Warp3D V5

API Changes

Table of Contents

Part I : Multitexturing In Warp3D

1 Functional overview

Multitexturing is only available for the Vertex Array functional group. The classic single-primitive drawing operations (like `W3D_DrawTriangle` and friends) cannot use multitexturing because the `W3D_Vertex` structure lacks the required texture coordinate(s) for additional texture units.

Illustration 1:Resulting image

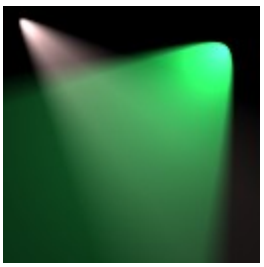


Illustration 2:Light Map

Illustration 3:Original Texture Map

For the vertex array functions, the `W3D_TexCoordPointer` already accepts an additional parameter that specifies the texture mapping unit (TMU) that this texture coordinate set belongs to. Likewise, the `W3D_BindTexture` call features the same field with the same purpose.

For multitexturing to be used, you must enable the `W3D_TEXMAPPING` *and* the `W3D_MULTITEXTURE` state; otherwise all but the first texture stage will be disabled by default. Both models imply the `W3D_GLOBALTEXENV` state, and whether it is enabled or not is ignored – enabling multitexturing will automatically assume it to be enabled.

2 Multi-Texture Mapping Models

There are two methods for multitexturing, although they look and operate similar to each other. Both use the same API function with slightly different inputs, but their use is mutually exclusive, and trying to mix them yields an undefined behavior. In general, the second (separated blend modes) form is much more general and more flexible, but not supported by all drivers (for example the Voodoo Avenger driver only supports the simplified combined model, while the Radeon and Voodoo Napalm support all models). The simplified is supported by all drivers which support multitexturing (See the section on query items, below).

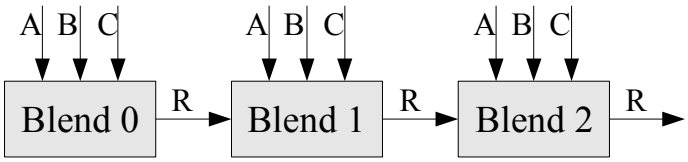
2.1 Combined Multitexture Model

The combined model works similar to the classic texture environment. In the classic environment, a specific texture environment mode specified how texture color is applied to the framebuffer. The most simple form is `W3D_REPLACE`, which simply ignores the incoming (iterated, either gouraud or flat shaded) color from the primitive and replaces it with the texture's color at the iterated u and v coordinates. This easily extends to multiple texture units. The “incoming” component in the first texture stage (TMU 0) is the “fragment” or “iterated” color (which can be, as mentioned above, either be vertex coloring based on gouraud shading, or a flat color set with `W3D_SetCurrentColor`). In all other tmu's, the incoming color is simply the output color of the previous stage, and the environment mode is treated exactly the same as on the first stage.

Setting the environment mode for each texture stage is done with the `W3D_SetTextureBlend` call, described below, with the input tags limited to `W3D_BLEND_STAGE` and `W3D_ENV_MODE`.

2.2 Separate Multitexturing Model

The multitexture pipeline consists of a variable (hardware-specific) number of blend units that are cascaded together to form a single pipeline:



Each unit computes a specific function with up to three inputs and exactly one output. The output of the last blend stage is written to the frame buffer. There is a separate identical chain of these blend units for the Alpha component.

The three inputs of each unit may be programmed separately. For example, to achieve the same effect as using the "MODULATE" texture environment function of the "classic" texture environment, you would set input A of unit 0 to "diffuse color" and unit B to "texture color". The blend unit 0 would then be programmed to perform the function "A*B". Unit 1 and all following units are disabled, so the output of "diffuse color" times "texture color" (the definition of MODULATE) is written to the frame buffer.

3 Warp3D API

The whole of the texture pipeline is controlled via a single Warp3D function:

```
uint32 W3D_SetTextureBlend(W3D_Context *, struct TagList *);
uint32 W3D_SetTextureBlendTags(W3D_Context *,...);
```

The function accepts a taglist as input and returns an error code in case of an unsupported or invalid input, or W3D_SUCCESS if successful.

The following table lists all allowed tag items:

Tag Item	*Tag Data*
W3D_BLEND_STAGE	An uint32 specifying the texture stage to affect. This must be the first tag item in any call to this function (except as described below). Any subsequent tag item affects the specified stage until a new stage is selected with this tag. See below for examples.
W3D_COLOR_ARG_A W3D_COLOR_ARG_B W3D_COLOR_ARG_C	Specify the color argument for the selected color blend stage. The argument is one of the W3D_ARG_#? constants as outlined below.
W3D_ALPHA_ARG_A W3D_ALPHA_ARG_B W3D_ALPHA_ARG_C	Specify the alpha argument for the selected alpha blend stage (which has the same index as the currently selected color blend stage). The argument is one of the W3D_ARG_#? constants specified below.
W3D_COLOR_COMBINE W3D_ALPHA_COMBINE	Specify the color (alpha) combination function for the selected texture unit (i.e. the operation that this blend stage is going to perform). The argument is one of the W3D_COMBINE_#? constants

Tag Item	*Tag Data*
	specified below.
W3D_BLEND_FACTOR	A per-stage constant blend factor, specified as a pointer to a W3D_Color structure.
W3D_COLOR_SCALE W3D_ALPHA_SCALE	An uint32 specifying a constant factor by which the result should be multiplied. The only accepted inputs are 1, 2 or 4.
W3D_ENV_MODE	The environment mode for this specific stage when using combined environment model. This tag is mutually exclusive with all others but W3D_BLEND_STAGE. See below for the possible values.

The following table lists all possible arguments for the W3D_COLOR_ARG_#? and W3D_ALPHA_ARG_#? Tags:

Constant	*Meaning*
W3D_ARG_PREVIOUS_COLOR W3D_ARG_PREVIOUS_ALPHA W3D_ARG_PREVIOUS	Use the output of the previous blend stage (or the diffuse color/alpha for stage 0). The third form, when used with a color combine, is equivalent to W3D_ARG_PREVIOUS_COLOR. Otherwise it is equivalent to W3D_ARG_PREVIOUS_ALPHA.
W3D_ARG_DIFFUSE_COLOR W3D_ARG_DIFFUSE_ALPHA W3D_ARG_DIFFUSE	Use the iterated diffuse color (alpha)
W3D_ARG_TEXTURE_COLOR W3D_ARG_TEXTURE_ALPHA W3D_ARG_TEXTURE	Use the color (alpha) from the texture bound to the matching texture unit, i.e. for stage 0 use texture 0 color/alpha
W3D_ARG_TEXTURE*n*_COLOR W3D_ARG_TEXTURE*n*_ALPHA W3D_ARG_TEXTURE*n*	Use the color (alpha) from the texture unit bound to unit *n*. Note that this input is not supported by all drivers. You need to query the W3D_Q_ENV_CROSSBAR query flag to ascertain that the mode is supported.
W3D_ARG_SPECULAR_COLOR	Use the specular color. This argument constant may not be used on Alpha stages and does not have an alpha component.
W3D_ARG_FACTOR	Use the per-stage constant blend factor as specified with the W3D_BLEND_FACTOR tag.
W3D_ARG_COMPLEMENT	This is a bit that may be logically or'ed with any of the above constants. The result is that the argument is complemented, i.e. Instead of using the color it will use one minus the color as argument.

The following table lists all constants that may be given to the W3D_COLOR_COMBINE or W3D_ALPHA_COMBINE tags.

Constant	*Formula*
W3D_COMBINE_DISABLE	Output of this stage is disabled, as well as all subsequent stages.

Constant	*Formula*
W3D_COMBINE_SELECT_A W3D_COMBINE_SELECT_B W3D_COMBINE_SELECT_C	Select the given argument as output. $RGB = A \quad \alpha = A$ $RGB = B \quad \alpha = B$ $RGB = C \quad \alpha = C$
W3D_COMBINE_MODULATE	Multiply Argument A and B $RGB = AB \quad \alpha = AB$
W3D_COMBINE_ADD	Add Argument A and B $RGB = A + B \quad \alpha = A + B$
W3D_COMBINE_SUBTRACT	Subtract argument A from B $RGB = A - B \quad \alpha = A - B$
W3D_COMBINE_ADDSIGNED	Add A to B with a -0.5 bias $RGB = A + B - 0.5 \quad \alpha = A + B - 0.5$
W3D_COMBINE_INTERPOLATE	Interpolate between Argument A and B by factor C $RGB = CA + (1 - C)B \quad \alpha = CA + (1 - C)B$
W3D_COMBINE_ACCUM	Multiply and accumulate $RGB = AB + C \quad \alpha = AB + C$
W3D_COMBINE_DOT3RGB W3D_COMBINE_DOT3RGBA	Calculate the dot product between color vector A and color vector B. The output is replicated to RGB (for the first form) or RGBA components (second form) $RGB[\alpha] = A_r B_r + A_g B_g + A_b B_b$

The following table lists the possible values for W3D_ENV_MODE:

Constant	*Function*
W3D_REPLACE W3D_DECAL W3D_BLEND W3D_MODULATE	See W3D_SetTexEnv
W3D_ADD	Add the color of the previous unit (or the iterated color on unit 0) to the texture color of the texture bound on this unit. Only available if a query of W3D_Q_ENV_ADD yields a positive result.
W3D_SUB	Subtract from the color of the previous unit (or the iterated color on unit 0) the color of the texture bound on this unit. Only available if a query of W3D_Q_ENV_SUB yields a positive result.
W3D_OFF	Disable this unit. Note that binding a NULL texture does NOT automatically disable a unit – you MUST explicitly set the texture environment to W3D_OFF to disable a stage when using the combined multitexture model. Likewise if the environment is set to W3D_OFF, any texture bound on the corresponding unit is ignored.

Example: Colored Lightmapping.

```
IWarp3D->W3D_BindTexture(0, gColorMap);
IWarp3D->W3D_BindTexture(1, gLightMap);

IWarp3D->W3D_SetTextureBlendTags(gContext,
 W3D_BLEND_STAGE,       0,
 W3D_COLOR_ARG_A,       W3D_ARG_TEXTURE_COLOR,
 W3D_ALPHA_ARG_A,       W3D_ARG_TEXTURE_ALPHA,

 W3D_COLOR_ARG_B,       W3D_ARG_DIFFUSE_COLOR,
 W3D_ALPHA_ARG_B,       W3D_ARG_DIFFUSE_ALPHA,

 W3D_COLOR_COMBINE,     W3D_COMBINE_MODULATE,
            W3D_ALPHA_COMBINE,      W3D_COMBINE_MODULATE,

      W3D_BLEND_STAGE,        1,
            W3D_COLOR_ARG_A,        W3D_ARG_PREVIOUS_COLOR,
            W3D_ALPHA_ARG_A,        W3D_ARG_PREVIOUS_ALPHA,

            W3D_COLOR_ARG_B,        W3D_ARG_TEXTURE_COLOR,
            W3D_COLOR_ARG_B,        W3D_ARG_TEXTURE_ALPHA,
            // To enhance the effect, scale it by two
            W3D_COLOR_SCALE,        2,

            W3D_COLOR_COMBINE,      W3D_COMBINE_MODULATE,
            W3D_ALPHA_COMBINE,      W3D_COMBINE_MODULATE,
      TAG_DONE);

if (IWarp3D->SetTextureBlend(gContext, NULL) != W3D_SUCCESS)
{
      printf("Error: Texture state invalid\n");
}
```

You may notice that in this example the `SetTextureBlend` function is called with an empty tag list. This can be done at any time and will evaluate the current state of texture blending, returning either a `W3D_SUCCESS` if the state is valid, or a `W3D_INVALIDINPUT` if there is an unsupported or invalid input.

> IMPORTANT: In the above case the first call to `SetTextureBlend` might return `W3D_SUCCESS` while the second one doesn't. The reason for this is that validation of the current state might be delayed to a later point in time by the driver. Calling `SetTextureBlend` with an empty tag list will enforce this validation and always return the appropriate result code.

You may also distribute the setting of the texture states over multiple `SetTextureBlend` calls. In that case, using a single `SetTextureBlend` call with a `NULL` or empty tag list will help to "centralize" the error checking.

In case of an unsupported or invalid state, the result of a subsequent drawing operation is undefined, and might fail to produce anything.

Another example: Colored lightmapping with the combined model

```
IWarp3D->W3D_BindTexture(0, gColorMap);
IWarp3D->W3D_BindTexture(1, gLightMap);

IWarp3D->W3D_SetTextureBlendTags(gContext,
      W3D_BLEND_STAGE,        0,
            W3D_ENV_MODE,     W3D_MODULATE,
      W3D_BLEND_STAGE,        1,
            W3D_ENV_MODE,     W3D_MODULATE,
      TAG_DONE);
```

3.1 Default Setup

The default setup is undefined. The driver will set up something that usually modulates the texture and iterated color on stage 0 and disables all other states. You should set the appropriate setup yourself. This will also ward against the possibility that the default setup changes at one point.

Part II : Interleaved Vertex Arrays And New Array Pointers

Warp3D V5 offers a new method of defining vertex arrays, and introduces two new pointers in the process.

1 New Vertex Pointers

1.1 Fog Coordinates

A new pointer is used to fetch explicit fog coordinates for a vertex. The previously used W coordinate is ignored when the fog coordinate pointer is given, and fog coordinates for a vertex are fetched from this pointer. It is set with the W3D_FogCoordPointer API call, and only used if W3D_FOG is active. See the autodoc for more info on this.

Fog parameters are still specified like before. The only difference is that the coordinate is fetched from the array instead of using the W coordinate, but the fog coordinate is still expressed as a W coordinate (as opposed to e.g. OpenGL).

1.2 Secondary (Specular) Color Array

The second new pointer specifies a color array used for specular coloring. The pointer is set with W3D_SecondaryColorPointer, and is ignored unless the W3D_SPECULAR state is active.

Note that the Secondary Color contains an alpha value, but the alpha value is ignored by most drivers that support secondary coloring.

2 Interleaved Array

Semantically, the interleaved array is another way of specifying up to all array pointers at the same time. Internally, though, the driver might decide to optimize the case. Therefore it is highly recommended that if in doubt you use the interleaved array scheme.

The interleaved array is specified with the W3D_InterleavedArray API call. If you call this function, it will make all other pointers invalid.

> NOTE: That means calling this function *and* calling e.g. W3D_VertexPointer **will not result** in the vertex coordinate data being fetched from the VertexPointer location with the rest from the location specified as the interleaved array!

That means that if you want to "switch" between the interleaved array and the separate arrays you have to re-set all arrays you need, although some drivers may by default use this function solely as a shorthand for calling the individual pointers. But you must not reply on such behaviour.

The format of the data stored in the array, and hence, which array pointers are semantically affected by the call (but keep in mind that all will become invalid), is specified by a bit mask. The following table lists the bits and their meaning. Note that the order at which the data appears in the table is also the order at which it appears in the array; any bit that is set takes up a certain amount of longwords (or floats) in the array, in this specific order. Any bit that is unset does not take any space, and hence enabling a state that requires the data will result in an error.

Bit Name	*Meaning*	*Data*
None	The positional part (usually set by W3D_VertexPointer) is always included in the format. These are three floats denoting x,y and z coordinates.	3 floats

Bit Name	*Meaning*	*Data*
W3D_VFORMAT_FOG	The vertex data contains a single floating point fog coordinate. This amounts to calling W3D_FogCoordPointer.	1 float
W3D_VFORMAT_COLOR	The vertex contains a primary vertex color, specified as four floating point values in the order red, green, blue and alpha. A color value of 0 means no intensity in this channel while a value of 1 means full intensity. Equivalent to calling W3D_ColorPointer. This bit is mutually exclusive with W3D_VFORMAT_COLOR_PACK.	4 floats
W3D_VFORMAT_COLOR_PACK	The vertex contains a primary vertex color, specified as a single unsigned long value with the format 0xRRGGBBAA. The values range from 0 for no intensity to 255 for full intensity. Equivalent to calling W3D_ColorPointer. Must not be specified together with W3D_VFORMAT_COLOR.	1 uint32
W3D_VFORMAT_SCOLOR	The vertex contains a seconday vertex color, specified as four floating point values in the order red, green, blue and alpha. A color value of 0 means no intensity in this channel while a value of 1 means full intensity. Equivalent to calling W3D_SecondayColorPointer. This bit is mutually exclusive with W3D_VFORMAT_SCOLOR_PACK.	4 floats
W3D_VFORMAT_SCOLOR_PACK	The vertex contains a secondary vertex color, specified as a single unsigned long value with the format 0xRRGGBBAA. The values range from 0 for no intensity to 255 for full intensity. Equivalent to calling W3D_SecondaryColorPointer. Must not be specified together with W3D_VFORMAT_COLOR.	1 uint32
W3D_VFORMAT_TCOORD_*n*	The vertex contains a set of three coordinates (u, v and w) as coordinates for texture unit *n*. This is equivalent to calling W3D_TexCoordPointer with a tmu parameter of *n*. Each set is three floating point values. The *flags* parameter of the W3D_InterleavedArray function can be used to specify whether the coordinates are normalized or not. If the flag W3D_TEXCOORD_NORMALIZED is set, then u and v are assumed to be in the range 0.0 to 1.0; otherwise they are assumed to be in the ranges 0..texwidth and 0..texheight, respectively. Note that only unbroken null-based chains of texture coordinate sets are allowed. That is, a vertex may contain sets for TMU 0 and TMU 1, but not for	3 floats

Bit Name	*Meaning*	*Data*
	TMU 0 and TMU 2 (not unbroken), or for TMU 1 and TMU 2 (not null-based).	

The interleaved array is used in exactly the same way as the separate array, i.e. calling `W3D_DrawArray` or `W3D_DrawElements` will draw using the data in the interleaved array.

2.1 Examples

```
/*
 * Example 1: position, floating point color, one texture coordinate set
 */

typedef struct
{
      float x,y,z;
      float r,g,b,a;
      float u,v,w;
} Vertex1;

#define VERTEX1_FORMAT (W3D_VFORMAT_COLOR | W3D_VFORMAT_TCOORD_0)

/*
 * Example 2: position, packed color, packed secondary color, 3 texture sets
 */

typedef struct
{
      float x,y,z;
      uint32 primary;
      uint32 secondary;
      struct
      {
            float u,v,w;
      } texcoord[3];
} Vertex2;

#define VERTEX2_FORMAT (W3D_VFORMAT_PACK_COLOR | W3D_VFORMAT_PACK_SCOLOR | \
      W3D_VFORMAT_TCOORD_0 | W3D_VFORMAT_TCOORD_1 | W3D_VFORMAT_TCOORD_2)

/*
 * Example 3: position and fog coordinate
 */

typedef struct
{
      float x,y,z;
      float fog;
} Vertex3;

#define VERTEX3_FORMAT (W3D_VFORMAT_FOG)
```

Part III : Other New Features

1 W3D_ClearBuffers

The `W3D_ClearBuffers` call replaces and deprecates the previous per-buffer calls `W3D_ClearDrawRegion`, `W3D_ClearZBuffer` and `W3D_ClearStencilBuffer`. The prototype of the new call is

```
uint32 W3D_ClearBuffers(W3D_Context *context, W3D_Color *clearColor,
 W3D_Double *clearDepth, uint32 *clearStencil);
```

The first argument, as usual, points to the context to use for the clear operation. The `clearColor`, `clearDepth` and `clearStencil` parameters may each be `NULL` or a valid pointer. If the `clearColor` is non-`NULL`, it specifies the color buffer (the drawing area) be cleared with the specified color. If the pointer is `NULL`, the color buffer is left untouched. Likewise, if the `clearDepth` parameter is non-`NULL`, the depth buffer is cleared to the value pointed to, otherwise the depth buffer is not cleared and left untouched. Finally, the `clearStencil` parameter, if non-`NULL`, specifies that the stencil buffer should be cleared with the value pointed to by `clearStencil`. Note that the stencil value passed this way will always be clipped to the depth of the stencil buffer, i.e. passing a 32 bit value when an 8-bit stencil buffer is used masks the lower 8 bit of the value and ignores the remaining 24 bits.

This call is much more effective than clearing each individual buffer, especially when stencil buffering is used. Since most graphics cards interleave the depth and stencil buffer, clearing only one buffer might include reading the current value from the buffer and modifying it for the clear operation. For this reason you should always use this call instead of single calls when clearing multiple buffers.

If any buffer is not allocated (i.e. the program requests the depth buffer to be cleared but no depth buffer is allocated) the value and operation on this buffer is silently ignored. Any other buffer that actually *is* present is cleared correctly, however.

2 Stencil Buffering

Prior to Warp3D V5, no driver actually implemented stencil buffering. With the V5 release, both the Napalm (a.k.a Voodoo 4 and Voodoo 5) and the Radeon drivers implement an 8 bit stencil buffer (the Avenger a.k.a. Voodoo 3 does not support stencil buffering).

There is only a minute change in the API, namely in the query items. The stencil buffer query items now return a bit mask that indicates the supported stencil test and stencil functions.

3 New Query Items

A number of additional items have been added to the query mechanism. The stencil buffer query items have been changed (see the previous section). The following table lists the new items and their meaning:

Query Item	*Meaning*
`W3D_Q_NUM_TMU`	The number of texture units a driver supports. That is, the maximum number of textures that can be bound at any one time.
`W3D_Q_NUM_BLEND`	The number of blend stages supported by the driver. This is usually the same as the number of texturing units, but 'usually' already expresses that this isn't necessarily always the case. The number of blend stages signifies the number of blend functions you can set at any one time.
`W3D_Q_ENV_COMBINE`	If `TRUE`, the driver supports the separate blend stage setup outlined in part I of this document. If `FALSE`, then `W3D_ENV_MODE` is the only combine function allowed.
`W3D_Q_ENV_ADD`	If `TRUE`, the driver supports the `W3D_ADD` environment mode as input for `W3D_ENV_MODE`.

Query Item	*Meaning*
W3D_Q_ENV_SUB	If TRUE, the driver supports the W3D_SUB environment mode as input for W3D_ENV_MODE.
W3D_Q_ENV_CROSSBAR	If TRUE, then the driver supports cross-bar texture inputs. That is, you can use W3D_ARG_TEXTURE*n*_COLOR or W3D_ARG_TEXTURE*n*_ALPHA on a blend stage that is not *n*. In other words, any blend stage can access the color or alpha components of any texture unit.