IDCMP_MOUSEBUTTONS|IDCMP_MOUSEMOVE|IDCMP_DELTAMOVE,
            WA_Flags,           WFLG_REPORTMOUSE|WFLG_RMBTRAP,
        TAG_DONE);

        if (!window) {
            printf("Unable to open window.\n");
            goto panic;
        }
```

Note that creating a suitable display might also be in the form of an AllocBitMap call, but you'll have to make sure that this bitmap is allocated in card memory. If you use an off-screen bitmap, or one that is attached to a window, you might use Warp3D in window mode, for

example by blit-copying the drawing bitmap to a window. Also note that there is no way of ensuring that clip regions other than those defined in the context's scissor region are correctly taken into account, so a window that is in front of your drawing window might get overdrawn.

Starting with Version 2, Warp3D provides a screen mode requester. This requester can be instructed to filter screen modes according to one of three rules, specified by the following tag items:

W3D_SMR_DRIVER
: The argument to this tag item is a pointer to a W3D_Driver structure, as provided by W3D_GetDrivers. If this tag is present, only screenmodes that are supported by this driver are shown in the requester. The screen modes may be filtered further (see below), but the modes are guaranteed to be supported by this driver.

W3D_SMR_DESTFMT
: The argument to this tag item is a mask made up from one or more of the W3D_FMT_#? bits. Each screen mode is compared against this mask before it makes its way into the mode requester. You may either use this to filter directly after the destination fomrat, or use it to further narrow down the selection of modes in conjunction with the W3D_SMR_DRIVER tag item.

W3D_SMR_TYPE
: The argument to this tag item is one of the constants W3D_DRIVER_CPU or W3D_DRIVER_3DHW, exclusively. Before a screen mode is presented in the mode requester, it is checked against all of the screen modes supported by the CPU drivers (if the argument is W3D_DRIVER_CPU) or by the hardware drivers (if W3D_DRIVER_3DHW was specified). This tag can be used in conjunction with W3D_SMR_DESTFMT to further narrow the selection of modes.

Further filtering may be done by using the ASLSM_MIN#? tags in conjunction with W3D_SMR_SIZEFILTER. This tag is a boolean tag that specifies if you want to filter screen modes according to certain size constraints. See the WarpTest.c for more details.

To allow for futher selection of a suitable driver, for example, when you want to open up on a public screen, you can use the W3D_TextMode function. This function accepts one argument, the ModeID of the screen, and returns a suitable W3D_Driver, either a HW driver if one is present, or a CPU driver (if one is available), or NULL if nothing is available.

1.7 Warp3D_Devel/Creating a Context

Creating a Context
==========

We are now ready to create a W3D_Context. But first, let us introduce the concept of a context.

A W3D_Context is a structure that stores information about the current rendering state, display state, and also driver-internal information. It serves as some kind of "anchor" or "handle" for Warp3D. The structure itself is off-limits, you should neither read or write it directly, but rather use the functions provided by the library.

A context is created with the W3D_CreateContext function. This function is fully described in the AutoDoc, we'll just look at an example call here to illustrate how the context is created for our full-screen example above (See Opening the display).

```
W3D_Context *context;
ULONG CError;

context = W3D_CreateContextTags(&CError,
    W3D_CC_BITMAP,      bm,
    W3D_CC_MODEID,      ModeID,
    W3D_CC_YOFFSET,     height,
    W3D_CC_DRIVERTYPE,  W3D_DRIVER_BEST,
TAG_DONE);
```

Note that there is usually no way for the programmer to influence driver selection. The User usually has the possibility to indicate a "preferred" CPU driver, but the HW driver is always selected automatically (if W3D_DRIVER_BEST is set).

The W3D_CC_YOFFSET tag item is used for double buffering the CyberGraphX style. This specifies the top scanline used for rendering. All coordinates will be relative to this line, meaning the the offset is automatically added to the Y coordinates. When you switch display with ScrollVPort, you will need to set a new drawing area with W3D_SetDrawRegion, you can change this offset.

Please note that you should start with an Y-Offset of the second (invisible) page. This way, rendering will be directed to the invisible page, and switching the display will reveal the stuff you just drawn.

The following table summarizes the tag items for W3D_CreateContext:

W3D_CC_BITMAP
: Specifies the bitmap where Warp3D should render into. This bitmap must either be allocated on the graphics card, or must be the bitmap of a screen or window.

W3D_CC_MODEID
: Specifies the ModeID of the display you want to use. This tag may be ommited if you open a window on the current public screen, otherwise it is mandatory. This tag is used to select the correct driver for the display.

W3D_CC_YOFFSET
: Used for 'fake' double buffering. Specifies the YOffset to add to all coordinates for drawing operations. Should generally only used if you use ScrollVPort double buffering.

W3D_CC_DRIVERTYPE

Specifies what type of driver should be active for this context. Must be one of three pre-defined symbols: W3D_DRIVER_BEST tries to obtain the best possible driver. W3D_DRIVER_3DHW makes the call fail if no suitable hardware driver was found. W3D_DRIVER_CPU will try to use a CPU driver.

W3D_CC_W3DBM
For CPU drivers, a simple bitmap might not be enough. A CPU driver might need a fastram buffer. This tagitem can be used to pass a W3D_Bitmap.

W3D_CC_INDIRECT
Sets the W3D_INDIRECT state bit on the created context. This means that all drawing operations will be queued. See Indirect Rendering for further details.

W3D_CC_GLOBALTEXENV
If this tagitem's value is W3D_TRUE, all texture environment changes will affect all textures, not just a single texture.

W3D_CC_DOUBLEHEIGHT
If this tagitem's value is W3D_TRUE, the bitmap is considered to be only half visible. This is used in conjunction with W3D_CC_YOFFSET to fake double buffering on older CyberGraphX systems.

W3D_CC_FAST
By setting this to W3D_TRUE, your are allowing the driver to alter the structures you pass it. This means that drawing will be slightly faster, because the structures need not be copied. This can speed things up when drawing many small triangles.

Caution: Do not use the tagitems W3D_CC_YOFFSET and W3D_CC_DOUBLEHEIGHT for any other purpose. They might not work on certain graphics systems or drivers, and might even become deprecated in the future, so this don't try to outsmart the documentation.

1.8 Warp3D_Devel/Context Queries

Context Queries
==========

Once the context is created, you should go and find out what the chip really can do in the environment you have provided it with. For this purpose, the W3D_Query function also accepts the context structure as an argument.

The template for calling the W3D_Query function is
```
res = W3D_Query(context, query, destfmt);
```

context is the pointer to the W3D_Context structure you created, and query is the item you want to query. The destfmt parameter is ignored.

```
  The result is one of the following three values:
W3D_FULLY_SUPPORTED
     This feature is fully supported by the selected driver.

W3D_PARTIALLY_SUPPORTED
     This feature in not fully supported, but may somehow still work.
     There might be graphics glitches, or the result might not look
     like you intended it to look. If possible, you should give users
     the possibility to bypass this feature if it can be simulated.

W3D_NOT_SUPPORTED
     This feature is not supported, and must be emulated in software.

  For a description of the supported items, refer to the AutoDocs, and
to the include file Warp3D/Warp3D.h.
```

1.9 Warp3D_Devel/Textures

```
Textures
********


  What is it                  What exactly is a texture
  Texture Infos               How to get Information on Textures
  Creating Textures           How to create and destroy textures
  Texture Images              How texture images are represented
  MIP-Mapping                 How MIP-Maps fit into this
  Using Textures              How textures are used
```

1.10 Warp3D_Devel/What is it

```
What is it
==========

  A Texture is a square or rectangular image map. That's about it. A
texture can be used for Texture Mapping. Texture Mapping is the process
of distorting the texture so that it looks as if it has been
perspectively draw onto a triangle. Most 3D hardware can do texture
mapping by itself, and that's what makes them so fast.

  A texture in Warp3D is a combination of a structure, called
W3D_Texture, and a few memory blocks.  Those memory blocks hold the
texture data, the MIP-Maps (See MIP-Mapping) and the converted texture
data(1).

  Textures can have several attributes associated with them. They can
be either RESIDENT, in which case they are in the texture memory of the
graphics board. They can be DIRTY, in which case the program has
updated the imagery of the texture, resulting in the need to convert or
upload them again.  More on this is covered in the next section.
```

---------- Footnotes ----------

(1) This data is generated by the 3D Hardware driver to suit the needs of the hardware.

1.11 Warp3D_Devel/Texture Infos

Texture Infos
==========

Different chips support different texture formats. Because of this, your textures might be in a format that the driver cannot use directly. If this is the case, your textures are automatically converted to a suitable format, without modifying the texture data you passed in. This step is handled completely transparent, without you ever noticing it.

However, this might not be desirable. For example, it might be necessary to convert your true-color textures to a chunky format because the chip can only use chunky textures. In this case, a very complicated process of finding closest matching colors must be performed, resulting in poor overall performance.

Warp3D offers a function that can determine if a texture format is directly supported by a given hardware driver. This function is called W3D_GetTexFmtInfo, and is called with one ULONG parameter, the format of the texture you want to verify.

The return value of this function is a bit mask. Currently, the following values are supported:

W3D_TEXFMT_SUPPORTED
: The specified format is supported by the driver, although it may need to be converted to be usable by the hardware.

W3D_TEXFMT_UNSUPPORTED
: The format is not supported by the driver, and no emulation or format conversion is defined for it.

W3D_TEXFMT_FAST
: The given format is directly supported by the hardware, without any conversion step

W3D_TEXFMT_CLUTFAST
: The given format is directly supported by the hardware in Chunky modes only.

W3D_TEXFMT_ARGBFAST
: The given format is directly supported by the hardware in direct color modes only.

Consider this example: You are using Chunky textures, but want to use a True-Color 15 bit display. To find out if your active driver supports this directly, you would use something like this

```
        ULONG info;
        // ...
        info = W3D_GetTexFmtInfo(W3D_CLUT);

        if ((info & W3D_TEXFMT_ARGBFAST) || (info & W3D_TEXFMT_FAST)) {
            // ... simply load the textures, it's ok to do so
        } else {
            // consider converting your textures to ARGB format once at the start
        }
```

If you found out that your textures aren't directly supported, you might want to convert them to a suitable format yourself, although this step is not needed due to the on-the-fly conversion of texture formats done by Warp3D. To find a suitable format, you can also use the W3D_GetTexFmtInfo function.

1.12 Warp3D_Devel/Creating Textures

```
Creating Textures
=================
```

Textures must be created with the W3D_AllocTexObj function. The calling template for this function looks like this:

```
    texture = W3D_AllocTexObj(context, error, ATOTags);
```

The context argument is a pointer to a context structure created with W3D_CreateContext (See Creating a Context). error is an ULONG * where the function deposits an error code in case of failure. Finally, ATOTags is a taglist of one or more tagitems defined in Warp3D/Warp3D.h. Currently, the following tag items are supported:

```
W3D_ATO_IMAGE
     (required) A pointer to the image data. See Texture Images for a
     description about how the texture data should be organized

W3D_ATO_FORMAT
     (required) A constant indicating the source format of the texture.
     See Texture Images for a description of these formats

W3D_ATO_WIDTH
     (required) The border width of the texture map. This must be 2^n
     with an integral n.

W3D_ATO_HEIGHT
     (required) The border height of the texture map. Same constraints
     as with the W3D_ATO_WIDTH.

W3D_ATO_MIPMAP
     If this tag item is specified, the texture map can be used for
     mipmapping. You can either provide your own mipmaps, or let Warp3D
```

```
     create some or all of them for you. For complete explanation of
     this tag item, See MIP-Mapping

W3D_ATO_MIPMAPPTRS
     This tag item must be specified if W3D_ATO_MIPMAP was used. The
     data of this tag item is a null-terminated array of pointers to
     the mipmaps you want to provide. This is described in more detail
     in MIP-Mapping

W3D_ATO_PALETTE
     (required for chunky textures only) This defines the color palette
     used for a texture. This is a pointer to an ULONG array defining
     256 entries of ARGB values, packed into one ULONG, with the bits
     31-24 defining Alpha, 23-16 defining red, 15-8 defining green and
     7-0 defining blue.
```

If the return value of this function is NULL, the ULONG pointed to by error is filled with an error value. You should always take appropriate action, and not ignore the value.

After you are done with the texture, it must be freed with W3D_FreeTexObj. This frees up all associated storage, but not the texture data pointer or mipmap pointers that you passed in. It is your responsibility to free this storage. See also Texture Images for further details.

1.13 Warp3D_Devel/Texture Images

```
Texture Images
==============
```

Textures are stored in one of six possible formats. The format of a texture specifies how a single pixel is represented. Textures are always stored one scanline at a time, with all scanlines successively stored in memory. Thus, a chunky texture of 128 by 128 pixels will occupy 16384 bytes of memory (if no mipmaps are used).

```
Supported Formats
-----------------
```

The following formats are supported by Warp3D:

```
Format         Pixel Size  Description
W3D_CHUNKY1 Byte      Palettized Data. Each unit servers as an
                           index into a color lookup table. (This
                           lookup table must be provided on
                           W3D_AllocTexObj with the W3D_ATO_PALETTE
                           tag item). The palette given must match
                           the screen palette, otherwise the result
                           is undefined.
W3D_A1R5G5B52 Bytes     True color data, one word per pixel. The
                           layout corresponds to a 15 bit screen
                           mode, i.e. there's 5 bits for each color,
                           but preceeded by a one bit alpha. If this
```

```
                           alpha is one, the pixel is fully opaque.
                           If the alpha is zero, the pixel is
                           invisible/fully transparent.
W3D_R5G6B52 Bytes     True color data, one word per pixel. This
                           layout corresponds to a 16 bit screen
                           mode, with each 5 bits for red and blue,
                           and 6 bits for green (The additional bit
                           is used for green rather than another
                           color because the human eye is more
                           sensitive to green than to any other
                           primary color). This format has no alpha
                           channel.
W3D_R8G8B83 Bytes     True color data, 3 bytes per pixel. Each
                           byte corresponds to one primary color,
                           with no alpha channel information. This
                           format is problematic because of the
                           uneven addresses of pixels, but can be
                           useful for true color textures since it
                           has no alpha channel, which might not
                           always be needed.
W3D_A4R4G4B42 Bytes     True color data, one word per pixel. This
                           format is the only "narrow" true color
                           format that supports more than just
                           on/off alpha channels. Uses Four bits per
                           color and alpha channel.
W3D_A8R8G8B84 Bytes     True color data, one longword per pixel.
                           This format uses 8 bit for each color and
                           alpha channel.
W3D_A81 Byte      Pure alpha data, one byte per pixel. This
                           format uses 8 bits of alpha information.
W3D_L81 Byte      8 bit luminance data. This format uses is
                           similar to R8G8B8 with each component set
                           to L, meaning that this texture specifies
                           a gray level map.
W3D_L8A82 Bytes     8 bit luminance and 8 bit alpha. This
                           format is a combination of W3D_L8 and
                           W3D_A8
W3D_I81 Byte      8 bit Intensity. This format uses 8 bit
                           and is similar to A8R8G8B8 with A=R=G=B=I
```

The format you choose for your textures is a matter of choice, need and hardware limitations. Some hardware might not be able to use True color textures on an 8 bit screen, so if you want to use and 8 bit screen, it might be more advisable to use the W3D_CHUNKY format instead and convert your true color data to 8 bit using a dither algorithm that might be time consuming (once) but may yield better quality than Warp3D's internal texture conversion algorithms, which are mainly tuned for speed. What format you finally choose is up to you, and it is very hard to give a general suggestion.

Generally, you should only use W3D_CHUNKY textures on an 8 bit screen, and true color texture formats on highcolor or truecolor screens.

Color Channels

As with the standard RGB model, the textures Warp3D uses do support all three color channels, including a forth, so called Alpha channel. The Alpha Channel is not really a color, but rather the "transparency" information associated with a pixel. An Alpha value of zero means the pixel is totally transparent, while full alpha means the pixel is completely opaque.

The real outcome of the pixel depends on the alpha blending mode (see the chapter about Alpha Blending for more detailed information about alpha blending modes). Alpha blending can be used for spectacular effects that are very hard to do in software.

Texture Memory and Images

Normally, when you allocate a texture, your source data is not copied. Rather, the data is used as specified, but might be internally converted to a more suitable format by the driver. This means that as long as your data pointer is associated with a texture, you may not freely modify the texture image without calling W3D_UpdateTexImage. Calling W3D_UpdateTexImage tells the driver that the image has changed.

To update only part of an image, use W3D_UpdateTexImage. See the AutoDoc for more information.

1.14 Warp3D_Devel/MIP-Mapping

MIP-Mapping
===========

Introduction

A common aliasing problem is loss of detail when textures move far away from the observer. For example, a skeleton sprite in a role playing game might look even more skinny when it is away from the player. A grate might not look like a grate from a distance because the bars have disappeared. Texture mapping a large texture onto a small polygon results in loss of pixels, and hence loss of (possibly important) details.

MIP-Mapping is a way to compensate this. Technically, a MIP-Map is a sequence of arrays of decreasing size of texture maps. Each new map is half as large as the previous one; for example, a valid mipmap sequence would be 128x128, 64x64, 32x32, 16x16, 8x8, 4x4, 2x2 and 1x1. In the case of non-square (read: rectangular) maps, this works a bit different: Each side is halved until it reaches 1, in which case it stays one. Such, a 32x16 texture would be MIP-Mapped as 32x16, 16x8, 8x4, 4x2, 2x1, 1x1.

Specifying MIP-Maps

MIP-Maps might either be programmer-supplied, or generated on the

fly, or a mixture of this. The Programmer might specify any or all of the MIP-Maps.

In any case, the mechanism for this is always identical. The tag items W3D_ATO_MIPMAP and W3D_ATO_MIPMAPPTRS are used for this while allocating the texture with W3D_AllocTexObj. The first tag item W3D_ATO_MIPMAP's data is a bit mask indicating which MIP-Maps are to be generated by Warp3D. A zero bit indicates this map is user supplied, while a 1 bit indicates that this MIP map should be generated. Regardless of this tag item's value, its presence always implies that this texture is a mip map. Note also that you may provide only part of the MIP maps, and that any holes in the sequence are filled in by Warp3D. Also note that automatic MIP-Map generation decreases performance, since those maps must be generated and filtered.

The second tag item involved is W3D_ATO_MIPMAPPTRS, which must be present if W3D_ATO_MIPMAP was specified. It's data is a pointer to a NULL-terminated array with pointer to the MIP-Map data, in the same format as the texture data.

Automatic MIP-Map creation filters the images when scaling down. The filter process is only supported on true-color maps. On 8 bit, the maps are simply scaled down to half the size. This means that MIP-Mapping essentially has no effect if automatic mipmap creation is used. Of course, using pre-defined MIP-Maps still gives good results, since an artist can decide which details should be carried over to the next level of detail.

The following example illustrates the use of MIP-Maps. In this case, we consider a program that wants to create a 64x64 MIP-Map and provide two additional maps, the 16x16 and 8x8 maps. We assume that there is a function void *LoadImageMap(int size, char *name) that loads an image file.

```
        void *Maps[3];
        void *MainMap;
        W3D_Texture *texture;
        ULONG error;

        extern W3D_Context *context;

        MainMap = LoadImageMap(64, "main.png");
        // Check for errors in the real world!

        Maps[0] = LoadImageMap(16, "main_m1.png");
        Maps[1] = LoadImageMap(8,  "main_m2.png");
        Maps[2] = NULL;

        texture = W3D_AllocTexObj(context, &error,
            W3D_ATO_IMAGE,      MainMap,
            W3D_ATO_FORMAT,     W3D_A1R5G5B5,
            W3D_ATO_WIDTH,      128,
            W3D_ATO_HEIGHT,     128,
            W3D_ATO_MIPMAP,     0x0049,     // == 00111001b
            W3D_ATO_MIPMAPPTRS, Maps,
        TAG_DONE);
```

```
        // --- Proceed
```

1.15 Warp3D_Devel/Using Textures

```
Using Textures
==============

Textures and Video Ram
----------------------

  In order to be used, it is not sufficient that the texture object is
allocated. Rather, it must be on the graphics card's texture memory to
be used. There are two possible ways to ensure this. You may either use
the the W3D_AUTOTEXMANAGEMENT state (See Context States, or you may
manually upload the textures. The automatic texture management is the
easiest to use, since this will take care of the details.

  If automatic texture management is on, Warp3D will automatically
upload textures when they are needed and are not resident in video
memory. Otherwise, you must upload them manually, using the
W3D_UploadTexture call:


        success = W3D_UploadTexture(context,texture);

  context is the drawing context you are using. texture is the texture
object to be uploaded.

     Caution: You may never use textures allocated under one context in
     a different context. This might lead to unpredictable behavior,
     including crashes

  The reverse process to uploading is releasing. A (single) texture can
be released with a call to the W3D_ReleaseTexture call. The call has
exactly the same parameters as the W3D_UploadTexture call, only the
reverse effect.

  To release multiple textures, you may call the W3D_FlushTextures
function with the context as a parameter. This will release all
textures from the video memory

The Fate of a Texture
---------------------

  When you're done with a texture, it must be deallocated. This is done
with a call to the W3D_FreeTexObj function. The texture pointer becomes
invalid after you freed the texture object. This will also free any
video memory associated with the texture, and deallocate all memory
that Warp3D allocated, including MIP-Maps created automatically.
However, all memory allocated by the programmer, including the main
texture image and the provided MIP-Maps, must be freed all by yourself.

Modifying Texture Images
------------------------
```

There might be situations where you would like to modify the image of a texture, or you might want to re-use a texture handle instead of disposing it and creating a new one if the overall parameters (size and format) match. The function W3D_UpdateTexImage must be used in this case. The teximage parameter might be either a pointer to a new image that should replace the old one, or NULL. In the latter case, Warp3D assumes that you have modified the original texture image and just want to inform the driver about this fact.

> Caution: You must call W3D_UpdateTexImage when you modified your texture. Certain drivers might not need to convert your source data, and CPU drivers might directly use your memory area and directly display the correct texture without this call. This does not mean, however, that this will be the case with every other driver.

In a similar line, you might want to update only parts of a texture. There are two possiblities for this, and both use the function W3D_UpdateTexSubImage.

If you want to replace parts of the texture with a new image, you can give a non-NULL pointer for the teximage parameter when calling W3D_UpdateTexSubImage. This essentially copies the image pointed to into the texture, replacing what was originally there. The area to update is specified with a pointer to a W3D_Scissor region in scissor. The image size should match the scissor region, if it doesn't, the srcbpr parameter should be used to specify the bytes to skip from the first pixel of the subimage to get to the next line.

You might also want to change parts of the original image and tell Warp3D the exact area where your changes took place, so that the driver might limit conversion of the image data to the are where the actual damage was done. In this case, teximage should be set to NULL.

Texture Image Considerations

The W3D_UpdateTexImage function gives you the possibility to update the texture image, however, it might not always be what you want. For example, the game Descent uses animated textures for monitors in the mine. You might chose to use the W3D_UpdateTexImage function each time the monitor switches an image, but that might be wasteful. The reason for this is that your program might not use the complete texture memory, so that all monitor images might fit. Using repetitive calls to the W3D_UpdateTexImage function will always convert and upload the new image.

As you can see from the above, a structured approach to texture management might be needed.

1.16 Warp3D_Devel/Context States

Context States

Introduction to Context States

Like OpenGL, Warp3D uses States to inform the graphics pipeline how certain situations should be handled. Some states affect internal workings of Warp3D, for example the W3D_AUTOTEXMANAGEMENT state which make texture uploading and releasing automatic, others affect the way primitives are drawn, for example, if the W3D_PERSPECTIVE state is active, triangles are drawn with perspective-corrected texture mapping.

States are set and cleared with a call to W3D_SetState. If the newstate argument equals W3D_ENABLE, the state is set. If it equals W3D_DISABLE, the state is reset. However, not every hardware supports every state. In order to find out what is supported, use the W3D_Query function (See Context Queries). Alternatively, you should check the return value of the W3D_SetState. If it returns W3D_SUCCESS, the state was successfully set or reset, otherwise this state cannot be enabled (or disabled, depending on the operation you attempted).

At the start of a program, when a context was created, all states are set to a well-defined initial state, with the exception of states that cannot be set to the desired value(1). To find out what states are set, use the W3D_GetState function. This function returns W3D_ENABLED if the state is active, and W3D_DISABLED otherwise.

Supported States

The following summerizes the available states, as well as the default values that are set by W3D_CreateContext.

W3D_AUTOTEXMANAGEMENT
: If this state is enabled, textures are automatically uploaded to the video memory as needed, and "old" textures are released if texture memory overflows. If disabled, the drawing functions return an error message when the texture supplied is not on the card. Unless you really need full control, there’s really no reason to disable this. Enabled by default.

W3D_SYNCHRON
: If disabled, any drawing function starts its operation and immediately returns, without waiting for the operation to finish. Thus, the main CPU can work in parallel with the hardware and use the extra cycles for calculations and the triangle setup stage. If enabled, all drawing functions wait until the operation completed before returning(2). Disabled by default.

W3D_INDIRECT
: If enabled, no drawing is done; rather, the primitive drawing functions are queued until one of three things happen: A W3D_Flush is called by the program, the buffer overflows, or the W3D_INDIRECT is reset. See the section on Indirect Rendering for further details. Disabled by default.

W3D_TEXMAPPING
     If enabled, the tex field of primitives are used to texture-map
     the primitive. If disabled, the current state of W3D_GOURAUD
     determines if the primitive will be gouraud-shaded or flat-shaded.
     Enabled if the hardware supports texture mapping.

W3D_PERSPECTIVE
     If enabled, textures are drawn with perspective correction.
     Otherwise, the textures are drawn with linear mapping. The latter
     is faster, but will look worse. Disabled by default, unless the
     hardware does not support linear mapping.

W3D_GOURAUD
     If enabled, and texture mapping is disabled, triangles are drawn
     with gouraud shading. Otherwise, flat shading is used. Enabled if
     the hardware supports gouraud shading.

W3D_ZBUFFER
     If enabled, the ZBuffer is used for depth sorting/ZBuffering (See
     ZBuffering). If disabled, pixels are always drawn, regardless of
     ZBuffer value. Disabled by default, unless the hardware won't do
     without it.

W3D_ZBUFFERUPDATE
     If enabled, the Z value of incoming pixels is used to update the
     ZBuffer if the ZBuffering test succeeded (See ZBuffering). If
     disabled, does not update the ZBuffer. Enabled by default, unless
     the ZBuffer cannot be updated.

W3D_BLENDING
     If enabled, Alpha Blending is turned on, and pixels are
     alpha-blended according to the current alpha blending mode.
     Otherwise, alpha blending is not performed. Disabled by default,
     unless it can't be disabled.

W3D_FOGGING
     If enabled, fogging is turned on (See Fogging). Otherwise, no
     fogging is done. Disabled by default, unless the hardware enforces
     the use of fogging.

W3D_ANTI_POINT
W3D_ANTI_LINE
W3D_ANTI_POLYGON
W3D_ANTI_FULLSCREEN
     If any of these is enabled, the appropriate operation is
     anti-aliased. The W3D_ANTI_FULLSCREEN affects the complete screen.
     If one is disabled, then one of the other states might still be
     active. All are disabled by default, unless they can't be
     disabled.

W3D_DITHERING
     If enabled, output is dithered using the default dithering
     method(3). Otherwise, pixels are written "as-is". Disabled by
     default, unless the hardware insists on dithering(4).

W3D_LOGICOP

```
     If enabled, Logic operations are performed (See Logic Operations)
     according to the currently set logic operation. Otherwise, pixels
     are drawn normally. Disabled by default, unless it must be enabled.

W3D_STENCILBUFFER
     If enabled, stencil buffering is performed (See
     Stencil Buffering) as specified by the stencil buffer mode.
     Otherwise, pixels are always written regardless of stencil values.
     Disabled by default, unless it can't be disabled.

W3D_DOUBLEHEIGHT
     If enabled, the bitmap pointer passed in assumed to be a
     double-height bitmap for faked double buffering.  The ZBuffer or
     Stencil Buffer will be allocated with half the height of this
     bitmap. If disabled, the bitmap is considered to be completely
     used for rendering. Disabled by default, unless the
     W3D_CC_DOUBLEHEIGHT tag item was specified on context creation.
Note: More states may be supported. Check the AutoDoc for a description
of all states.

  ---------- Footnotes ----------

  (1) For example, W3D_DITHERING cannot be disabled with the ViRGE
driver

  (2) The hardware driver will always wait for completion of the last
operation before it writes any registers.

  (3) Chances are this is Ordered Dithering

  (4) The ViRGE does indeed
```

1.17 Warp3D_Devel/Drawing

```
Drawing
*******


  Starting to Draw         How to start, what to do
  Locking                  Locking and Unlocking the display
  Coordinates              X/Y/Z and U/V/W
  Triangles                The essentials on Triangles
  Lines                    Line Drawing
  Points                   Point Drawing
  Fogging                  Fog and Depth Cueing
  Logic Operations         What the LogicOp stuff means
  Stencil Buffering        Cookie Cutting
  ZBuffering               Using the Z Buffer
  Alpha Blending           Transparency and other stuff
  Light                    Lighting
```

1.18 Warp3D_Devel/Starting to Draw

```
Starting to Draw
================
```

Let's now get to the most important part of Warp3D programming: Drawing. In order for the user or player to see anything, we must bring our expensively calculated 3D objects to the screen.

In spite of the name, Warp3D does not really do any 3D calculation. The coordinates passed in must be in so-called Screen Space, that is, they must have been projected and clipped so that they lie on screen, or rather, in the currently visible buffer. Under Picasso96 or CyberGraphX 3, you can use the ChangeScreenBuffer call for double buffering.

To define the area on which you want to draw, you must call the W3D_SetDrawRegion function. The bm parameter is a pointer to the bit map you want the driver to draw into, that is, it must be compatible with Warp3D (W3D_CreateContext checks this). The yoffset parameter indicates the vertical offset of the top left edge of the screen, and is used to do the double buffering under CyberGraphX V2.

Basically, both the ScrollVPort trick and the normal Kick 3.x double buffering functions can be used for double buffering under Warp3D. However, the latter method has the advantage that if you do want to do direct rendering with the processor, you do not need to add an offset to determine the correct buffer. Otherwise, both functions work equally good, and with comparable speed.

The scissor parameter is a pointer to a filled-in W3D_Scissor structure. This defines the area of the screen that drawing should be restricted to. It is generally a good idea to set this, even if you don't really intent to draw into s screen section only, as it protects the memory outside your drawing area. An example for this can be found in the WarpTest.c source file.

When the selected driver is a CPU driver, it might be more reasonable to specify a RAM-Buffers instead of a bitmap for drawing. To that end, Warp3D specifies a W3D_Bitmap structure. This can be used to pass such a pointer in the dest field upon creation of the context, with the help of the W3D_CC_W3DBM tag item. Refer to the Warp3D.h include file for further details.

1.19 Warp3D_Devel/Locking

```
Locking
=======
```

In order to be able to draw anything, you must first lock the hardware. This is done with a single call to W3D_LockHardware. This call grants exclusive access to the hardware. Unless the W3D_INDIRECT state is enabled, this call must preceed any drawing operation. (see

See Indirect Rendering for details about indirect rendering).

  Locking ensures that the hardware is ready to draw, and also assures that the bitmap you want to draw to is residing in graphics memory. This can result in a certain temporal overhead. Although this is not too much, you should not lock the hardware for every primitive, This tactic should only be used if time is very unimportant for your application, or otherwise a per-frame locking is not possible(1). In this case, it might be a better idea to use indirect rendering, as discussed in Indirect Rendering.

  Certain constraints of the graphics subsystem also affect locking. For example, CyberGraphX might have severe difficulties when a lock is held for a certain time. However, you should generally try and lock only on a frame basis, i.e. you should lock your hardware, draw your stuff, and finally release the hardware.

  Other strategies include locking for a certain time only. For example, if you can predict the amount of work required for an operation, you might predict the time for which you will hold the lock. However, such things are hard to implement, so the best bet is to lock the entire frame.

  ---------- Footnotes ----------

  (1) This might be the case if you are porting a 3D-Engine from another platform, or use Warp3D as a rasterizer for a 3D-Graphics library

1.20 Warp3D_Devel/Coordinates

Coordinates
===========

  Coordinates are passed to Warp3D as floating point numbers of type W3D_Float. This allows for sub-pixel accuracy, i.e. it avoids "jumping" polygons, and generally looks better than using integer coordinates (of course, your 3D engine must support this). For this reason, Warp3D only runs on systems with a floating point unit.

  X and Y coordinates are screen coordinates, also called device coordinates, because they reference individual pixels on the display directly. They need not be passed in any particular order (i.e. clockwise/counterclockwise), although a certain direction is almost always imposed by the 3D engine.

  The Z coordinate is the vertex depth coordinate, and is used primarily for Z Buffering (See ZBuffering). The Z value must be normalized to a value between 0 and 1, where one is the smallest depth (i.e. 0 is directly behind the glass of the monitor).

  There has been much confusion on the meaning of the W coordinate. Traditionally, the W coordinate used to be the fourth component of a 3D vector in a homogenous coordinate system. In such a system, a point in

```
3-space is represented as v = (x y z w), and for projection, this is
transformed as v' = (x/w y/w z/w 1), but this is not what it stands for
in Warp3D.

  In Warp3D, the W coordinate is used as the inverse of the z
coordinate, i.e. w = 1/z. The reason for this is that for texture
mapping, the z value is not linear, but 1/z is(1). Note that the z
value we are talking about here is the z value in camera or eye space,
that is, before the perspective transformation took place. This is
because with some approaches, the z coordinate is also perspectively
foreshorted, which is not what we want here. The W coordinate should
generally be set to w = scale/z. The scale value can be any value, you
should define it in a way that all W values in your engine fit into the
area [0..1]. Or, in other words: if you multiply all W values in your
engine by 2, the result will be the same (unless overflow or precision
errors occur).  Finally, the W value has no meaning as an absolute
value, it only contains information when compared to the W values of
other vertices. It is finally only used for transformations between
linear and perspective space.  (Note to OpenGL implementors: the
correct value for W is the inverse of W in clip coordinate space,
eventually scaled to achieve better precision).  While it is desired to
scale the W values into the range [0..1], it is possible to pass values
outside this range, but it may result in a slight performance loss.
Often, this is necessary, for example when the world starts close to 0,
or if negative Z coordinate appear (OpenGL is very flexible here).
Note that you must not pass negative W values.  See the example source
WarpTest.c for an example on how to do this.

  The texture coordinates, U and V, are always given as pixel
coordinates. This means that if you are using a 128x128 texture map,
the upper left pixel is at (0,0), while the lower right is at
(127,127). These values are floating point, so you may even specify
subpixels.

  Light or color "coordinates" are given in a range of [0..1], with 0
being lowest, and 1 highest intensity. These values should be clamped
to the interval to avoid errors. Also, alpha values are given in the
same range, with 1 being full opaque, and 0 translucent.

  ---------- Footnotes ----------

  (1) This is quite easy to see, for example, the function f(x) = 1/x
is not linear, while g(x) = 1/f(x) is the identity, which is linear.
```

1.21 Warp3D_Devel/Triangles

```
Triangles
=========

  Triangle drawing is the most essential operation in 3D graphics.
Usually, objects are represented using the so-called Polygon Boundary
Representation, which means that the surface of the object is
represented as a series of polygons, which usually should be planar.
Since Planarity can only be guaranteed for triangles, most 3D
```

accelerators offer triangles as a drawing primitive.

Triangles are defined by three vertices at their corner. A vertex is specified using a W3D_Vertex structure. Basically, this structure defines the vertex screen coordinate, it's depth and texture space coordinate as well as the vertex color. The latter is used for gouraud shading, while the depth and texture space coordinates are used for Z-Buffering, texture mapping, and fogging.

Triangles are drawn with one of three functions. W3D_DrawTriange draws a single triangle specified by a W3D_Triangle structure. The advantage of this function is that it directly returns if W3D_SYNCHRON is not set, giving you extra time to do other stuff while the 3D chip is busy writing the triangle to the screen.

If you want to draw more than one triangle at once, it might be more advisable to use one of the other two functions. Those can actually draw polygons, although they expect the vertices of these polygons to be in sequence. The functions are called W3D_DrawTriFan and W3D_DrawTriStrip. Since they are very similar, they are discussed here together.

Both use a W3D_Triangles structure. This structure contains a pointer to the vertices that define the polygon. Texture information is extracted from the tex field in the W3D_Triangles structure. The v field points to an array of W3D_Vertex structures. These need not be specially terminated, so this can also be a slice of an array, the vertexcount field determines the number of edges your polygon has.

The difference between the functions is that W3D_DrawTriFan function draws a triangle fan, while the W3D_DrawTriStrip draws a triangle strip. For a detailed description about these, see the OpenGL Specification.

The following image shows the order in which vertices have to be specified in the W3D_Triangles structure:

(press this link to see the image)

1.22 Warp3D_Devel/Lines

Lines
=====

Lines are defined by their two endpoints. These are specified using W3D_Vertex structures, just like in the case with triangles (See Triangles). On some hardware, lines can be textures; in this case, the tex element of the W3D_Line structure is used.

If the driver supports arbitrary line widths, the linewidth element can be used for this. If you do not want to use this feature, or if the current driver does not support this, you should still set the linewidth to 1.0, so that your code also works on other hardware.

Vertices (v1 and v2) are used the same way as with triangles, and this applies to all of the entries in the W3D_Vertex structure. Most notably, the z coordinate is still used for Z-Buffering (if enabled) and the w coordinate is also used for fogging (if enabled) and texturing (if supported/enabled). Colors are also taken into account.

To draw the line, you call the W3D_DrawLine function.

1.23 Warp3D_Devel/Points

Points
======

Points only take one W3D_Vertex element. They may be textured, if the driver supports this. As with lines (See Lines), given the right driver, you may specify the size of the point in the pointsize entry. This must be set to 1.0 if you just wish pixel-sized dots, even if the driver does not support arbitrary point sizes, so that your code works on other drivers too.

Points are drawn with a call to the W3D_DrawPoint function.

1.24 Warp3D_Devel/Fogging

Fogging
=======

Fogging is the process of blending an incoming pixel (i.e. a pixel that is about to be drawn) with a certain amount of a specified color, the fog color. The amount of color blending is determined by the incoming pixel's W coordinate and the current fog function.

Fogging is controlled by the W3D_SetFogParams function call and the W3D_FOGGING state. The state is controlled with the W3D_SetState function (See Context States).

To set the fogging parameters, you'll have to fill out a W3D_Fog structure. The fog_start and fog_end elements specify where the fog starts to gather up, and where it is so think that pixels are only drawn in the fog color (specified in the fog_color field). The start and end values are in W-Coordinates, meaning that a fog_start of 1.0 means the fog starts right behind the glass of the monitor. The fog_density field is only used with non-linear fog (see below).

The fogmode parameter in the W3D_SetFogParams function call specifies the type of fogging you want to use (the fog function). W3D_FOG_LINEAR means that the fog should ramp up from fog_start to fog_end. The W3D_FOG_EXP and W3D_FOG_EXP_2 are called exponential fogging and square exponential fogging respectively (also sometimes called non-linear fog modes). The exact formula is given in the OpenGL specification, and

since Texinfo isn't that good at math formulas, I won't give it here.

1.25 Warp3D_Devel/Logic Operations

Logic Operations
==========

  This chapter has not been written yet.

1.26 Warp3D_Devel/Stencil Buffering

Stencil Buffering
==========

  Stencil buffering is covered extensively in the (guess where) OpenGL specification, so I will only briefly describe it here. It is not available in the ViRGE driver.

  Like the Z Buffer (See ZBuffering) , the Stencil Buffer is a buffer as large as the drawing area. Each incoming pixel is assigned a stencil value if stencil buffering is enabled. This value is compared against the value in the buffer, and depending on the outcome of this comparison, the pixel is either drawn or discarded. Furthermore, the stencil value of the frame buffer may be updated in a certain way, as specified in the stencil test operation. These operations include incrementing or decrementing the buffer value, setting it to zero, replace it with the incoming value, and so on.

  Stencil buffering is used (among other things) to cookie-cut shapes from an image.

1.27 Warp3D_Devel/ZBuffering

ZBuffering
==========

  ZBuffering (sometimes also called depth buffering) is a method for hidden surface removal. The ZBuffer itself is a buffer with the same size as the drawing area.

Theory of Operation

  When a pixel is about to be drawn, it's Z value (which is taken from the vertex structure's Z value interpolated along the primitive) is compared against the current ZBuffer Z value using the current compare function. If the test fails, the pixel is discarded. If it succeeds,

the pixel is drawn. After this, if ZBuffer updating is enabled, the Z value of the pixel overwrites the value in the ZBuffer only if the pixel was also drawn.

  The following table summarizes the available Z Comparison functions, along with their meaning:

W3D_Z_NEVER
     The comparison never passes, the pixel is always discarded.

W3D_Z_LESS
     The comaprison passes if the incoming Z value is less than the ZBuffer’s value

W3D_Z_GEQUAL
     The comaprison passes if the incoming Z value is greater or equal the ZBuffer’s value

W3D_Z_LEQUAL
     The comaprison passes if the incoming Z value is less than or equal to the ZBuffer’s value

W3D_Z_GREATER
     The comaprison passes if the incoming Z value is greater than the ZBuffer’s value

W3D_Z_NOTEQUAL
     The comaprison passes if the incoming Z value is not equal to the ZBuffer’s value

W3D_Z_EQUAL
     The comaprison passes if the incoming Z value is equal to the ZBuffer’s value

W3D_Z_ALWAYS
     The comaprison always passes, the pixel is always drawn.

Using the ZBuffer

  The comparison function is set using the W3D_SetZCompareMode function. The mode parameter must be a value from the above table. In this context, the W3D_Z_LESS comparison function is what usually is understood as ZBuffering, meaning that pixels closer to the screen are drawn, while those further away are discarded. On the other hand, the W3D_Z_EQUAL or W3D_Z_NOTQUAL modes can be used as a kind of ,,poor mans stencil buffering".

  In order to use the ZBuffer, it must have been allocated before. This is done with a single call to the function W3D_AllocZBuffer. This makes the ZBuffer available if the return value was W3D_SUCCESS. After you are done with it, it must be freed again with a call to W3D_FreeZBuffer.

  In addition to the above, there are other functions to read and write to the ZBuffer. These are only explained here in brief, please refer to the appropriate AutoDocs for more details.

  W3D_ClearZBuffer clears the ZBuffer with a specified value. This

value must be in the range 0 through 1.

W3D_ReadZPixel and W3D_ReadZSpan read a single pixel or a number of pixels from the ZBuffer. The result is a single or an array of floats, all in range 0 through 1.

1.28 Warp3D_Devel/Alpha Blending

Alpha Blending
==========

Alpha Blending is a process where the color of a pixel to be drawn is blended with what is already in the frame buffer at this time. The way how each pixel affects the other is defined by the blend functions on a per-texture basis.

There are two blending functions, named source and destination mode. The source function affects the alpha of the current pixel to be drawn, while the destination function affects the framebuffer pixel. For example, a destination function of W3D_ONE completely ignored the alpha value of the framebuffer pixel (assumes it to be one, i.e. opaque) and just uses the alpha from the incoming pixel.

An alpha value of 1.0 means this pixel is completely opaque, or more mathematically, this pixel's weight is 1.0. The resulting color will be a weighed average of both pixels, a linear interpolation between both pixels with the interpolation factor being the calculated alpha value.

For a deeper discussion of alpha blending, consult the OpenGL specification.

1.29 Warp3D_Devel/Light

Light
=====

Gouraud shading is an algorithm for simulating lighting of a triangle. For Gouraud shading to work, only the normals at the vertices are needed to calculate the color values. Gouraud shading has some shortcommings, especially if compared with the more complex Phong shading. Most notably, it will never catch hightlights that are not on a vertex(1). Moreover, it also never takes perspective into account.

Most modern 3D hardware support gouraud shading. Gouraud shading is used either for shading untextured triangles, or for "lighting" textured polygons.

On untextured triangles, the vertex colors are linearly interpolated over the triangle. The alternative to this is flat shading, where one color is used for the entire triangle.

On textured triangles, the vertex colors are also interpolated over the triangle, but depending on the environment mode selected for the texture, the light color is combined with the texture color.

There are currently four possible environment modes for textures. These are set with a single call to W3D_SetTexEnv. The possible mode are(2)

W3D_REPLACE
: The texture color is used for the triangle. No lighting whatsoever is done

W3D_DECAL
: The texture color is blended with the light color depending on the alpha value. This means the alpha value of the texture is used as a linear interpolation factor between texture and light color.

W3D_MODULATE
: The texture color and light color are multiplied, and the result is used as the pixel color.

W3D_BLEND
: This mode behave much like W3D_MODULATE, only that the specified envcolor is blended with the result.

---------- Footnotes ----------

(1) A relatively new algorithm named fence shading will catch those on edges too, but still not those that are completely inside the triangle

(2) The ViRGE only supports the first three modes, and the alpha for blending must be set to 1.0. Furthermore, W3D_REPLACE and W3D_MODULATE are treated equally

1.30 Warp3D_Devel/Hinting

Hinting
****

Hints are a means of controlling output quality of the Warp3D system without knowing the capability of the underlying hardware. Warp3D defines a number of categories and three different levels of quality. The function W3D_Hint is used to tell Warp3D what quality is desired for a certain category under the current context.

The following table summerizes the possible quality levels:

W3D_H_FAST
: Use the fastest possible solution for this category, even if this means to downgrade the quality of output. Using this level might result in graphics glitches sometimes, but is always the fastest possible setting. This or the next mode should be used for games.

W3D_H_AVERAGE
     Try to use a compromise between speed and quality.

W3D_H_NICE
     With this setting, everything is optimized for maximum output
     quality, regardless of output speed. This mode is most suitable
     for interactive software that does not need all the possible
     speed, like modellers or some OpenGL applications.

  Currently, Warp3D specifies the following categories for Hinting:

W3D_H_TEXMAPPING
     Affects the quality of texture mapping.

W3D_H_MIPMAPPING
     Affects the quality of mipmapping

W3D_H_BILIENARFILTER
     Affects the quality of filtering. For example, there is a faster
     bilinear filter mode available on the ViRGE graphics processor
     (due to a bug, YUV textures are treated normally, but with a
     faster filter mode). The quality of this is a bit lower.

W3D_H_MMFILTER
     Affects the quality of depth filtering

W3D_H_PERSPECTIVE
     Affects the quality of perspective mapping. For example, the ViRGE
     processor has problems with wrapping in perspective mode, but the
     driver has the ability to subdivide triangles on texture wrap
     borders.

W3D_H_BLENDING
     Affects the quality of alpha blending.

W3D_H_FOGGING
     Affects the quality of fogging. For example, some graphics chips
     cannot have discontinuities in fogging, which means that when the
     fogging border is within the triangle drawn, the linear fog
     interpolation will not draw the triangle correctly. Again, some
     drivers can subdivide the triangle to get this effect right.

W3D_H_ANTIALIASING
     Affects the quality of anti-aliasing

W3D_H_DITHERING
     Affects the quality of dithering. For example, some driver might
     be able to select ordered dithering or floyd-steinberg depending
     on this setting.

W3D_H_ZBUFFER
     Affects the accuracy of the Z buffer, i.e. bit depth.

1.31 Warp3D_Devel/Indirect Rendering

```
Indirect Rendering
******************

  If the context was created with the W3D_CC_FAST tag set to W3D_TRUE,
or the state W3D_FAST was set, Warp3D is in fast mode. This means that
structures passed to the driver are not copied and may be modified.

  In fast mode, locking the hardware is no longer necessary. All
rendering calls are internally stored. Each time the queue is full, or
certain functions are called, the queue is flushed (The same effect can
be achieved with the W3D_Flush function).

  The following table lists all functions that are queued:
   * W3D_DrawLine

   * W3D_DrawPoint

   * W3D_DrawTriangle

   * W3D_DrawTriFan

   * W3D_DrawTriStrip

   * W3D_ClearZBuffer

   * W3D_ClearStencilbuffer

   * W3D_SetCurrentColor

   * W3D_SetCurrentPen

   * W3D_Hint

  These flush the queue:
   * All Stencil buffer functions except W3D_AllocStencliBuffer, and
     the functions that are queued.

   * All ZBuffer functions, except W3D_AllocZBuffer, and the functions
     that are queued.

   * All functions that change the visual appearance of rendering, i.e
     W3D_SetFogParams

   * All functions changing textures, i.e. W3D_UpdateTexImage
```

1.32 Warp3D_Devel/Indices

```
Indices
*******
```

1.33 Warp3D_Devel/Concept Index

Concept Index
****

1.34 Warp3D_Devel/Function Index

Function Index
****

1.35 Warp3D_Devel/Type Index

Type Index
****