M l i e i g i Di ec X 6

by Jason L. Mitchell, Michael Tatro and Ian Bullard

One of the most interesting features introduced to Direct3D in the recent release of DirectX 6 is multiple texturing. Unfortunately, it's also one of the more confusing new features. This article will introduce multiple texture mapping into the context of the traditional pixel pipeline. We will describe the multitexture programming model, provide programming examples, and spend some time addressing the issues involved in robustly taking advantage of multitexturing hardware while maintaining fallback paths for application-level multipass methods. We have also created MulTex, a simulator that interactively illustrates this potentially puzzling new feature of Direct3D. Experimenting with MulTex is a good way to gain some familiarity with the texture blending abstraction. MulTex is available from the *Game Developer* web site and is definitely useful to have by your side as you read this article. (MFCTEX, a similar tool written by Microsoft, ships with the Microsoft DirectX 6 SDK.)

The Traditional Pixel Pipeline

In previous versions of DirectX, the texture mapping phase of the Direct3D pixel pipeline has only involved fetching texels from a single texture. The two gray pipeline segments in Figure 1 are the stages in the traditional pipeline that deal with determining texel color and blending that color with the color of the primitive interpolated from the vertices. These two stages of the pipeline are replaced by the new multitexturing abstraction. The rest of the pipeline remains untouched.

Texture Operation Units

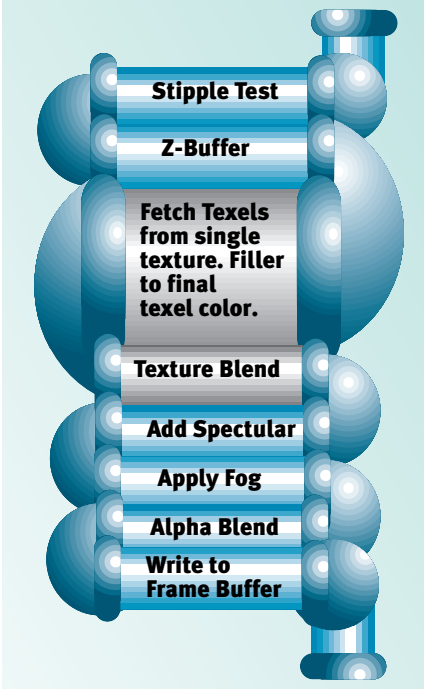
DirectX 6 introduces the concept of a texture operation unit. Each unit may have a single texture associated with it, and up to eight texture operation units can be cascaded together to apply multiple textures to a common primitive.

Each texture operation unit has six associated render states, which control the flow of pixels through the unit, as well as additional render states associated with filtering, clamping, and so on. Figure 2 shows two texture operation units cascaded together. We'll limit our discussion here to the dual texture case to keep things simple and because most of the near-term 3D hardware will support only two textures.

Three of the render states in each texture operation unit are associated with RGB (color), and another three are associated with alpha. For RGB color, the render states **D3DTSS_COLORARG1** and **D3DTSS_COLORARG2** control arguments, while **D3DTSS_COLOROP** controls the operation on the arguments. Likewise, **D3DTSS_ALPHAARG1** and **D3DTSS_ALPHAARG2** control arguments to **D3DTSS_ALPHAOP**. Essentially, the **D3DTSS_COLOR***x* render states control the flow of an RGB vector, while the **D3DTSS_ALPHA***x* render states govern the flow of the scalar alpha through parallel segments of the pixel pipeline, as shown in Figure 2.

Jason L. Mitchell (JasonM@atitech.com) is a software engineer in the 3D Application Research Group at ATI Research Inc. Michael J. Tatro (mike@stainlesssteelstudios.com) is a software engineer at Stainless Steel Studios in Cambridge, Mass. Ian Bullard (ianb@technologist.com) is a software engineer at New World Computing.

FIGURE 1. *The Direct3D pixel pipeline.*



Arguments

Using the argument states, you can direct input, such as interpolated diffuse color or texel color, into the texturing operations. Table 1 shows a complete list.

Additionally, you can invert the arguments or replicate their alpha channel across the RGB channels. In the API, you can bitwise OR in the constants D3DTA_COMPLEMENT and D3DTA_ALPHAREPLICATE with any of these render states to achieve the desired effect. D3DTA_COMPLEMENT simply inverts each of the color channels, while D3DTA_ALPHAREPLICATE replicates the alpha from the argument across the R, G, and B channels. Naturally, the D3DTA_ALPHAREPLICATE flag isn't meaningful if it's used with D3DTSS_ALPHAARGx. Also, D3DTA_CURRENT doesn't make sense for the 0 texture operation unit because there is no previous texture operation unit.

Operators

The operators in each unit can operate on one or both of the corresponding arguments. The operator render states can be set to any of the values in Table 2.

This long list of operations may seem a bit daunting at first, but with some experimentation, the abstraction is actually quite approachable. To get you started with the model, the next section illustrates some common multitexture techniques and how they can be programmed in DirectX 6. We suggest that you follow along with MulTex.

FIGURE 2. *This screenshot, taken from the MulTex utility, shows two cascaded texture operation units.*

Each texture operation unit also has states for texture addressing and filtering associated with it. The application programmer can set these render states independently for each texture operation unit. A common example would be to set the base texture of an object, such as the brick texture in the following dark mapping example (Figure 3), to use D3DTADDRESS_WRAP texture addressing, while the texture operation unit for the dark map uses D3DTADDRESS_CLAMP.

Multitexture Examples

DARK MAPPING. Naturally, our first example of multiple texture mapping is the dark map described by Brian Hook in the August 1997 issue of *Game Developer* ("Multipass Rendering and the Magic of Alpha Rendering"). Dark mapping is commonly used in lieu of vertex lighting, where one of the two textures contains an unlit base texture and the other contains a lighting texture (the dark map). Using the new multiple texturing API, one might implement this technique as shown in Figure 2. In the figure, the two large blue boxes represent texture operation units, and the red lines show the flow of data through the pipeline. The first texture operation unit merely passes data from texture 0 to the next stage. The second texture operation unit receives these texels via Arg2 and also fetches texels from texture 1 via Arg1. The results are modulated, giving the final texel color as shown on the right-hand side of Figure 3. Nothing interesting is being done with the alpha channel of the pipeline in this case. Code (generated by MulTex) for dark mapping is shown in Listing 1.

MODULATE2X. The preceding technique is called dark mapping rather than light mapping because the resulting texel can only be a darker version of the unlit texel from the primary map. For this reason, some applications use a variant of the modulate technique, where the resulting texel is brightened by a factor of two using the D3DTOP_MODULATE2X operation instead of D3DTOP_MODULATE. (This can also be done in two passes using the alpha blending operation of Src*Dest + Dest*Src.) One notable engine that uses this technique is AnyChannel's AnyWorld engine, used in the upcoming Postlinear/SegaSoft game, VIGILANCE (Figure 4). The AnyWorld engine uses a radiosity lighting model that requires precomputed light maps from LightScape, converted for use in the engine. The advantage to using multitexture rendering in this case is that the base texture maps can be tiled and reused, while the light maps, which are unique to each polygon and very low resolution, are used with clamping. This trades the high polygon count associated with radiosity rendering for a fairly large texture footprint.

SPECULAR HIGHLIGHTS AND ENVIRONMENT MAPPING. In order to incorporate view-dependent reflection of light sources into the lighting model, a specular term

TABLE 1. *DirectX 6 texturing operations.*

D3DTA_TFACTOR	Take pixel data from API-level factor. This factor is set with RENDERSTATE_TEXTUREFACTOR.
D3DTA_DIFFUSE	Use interpolated diffuse color (Gouraud shading).
D3DTA_CURRENT	Use color from previous texture operation unit.
D3DTA_TEXTURE	Use color from texture associated with this unit.

TABLE 2. *Operator render states.*

D3DTOP_DISABLE	Disable this and any later texture operation units
D3DTOP_SELECTARG1	Pass argument 1 untouched
D3DTOP_SELECTARG2	Pass argument 2 untouched
D3DTOP_MODULATE	Multiply both arguments together
D3DTOP_MODULATE2X	Multiply both arguments and shift 1 bit
D3DTOP_MODULATE4X	Multiply both arguments and shift 2 bits
D3DTOP_ADD	Add Arguments together
D3DTOP_ADDSIGNED	Add Arguments with -0.5 bias
D3DTOP_ADDSIGNED2X	Add Arguments with -0.5 bias and shift 1 bit
D3DTOP_SUBTRACT	Subtract Arg2 from Arg1, with no saturation
D3DTOP_ADDSMOOTH	Add arguments and subtract product
D3DTOP_BLENDDIFFUSEALPHA	Blend arguments based on interpolated alpha
D3DTOP_BLENDTEXTUREALPHA	Blend arguments based on texture alpha
D3DTOP_BLENDFACTORALPHA	Blend arguments based on factor alpha
D3DTOP_BLENDTEXTUREALPHAPM	Linear alpha blend with premultiplied Arg1 Arg1 + Arg2*(1-Alpha)
D3DTOP_BLENDCURRENTALPHA	Blend arguments based on current alpha
D3DTOP_PREMODULATE	Modulate with next texture before use
D3DTOP_MODULATEALPHA_ADDCOLOR	Arg1.RGB + Arg1.A*Arg2.RGB
D3DTOP_MODULATECOLOR_ADDALPHA	Arg1.RGB*Arg2.RGB + Arg1.A
D3DTOP_MODULATEINVALPHA_ADDCOLOR	(1-Arg1.A)*Arg2.RGB + Arg1.RGB
D3DTOP_MODULATEINVCOLOR_ADDALPHA	(1-Arg1.RGB)*Arg2.RGB + Arg1.A
D3DTOP_BUMPENVMAP	Per pixel environment map perturbation
D3DTOP_BUMPENVMAPLUMINANCE	Environment map perturbation w/luminance channel
D3DTOP_DOTPRODUCT3	A per-pixel dot product that could be used for specification of surface normal vector data in texture maps. The result is (Arg1.R*Arg2.R + Arg1.G*Arg2.G + Arg1.B*Arg2.B) where each component is scaled and offset to make it signed.

is added to the view-independent (diffuse) term. In Direct3D, an application can provide the renderer with specular values at polygon vertices by passing them in the vertex structures. The interpolated specular colors are then added to the lighted texture color as shown in Figure 1. The problem with this method is that specular reflections tend to be fairly localized on an object, and their appearance can vary wildly depending on the tessellation of the object in the area of the specular reflection. Theoretically, the peak brightness of a traditional specular highlight can fall within a polygon (as in, not at a vertex), but this interpolation scheme doesn't reproduce such behavior. Figure 5A shows the artifacts caused by using vertex specular lighting. These artifacts are even more pronounced when the object and/or viewer are in motion.

One solution to this problem is to use a secondary texture as a specular light map, as shown in Figure 5B. The secondary texture coordinates of these polygons are generated by projecting into the light map (perhaps using a hemispherical environment mapping technique). The same effect, with a full environment map, is shown in Figure 5C. In Figures 5B and 5C, the teapot is rendered using traditional diffuse lighting at the vertices, which is modulated with the primary (woodgrain) texture. To this product, the second blending unit adds the specular map. Direct3D multiple texturing syntax is shown in Listing 2.

LINEAR BLENDING. Multitexturing also lets you linearly blend between two textures for morphing effects. You can use any of the D3DTOP_BLEND*x*ALPHA operations, but D3DTOP_BLENDFACTORALPHA or D3DTOP_BLENDDIFFUSEALPHA are most efficient for frame-to-frame variation of the linear blend factor. A good example of using a linear blend for morphing is the ATI Knight Demo, first shown at the CGDC in 1997 and illustrated in Figure 6. In this example, the stone texture map on the knight statue is the primary texture, and the "living" texture map is secondary. From frame to frame during the morph, varying the blend factor causes the living texture to fade in until it is the only texture visible. For efficiency, before and after the morph, traditional single texture mapping is used with the appropriate texture (Figures 6A and 6D).

WHAT ABOUT PLAIN OLD DIFFUSE VERTEX LIGHTING? As show in Figure 1, DirectX 6 has rolled the whole texture blending phase of the traditional pixel pipeline into the multiple texture mapping model. As a result, there is no separate texture blending render state like the one that existed in previous versions of Direct3D. Instead, the program needs to use a single texture operation unit to perform common vertex lighting. In order to use common vertex lighting on a single texture, program texture blending unit 0 as shown in Listing 2, but disable texture blending unit 1.

OTHER TEXTURE OPERATIONS. What we've illustrated here are techniques that are likely to be of immediate use to developers, given the effects popular today and the capabilities of available hardware. Microsoft has provided illustrations of a variety of multitexturing effects and their multipass equivalents in the DirectX 6 SDK, though some of the single-pass versions of the techniques may not be supported by current or even next-generation hardware. Of course, users of Direct3D applications are likely to have cards with varying coverage of multitexture features (including no multitexturing features at all). How can an application determine the feature coverage of the hardware it is running on and use the most optimal multitexturing technique? Fortunately,

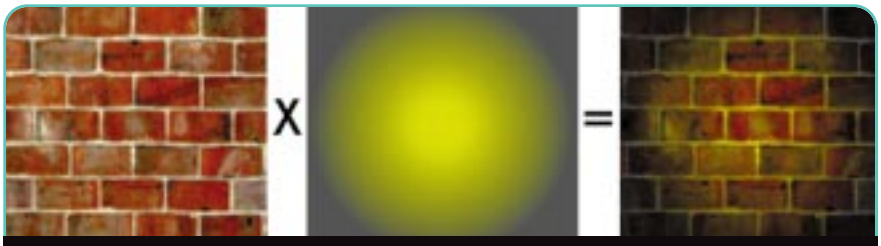


FIGURE 3. *Dark mapping in action.*

FIGURE 4. *The AnyWorld engine. The walls and floor are tiled with smaller textures while large light maps add highly detailed lighting to the scene.*

the new API includes a means for validating the multitexturing techniques an application will use. In the next section, we cover this validation scheme as well as an architecture for incorporating fallback techniques so that an application will be able to take advantage of multitexturing hardware when it's available and robustly fall back to multipass techniques when it isn't.

The Reality of Hardware Support

You'll note that throughout this article, we've consistently referred to the new texture abstraction as just that, an abstraction. The model, or abstraction, that's illustrated here and documented in Microsoft's DirectX 6 SDK doesn't necessarily map directly to the silicon designed by 3D video card makers. In fact, in some cases, the silicon predates the API. Additionally, the full feature set defined in the model isn't implemented on any 3D cards available to date. As a result, programmers may initially find support of multiple texture mapping modes somewhat sparse relative to the extreme flexibility of the API. Microsoft has written a reference rasterizer for DirectX 6 that implements the full multiple texture mapping model; software developers can use these to experiment with new effects. Also under DirectX 6, the software rasterizer (not the same as the reference rasterizer) has support for two texture operation units and a reasonable subset of the operations defined by the API. MulTex can also serve as a tool for experimenting with new techniques. For the foreseeable future, we recommend that developers plan to implement both multipass and single-pass versions of techniques in their titles, where the multipass code is executed when the application is running on boards that cannot provide the desired functionality in a single pass.

Typically, we expect developers to define a set of materials that they will use in their applications. These materials are defined by the texture operation units' arguments and operations that will be used when rendering a polygon of that material type, as well as a few other criteria, which we'll touch on in a moment. Each multitextured material will have multiple ways that it can be rendered: the ideal single-pass case, the multipass fallback case, and any intermediate cases. For example, a wall with a static diffuse light map and a dynamic light map might have three rendering methods (Table 3).

At application initialization time, the 3D application should run through its

LISTING 1. *Dark mapping. In this code snippet, unspecified render states are left in their default states for brevity. In practice, an application should be more defensive than this.*

```
// Program Stage 0:
lpDev->SetTexture(0, pTex0 );
lpDev->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
lpDev->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);

// Program Stage 1:
lpDev->SetTexture(1, pTex1 );
lpDev->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE);
lpDev->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT);
lpDev->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE);
```

LISTING 2. *Specular mapping as shown in Figures 5B and 5C.*

```
// Program Stage 0:
lpDev->SetTexture(0, pTex0 );
lpDev->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
lpDev->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
lpDev->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);

// Program Stage 1:
lpDev->SetTexture(1, pTex1 );
lpDev->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE);
lpDev->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT);
lpDev->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADD);
```

FIGURE 5. *Vertex specular (A) versus specular mapping (B) and environment mapping (C).*

TABLE 3. *A three-tiered approach to multitexture rendering a wall with a static light map and a dynamic light map.*

Top Tier	Single-pass	(Static Light Map + Dynamic Light Map) * Wall Texture
Middle Tier	Two-pass	Frame buffer = Static Light Map + Dynamic Light Map Alpha blend to get Wall Texture * Frame buffer
Bottom Tier	Three-pass	Frame buffer = Static Light Map Alpha blend to get Frame buffer = Static Light Map + Dynamic Light Map Alpha blend to get Frame buffer = (Static Light Map + Dynamic Light Map) * Wall Texture

materials to determine which rendering method it can use for each material on the given hardware. The application would program the texture operation units for the top tier and validate this set of states by calling the new function `IDirect3DDevice3::ValidateDevice()`. If the function passes, the application can use this tier when this material is being rendered. If `ValidateDevice()` fails, the application should keep moving downward until a tier passes validation. When `ValidateDevice()` fails, it returns an error code that indicates why it failed; applications should be prepared to handle this successfully. The error codes are shown in Table 4.

These return codes are the additional parameters that make up what we are considering a material. For example, an application that plans to use dark mapping techniques, sometimes using trilinear filtering and sometimes using bilinear filtering, should consider each of these cases separately and validate accordingly. This distinction will probably become less critical in a year or two, but with the first crop of multitexture hardware, it's very important to pay attention to validation.

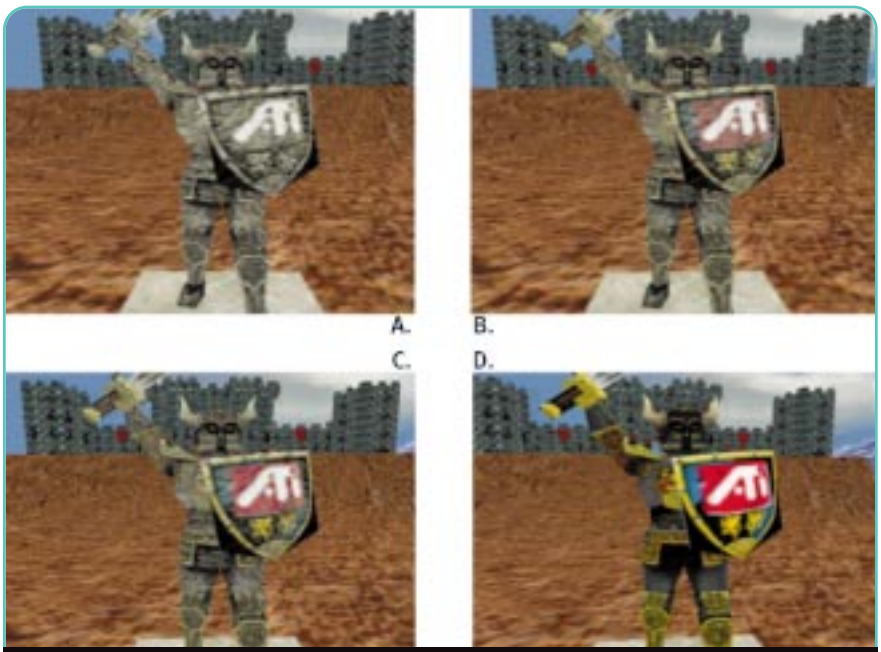


FIGURE 6. *Four frames of the ATI Knight Demo from CGDC 97 showing a texture morph using a linear blend factor. The camera is moving slightly during this series of shots. The full .AVI is available from the* Game Developer *web site.*

TABLE 4. `IDirect3DDevice3::ValidateDevice()` *return codes.*

WRONGTEXTUREFORMAT	The hardware cannot support the current state in the selected texture format.
UNSUPPORTEDCOLOROPERATION	The specified color operation is unsupported.
UNSUPPORTEDCOLORARG	The specified color argument is unsupported.
UNSUPPORTEDALPHAOPERATION	The specified alpha operation is unsupported.
UNSUPPORTEDALPHAARG	The specified alpha argument is unsupported.
TOOMANYOPERATIONS	The hardware can't handle the specified number of operations.
CONFLICTINGTEXTUREFILTER	The hardware can't do both trilinear filtering and multitexture at the same time.
UNSUPPORTEDFACTORVALUE	The hardware can't support `D3DTA_TFACTOR` greater than 1.0.

Chipsets with Multiple Texture Support

At press time, the ATI Rage Pro and the 3Dfx Voodoo2 are the only shipping cards with multitexture support. So developers already have access to the first crop of available hardware. For up-to-date information, DirectX 6 drivers, and a list of which texture operations are supported, keep an eye on the developer support material on the ATI and 3Dfx web sites. ■

Acknowledgements

Thanks to AnyChannel, John Pallett-Plowright, and our colleagues at ATI for their input.

FOR FURTHER INFO

3Dfx Inc.
http://www.3dfx.com

ATI Research Inc.
http://www.ati.com

AnyChannel's AnyWorld engine
http://www.anychannel.com/anyworld.html