Chapter 11

Basic Texturing

"Without drawing, I feel myself but
half invested with language."
S. T. Coleridge: *Notebooks*, 1803

11.1 Overview

With lighting, we saw how we could create an appearance for our models by specifying lighting and material properties to compute a color for each vertex. With vertex shaders, we were able to extend the fixed-function lighting and create per-vertex color components from arbitrary computations. With either method, we computed color components for vertices which were interpolated across the primitive by rasterization. If a very detailed appareance to an object was desired, each detail must be modelled with geometry and lighting parameters.

With texturing we can provide complex visual detail using an image instead of complex geometry. When low resolution geometry is combined with textures, we can create visually interesting elements for a scene that render very quickly.

Texturing allows us to compute the diffuse color of each pixel by table lookup. The indices to the table are computed from the texture coordinates associated with each vertex which are interpolated across the primitive. The texture coordinates can be included directly as vertex components or can be generated with vertex processing. The table is defined by a texture resource, organized as a regular grid of texel values. A texel is similar to a pixel, but is a more generalized idea of a sample, not necessarily just a sample of a color used in a picture.

The texture coordinates are scalar, two-dimensional, three-dimensional or four-dimensional floating-point values. The texture can be rectangular, cubic, or volumetric in organization, while the texels themselves can be almost any enumerant of `D3DFORMAT`, including vendor-specific four character code formats.

With a single texture, the result of the table lookup can be combined with other color inputs from rasterization to produce the diffuse color sent to the frame buffer. With multiple textures, the results of one texture are used as input to the next stage of texture processing, with the final stage producing the diffuse color.

Fixed-function pixel processing provides a number of inputs that can be combined through one or more texture stages to create very complex coloring, lighting and shading effects for primitives. Programmable pixel processing extends the range of achievable effects tremendously. This chapter will introduce the multitexturing architecture provided by Direct3D and cover the details of the application of a single texture. In the next chapter, we will discuss advanced texturing techniques using multiple stages. In chapter 13 we will cover the details of programmable pixel processing and show how it corresponds to fixed-function pixel processing.

We start by discussing the general architecture of multitexturing in Direct-3D. Texturing is organized as a series of stages, with each successive stage fed by the result of the previous stage. The operation of each stage is controlled by the texture stage states associated with the stage. Each stage is bound to a texture resource containing the table.

Next we examine the texturing resources we can create with the device: rectangular textures, cube textures and volume textures. We describe the interfaces for texture resources and their management.

With the groundwork out of the way, we'll next look at the texture coordinate processing in detail. Texture coordinates can be provided in the vertex data or can be generated by Direct3D. Each coordinate can be transformed by a transformation matrix before being presented to the stage.

Once a texture coordinate is associated with a source pixel and a textures is bound to a stage, we will show how the texture is sampled by the coordinates to produce a texel value. We show how textures can be sampled using different methods to avoid aliasing artifacts in producing the final texel.

Next we show how the sampled texel is combined with the vertex color components to produce the diffuse color sent to the frame buffer. Each stage allows for independent processing of the RGB color channels and the alpha channel. We will show how each of these can be controlled independently.

Finally, we close the chapter with a sample program that demonstrates how to create, fill and use a single texture. The program lets you explore the options for the minification, magnification and mipmap filters supported by the device. Texture addressing modes can be independently selected for each texture coordinate and scenes with one-, two- and three-dimensional textures can be selected.

11.2 Direct3D Multitexture Architecture

The architecture of Direct3D multitexturing is shown in figure 11.1. Interpolated vertex lighting components from the rasterizer are fed into the texture

Rasterization
Stage 0
Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Stage 6
Stage 7
Frame Buffer

Inputs
Stage
RGB Unit
Alpha Unit
Output

Texture Coordinate
Texture Transform
Texture Addressing
Texture Sampling
Texels

Figure 11.1: The multitexture cascade, consisting of up to eight texture stages, with the output of each stage fed as an input to the successive stage. Each stage consists of an RGB unit that computes the color for the stage and an alpha unit that computes the transparency for the stage.

cascade. Each stage of the cascade computes some output from the inputs to the stage using a color processing unit and an alpha processing unit. Processing continues through each successive enabled texture stage, with the last stage producing the output sent to the frame buffer. The alpha and color processing units can be controlled independently, but cannot be enabled or disabled separately.

The operation of each stage is controlled through a series of state variables, identified by the `D3DTEXTURESTAGESTATETYPE` enumerated type, listed on page 97. The state variables select the inputs and output of each stage, the computation performed in the RGB and alpha processing units, as well as how the associated texture is addressed by the texture coordinates used by each stage.

```
HRESULT GetTextureStageState(DWORD stage,
            D3DTEXTURESTAGESTATETYPE kind,
            DWORD *value);
HRESULT SetTextureStageState(DWORD stage,
            D3DTEXTURESTAGESTATETYPE kind,
```

```
            DWORD value);
```

Each stage can be bound to a texture resource with the `SetTexture` and `GetTexture` methods on the device. The `MaxTextureBlendStages` member of `D3DCAPS9` gives the maximum number of texture stages supported by the device, while the `MaxSimultaneousTextures` member gives the maximum number of stages that can be bound to a texture at a time. The difference is that some devices expose additional stages that can be used to provide processing, but they don't have associated texture units and therefore cannot address and sample a texture.

```
HRESULT GetTexture(DWORD index,
            IDirect3DBaseTexture9 **value);
HRESULT SetTexture(DWORD index,
            IDirect3DBaseTexture9 *value);
```

The type of this device property is `IDirect3DBaseTexture9`, the base interface for all texture resources as shown in figure 3.1 on page 82. The interfaces for textures are described in the next section.

After calling `GetTexture`, you can determine the type of texture by examining its resource type with the `GetType` method on the `IDirect3DResource9` interface as described in section 3.5 on page 82. Once the resource type is known, `QueryInterface` can be used to obtain the appropriate interface pointer. This is one of the few places where you might need to call `QueryInterface` from a Direct3D application; you cannot use a C++ style cast to obtain the appropriate interface pointer. The following code calls `GetTexture` and then obtains an `IDirect3DTexture9` interface pointer.

```
IDirect3DTexture9 *texture = 0;
IDirect3DBaseTexture9 *base = 0;
THR(device->GetTexture(0, &base));
THR(base->QueryInterface(IID_IDirect3DTexture9,
    reinterpret_cast<void **>(&texture)));
base->Release();
// use texture pointer for something...
texture->Release();
```

With the `CComPtr<>` smart pointer class provided by ATL in `<atlbase.h>`, we can eliminate the use of `reinterpret_cast` and the manual management of reference counts.

```
CComPtr<IDirect3DTexture9> texture;
CComPtr<IDirect3DBaseTexture9> base;
THR(device->GetTexture(0, &base));
THR(base->QueryInterface(IID_IDirect3DTexture9, &texture));
// use texture pointer for something...
// IUnknown::Release called in ~CComPtr
```

ATL provides another smart pointer class, `CComQIPtr<>`, designed specifically for this usage of `QueryInterface`. `QueryInterface` will be called in the constructor for `CComQIPtr<>`; if the interface could not be obtained, then the interface pointer will be zero. The obvious way to use it is as follows:

```
CComPtr<IDirect3DBaseTexture9> base;
THR(device->GetTexture(0, &base));
CComQIPtr<IDirect3DTexture9> texture = base;
if (!texture) THR(E_NOINTERFACE);
// use texture pointer for something...
// IUnknown::Release called in ~CComPtr and ~CComQIPtr
```

However, this will result in a compile error even though there are no syntactical errors in this usage. The problem stems from the exact mechanism used by `CComQIPtr<>` to determine the interface GUID `IID_IDirect3DTexture9`. `CComQIPtr<>` uses a Microsoft extension to C++ that associates the COM interface GUID with the C++ interface structure via the `uuid()` declaration specification. While many standard Win32 header files associate the COM interface GUIDs with the corresponding C++ interface structures, the Direct3D header files do not. Fortunately, we can associate the COM interface GUID, defined in `<d3d9.h>`, with the C++ struct using the following code.

```
struct
__declspec(uuid("{85C31227-3DE5-4f00-9B3A-F11AC38C18B5}"))
IDirect3DTexture9;
```

The header file `<rt/iid.h>` in the sample code associates the appropriate COM interface GUIDs with their corresponding Direct3D interfaces.

11.3 Texture Resources

Before we delve into the specifics of how textures are addressed with coordinates, let's take a look at the textures themselves. Textures are described to the device through a collection of resource interfaces that derive from `IDirect3DBaseTexture9`. The texture interfaces allow expose the Texture resources are COM objects exposing the texel samples within a texture. Each texture resource contains a regular grid of texel samples that are used in pixel processing.

As we saw in chapter 1, sampling can introduce aliasing. With texturing, there are two possible sources of aliasing. First, the process of producing the texels themselves may introduce aliasing. Second, the texels themselves are used to reconstruct a continuous function which is sampled during pixel processing. The sampling of the reconstructed function can also introduce aliasing.

Direct3D provides no direct means for eliminating the first source of aliasing. Usually texture resources are created by artists in a paint program, or by scanning a photographic original. In both cases, the effects of aliasing can be

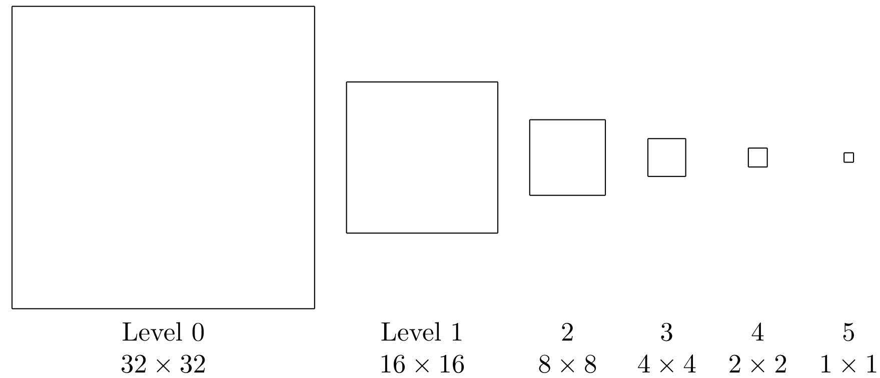


Figure 11.2: Organization of mipmapped textures. Each successive level in a mipmap reduces the dimensions of the previous level by one half. Rectangular mipmap level dimensions decrease as with square textures, with the smaller dimension decreasing to 1 until the larger dimension is also 1. An 16×4 rectangular texture would have mipmap levels of dimension 16×4, 8×2, 4×1, 2×1 and 1×1.

minimized by creating the texture resources from supsersampled original images, which are possibly filtered to fit the size of the texture resource. For the second source of aliasing, Direct3D provides mipmapping and texture filtering to minimize the effect of aliasing.

In the mapping from texture coordinates to screen space, a texture's pixels can be compressed to be smaller than a screen-space pixel, or may be enlarged to be larger than a screen-space pixel. With perspective foreshortening, both of these effects may take place within a single primitive. When a texture is compressed, it is said to be **minified** and when it is enlarged, it is said to be **magnified**.

During minification, the fine details of a texture may be lost due to aliasing. We could minimize this aliasing, if we knew how many texels were covered by a screen-space pixel. A weighted average of the texels covered by the pixel can be computed as the texel to be used for that pixel. However, computing this weighted average could be very costly as we could compress an entire 2048×2048 sized texture into less than a screen pixel with a suitable choice of geometry and texture coordinates.

Mipmaps

mip: *multum in parvo*, Latin for "many things in a small place".

Mipmapping is a preprocessing technique that allows us to compute an approximation to the exact weighted average quickly. The idea behind mipmapping is to create filtered versions of the most detailed texture image. Each application of the filter to an image results in a new image half as large as the original image in each dimension. The filter is chosen to eliminate the higher frequency components in the source image to produce a smaller image with lower frequency

components. This process starts with the most detailed texture image and is repeated on successive filtered images until, for a square texture, a single pixel image is obtained. This single pixel will represent the average sample value over the entire original texture. The mipmap itself is a collection of images, each stored in a level of the mipmap. Level 0 of the mipmap is the highest resolution image in the mipmap and generally contains the original image from which the other levels were generated through filtering. Level 1 of the mipmap is half the size of level 0, and so-on, down to the smallest level in the mipmap, as shown in figure 11.2.

With a mipmap, we can compute the weighted average of the texels covered by a screen-space pixel by selecting the appropriate mipmap level, and selecting a texel from the mipmap level. Because each successive mipmap level, starting from 2, is half the size of the previous mipmap level, each texel covers twice as much area in screen-space than the previous mipmap level. We'll see the exact formula for selecting mipmap levels and texels in the following sections.

Textures are not required to be mipmaps and may consist of only a single level. However, textures that are likely to be minified should probably be in mipmap form. Mipmaps trade off the storage space of the map for the speed in computing the filtered texel. The space penalty for a mipmap is $\frac{1}{3}$ the size of level 0. On some architectures, use of mipmaps may have a positive effect on performance, see chapter 23.

Texture Resource Management

As you might imagine, textures can consume quite a bit of memory. Imagine an application with 10 textures, each 512 texels square, with 32-bit texels. As a single level texture, each texture will occupy at least 1 MB of memory, for a total of 10 MB of texture memory. An additional $3\frac{1}{3}$ MB of memory will be required if all the textures are mipmapped.

The resource manager described in section 3.5 can be used with texture resources to allow a virtually unlimited amount of texture resources to be used with a device. The statistics returned by the resource manager are helpful in tuning your application's use of texture resources, see pg. 90.

Several bits in `D3DCAPS9::DevCaps` give information about the characteristics of the device's texture memory. If the `D3DDEVCAPS_SEPARATETEXTUREMEMORIES` bit is set, the device can use textures from different memory pools. If the `D3DDEVCAPS_TEXTURENONLOCALVIDMEM` bit is set, the device can use textures in non-local video memory, such as AGP memory. If the `D3DDEVCAPS_TEXTURESYSTEMMEMORY` bit is set, the device can use textures in system memory. If the `D3DDEVCAPS_TEXTUREVIDEOMEMORY` bit is set, the device can use textures in video memory.

11.3.1 `IDirect3DBaseTexture9`

As shown in figure 3.1 on page 82, all texture resources derive from `IDirect3DBaseTexture9`. This interface consists only of two properties: a read-only

mipmap level count, determined at resource creation time, and a level of detail.

Interface 11.1: Summary of the `IDirect3DBaseTexture9` interface.

IDirect3DBaseTexture9

Read-Only Properties	
`GetLevelCount`	The number of mipmap levels in the texture.
Properties	
`GetAutoGen-FilterType` `SetAutoGen-FilterType`	Mipmap generation filter.
`GetLOD` `SetLOD`	Most detailed mipmap level stored in the device.
Methods	
`GenerateMip-SubLevels`	Filter a level to generate mipmap sublevels.

```
interface IDirect3DBaseTexture9 : IDirect3DResource9
{
    // read-only properties
    DWORD GetLevelCount();

    // read-write properties
    DWORD GetLOD();
    DWORD SetLOD(DWORD value);
};
```

Only managed textures support the level of detail property. For nonmanaged texture resources, the property has a value of 0. We can think of each successive level of a mipmap as a reduced level of detail when compared to the previous level.

The level of detail property is the highest mipmap level that will be streamed into the device for a managed texture. Dropping one level of detail from a texture will reduce its size to $\frac{1}{4}$ of its previous size.

Dirty Regions

Textures maintain a "dirty region list" to obtain more efficient streaming of modified texels into device memory. The basic idea is to maintain two versions of the texture, one residing in the system memory pool and another residing in the default memory pool. The application makes modifications to the system

memory pool version, updating the dirty region list to reflect the modified region. The dirty regions are then copied to the default pool version and the dirty region list is cleared. The dirty region list is maintained for the most detailed mipmap level in the source texture and corresponding regions in the successive mipmap levels are also considered dirty.

For a rectangular texture, the dirty region is a list of rectangles, while for a volumetric texture the dirty region is a list of rectangular parallelpipeds, or "boxes". When the texture is created, the dirty region list initially covers the entire texture. Locking a region on a texture adds that region to the dirty list, as described in the following sections. Using a texture as the target of a `StretchRect` operation marks the entire texture as dirty. Each texture resource also provides a method to explicitly add regions to the dirty list.

The `UpdateTexture` method on the device copies the dirty regions from the system memory texture to device memory texture and then clears the dirty region list.

```
HRESULT UpdateTexture(IDirect3DBaseTexture9 *source,
            IDirect3DBaseTexture9 *destination);
```

The source texture must be in the system memory pool and the destination texture must be in the default memory pool, or an error results. The two textures must match in most respects: they must be textures of the same resource type, they must have the same format and must have equal dimensions for corresponding mipmap levels or an error results. The destination texture may have fewer mipmap levels than the source texture, in which case only the matching mipmap levels are copied.

11.3.2 `IDirect3DTexture9`

The `IDirect3DTexture9` interface exposes a single collection of mipmap levels for a rectangular texture. The `CreateTexture` method on the device is used to create a texture resource.

```
HRESULT CreateTexture(UINT width,
           UINT height,
           UINT levels,
           DWORD usage,
           D3DFORMAT format,
           D3DPOOL pool,
           IDirect3DTexture9 **result);
```

Unlike image surfaces, which can be created in almost any size and format, textures typically have more restrictions in terms of size and format. As with other resources, the format and pool constraints for a texture can be checked with the `IDirect3D9` method `CheckDeviceFormat`. A device indicates the size requirements through the `D3DCAPS9` structure.

The `levels` parameter indicates the number of mipmap levels in the texture, with zero and one being the most common values. A value of one creates a non-mipmapped texture of a single level. A value of zero creates a full set of mipmap levels down to a level with dimensions 1×1. Values greater than one create a partial set of mipmap levels, provided the number of levels is consistent with the size of the texture. A partial set of mipmap levels can be used when you know the range of minification that will be used on textured primitives will not exceed the number of levels created. A device supports mipmapped rectangular textures if the `D3DPTEXTURECAPS_MIPMAP` bit of `D3DCAPS9::TextureCaps` is set.

Texture hardware often imposes constraints on the dimensions of textures to obtain efficient rendering of textured primitives. The maximum supported dimensions for a texture are given by the `MaxTextureWidth` and `MaxTextureHeight` members of `D3DCAPS9`. For rectangular textures, the maximum supported aspect ratio (width divided by height) is given by the `MaxTextureAspectRatio` member. Beyond these maximum dimensions, a texture may be required to have dimensions that are powers of two or equal dimensions for the width and height. The former is required if the `D3DPTEXTURECAPS_POW2` bit of `TextureCaps` is set. The latter is required if the `D3DPTEXTURECAPS_SQUAREONLY` bit of `TextureCaps` is set.

Textures whose dimensions are not powers of two may be used if the `D3DPTEXTURECAPS_NONPOW2CONDITIONAL` bit of `TextureCaps` is set and the texture meets these additional constraints:

1. The texture is used with the clamp addressing mode.
2. Texture coordinate wrapping is not used.
3. The texture is not mipmapped.
4. The texture is not stored in a compressed format.

As you can see, the size constraints for textures are a little involved. The easiest way to deal with the constraints is to create square textures with dimensions that are powers of two. This is most easily handled by adjusting your artwork generation process so that the textures meet these constraints from the beginning. Alternatively, an application can pack multiple images into a single texture resource and adjust the texture coordinates to address the proper region for each image. Images from external sources can also be resampled to fit the size of the texture resource with texture coordinates addressing the full size of the resource, or the external image source can be placed into the next largest size texture resource with the texture coordinates adjusted to address the range of the image within the texture. D3DX provides functions for creating textures from image files and functions for adjusting the dimensions of texture resources to comply with device requirements, see chapter 15.

The `usage` parameter to `CreateTexture` can include zero or more of the following flags:

```
#define D3DUSAGE_DEPTHSTENCIL 0x00000002L
```

```
#define D3DUSAGE_DYNAMIC      0x00000200L
#define D3DUSAGE_RENDERTARGET 0x00000001L
```

Textures created with a render target usage can be used as the render target for rendering by obtaining the appropriate surface interface pointer for a texture mipmap level and calling `SetRenderTarget`. Textures with a depth/stencil usage can be used to create shadow maps and are discussed in the next chapter.

Textures can be created with the dynamic usage flag if the `D3DCAPS2_DYNAMICTEXTURES` bit of `D3DCAPS9::Caps2` is set. Dynamic textures are designed for situations where the contents of the texture are changed frequently, such as once per frame. This is often the case with procedural textures where the contents of the texture are generated from an algorithm instead of read from an image file. Dynamic textures cannot be created in the managed memory pool or an error will result.

`IDirect3DTexture9` exposes a mipmap chain as a collection of surfaces with `GetLevelDesc` and `GetSurfaceLevel`. `GetLevelDesc` obtains the surface description for a texture level and is identical in semantics to the `GetDesc` method for surfaces, described on page 113. `GetSurfaceLevel` returns a surface interface pointer for the given mipmap level of the texture. The `AddDirtyRect` method allows an application to explicitly add a rectangle to the dirty list for the texture. The `LockRect` and `UnlockRect` methods provide for direct access to the texels within a mipmap level.

Interface 11.2: Summary of the `IDirect3DTexture9` interface.

IDirect3DTexture9	
Read-Only Properties	
`GetLevelDesc`	A description of the texel data for a mipmap level.
`GetSurfaceLevel`	The surface interface for a mipmap level.
Methods	
`AddDirtyRect`	Adds a rectangle to the dirty region list.
`LockRect`	Obtains direct access to the contained texel data.
`UnlockRect`	Releases direct access to the contained texel data.

```
interface IDirect3DTexture9 : IDirect3DBaseTexture9
{
    // read-only properties
    HRESULT GetLevelDesc(UINT level,
                D3DSURFACE_DESC *value);
    HRESULT GetSurfaceLevel(UINT level,
                IDirect3DSurface9 **value);

    // methods
```

```
    HRESULT AddDirtyRect(const RECT *rect);
    HRESULT LockRect(UINT level,
                D3DLOCKED_RECT *data,
                const RECT *rect,
                DWORD flags);
    HRESULT UnlockRect(UINT Level);
};
```

The flags argument to LockRect is a bitwise combination of zero or more of the following flags:

```
#define D3DLOCK_DISCARD         0x00002000L
#define D3DLOCK_NO_DIRTY_UPDATE 0x00008000L
#define D3DLOCK_NOOVERWRITE     0x00001000L
#define D3DLOCK_NOSYSLOCK       0x00000800L
#define D3DLOCK_READONLY        0x00000010L
```

The discard, no overwrite, read only and system lock flags have the same meaning as they do for the LockRect method on a surface, described on page 114. The no dirty update flag specifies that the locked region is not to be added to the dirty region list for this texture. Each call to LockRect must be matched by a corresponding call to UnlockRect.

As we did for surfaces in section 4.3, we can create a locking helper class for locking rectangles on a texture mipmap level. The file <rt/texture.h> in the sample code includes such a helper class that locks a rectangle on a mipmap level in its constructor and unlocks the rectangle in its destructor. Its implementation is similar to the surface lock class, with the addition of the mipmap level argument to the constructor of the lock.

11.3.3 IDirect3DCubeTexture9

A cube texture resource is used in advanced texturing techniques such as environment mapping, which we will discuss in the next chapter. A cube texture models a square texture applied to each surface of a cube centered on the origin, as shown in figure 11.3. A cube texture is created with the CreateCubeTexture method on the device.

```
HRESULT CreateCubeTexture(UINT edge_length,
            UINT levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            IDirect3DCubeTexture9 **result);
```

The levels, usage, format and pool parameters to CreateCubeTexture have the same semantics as for CreateTexture. A device supports cube textures if the D3DPTEXTURECAPS_CUBEMAP bit of D3DCAPS9::TextureCaps is set. A

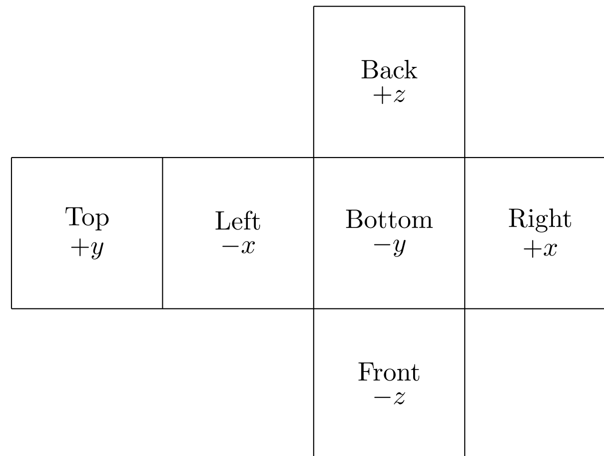


Figure 11.3: Cube texture organization. When a cube, centered on the origin in a left-handed coordinate system and aligned with the coordinate axes, is unfolded onto the plane the result is six squares in the plane. Each face of the cube is labelled by the coordinate axis that intersects the center of the face. Each label corresponds to an enumerant of `D3DCUBEMAP_FACES`.

device supports mipmapped cube textures if the `D3DPTEXTURECAPS_MIPCUBEMAP` bit of `TextureCaps` is set.

Unlike a rectangular texture, a cube texture always has a width equal to its height, with the `edge_length` parameter defining the size of each face of the cube in texels. If the `D3DPTEXTURECAPS_CUBEMAP_POW2` bit of `TextureCaps` is set, then the device requires the `edge_length` parameter to be a power of two.

Interface 11.3: Summary of the `IDirect3DCubeTexture9` interface.

IDirect3DCubeTexture9

Read-Only Properties

`GetCubeMapSurface`	The surface interface for a cube face.
`GetLevelDesc`	A description of the texel data for a cube face.

Methods

`AddDirtyRect`	Adds a rectangle to the dirty region list.
`LockRect`	Obtains direct access to the contained texel data.
`UnlockRect`	Releases direct access to the contained texel data.

```
interface IDirect3DCubeTexture9 : IDirect3DBaseTexture9
{
    // read-only properties
    HRESULT GetCubeMapSurface(D3DCUBEMAP_FACES which,
                UINT level,
                IDirect3DSurface9 **value);
    HRESULT GetLevelDesc(UINT level,
                D3DSURFACE_DESC *value);

    // methods
    HRESULT AddDirtyRect(D3DCUBEMAP_FACES which,
                CONST RECT *rect);
    HRESULT LockRect(D3DCUBEMAP_FACES which,
                UINT level,
                D3DLOCKED_RECT *data,
                const RECT *rect,
                DWORD flags);
    HRESULT UnlockRect(D3DCUBEMAP_FACES which,
```

The GetCubeMapSurface method returns a surface interface pointer for a mipmap level of one of the cube faces. The cube faces are identified by the values of the D3DCUBEMAP_FACES enumeration.

```
typedef enum _D3DCUBEMAP_FACES
{
    D3DCUBEMAP_FACE_POSITIVE_X = 0,
    D3DCUBEMAP_FACE_NEGATIVE_X = 1,
    D3DCUBEMAP_FACE_POSITIVE_Y = 2,
    D3DCUBEMAP_FACE_NEGATIVE_Y = 3,
    D3DCUBEMAP_FACE_POSITIVE_Z = 4,
    D3DCUBEMAP_FACE_NEGATIVE_Z = 5
} D3DCUBEMAP_FACES;
```

AddDirtyRect adds a rectangle on a face of the cube texture to its dirty region list. The LockRect and UnlockRect methods provide direct access to the texel data for a mipmap level of a cube face. Each call to LockRect must be matched by a corresponding call to UnlockRect. The header file <rt/texture.h> in the sample code includes a helper class that calls LockRect in its constructor and UnlockRect in its destructor, similar to the helper class for rectangular textures.

11.3.4 IDirect3DVolume9

While a rectangular texture contains mipmap levels that are described by surfaces, a volume texture contains mipmap levels that are described by volumes. A volume is a three-dimensional regular grid of texel samples, as shown in figure 11.4, and has an interface similar to that of a surface. Unlike surfaces, volumes cannot be created directly; they can only be created as the levels of a volume texture. Before we describe a volume texture, we will look at the interface for volumes.

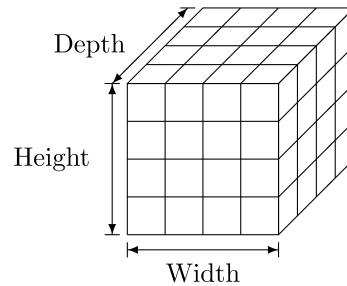


Figure 11.4: Volume organization. A volume arranges texels in a regular three-dimensional grid. The volume can be considered as a stack of "slices" in depth, with each slice containing *Width* × *Height* texels.

Interface 11.4: Summary of the `IDirect3DVolume9` interface.

IDirect3DVolume9

Read-Only Properties	
`GetContainer`	The containing resource or device.
`GetDesc`	A description of the contained texel data.
`GetDevice`	The containing device.

Properties	
`GetPrivateData`	Association of the volume with private data.
`SetPrivateData`	

Methods	
`FreePrivateData`	Removes the association with private data.
`LockBox`	Obtains direct access to the contained texel data.
`UnlockBox`	Releases direct access to the contained texel data.

```
interface IDirect3DVolume9 : IUnknown
{
    // read-only properties
    HRESULT GetContainer(REFIID iid,
                void **container);
    HRESULT GetDesc(D3DVOLUME_DESC *pDesc);
    HRESULT GetDevice(IDirect3DDevice9 **value);
```

```
    // read/write properties
    HRESULT GetPrivateData(REFGUID key,
                void *data,
                DWORD *size);
    HRESULT SetPrivateData(REFGUID key,
                const void *data,
                DWORD size,
                DWORD flags);

    // methods
    HRESULT FreePrivateData(REFGUID key);
    HRESULT LockBox(D3DLOCKED_BOX *data,
                const D3DBOX *volume,
                DWORD flags);
    HRESULT UnlockBox();
};
```

The volume interface derives directly from `IUnknown` and contains properties similar to those of a surface interface. The `GetContainer`, `GetDevice`, `GetPrivateData`, `SetPrivateData`, and `FreePrivateData` methods behave exactly as the corresponding methods on the `IDirect3DSurface9` or `IDirect3DResource9` interfaces.

The `GetDesc` method returns a description of the volume in a `D3DVOLUME_DESC` structure.

```
typedef struct _D3DVOLUME_DESC
{
    D3DFORMAT       Format;
    D3DRESOURCETYPE Type;
    DWORD           Usage;
    D3DPOOL         Pool;
    UINT            Size;
    UINT            Width;
    UINT            Height;
    UINT            Depth;
} D3DVOLUME_DESC;
```

This structure is similar to the `D3DSURFACE_DESC` structure, with the addition of the `Depth` member. The `Depth` member describes the extent of the volume in the third dimension, as shown in figure 11.4. The remaining members are identical in their semantics to the corresponding members of the `D3DSURFACE_DESC` structure.

The `LockBox` and `UnlockBox` methods provide direct access to the contained texel data. Each call to `LockBox` must be matched by a call to `UnlockBox`. Just as with surfaces or textures, we can specify a portion of the entire resource to be locked. With a volume the portion to be locked is specified by a `D3DBOX` structure. When the `volume` argument is `NULL`, the entire volume is locked.

```
typedef struct _D3DBOX
{
    UINT Left;
    UINT Top;
    UINT Right;
    UINT Bottom;
    UINT Front;
    UINT Back;
} D3DBOX;
```

The `Left`, `Top`, `Right`, and `Bottom` members correspond to the members of the same name in a `RECT` structure, while the `Front` and `Back` members describe the extent of the volume to be locked in the third dimension. When the volume is successfully locked, a pointer to the texel data is returned in a `D3DLOCKED_BOX` structure.

```
typedef struct _D3DLOCKED_BOX
{
    INT   RowPitch;
    INT   SlicePitch;
    void *pBits;
} D3DLOCKED_BOX;
```

Just as the pitch between scanlines must be observed when locking surfaces or textures, for locked volumes both the pitch between scanlines and the pitch between slices at different depths must be observed. The `RowPitch` member gives the pitch between scanlines within a slice, while the `SlicePitch` member gives the pitch between successive slices in the volume. You can think of each slice of the volume as an image.

The following code snippet locks a volume and fills it with opaque white.

```
D3DVOLUME_DESC desc;
THR(volume->GetDesc(&desc));
D3DLOCKED_BOX lb = { 0 };
THR(volume->LockBox(&lb, NULL, 0));
for (UINT z = 0; z < desc.Depth; z++)
{
    BYTE *plane = static_cast<BYTE *>(lb.pBits) +
        z*lb.SlicePitch;
    for (UINT y = 0; y < desc.Height; y++)
    {
        D3DCOLOR *scanline = reinterpret_cast<D3DCOLOR *>
            (plane + y*lb.RowPitch);
        for (UINT x = 0; x < desc.Width; x++)
        {
            scanline[x] = D3DCOLOR_XRGB(255, 255, 255);
        }
```

```
    }
}
THR(volume->UnlockBox());
```

As with other resources, we can create a helper class that ensures matched calls to `LockBox` and `UnlockBox`. The include file `<rt/texture.h>` in the sample code contains a helper class for locking volumes.

11.3.5 IDirect3DVolumeTexture9

The `IDirect3DVolumeTexture9` interface exposes a collection of mipmap levels for a volume texture, as shown in figure 11.5. The `CreateVolumeTexture` method on the device is used to create a volume texture resource.

```
HRESULT CreateVolumeTexture(UINT width,
            UINT height,
            UINT depth,
            UINT levels,
            DWORD usage,
            D3DFORMAT format,
            D3DPOOL pool,
            IDirect3DVolumeTexture9 **result);
```

A device supports volume textures if the `D3DPTEXTURECAPS_VOLUMEMAP` bit of `D3DCAPS9::TextureCaps` is set. The `usage` parameter must be zero. The `levels`, `format`, and `pool` parameters have the same semantics as the corresponding arguments to `CreateTexture`. A mipmapped volume can be created if the `D3DPTEXTURECAPS_MIPVOLUMEMAP` bit of `TextureCaps` is set. The `width`, `height` and `depth` parameters give the dimensions in texels of the most detailed mipmap level for the volume texture. The maximum value for all three dimensions of a volume texture is given by the `MaxVolumeExtent` member of `D3DCAPS9`. If the `D3DPTEXTURECAPS_VOLUMEMAP_POW2` bit of `TextureCaps` is set, then all three dimension arguments must be powers of two.

Interface 11.5: Summary of the `IDirect3DVolumeTexture9` interface.

IDirect3DVolumeTexture9

Read-Only Properties	
`GetLevelDesc`	A description of the texel data for a mipmap level.
`GetVolumeLevel`	The volume interface for a mipmap level.

Methods	
`AddDirtyBox`	Adds a box to the dirty region list.
`LockBox`	Obtains direct access to the contained texel data.
`UnlockBox`	Releases direct access to the contained texel data.

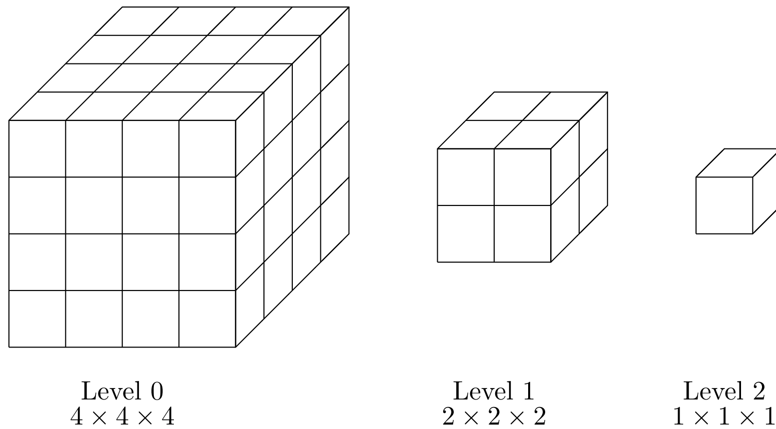


Figure 11.5: Organization of mipmapped volume textures. Each successive volume in the mipmap reduces the dimensions of the previous level by one half, stopping at a value of 1. Rectangular parallelpiped mipmap volume dimensions decrease as with cube volumes, with the smaller dimension decreasing to 1 until the larger dimension is also 1. A volume texture with dimensions $16 \times 8 \times 4$ for level zero would have mipmap levels of dimension $16 \times 8 \times 4$, $8 \times 4 \times 2$, $4 \times 2 \times 1$, $2 \times 1 \times 1$, and $1 \times 1 \times 1$.

```
interface IDirect3DVolumeTexture9 : IDirect3DBaseTexture9
{
    // read-only properties
    HRESULT GetLevelDesc(UINT level,
                D3DVOLUME_DESC *value);
    HRESULT GetVolumeLevel(UINT level,
                IDirect3DVolume9 **value);

    // methods
    HRESULT AddDirtyBox(const D3DBOX *volume);
    HRESULT LockBox(UINT level,
                D3DLOCKED_BOX *data,
                const D3DBOX *volume,
                DWORD flags);
    HRESULT UnlockBox(UINT level);
};
```

The methods in the `IDirect3DVolumeTexture9` interface mirror those in the `IDirect3DVolume9` interface, with the addition of a mipmap level parameter. As with other resources, we can define a helper class that ensures matched calls to `LockBox` and `UnlockBox`. The include file `<rt/texture.h>` in the sample code contains such a helper class for locking volume textures.

11.4 Texture Formats

As we mentioned in section 2.7, the D3DFORMAT enumeration describes the format of resource data, including texture resources. Textures are usually created using one of the RGB format types, but other format types apply equally well to textures. If the D3DPTEXTURECAPS_ALPHA bit of D3DCAPS9::TextureCaps is set, then the device supports textures containing an alpha channel.

luminance: A scalar value indicating an overall brightness.

The **luminance** formats, D3DFMT_A4L4, D3DFMT_L8, and D3DFMT_A8L8, define a luminance value with an optional alpha channel. The luminance value is replicated to all three color channels before any texture processing occurs. If a luminance texture has no associated alpha channel, Direct3D supplies an alpha value of 1.0. The D3DFMT_P8 and D3DFMT_A8P8 formats are pallette based formats, with the P channel containing indices into a pallete associated with the texture. The D3DFMT_DXTn formats are used to define texture resources whose texel data is stored in a compressed representation. The formats D3DFMT_L6V5U5, D3DFMT_V8U8, D3DFMT_A2W10V10U10, D3DFMT_Q8W8V8U8, D3DFMT_V16U16, D3DFMT_W11V11U10, and D3DFMT_X8L8V8U8 are used in bump-mapping techqniques described in the next chapter.

11.4.1 Palette Based Textures

Direct3D does not support indexed display modes, but it does support an indexed color format for texture resources. The formats D3DFMT_P8 and D3DFMT_A8P8 may be supported by the device for texture resources, as reported by CheckDeviceFormat. Devices expose an array of 16,384 palettes, with only one of the palettes being active at any one time. The palette is used only by the texturing units. The active palette is changed with the GetCurrentTexturePalette and SetCurrentTexturePalette methods. As with light numbers, an application should choose palette numbers beginning with zero and increasing sequentially to minimize the amount of memory used by the device.

```
HRESULT GetCurrentTexturePalette(UINT *value);
HRESULT SetCurrentTexturePalette(UINT value);
```

Each palette is an array of 256 PALETTEENTRY structures and is manipulated with the GetPaletteEntries and SetPaletteEntries methods. As discussed in chapter 1, PALETTEENTRY structures do not have a bit layout that corresponds to D3DCOLOR, so take care when assigning between the two color representations.

```
HRESULT GetPaletteEntries(UINT index,
            PALETTEENTRY *value);
HRESULT SetPaletteEntries(UINT index,
            const PALETTEENTRY *value);
```

Direct3D uses the peFlags member of the PALETTEENTRY structure to store the alpha value for the color. A device supports alpha in the palette colors if the D3DPTEXTURECAPS_ALPHAPALETTE bit of D3DCAPS9::TextureCaps is set.

$0, 3$	$1, 3$	$2, 3$	$3, 3$
$0, 2$	$1, 2$	$2, 2$	$3, 2$
$0, 1$	$1, 1$	$2, 1$	$3, 1$
$0, 0$	$1, 0$	$2, 0$	$3, 0$

Figure 11.6: Texel organization within a block for compressed textures. Shown are the coordinates of each texel within a block, encoded into `WORD`s as indicated in the text.

A palette-based texture format can reduce the amount of storage required for textures with a limited number of colors and allow an application to update the contents of a texture quickly, provided the update can be performed as a simple palette change. If the device does not support alpha values in the palette entries and the texture format does not contain an alpha channel, then an alpha value of 1.0 is supplied, producing completely opaque colors.

11.4.2 Compressed Texture Formats

One way to reduce the memory footprint of a texture is to use a texture with smaller dimensions, but that may adversely impact the image quality of the rendered scene. Another way to reduce the memory footprint of a texture is to store the texture in a compressed format. Direct3D provides the `D3DFMT_-DXT`n formats for storing textures in a compressed format. Most applications needn't be aware of the internal storage representation of a compressed texture and can use the functions in the D3DX library to convert between format representations. However, knowledge of the compression format is useful for writing off-line tools for producing compressed textures for later use by the application.

The formats can be roughly categorized into four groups: opaque textures (DXT1), textures with a single bit transparency mask per texel (DXT1), textures with explicit transparency for each texel (DXT2, DXT3) and textures with interpolated transparency for each texel (DXT4, DXT5).

All compressed texture formats encode data in 4×4 blocks of texels and all dimensions of compressed textures must be powers of two. When locking any resource in a compressed texture format, the `Pitch` member will reflect the pitch of 4 scanlines, i.e. a scanline of blocks. When locking a `RECT` of a resource in a compressed format, the coordinates of the `RECT` must be aligned to a 4×4 block boundary or an error will result.

White	White	Black	Black
Black	White	Black	Black
White	Black	Black	Black
Black	Black	Black	Black

Figure 11.7: Sample block of texels to be stored in a compressed texture format. The block encodes the numbers $0, \ldots, 3$ as a binary sequence, increasing from bottom to top. The binary sequence is left-justified within each scanline of the block.

DXT1

The `D3DFMT_DXT1` format encodes opaque textures and textures with a single bit transparency mask. Each 4×4 block of texels is encoded in a chunk of 64 bits resulting in 4 bits per texel. Each chunk contains two colors stored in 16 bits each and a value for each texel stored in 2 bits. Each texel in the block can be one of four values: either of the two colors stored explicitly in the chunk or a color interpolated between the two colors. The layout of the chunk is four `WORD`s as follows, with the most significant bit of each word on the left.

Word	**Bits**	**Contents**
0	`rrrrr gggggg bbbbb`	C_0: Color 0
1	`rrrrr gggggg bbbbb`	C_1: Color 1
2	`31 21 11 01 30 20 10 00`	Scanlines 1, 0
3	`33 23 13 03 32 22 12 02`	Scanlines 3, 2

Each color is stored as a pixel in `D3DFMT_R5G6B5` format. For the scanline data, the table shows the texel coordinates from figure 11.6 for each pixel within the word. Each texel is represented by a two bit code within the scanline data. The interpretation of the texel code depends on the relative magnitude of the two colors and is made on a block-by-block basis within the texture. If the first `WORD` is greater than the second `WORD` in the chunk, then the code designates 4 opaque texel colors; otherwise the code designates 3 opaque texel colors and transparent black.

$C_0 > C_1$		$C_0 \leq C_1$	
Code	**Color**	**Code**	**Color**
0	C_0	0	C_0
1	C_1	1	C_1
2	$\frac{2}{3}C_0 + \frac{1}{3}C_1$	2	$\frac{1}{2}C_0 + \frac{1}{2}C_1$
3	$\frac{1}{3}C_0 + \frac{2}{3}C_2$	3	$\langle 0, 0, 0, 0 \rangle$

As an example, consider the block of texels shown in figure 11.7 containing the colors opaque white and opaque black. This block would be encoded as an opaque chunk with the following `WORD`s:

Word	Value	Contents
0	`0xffff`	C_0: opaque white
1	`0x0000`	C_1: opaque black
2	`0x5455`	Scanlines 3, 2
3	`0x5051`	Scanlines 1, 0

If the opaque black pixels were made transparent, the encoded `WORD`s would be as follows:

Word	Value	Contents
0	`0x0000`	C_0: opaque black
1	`0xffff`	C_1: opaque white
2	`0xfdff`	Scanlines 3, 2
3	`0xf5f7`	Scanlines 1, 0

DXT2 and DXT3

The compressed texture formats `D3DFMT_DXT2` and `D3DFMT_DXT3` extend the concepts in `D3DFMT_DXT1` to include a per-texel alpha value. They are identical in all respects except for the interpretation of the alpha channel. In the DXT2 format, the texels are interpreted as having associated, or premultiplied, alpha while in DXT3 the texels are interpreted as having unassociated alpha. Recall from chapter 1 that an unassociated alpha color $\langle \alpha, r, g, b \rangle$ can have any of r, g, or b values larger than the α value. Associated alpha colors all have r, g, and b values less than or equal to the α channel value. Different formats are used to represent these two cases as the associativity of the alpha cannot be inferred directly from the data values.

Each 4×4 block of texels is encoded in two 64-bit chunks: first a 64-bit chunk describing the transparency of the pixels, followed by a 64-bit chunk encoding the color of the pixels. The color chunk is exactly the same as the 64-bit DXT1 chunk that encodes a 4×4 block of opaque pixels. The 64-bit alpha chunk encodes the transparency of each texel directly using 4 bits of alpha per texel, or four texels per `WORD`, with the pixels filled across the scanline from the least-significant bits first. Note that the scanline order for the alpha data is different from that of the colors in a DXT1 chunk.

Word	Bits	Contents
0	`3333 2222 1111 0000`	α Scanline 0
1	`3333 2222 1111 0000`	α Scanline 1
2	`3333 2222 1111 0000`	α Scanline 2
3	`3333 2222 1111 0000`	α Scanline 3

The 4×4 block of texels in figure 11.7 would be encoded in two 64-bit chunks as the following `WORD`s:

Word	Value	Contents
0	0x000f	Alpha Scanline 0
1	0x00f0	Alpha Scanline 1
2	0x0f00	Alpha Scanline 2
3	0xf000	Alpha Scanline 3
4	0xffff	C_0: opaque white
5	0x0000	C_1: opaque black
6	0x5455	Color Scanlines 3, 2
7	0x5051	Color Scanlines 1, 0

DXT4 and DXT5

The compressed texture formats D3DFMT_DXT4 and D3DFMT_DXT5 extend the concepts in D3DFMT_DXT1 to include two alpha values and a per-texel code that interpolates between the two alpha values. They are identical in all respects except for the interpretation of the alpha channel: in DXT4 the texels are interpreted as having associated, or premultiplied, alpha while in DXT5 the texels are interpreted as having unassociated alpha.

Each 4×4 block of texels is encoded in two 64-bit chunks: first a 64-bit chunk describing the transparency of the pixels, followed by a 64-bit chunk encoding the color of the pixels. The color chunk is exactly the same as the 64-bit DXT1 chunk that encodes a 4×4 block of opaque pixels. The 64-bit alpha chunk gives two explicit 8-bit BYTE alpha values and encodes each texel's alpha value by a 3-bit code. As three bits are used for each texel, the texel codes span BYTE boundaries within the chunk.

Byte	Bits	Contents
0	aaaaaaaa	α_0
1	aaaaaaaa	α_1
2	333 222 11	Scanline 0
3	1 000 333 2	Scanlines 0, 1
4	22 111 000	Scanline 1
5	333 222 11	Scanline 2
6	1 000 333 2	Scanlines 2, 3
7	22 111 000	Scanline 3

The interpretation of the 3-bit alpha code depends on the relative magnitude of the two alpha vlaues. If the first BYTE is greater than the second BYTE in the chunk, then the alpha codes identify eight distinct alpha values interpolated evenly between the two explicit values. Otherwise, the alpha codes identify six distinct alpha values interpolated evenly between the two explicit values and the additional values 0.0 and 1.0, for fully transparent and fully opaque α, respectively.

$\alpha_0 > \alpha_1$		$\alpha_0 \le \alpha_1$	
Code	**Alpha**	**Code**	**Alpha**
0	α_0	0	α_0
1	α_1	1	α_1
2	$\frac{6}{7}\alpha_0 + \frac{1}{7}\alpha_1$	2	$\frac{4}{5}\alpha_0 + \frac{1}{5}\alpha_1$
3	$\frac{5}{7}\alpha_0 + \frac{2}{7}\alpha_1$	3	$\frac{3}{5}\alpha_0 + \frac{2}{5}\alpha_1$
4	$\frac{4}{7}\alpha_0 + \frac{3}{7}\alpha_1$	4	$\frac{2}{5}\alpha_0 + \frac{3}{5}\alpha_1$
5	$\frac{3}{7}\alpha_0 + \frac{4}{7}\alpha_1$	5	$\frac{1}{5}\alpha_0 + \frac{4}{5}\alpha_1$
6	$\frac{2}{7}\alpha_0 + \frac{5}{7}\alpha_1$	6	0.0
7	$\frac{1}{7}\alpha_0 + \frac{6}{7}\alpha_1$	7	1.0

As an example, consider the block of texels in figure 11.7, with the black texels as transparent. The block would be encoded as the following sequence of `WORD`s in the DXT4 format:

Word	Value	Contents
0	`0x00ff`	α_0, α_1
1	`0x8249`	Alpha Scanlines 0, 1
2	`0x4124`	Alpha Scanlines 1, 2
3	`0x2402`	Alpha Scanlines 2, 3
4	`0xffff`	C_0: white
5	`0x0000`	C_1: black
6	`0x5455`	Color Scanlines 1, 0
7	`0x5051`	Color Scanlines 3, 2

11.5 Texture Coordinate Processing

Now that we've become acquainted with the different texture resources, let's take a look at how texture coordinates are processed to address texels within those resources. As we saw in section 5.6, texture coordinates can be included with a vertex, either through FVF flags or a vertex shader declaration. Conceptually a texture is addressed by coordinates in the unit interval $[0, 1]$, as shown in figure 11.8. A "linear" texture, addressed by a single texture coordinate value, is not exposed as an explicit resource type: it is just a rectangular texture with a height of one. A cube texture is slightly different — it is addressed by coordinates in the range $[-1, 1]$, with the coordinate having the largest magnitude selecting the face of the cube and the remaining two coordinates addressing within the selected face.

After a primitive has been projected into screen space through vertex processing, rasterization interpolates the texture coordinates across the primitive. At this point, for a texture coordinate u Direct3D knows the change in the texture coordinate, Δu, across each pixel from rasterization. If we scale the change in texture coordinates by the appropriate dimension of the texture we will know how many texels correspond to this source pixel. Using that information we can select an appropriate mipmap level to use for texturing this source pixel.

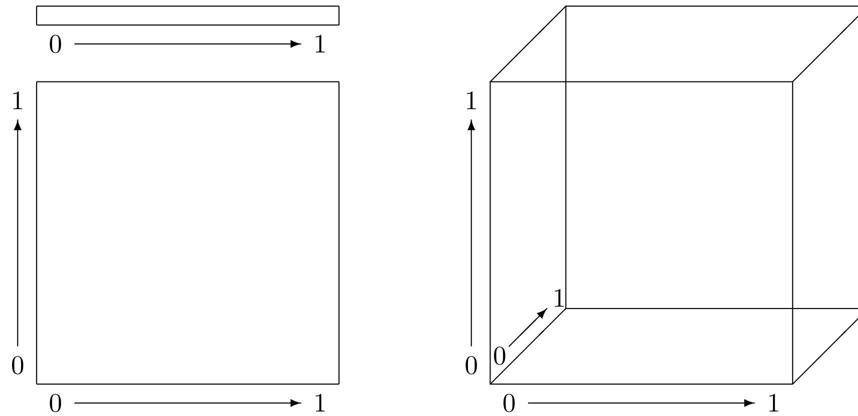


Figure 11.8: Texture address space for linear, rectangular and volume textures. Each texture is conceptually addressed by coordinates in the unit interval $[0, 1]$. A cube texture is addressed slightly differently: the largest coordinate is used to select a face of the cube and the remaining two texture coordinates address texels within the face, as in a rectangular texture.

As an example, suppose we have a square texture with mipmap levels 16×16, 8×8, 4×4, 2×2 and 1×1 addressed by two-dimensional texture coordinates u and v. If the change in texture coordinates across a source pixel were $\Delta u = 0.0625, \Delta v = 0.0625$, then this source pixel corresponds to a 1×1 texel area in the texture ($0.0625 \times 16 = 1$). Consequently Direct3D takes texels from level 0 of the mipmap to produce a texel value for this source pixel. If $\Delta u = 0.125, \Delta v = 0.125$, then the source pixel would correspond to a 2×2 texel area in the texture and Direct3D would take texels from level 1 of the mipmap to produce a texel value.

At times we may want to map texels directly to screen pixels, such as when drawing user interface elements with textured primitives. To achieve a one-to-one mapping between texels and source pixels, the primitive coordinates should be offset by $\frac{1}{2}$ relative to the screen space location. This is due to the rasterization rules for triangles discussed in chapter 10, where integer coordinates address the center of a screen space pixel. The offset to the triangle's coordinate aligns a texel with a screen space pixel. The effect is most easily obtained by using transformed vertices whose position is already in screen space. The following excerpt from `app.cpp` in the `rt_FrameBuffer` sample application from chapter 14 computes vertices to tile the back buffer with a texture.

```
// build s_screen_vertex structures for stippling the window
m_stipple_verts = 0;
m_num_stipple_quads = (width + STIPPLE_SIZE - 1)/STIPPLE_SIZE*
    (height + STIPPLE_SIZE - 1)/STIPPLE_SIZE;
```

```
THR(m_pd3dDevice->CreateVertexBuffer(m_num_stipple_quads*
    6*sizeof(s_screen_vertex), D3DUSAGE_WRITEONLY,
    s_screen_vertex::FVF, D3DPOOL_MANAGED, &m_stipple_verts));
rt::vertex_lock<s_screen_vertex> verts(m_stipple_verts);
s_screen_vertex *quad = verts.data();
for (y = 0; y < height; y += STIPPLE_SIZE)
{
    for (UINT x = 0; x < width; x += STIPPLE_SIZE)
    {
        const float right = min(x + STIPPLE_SIZE, width);
        const float bottom = min(y + STIPPLE_SIZE, height);

        quad[0].x = x + 0.5f;       quad[0].y = y + 0.5f;
        quad[0].z = 0.99f;          quad[0].rhw = 0.99f;
        quad[0].u = 0;              quad[0].v = 1;

        quad[1].x = right + 0.5f;   quad[1].y = y + 0.5f;
        quad[1].z = 0.99f;          quad[1].rhw = 0.99f;
        quad[1].u = 1;              quad[1].v = 1;

        quad[2].x = x + 0.5f;       quad[2].y = bottom + 0.5f;
        quad[2].z = 0.99f;          quad[2].rhw = 0.99f;
        quad[2].u = 0;              quad[2].v = 0;

        quad[3].x = x + 0.5f;       quad[3].y = bottom + 0.5f;
        quad[3].z = 0.99f;          quad[3].rhw = 0.99f;
        quad[3].u = 0;              quad[3].v = 0;

        quad[4].x = right + 0.5f;   quad[4].y = y + 0.5f;
        quad[4].z = 0.99f;          quad[4].rhw = 0.99f;
        quad[4].u = 1;              quad[4].v = 1;

        quad[5].x = right + 0.5f;   quad[5].y = bottom + 0.5f;
        quad[5].z = 0.99f;          quad[5].rhw = 0.99f;
        quad[5].u = 1;              quad[5].v = 0;

        quad += 6;
    }
}
```

Multiple sets of texture coordinates can be included with a vertex. By default the first set of texture coordinates is used for the first texture stage, the second set is used for the second stage and so-on. This mapping of texture coordinate sets to texture stages can be changed with TSS Tex Coord Index. This allows multiple texture stages to use the same texture coordinate set. TSS Tex Coord Index for a stage is set to the zero-based index of the texture coordinate to be

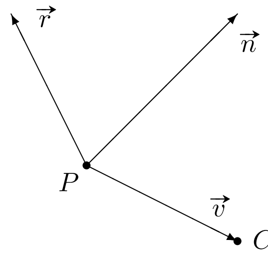


Figure 11.9: Illustration of the reflection vector for a vertex. The vector $\vec{v}$ is from the vertex position P to the camera position C, the origin of camera space. The reflection vector $\vec{r}$ makes an angle to the surface normal $\vec{n}$ equal to that made by $\vec{v}$, but reflected about $\vec{n}$.

used for that stage.

11.5.1 Texture Coordinate Generation

In addition to specifying texture coordinates as a vertex component, Direct3D can generate texture coordinates from the vertex directly. A device supports the generation of texture coordinates from the vertex if the `D3DVTXPCAPS_TEXGEN` bit of `D3DCAPS9::VertexProcessingCaps` is set. The coordinates can be generated from the vertex position, the vertex normal, or the reflection vector of the vertex, all in camera space. Recall that camera space is the coordinate system resulting from the application of the world and view matrices. The reflection vector is the reflection about the vertex normal of the vector from the vertex position to the camera, as shown in figure 11.9.

To generate texture coordinates, set TSS Tex Coord Index to one of the following values. The `D3DTSS_TCI_PASSTHRU` value is shown for completeness and corresponds to the situation where a texture coordinate set index is used.

```
#define D3DTSS_TCI_PASSTHRU                      0x00000000
#define D3DTSS_TCI_CAMERASPACENORMAL             0x00010000
#define D3DTSS_TCI_CAMERASPACEPOSITION           0x00020000
#define D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR 0x00030000
```

11.5.2 Texture Coordinate Transformation

Just as a transformation matrix allowed us to reposition and reorient primitives without editing the vertices, texture coordinates can also be transformed by a matrix during vertex processing. Each stage has its own transformation matrix identified by the `D3DTS_TEXTURE`n enumerants of `D3DTRANSFORMSTATETYPE`. The `GetTransform`, `SetTransform`, and `MultiplyTransform` methods are used to manipulate the matrix.

Texture transformation matrices are slightly different from other transformation matrices on the device. For a one-dimensional texture coordinate, 2×2

submatrix in the upper-left of the matrix is used for the transformation. Similarly, for two-dimensional texture coordinates, the 3×3 submatrix in the upper-left is used for the transformation. The exact elements of the matrix used for transforming the texture coordinate are determined by the dimensionality of the texture coordinate set and the value of TSS Texture Transform Flags. This texture stage state has one of the values from the `D3DTEXTURETRANSFORMFLAGS` enumeration.

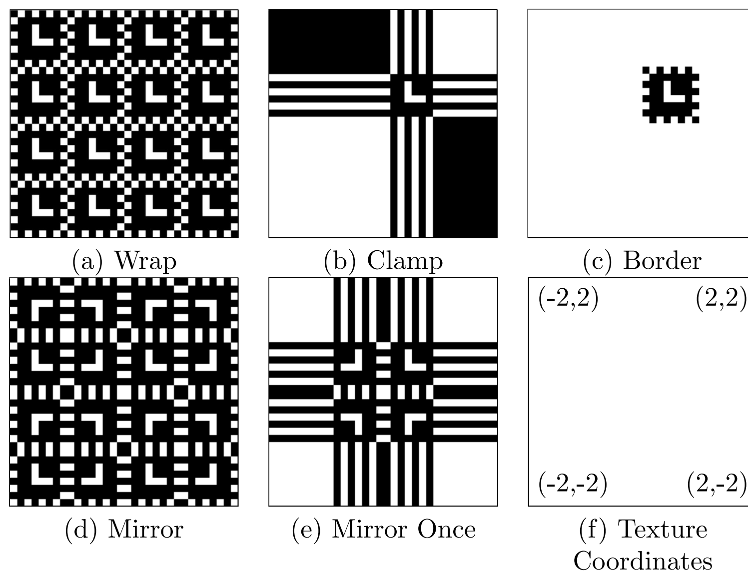
```
typedef enum _D3DTEXTURETRANSFORMFLAGS {
    D3DTTFF_DISABLE   = 0,
    D3DTTFF_COUNT1    = 1,
    D3DTTFF_COUNT2    = 2,
    D3DTTFF_COUNT3    = 3,
    D3DTTFF_COUNT4    = 4,
    D3DTTFF_PROJECTED = 256
} D3DTEXTURETRANSFORMFLAGS;
```

`D3DTTFF_DISABLE` disables texture coordinate transformation and the texture matrix for the stage is ignored. The `D3DTTFF_COUNT`n values instruct the rasterizer that n-dimensional texture coordinates result from the transformation.

The count can be bitwise ored with the `D3DTTFF_PROJECTED` value to indicate that the resulting coordinates are homogeneous coordinates and not cartesian coordinates. Homogeneous texture coordinates are converted to cartesian coordinates before the texture is addressed with the coordinate set. A device supports projective texture coordinates by performing the divide per-pixel when the `D3DPTEXTURECAPS_PROJECTED` bit of `TextureCaps` is set. Otherwise, the Direct3D runtime performs the divide per-vertex.

As an example, consider a vertex with two-dimensional texture coordinates. With TSS Texture Transform Flags set to `D3DTTFF_COUNT2`, the upper 2×2 submatrix of the texture transformation matrix will be used. With three-dimensional texture coordinates, such as those resulting from automatic texture coordinate generation, and TSS Texture Transform Flags set to `D3DTTFF_COUNT2`, the upper left 2×3 submatrix will be used.

We can use regular 4×4 matrix functions to generate any necessary matrices and then shuffle the matrix elements to suit the dimensionality of the texture coordinate set and the value of TSS Texture Transform Flags. We can use a custom class to construct the matrix in the appropriate form. For example, here is a class that constructs the appropriate 2×2 submatrix from a standard 4×4 transformation matrix. It works by extending the D3DX matrix class to construct the appropriate matrix.



(a) Wrap (b) Clamp (c) Border (d) Mirror (e) Mirror Once (f) Texture Coordinates

Figure 11.10: Texture coordinate addressing modes values for the SS Address U, SS Address V and SS Address W texture stage states. A black and white texture with texture coordinates spanning the interval $[-2, 2]$ in both u and v is shown. The border color is white.

```
class mat_tex1d : public D3DXMATRIX
{
public:
    mat_tex1d(const D3DXMATRIX &rhs)
        : D3DXMATRIX(rhs._11, rhs._14, 0, 0,
                     rhs._41, rhs._44, 0, 0,
                     0, 0, 1, 0,
                     0, 0, 0, 1)
    {}
};
```

Similar classes can be used to construct other special forms of texture transformation matrices. The include file `<rt/mat.h>` in the sample code contains several classes for constructing texture coordinate transformation matrices in this manner.

11.5.3 Texture Addressing

So far we have discussed texture coordinates drawn from the unit interval $[0, 1]$. You can use texture coordinates outside the unit interval as well. The range of coordinates outside the unit interval accepted by a device is given by the `MaxTextureRepeat` member of `D3DCAPS9`. If this value is m, then the device

supports texture coordinates in the interval $[-m, m]$. If the D3DPTEXTURECAPS_-TEXREPEATNOTSCALEDBYSIZE bit of TextureCaps is not set, then the device scales texture coordinates to the size of the texture before interpolation and the MaxTextureRepeat value indicates the range accepted by the device after this scaling takes place.

Texture coordinates outside the unit interval are processed based on the texture addressing mode set for the stage. SS Address U, SS Address V, and SS Address W control the addressing modes for the first, second and third texture coordinate values, respectively, for a texture stage. There is no addressing mode specified for the fourth coordinate, which is used only for projective texturing of three-dimensional textures. Each of these texture stage states can be one of the values in the D3DTEXTUREADDRESS enumeration.

```
typedef enum _D3DTEXTUREADDRESS {
    D3DTADDRESS_WRAP       = 1,
    D3DTADDRESS_MIRROR     = 2,
    D3DTADDRESS_CLAMP      = 3,
    D3DTADDRESS_BORDER     = 4,
    D3DTADDRESS_MIRRORONCE = 5
} D3DTEXTUREADDRESS;
```

The default texture addressing mode for all coordinates is D3DTADDRESS_-WRAP, where a coordinate outside $[0, 1]$ is mapped to the unit interval by modulo arithmetic. With the wrap address mode, a texture coordinate of 1.75, -0.25 or 0.75 address the same position within a texture. Mathematically, the original coordinate u is transformed into the coordinate u' before addressing the texture as follows:

$$u' = \begin{cases} u + \lceil u \rceil, & u < 0 \\ u, & 0 \leq u \leq 1 \\ u - \lfloor u \rfloor, & 1 < u \end{cases}$$

The ceiling function $\lceil u \rceil$ returns the smallest integer k larger than the argument u. Similarly, the floor function $\lfloor u \rfloor$ returns the largest integer k smaller than the argument u. The clamp addressing mode clamps a texture coordinate to the interval $[0, 1]$:

$$u' = \begin{cases} 0, & u < 0 \\ u, & 0 \leq u \leq 1 \\ 1, & 1 < u \end{cases}$$

The border addressing mode is similar to clamp, but the texture color itself is substituted with the D3DCOLOR value of SS Border Color when the texture coordinate is outside the unit interval.

The mirror address mode is similar to the wrap address mode, but at each repetition of the texture space it is mirrored. The mirror once address mode

mirrors the texture only once, about the zero point of the unit interval. The different addressing modes are illustrated in figure 11.10. Mathematically, for the mirror addressing mode the texture coordinate u is transformed into the coordinate u' before addressing the texture as follows:

$$u' = \begin{cases} u - \lfloor u \rfloor, & k \leq \lfloor u \rfloor \leq k+1, \quad k \text{ even} \\ 1 - u + \lceil u \rceil, & k \leq \lceil u \rceil \leq k+1, \quad k \text{ odd} \end{cases}$$

The mirror once addressing mode transforms the texture coordinate value u into the value u' as follows:

$$u' = \begin{cases} 1, & u < -1 \\ -u, & -1 \leq u \leq 0 \\ u, & 0 < u \leq 1 \\ 1, & 1 < u \end{cases}$$

The `TextureAddressCaps` and `VolumeTextureAddressCaps` members of `D3DCAPS9` give the addressing mode support for the device. `TextureAddressCaps` applies to rectangular and cubemap textures, while `VolumeTextureAddressCaps` applies only to volume textures. Both members are a combination of one or more flags indicating the texture addressing modes supported. The device supports independent addressing modes for the texture coordinates if the `D3DTADDRESSCAPS_INDEPENDENTUV` bit is set, otherwise the device supports only a single addressing mode for all texture coordinates, given by SS Address U.

```
#define D3DPTADDRESSCAPS_WRAP           0x00000001L
#define D3DPTADDRESSCAPS_MIRROR         0x00000002L
#define D3DPTADDRESSCAPS_CLAMP          0x00000004L
#define D3DPTADDRESSCAPS_BORDER         0x00000008L
#define D3DPTADDRESSCAPS_INDEPENDENTUV  0x00000010L
#define D3DPTADDRESSCAPS_MIRRORONCE     0x00000020L
```

11.5.4 Texture Wrapping

Texture wrapping changes how the rasterizer interpolates the texture coordinates across a primitive. Be careful not to confuse texture wrapping with the wrap texture addressing mode. Texture wrapping is controlled for each texture stage by the render states RS Wrap 0, ..., RS Wrap 15. When wrapping is enabled for a texture coordinate, the rasterizer interpolates the coordinate across a primitive with the shortest possible path between the two coordinate values, treating the texture space as an infinitely repeating grid, as illustrated in figure 11.11.

For example, suppose a line segment had texture coordinates of 0.1 and 0.9 at its two endpoints. With wrapping disabled, the rasterizer would interpolate the coordinate u in the range $[0.1, 0.9]$, spanning a distance of 0.8 in texture space. However, with coordinate wrapping enabled, the rasterizer would interpolate

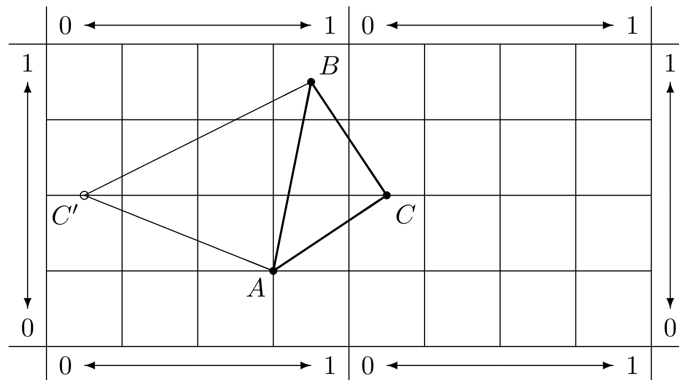


Figure 11.11: Texture coordinate wrapping with the RS Wrap 0, ..., RS Wrap 15 render states. A triangle with the texture coordinates $A = (0.75, 0.25)$, $B = (0.875, 0.875)$ and $C = (0.125, 0.5)$ is shown with wrapping enabled for the horizontal coordinate. The triangle $\triangle ABC$ spans the portion of texture space drawn when wrapping is enabled, while the triangle $\triangle ABC'$ spans the portion of texture space drawn when wrapping is disabled.

the coordinate u in the range $[0.1, 0]$ followed by the range $[1, 0.9]$, spanning a distance of 0.2 in texture space.

The RS Wrap 0, ..., RS Wrap 15 render states are set to a combination of zero or more flags to enable wrapping for a particular coordinate on a texture stage. To disable texture coordinate wrapping completely for a given stage, set the appropriate render state to zero. The D3DWRAPCOORD_0, D3DWRAPCOORD_1, and D3DWRAPCOORD_2 values are synonyms for the values D3DWRAP_U, D3DWRAP_V and D3DWRAP_W, respectively.

```
#define D3DWRAP_U 0x00000001L
#define D3DWRAP_V 0x00000002L
#define D3DWRAP_W 0x00000004L

#define D3DWRAPCOORD_0 0x00000001L
#define D3DWRAPCOORD_1 0x00000002L
#define D3DWRAPCOORD_2 0x00000004L
#define D3DWRAPCOORD_3 0x00000008L
```

11.6 Texture Sampling

In section 11.3, we mentioned that there were two sources of aliasing with texturing: the creation of texture samples and the sampling of the reconstructed function represented by the texture samples. In section 11.5, we saw how texels can be selected from a mipmap level based on the change in texture coordinates across a source pixel. The selection of a texel value for a source pixel is a

sampling operation on the texture.

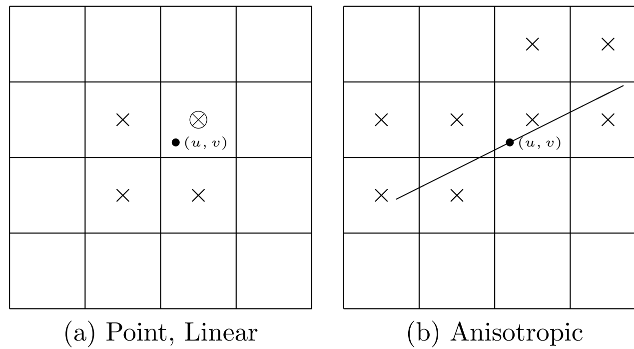
If the amount of texture space spanned by a source pixel is exactly a power of two in all dimensions of the texture and we have a mipmapped texture level corresponding to that power of two, then the sampling can select a value from the mipmap directly. Most of the time, a source pixel will span some other amount of texture space. Direct3D provides three texture stage states to control the sampling of the texture to produce a texel value used in texture processing: SS Min Filter when the texture is minified, SS Mag Filter when the texture is magnified, and SS Mip Filter when the texture is mipmapped. Each of these render states can be one of the values of the `D3DTEXTUREFILTERTYPE` enumeration.

```
typedef enum _D3DTEXTUREFILTERTYPE
{
    D3DTEXF_NONE          = 0,
    D3DTEXF_POINT         = 1,
    D3DTEXF_LINEAR        = 2,
    D3DTEXF_ANISOTROPIC   = 3,
    D3DTEXF_FLATCUBIC     = 4,
    D3DTEXF_GAUSSIANCUBIC = 5
} D3DTEXTUREFILTERTYPE;
```

During minification, SS Min Filter specifies how multiple texel samples should be combined to produce a single texel value. You will recall that we can create a partial mipmap, so it is possible that the texture will be compressed more than the smallest mipmap level available for the texture. During magnification, SS Mag Filter specifies how to interpolate between texel samples. Again, we can magnify a texture beyond the most detailed level available for the texture. Instead of selecting texel samples from only a single mipmap level, we can perform the appropriate minified or magnified filtering on two adjacent mipmap levels and combine them with the filter specified by SS Mip Filter. Combining texels from two mipmaps provides a smooth change in appearance as a primitive moves in a scene and the amount of texture minification or magnification changes.

Not all of the enumerants of `D3DTEXTUREFILTERTYPE` apply to all the filtering texture stage states. `D3DTEXF_POINT` and `D3DTEXF_LINEAR` can be used for minification, magnification and mipmap filtering. `D3DTEXF_NONE` is used to disable mipmap filtering. `D3DTEXF_ANISOTROPIC` can be used for minification or magnification filtering, while the filters `D3DTEXF_PYRAMIDALQUAD` and `D3DTEXF_GAUSSIANQUAD` can only be used for magnification filtering.

Filter	**Minification**	**Magnification**	**Mipmap**
None	No	No	Yes
Point	Yes	Yes	Yes
Linear	Yes	Yes	Yes
Anisotropic	Yes	Yes	No
Flat Cubic	No	Yes	No
Gaussian Cubic	No	Yes	No



(a) Point, Linear (b) Anisotropic

Figure 11.12: Texture minification filtering with the SS Min Filter texture stage state. (a) The point addressed by the texture coordinates (u, v) selects the texel marked $\otimes$ with point filtering to produce the sampled texel. When linear filtering is selected the 2×2 region (texels marked $\otimes$ and $\times$) around the addressed point is combined by linear interpolation to produce the sampled texel. (b) With anisotropic filtering, the line of anisotropy for the source pixel is projected into texture space. The texels nearest the line of anisotropy are combined by interpolation along the line of anisotropy. In this figure, the anisotropy has a value of approximately 2. The more the anisotropy, the more texels are combined by linear interpolation to produce the sampled texel.

A point filter selects the nearest texel or mipmap level corresponding to the texture coordinate for the source pixel. The linear filter, when used for minification or magnification, takes a 2×2 neighbourhood of texels and uses the texture coordinate to interpolate between the four pixels. Point and linear filtering for minification are shown in figure 11.12. Linear mipmap filtering takes samples from the two nearest mipmap levels and interpolates between them. In linear mipmap filtering, the minification and magnification filters are first applied to obtain the two texel values taken from the two mipmap levels. The cubic filters provide for a more accurate interpolation between texel values during magnification, using a 3×3 texel neighbourhood from the magnified mipmap level.

Anisotropic filtering samples a non-square area of the texture, based on the change in texture coordinates across the source pixel. This allows for more accurate filtering when the texture is compressed more along one coordinate than another, such as when a plane tilted relative to the view plane is viewed in perspective. A device supports anisotropic filtering if the `D3DPRASTERCAPS_ANISOTROPY` bit of `RasterCaps` is set. SS Max Anisotropy controls the maximum amount aspect ratio of the anisotropic filtering region. The larger the value in this texture stage state, the more rectangular an area will be filtered to produce a texel value when `D3DTEXF_ANISOTROPIC` is selected. The maximum value for SS Max Anisotropy that is supported by the device is given by the `MaxAnisotropy` member of `D3DCAPS9`.

The supported texture filters vary with the type of texture resource and the filtering texture stage stage. The `TextureFilterCaps`, `CubeTextureFilterCaps` and `VolumeTextureFilterCaps` members of `D3DCAPS9` describe the filters supported for the corresponding type of texture resource. They can be zero or more of the following flags.

```
#define D3DPTFILTERCAPS_MINFPOINT         0x00000100L
#define D3DPTFILTERCAPS_MINFLINEAR        0x00000200L
#define D3DPTFILTERCAPS_MINFANISOTROPIC   0x00000400L
#define D3DPTFILTERCAPS_MIPFPOINT         0x00010000L
#define D3DPTFILTERCAPS_MIPFLINEAR        0x00020000L
#define D3DPTFILTERCAPS_MAGFPOINT         0x01000000L
#define D3DPTFILTERCAPS_MAGFLINEAR        0x02000000L
#define D3DPTFILTERCAPS_MAGFANISOTROPIC   0x04000000L
#define D3DPTFILTERCAPS_MAGFAFLATCUBIC    0x08000000L
#define D3DPTFILTERCAPS_MAGFGAUSSIANCUBIC 0x10000000L
```

Mipmap filtering is also influenced by the level of detail set on a texture via the `SetLOD` method on `IDirect3DBaseTexture9` and the values of the texture stage states SS Max Mip Level and SS Mip Map LOD Bias. SS Max Mip Level specifies the highest mipmap level number that will be used during texture sampling. When this value is zero, all mipmap levels are available for use. This texture stage state can be set for textures in all resource pools, while `SetLOD` is only available for managed textures. SS Max Mip Level is bounded from above by the number of mipmap levels in the texture resource.

SS Mip Map LOD Bias allows an application to shift the mipmap level selection towards higher mipmap levels to artifically "blur" the texture, or towards lower mipmap levels to artficially "sharpen" the texture. In reality, these are not true blur or sharpen operations as performed in image processing, they only bias the texturing units towards mipmap levels with smaller or larger dimensions, respectively. The value is a `float`, with a value of 1.0 shifting the mipmap selection to the next smaller level of detail, and a value of -1.0 shifting the mipmap selection to the next larger level of detail. A device supports SS Mip Map LOD Bias if the `D3DPRASTERCAPS_MIPMAPLODBIAS` bit of `RasterCaps` is set.

There is no one set of texture sampling state that works best for all applications, or even for all textures within a single application. The sample program for this chapter allows you to interactively explore the affects of texture sampling state on the appearance of a texture. In general, linear filtering for minification, magnification and mipmap filtering provides high quality results at the cost of additional processing. Point sampling is quick, but results in aliasing. Anisotropic filtering may be required to acceptably render fine details in primitives not parallel to the viewing plane. For textures that are often magnified, the cubic magnification filters can provide superior quality.

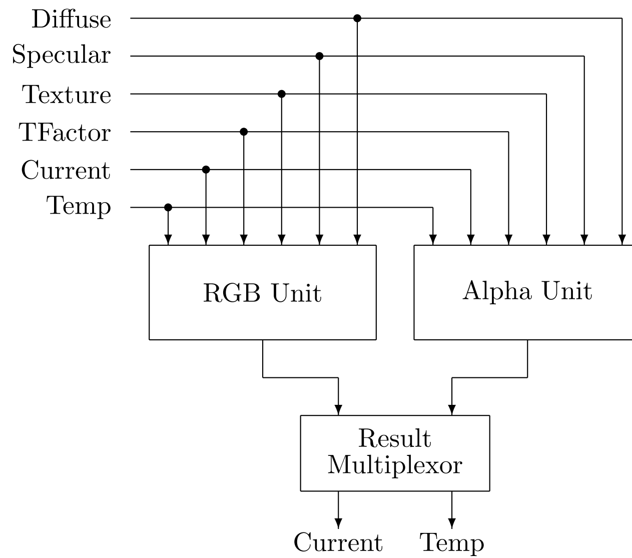


Figure 11.13: Texture stage processing.

11.7 Texture Stage Processing

After a texture has been addressed and sampled, a texel value is produced. This texel value can be combined with other inputs to the texture stage to produce a diffuse color value for the source pixel. The diffuse color value can also be passed on to subsequent stages for further processing, as discussed in the next chapter. Each texture stage contains an RGB processing unit that produces the color output and an alpha processing unit that produces the alpha output. While the API exposes the two units independently, they must always be controlled in tandem. You should always set the RGB and alpha unit state together for a stage. A common mistake is to set the state for one and forget to set the state for the other unit.

Each unit can select up to three arguments from any of six inputs that are combined through an operation to produce an output. The result of the RGB and alpha units for the stage can then be sent to one of two possible destinations for further processing. The flow of data from one stage to the next is illustrated in figure 11.13.

The six available inputs are: the interpolated diffuse color from the rasterizer, the interpolated specular color from the rasterizer, the texel sampled from the texture, the value of RS Texture Factor, the result of the previous texture stage, or the contents of the temporary register texture stage output. Each of these inputs is identified by one of the following values. RS Texture Factor is a `D3DCOLOR` value. `D3DTA_SELECTMASK` is a mask giving the bits used to select an input.

```
#define D3DTA_SELECTMASK      0x0000000f
#define D3DTA_DIFFUSE         0x00000000
#define D3DTA_CURRENT         0x00000001
#define D3DTA_TEXTURE         0x00000002
#define D3DTA_TFACTOR         0x00000003
#define D3DTA_SPECULAR        0x00000004
#define D3DTA_TEMP            0x00000005
```

Each of the input arguments to a processing unit can be combined with zero or more of the following modifier flags. D3DTA_COMPLEMENT inverts the argument C to supply $1 - C$ as the input. D3DTA_ALPHAREPLICATE takes the argument $\langle a, r, g, b\rangle$ and supplies $\langle a, a, a, a\rangle$ as the input.

```
#define D3DTA_COMPLEMENT      0x00000010
#define D3DTA_ALPHAREPLICATE 0x00000020
```

The RGB and alpha units each can compute functions of up to three arguments. Each argument is bound to one of the six inputs with a texture stage state: TSS Color Arg 0, TSS Color Arg 1, TSS Color Arg 2, TSS Alpha Arg 0, TSS Alpha Arg 1, and TSS Alpha Arg 2. The functions to be computed for the RGB and alpha units are designated by the TSS Color Op and TSS Alpha Op states. The operations that can be performed are given by the D3DTEXTUREOP enumeration.

```
typedef enum _D3DTEXTUREOP
{
    D3DTOP_DISABLE                    = 1,
    D3DTOP_SELECTARG1                 = 2,
    D3DTOP_SELECTARG2                 = 3,
    D3DTOP_MODULATE                   = 4,
    D3DTOP_MODULATE2X                 = 5,
    D3DTOP_MODULATE4X                 = 6,
    D3DTOP_ADD                        = 7,
    D3DTOP_ADDSIGNED                  = 8,
    D3DTOP_ADDSIGNED2X                = 9,
    D3DTOP_SUBTRACT                   = 10,
    D3DTOP_ADDSMOOTH                  = 11,
    D3DTOP_BLENDDIFFUSEALPHA          = 12,
    D3DTOP_BLENDTEXTUREALPHA          = 13,
    D3DTOP_BLENDFACTORALPHA           = 14,
    D3DTOP_BLENDTEXTUREALPHAPM        = 15,
    D3DTOP_BLENDCURRENTALPHA          = 16,
    D3DTOP_PREMODULATE                = 17,
    D3DTOP_MODULATEALPHA_ADDCOLOR     = 18,
    D3DTOP_MODULATECOLOR_ADDALPHA     = 19,
    D3DTOP_MODULATEINVALPHA_ADDCOLOR = 20,
    D3DTOP_MODULATEINVCOLOR_ADDALPHA = 21,
```

```
    D3DTOP_BUMPENVMAP              = 22,
    D3DTOP_BUMPENVMAPLUMINANCE     = 23,
    D3DTOP_DOTPRODUCT3             = 24,
    D3DTOP_MULTIPLYADD             = 25,
    D3DTOP_LERP                    = 26
} D3DTEXTUREOP;
```

The mathematical function computed by most of these operators is summarized in table 11.1. The bump mapping operators are discussed in the next chapter. D3DTOP_DISABLE disables processing on a texture stage and must be set on both the RGB and alpha units of a stage or an error will result. The modulating operators D3DTOP_MODULATEALPHA_ADDCOLOR, D3DTOP_MODULATECOLOR_ADDALPHA, D3DTOP_MODULATEINVALPHA_ADDCOLOR and D3DTOP_MODULATEINVCOLOR_ADDALPHA can only be specified for the RGB unit with TSS Color Op. An error results if an attempt is made to specify these operations on the alpha unit.

The select operators are handy for quickly selecting an input as the resulting color. For instance, the following settings use the texture color and alpha directly as the resulting diffuse color and alpha:

TSS Color Arg 1 = D3DTA_TEXTURE
TSS Color Op = D3DTOP_SELECTARG1
TSS Alpha Arg 1 = D3DTA_TEXTURE
TSS Alpha Op = D3DTOP_SELECTARG1

Most of the texture operations are functions of two arguments as given by TSS Color Arg 1, TSS Color Arg 2, TSS Alpha Arg 1 and TSS Alpha Arg 2. The two triadic operations D3DTOP_MULTIPLYADD and D3DTOP_LERP take the third argument from TSS Color Arg 0 and TSS Alpha Arg 0. A device indicates basic support for the various texture operations through the TextureOpCaps member of D3DCAPS9. Each supported operation corresponds to a set bit in TextureOpCaps. A device always supports D3DTOP_DISABLE so that texturing can be disabled if necessary.

```
#define D3DTEXOPCAPS_DISABLE                    0x00000001L
#define D3DTEXOPCAPS_SELECTARG1                 0x00000002L
#define D3DTEXOPCAPS_SELECTARG2                 0x00000004L
#define D3DTEXOPCAPS_MODULATE                   0x00000008L
#define D3DTEXOPCAPS_MODULATE2X                 0x00000010L
#define D3DTEXOPCAPS_MODULATE4X                 0x00000020L
#define D3DTEXOPCAPS_ADD                        0x00000040L
#define D3DTEXOPCAPS_ADDSIGNED                  0x00000080L
#define D3DTEXOPCAPS_ADDSIGNED2X                0x00000100L
#define D3DTEXOPCAPS_SUBTRACT                   0x00000200L
#define D3DTEXOPCAPS_ADDSMOOTH                  0x00000400L
#define D3DTEXOPCAPS_BLENDDIFFUSEALPHA          0x00000800L
#define D3DTEXOPCAPS_BLENDTEXTUREALPHA          0x00001000L
#define D3DTEXOPCAPS_BLENDFACTORALPHA           0x00002000L
```

Operation	**Formula** $f(A, B, C)$
Select Arg 1	A
Select Arg 2	B
Modulate	AB
Modulate 2X	$2AB$
Modulate 4X	$4AB$
Add	$A + B$
Add Signed	$A + B - 0.5$
Add Signed 2X	$2(A + B - 0.5)$
Subtract	$A - B$
Add Smooth	$A + (1 - A)B$
Blend Current Alpha	$A\alpha_c + B(1 - \alpha_c)$
Blend Diffuse Alpha	$A\alpha_d + B(1 - \alpha_d)$
Blend Factor Alpha	$A\alpha_f + B(1 - \alpha_f)$
Blend Texture Alpha	$A\alpha_t + B(1 - \alpha_t)$
Blend Texture Alpha PM	$A + B(1 - \alpha_t)$
Modulate Alpha Add Color	$\langle r_a + \alpha_a r_b,\ g_a + \alpha_a g_b,\ b_a + \alpha_a b_b \rangle$
Modulate Color Add Alpha	$\langle r_a r_b + \alpha_a,\ g_a g_b + \alpha_a,\ b_a b_b + \alpha_a \rangle$
Modulate Inverse Alpha Add Color	$\langle r_a + f r_b,\ g_a + f g_b,\ b_a + f b_b \rangle$, $f = 1 - \alpha_a$
Modulate Inverse Color Add Alpha	$\langle r_f r_b + \alpha_a,\ g_f g_b + \alpha_a,\ b_f b_b + \alpha_a \rangle$, $\langle r_f,\ g_f,\ b_f \rangle = \langle 1 - r_a,\ 1 - g_a,\ 1 - b_a \rangle$
Dot Product 3	$\langle f,\ f,\ f \rangle$, $f = r_a r_b + g_a g_b + b_a b_b$
Multiply Add	$AB + C$
Lerp	$AB + (1 - A)C$

Table 11.1: Summary of operations computed in the RGB and alpha units of a texture stage. $A = \langle \alpha_a,\ r_a,\ g_a,\ b_a \rangle$ is the argument selected by TSS Color Arg 1 or TSS Alpha Arg 1. $B = \langle \alpha_b,\ r_b,\ g_b,\ b_b \rangle$ is the argument selected by TSS Color Arg 2 or TSS Alpha Arg 2. C is the argument selected by TSS Color Arg 0 or TSS Alpha Arg 0. AB represents $\langle r_a r_b,\ g_a g_b,\ b_a b_b \rangle$ or $\langle \alpha_a \alpha_b \rangle$, the color or alpha resulting from the component-wise product of the two colors A and B. α_c, α_d, α_f and α_t represent the alpha component of the `D3DTA_CURRENT`, `D3DTA_DIFFUSE`, `D3DTA_FACTOR`, `D3DTA_TEXTURE` inputs, respectively. Not all operations apply to both the RGB and alpha units; see the text for the details. `D3DTOP_PREMODULATE`, `D3DTOP_BUMPENVMAP`, and `D3DTOP_BUMPENVMAPLUMINANCE` are bump mapping operations discussed in chapter 12.

```
#define D3DTEXOPCAPS_BLENDTEXTUREALPHAPM       0x00004000L
#define D3DTEXOPCAPS_BLENDCURRENTALPHA         0x00008000L
#define D3DTEXOPCAPS_PREMODULATE               0x00010000L
#define D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR    0x00020000L
#define D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA    0x00040000L
#define D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR 0x00080000L
#define D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA 0x00100000L
#define D3DTEXOPCAPS_BUMPENVMAP                0x00200000L
#define D3DTEXOPCAPS_BUMPENVMAPLUMINANCE       0x00400000L
#define D3DTEXOPCAPS_DOTPRODUCT3               0x00800000L
#define D3DTEXOPCAPS_MULTIPLYADD               0x01000000L
#define D3DTEXOPCAPS_LERP                      0x02000000L
```

The inputs D3DTA_CURRENT and D3DTA_TEMP represent two distinct data paths that flow through the texture cascade: the current input value and a temporary input value. For stage zero, the current input is initialized to the interpolated diffuse color for the source pixel and the temporary input is initialized to transparent black ($\langle 0, 0, 0, 0\rangle$). The temporary input allows a multistage computation where two of the input values to a successive stage are computed by previous stages. One stage computes its value to the current data path and a different stage computes its value to the temporary data path, while a third stage combines the results of the two previous stages. The temporary input is supported by a device if the D3DPMISCCAPS_TSSARGTEMP bit of PrimitiveMiscCaps is set, otherwise D3DTA_TEMP cannot be used.

Each stage can send the result of its computation to either the current data path or the temporary data path for use by successive stages. TSS Result Arg can be either D3DTA_CURRENT or D3DTA_TEMP to specify the destination for the stage's computed value.

While Direct3D does provide information about the supported texturing capabilities in the D3DCAPS9 structure, it would be prohibitive to enumerate every allowed combination of arguments and texture operation in the capabilities. Instead, Direct3D provides the ValidateDevice method to validate the existing state of the device. Most often this is used for validating a combination of texture stage states, but ValidateDevice checks all device state for validity, not just the state of the texture stages.

```
HRESULT ValidateDevice(DWORD *passes);
```

The passes argument returns the number of rendering passes required to perform the texturing operation currently set in the device. If this value is larger than one, it usually indicates that the requested operation is using more simultaneous textures than the device supports and the operation must be decomposed into multiple passes to achieve the end result. For instance, alpha blending can be used to add the source pixels from rendering with the destination pixels in the frame buffer. So a two stage texturing operation that performed an addition of two terms could be achieved as a two-pass single stage

rendering operation by using the frame buffer to add the two values together. Alpha blending operations are discussed in chapter 14.

If `ValidateDevice` fails, the return value can be one of the following errors indicating the lack of support for the requested operation. These are not the only failure codes that could be reported by `ValidateDevice`. When checking `HRESULT`s, always check for general failure in addition to any specific failure codes.

`D3DERR_CONFLICTINGTEXTUREFILTER` The selected texture filters (minification, magnification, mipmap) cannot be used together.

`D3DERR_CONFLICTINGTEXTUREPALETTE` The current textures cannot be used simultaneously. Some devices that support palette based textures require that only a single palette based texture be used at any one time.

`D3DERR_TOOMANYOPERATIONS` The device has a limited amount of texture filtering capacity and the requested combination of minification, magnification and mipmap filtering exceeds the capacity of the device.

`D3DERR_UNSUPPORTEDALPHAARG` One of the values for TSS Alpha Arg 0, TSS Alpha Arg 1, or TSS Alpha Arg 2 is unsupported.

`D3DERR_UNSUPPORTEDALPHAOPERATION` The value of TSS Alpha Op is unsupported.

`D3DERR_UNSUPPORTEDCOLORARG` One of the values for TSS Color Arg 0, TSS Color Arg 1, or TSS Color Arg 2 is unsupported.

`D3DERR_UNSUPPORTEDCOLOROPERATION` The value of TSS Color Op is unsupported.

`D3DERR_UNSUPPORTEDFACTORVALUE` The device does not support the value specified for RS Texture Factor.

`D3DERR_UNSUPPORTEDTEXTUREFILTER` One of the values specified for SS Min Filter, SS Mag Filter or SS Mip Filter is unsupported.

`D3DERR_WRONGTEXTUREFORMAT` One of the textures set on a stage is an unsupported format.

Unfortunately there are no definite rules of which texture stage configurations will always work. Many hardware vendors use the texturing capabilities of their hardware to differentiate themselves within the marketplace and within their own product line. However, there are some guidelines that will increase the likelihood that your texture stage configuration will pass `ValidateDevice`.

1. Select the diffuse input as Arg2 (color or alpha).

2. Select the texture input as Arg1 (color or alpha).

3. Use the diffuse input in only the first or last texture stage of a multistage operation.

4. Use similar operations for the color and alpha units in a stage.

When `ValidateDevice` does fail because of a limitation of the hardware, the best way to deal with the situation is to fall back to a simpler rendering strategy. This may involve switching from a multistage single pass rendering to single stage multipass rendering. It may mean that some of your sophisticated effects achieved with texturing are simply disabled on less capable hardware. The exact solution varies from application to application, with the general advice of falling back to simpler renderings when possible. A simpler rendering is generally preferable to a user than a dialog box complaining that the application requires fancier hardware.

11.8 Examples

Because the arguments and operation for the RGB and alpha units in a texture stage are typically set all at once we can define inline functions to set these three texture stage states in tandem.

```
inline HRESULT
set_color_op(IDirect3DDevice9 *device, UINT stage,
             DWORD arg1, D3DTEXTUREOP op, DWORD arg2)
{
    HRESULT hr = device->SetTextureStageState(stage,
        D3DTSS_COLORARG1, arg1);
    if (FAILED(hr)) return hr;
    hr = device->SetTextureStageState(stage,
        D3DTSS_COLORARG2, arg2);
    if (FAILED(hr)) return hr;
    return device->SetTextureStageState(stage,
        D3DTSS_COLOROP, op);
}

inline HRESULT
set_alpha_op(IDirect3DDevice9 *device, UINT stage,
             DWORD arg1, D3DTEXTUREOP op, DWORD arg2)
{
    HRESULT hr = device->SetTextureStageState(stage,
        D3DTSS_ALPHAARG1, arg1);
    if (FAILED(hr)) return hr;
    hr = device->SetTextureStageState(stage,
        D3DTSS_ALPHAARG2, arg2);
    if (FAILED(hr)) return hr;
    return device->SetTextureStageState(stage,
```

```
        D3DTSS_ALPHAOP, op);
}
```

Using these functions, we can set the RGB and alpha units in a couple of lines of code. The following snippet modulates the diffuse color by the texture's color as the resulting color and selects the diffuse alpha as the resulting alpha.

```
set_color_op(device, 0,
             D3DTA_TEXTURE, D3DTOP_MODULATE, D3DTA_DIFFUSE);
set_alpha_op(device, 0,
             D3DTA_TEXTURE, D3DTOP_SELECTARG2, D3DTA_DIFFUSE);
```

There is quite a bit of state that can be set on each texture stage resulting in many different possibilities. How do you determine the right settings to achieve a particular effect? Start by writing down a formula that represents what you want to compute at each pixel. Write down separate equations that compute the color and the alpha values for the effect. Next, identify the computations in your formula and how they could be expressed as texture stage operations. There is a limited amount of work you can do in a single texture stage; more sophisticated techniques require the use of multiple stages and are discussed in the next chapter.

In each of the following examples, we show the formula for each of the texture configurations and then the code that sets the configuration. The equations in each example use $D = \langle \alpha_d, r_d, g_d, b_d \rangle$ as the diffuse color and $T = \langle \alpha_t, r_t, g_t, b_t \rangle$ as the texture color and C' as the color resulting from the texture stage processing.

11.8.1 Single Stage Light Maps

In chapter 8, we saw how the diffuse and specular colors for an object were computed from the light in the environment. Looking back at those formulas you can see that the total reflected light is a sum of components, where each component models some affect of light from the environment on the object. Instead of computing the reflection information dynamically from light structures and vertex positions, we can precompute this information and store it in a texture. Such a texture is called a "light map" and can provide very sophisticated lighting effects on low resolution geometry. Direct3D computes the lighting only at vertices, so if we wanted detailed lighting effects we needed a dense mesh of vertices to sufficiently sample the lighting.

With the lighting computed directly into the texture, we simply replace the diffuse color with the texture contents. Since we're computing the light from the texture, we turn off lighting to avoid unnecessary work.

$$C' = C_t = \langle \alpha_t, r_t, g_t, b_t \rangle$$

```
device->SetRenderState(D3DRS_LIGHTING, FALSE);
set_color_op(device, 0,
             D3DTA_TEXTURE, D3DTOP_SELECTARG1, D3DTA_DIFFUSE);
set_color_op(device, 0,
             D3DTA_TEXTURE, D3DTOP_SELECTARG1, D3DTA_DIFFUSE);
```

However, because we've encoded the object's diffuse color directly into the texture map, we'd have to recompute the contents of the texture map if we want to change the color of the object. The light map doesn't really need the object's color in it – all we really need is the amount of the diffuse color reflected by the object. If we encode the object's reflectance into an eight bit alpha channel of the texture map, where 0 indicates no reflectance and 255 indicates full reflectance, we can use this to modulate the interpolated diffuse color. We'll also change our example slightly to use the alpha from the diffuse color so that we can have transparent objects illuminated by our light map. We've used the alpha replicate modifier so that each color channel in the diffuse color is modulated by the alpha value from the texture.

$$C' = \langle \alpha_d, \alpha_t r_d, \alpha_t g_d, \alpha_t b_d \rangle$$

```
set_color_op(device, 0, D3DTA_TEXTURE | D3DTA_ALPHAREPLICATE,
             D3DTOP_MODULATE, D3DTA_DIFFUSE);
set_alpha_op(device, 0,
             D3DTA_TEXTURE, D3DTOP_SELECTARG2, D3DTA_DIFFUSE);
```

This monochrome light map modulates all the color channels equally, which is fine for a white light, but we may have colored lights in our scene that will modulate the different channels in the diffuse color by varying amounts. If we modulate the diffuse color by the texture color in the RGB unit, we can achieve this effect.

$$C' = \langle \alpha_d, r_t r_d, g_t g_d, b_t b_d \rangle$$

```
set_color_op(device, 0,
             D3DTA_TEXTURE, D3DTOP_MODULATE, D3DTA_DIFFUSE);
set_alpha_op(device, 0,
             D3DTA_TEXTURE, D3DTOP_SELECTARG2, D3DTA_DIFFUSE);
```

11.8.2 Decals

Sometimes you want to render an object with its lighting and material properties and "paste" an image onto the surface of the object, as if the image were a decal applied to the object. One way to achieve this in a single pass is to use

the texture border color as the color for outside the decal and draw the object with the appropriate texture coordinates so that the border color appears on the object everywhere except where the texture is defined. This works well when only a simple color is desired for the portion of the object outside the texture; the border color won't be used in any lighting computations by Direct3D as texturing occurs in the pipeline after lighting.

Another approach is to use a two-pass rendering technique. The first pass draws the object with its lighting and material properties but no texturing. A second pass is then drawn with only the texture and the appropriate texture coordinates on the object to position the decal properly on the model. The Z buffer is disabled for writing and the test changed from D3DCMP_LESS to D3DCMP_LESSEQUAL during the second pass to prevent "Z fighting" artifacts between the pixels of the first and second pass. However, the Z test is still enabled so that occluded decal pixels will still be rejected by the Z buffer. The details of the operation of the Z buffer are described in chapter 14.

A single pass approach can be used when D3DTOP_BLENDTEXTUREALPHA is supported. We can use the alpha from the texture to blend the texture color over top of the diffuse color, preserving the computed lighting in the diffuse component while still rendering in a single pass.

$$C' = \langle \alpha_d,\, r_d(1-\alpha_t) + r_t\alpha_t,\, g_d(1-\alpha_t) + g_t\alpha_t,\, b_d(1-\alpha_t) + b_t\alpha_t \rangle$$

```
set_color_op(device, 0,
    D3DTA_TEXTURE, D3DTOP_BLENDTEXTUREALPHA, D3DTA_DIFFUSE);
set_alpha_op(device, 0,
    D3DTA_TEXTURE, D3DTOP_SELECTARG2, D3DTA_DIFFUSE);
```

If the colors in the texture represent colors with associated alpha we can use D3DTOP_BLENDTEXTUREALPHAPM.

$$C' = \langle \alpha_d,\, r_d(1-\alpha_t) + r_t,\, g_d(1-\alpha_t) + g_t,\, b_d(1-\alpha_t) + b_t \rangle$$

```
set_color_op(device, 0,
    D3DTA_TEXTURE, D3DTOP_BLENDTEXTUREALPHAPM, D3DTA_DIFFUSE);
set_alpha_op(device, 0,
    D3DTA_TEXTURE, D3DTOP_SELECTARG2, D3DTA_DIFFUSE);
```

11.9 Specular Addition

After any texture processing has been performed, the diffuse color produced by the result of the multitexture cascade can be combined with the interpolated specular color component. RS Specular Enable controls not only the generation specular lighting and its interpolation, it also controls the combination of the diffuse and specular colors by addition. When this render state is set to TRUE, the specular color is added to the result of the multitexture cascade. This can eliminate the use of a texture stage that is used only to perform the addition of the specular component.

11.10 Textured Primitives

Textured triangles and quadrilaterals are often used to emulate other kinds of "primitives" that are not provided directly by Direct3D. Here we briefly discuss some of the primitives that can be constructed in this manner.

Sprites

There are several different ways that an application can draw a **sprite**. In chapter 5, we mentioned that point sprites were textured squares. Point sprites are affected by the texture stage states during rendering and require a texture to be set on the device in order to render properly. Point sprites are demonstrated by the `PointSprite` sample in the SDK.

sprite: A small textured rectangle parallel to the view plane.

Point sprites may not be supported by the hardware, or the range of texture coordinates used by point sprites – effectively dedicating an entire texture to a single sprite – may be too limiting for your application. In that situation, you can draw sprites as two textured triangles arranged to form a quadrilateral – or any other shape necessary for your sprite.

Another alternative to the point sprite primitive is to use the `ID3DXSprite` interface provided by D3DX. This interface provides a sprite as commonly found in two-dimensional applications and is described in section 17.4.

Billboards

A billboard is very similar to a point sprite, but instead of having Direct3D align the primitive parallel to the viewing plane, the application arranges for this directly by appropriately orienting the primitive during rendering. The Billboard sample in the SDK demonstrates the technique.

Impostors

When we draw a complex piece of geometry that is projected to a small screen-space area, most of the geometric detail isn't discernible. When such a piece of geometry is repeated many times in a scene, we are doing lots of geometric processing for little visual benefit. We can achieve the same discernible level of detail by rendering the geometry into a texture render target and substituting a textured quadrilateral with the rendered image for the geometry in the scene. The textured quadrilateral is referred to as an "imposter" for the real geometry.

Text

Direct3D doesn't contain a text primitive, so how do we draw text in a scene? Text can be drawn with a vector font as a line list or line strip, but this results in poor quality text. Another approach is to rasterize the character glyphs in a font into a texture image and then draw each character as a textured quadrilateral or triangle with the appropriate texture coordinates and texture stage state.

This approach works well for 8 bit fonts of moderate size where the memory requirement for the texture is reasonable. A more sophisticated "glyph caching" approach may be needed for large fonts, or Asian fonts with many glyphs where the memory requirement is prohibitive. The `CD3DFont` class in the SDK sample framework uses this approach.

Another alternative is to use GDI to render text into a bitmap, copy the bitmap contents to a texture and then render a textured quad. As this approach uses GDI, it correctly handles the issue of Asian and other 16 bit fonts as well as properly handling kerning between characters within a text string. The `ID3DXFont` interface provided by D3DX operates in this manner. However, the additional quality of rendering comes at the cost of calling GDI and transferring the bitmap contents to a texture every time the text is drawn.

A third alternative does not use texturing at all, but uses the parametric glyph outline of a character in a TrueType font to create an extruded triangle mesh directly from the glyph outline data. This results in true three-dimensional text that can be oriented and shaded with Direct3D. The D3DX function `D3DX-CreateText` creates such a mesh from a TrueType font and a text string. An application can also extract the glyph data directly using the GDI function `::GetGlyphOutline` to construct other variations, such as a text outline with line primitives.

The `rt_Text` sample draws text in all three styles for comparison. The Text3D sample in the SDK demonstrates the use of `CD3DFont` to draw text as a texture on a quadrilateral that is not aligned with the viewing plane.

Wide Lines

While Direct3D provides a line primitive, it does not provide a way to draw lines wider than a single pixel. Using a combination of billboarding and texturing, we can draw lines with arbitrary widths. The idea is to render each line segment as a quadrilateral parallel to the view plane of the appropriate width in screen space.

Textured Clipping

A texture can also be used to clip arbitrary portions of primitives. The idea here is to use a texture as a mask that modulates the diffuse color of the primitive. The mask is oriented on the primitive by the texture coordinates used by the texture stage – automatically generated texture coordinates and the texture transformation matrix are particularly useful here. The mask texture is opaque white where the primitive should show and transparent black where the primitive should be masked. The texture operation modulates the diffuse color with the texture, leaving the original diffuse color unchanged in the white portions and replacing the original diffuse with transparent black in the black portions. The alpha test, as described in chapter 14, can then be used to reject the transparent pixels, giving the appearance of clipped geometry. This technique is particularly useful when user clip planes are not supported by the device.

11.11 Further Reading

A significant amount of the progress towards real-time photorealistic rendering is being achieved through the use of texturing. New uses and ideas for texturing are constantly being published at conferences such as the Game Developer's Conference (GDC), or the Association for Computing Machinery's Special Interest Group on Computer Graphics (SIGGRAPH). Proceedings for these conferences are published every year and contain many of the newest ideas in computer graphics.

The book series Graphics Gems and its sucessor, the Journal of Graphic Tools provide an ongoing forum for the dissemination of useful techniques. The Game Programming Gems series is similar, but contains other techniques besides those dealing with rendering for use in game applications.

GDC Conference Proceedings. Available through the GDC web site. ⟨`http://www.gdconf.com/`⟩

SIGGRAPH Conference Proceedings. Available through the SIGGRAPH web site. ⟨`http://www.siggraph.org/`⟩

Pyramidal Parametrics, Williams, Lance. Computer Graphics, Vol. 7, No. 3, July 1983, pp. 1-11. This is the paper that introduced mipmaps and it discusses the details of mipmapping and the necessary filtering to avoid aliasing.

High Performance Rendering Using the Talisman Architecture, Barkans, Anthony C., Proceedings of the 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 79-88, August 1997. This paper describes the details of anisotropic filtering.

Textured Lines In D3D, Terdiman, Pierre. Gives a code snippet for drawing wide, textured lines. ⟨`http://www.flipcode.com/cgi-bin/msg.cgi?showThread=COTD-TexturedLinesInD3D&forum=cotd&id=-1`⟩

Texture Based Clipping Demo, Dunlop, Robert. Demonstrates the use of a texture to clip geometry. ⟨`http://www.mvps.org/directx/articles/clipcube.htm`⟩

11.12 Sample Applications

Texturing is such a basic technique and so widely available on modern graphics hardware that it appears in almost every graphics program. The following subsections briefly describe the use of texturing in the samples for this book and the SDK before describing the sample application for the chapter.

11.12.1 Book Samples

Many of the samples accompanying this book utilize textures. The main sample for this chapter is `rt_Texture` described later in this section. A brief description of other samples that use textures is given here.

`rt_D3DXSprite` Demonstrates sprite drawing with the `ID3DXSprite` interface.

`rt_FrameBuffer` Uses a texture to create a stipple pattern in the stencil planes.

`rt_GingerBread` Demonstrates automatic texture coordinate generation with point primitives.

`rt_IFS` Visualizes a three-dimensional iterated function system (IFS) fractal, using a stack of planes with three-dimensional texture coordinates and a volume texture. The planes are drawn parallel to the viewer and the IFS fractal is stored in the volume texture. The fractal is oriented by applying transformations to the texture coordinates while keeping the orientation of the planes fixed.

`rt_Text` Similar to the Text3D sample in the SDK, this sample also demonstrates rendering text with the `ID3DXFont` interface.

11.12.2 SDK Samples

Almost every sample in the SDK utilizes texturing. The following is a brief summary of the SDK samples that use basic texturing.

Billboard Renders a scene containing a textured terrain populated with textured trees. Each tree is drawn as a billboard.

DxTex A texture format conversion tool. This tool is useful for examining the contents of `.dds` files, which can contain any of the texture formats or resource types defined by Direct3D.

MFCTex An MFC application that allows you to interactively explore the combinations of TSS Color Arg 1, TSS Color Arg 2, TSS Color Op, TSS Alpha Arg 1, TSS Alpha Arg 2 and TSS Alpha Op for up to three texture stages. It can also produce a code snippet that sets the currently selected items on a texture stage.

PointSprite Draws a simple scene containing some fireworks, where the firework effect is achieved using point sprites.

Text3D Demonstrates two-dimensional and three-dimensional text via `CD3DFont` as well as text drawn as a mesh.

VolumeTexture Demonstrates a simple use of a volume texture.

11.12.3 rt_Texture Sample Application

Finally, we close the chapter with a sample program that demonstrates how to create, fill and use a single texture. The program lets you explore the options for the minification, magnification and mipmap filters supported by the device. Texture addressing modes can be independently selected for each texture coordinate and scenes with one-, two- and three-dimensional textures can be selected.

The scene consists of a textured quadrilateral that can be oriented with the use of the arrow keys on the keyboard. Changing the orientation allows you to see the effect of mipmapping and filtering as the texture distortion changes across the quadrilateral. The maximum level of detail for the texture set with SetLOD can be interactively changed as well as the value of SS Max Mip Level and SS Mip Map LOD Bias. The level of anisotropy can also be changed interactively.

This sample offers a "split-screen" mode for visualizing the effect of texture filtering and sampling state on the resulting quality of the rendering. The left side of the screen is drawn with the selected filtering, while the right side is drawn with point sampling only.

Shown here is the "interesting" portion of the program, with the remainder of the code in the sample source accompanying this book. A number of helper classes, also included in the sample code, are used to make common tasks easier. Each helper class or function is in the rt namespace and prefixed with the namespace to make their origin clear in the code. The C++ standard library is used along with D3DX to handle some of the low-level details.

Listing 11.1: rt_Texture.cpp: Demonstration of texturing.

```
1   ///////////////////////////////////////////////////////////
2   // rt_Texture.cpp
3   //
4   // Demonstrates the following texture related pipeline
5   // features: texture filtering, texture addressing,
6   // automatic mipmap generation, texture formats, texture
7   // stage states, filling textures with HLSL functions
8   //
9
10  // C++ includes
11  #include <algorithm>
12  #include <cmath>
13  #include <sstream>
14  #include <vector>
15
16  // Win32 includes
17  #define STRICT
18  #include <windows.h>
19  #include <commctrl.h>
20  #include <commdlg.h>
```

```
21  #include <basetsd.h>
22  #include <tchar.h>
23
24  // ATL includes
25  #include <atlbase.h>
26
27  // Direct3D includes
28  #include <d3dx9.h>
29
30  // D3DFrame includes
31  #include "DXUtil.h"
32  #include "D3DEnumeration.h"
33  #include "D3DSettings.h"
34  #include "D3DApp.h"
35  #include "D3DFont.h"
36  #include "D3DUtil.h"
37
38  // RT includes
39  #include "rt/app.h"
40  #include "rt/hr.h"
41  #include "rt/mat.h"
42  #include "rt/media.h"
43  #include "rt/mesh.h"
44  #include "rt/misc.h"
45  #include "rt/states.h"
46  #include "rt/surface.h"
47  #include "rt/texture.h"
48  #include "rt/tint.h"
49  #include "rt/tstring.h"
50
51  // sample includes
52  #include "resource.h"
53  #include "rt_Texture.h"
54
55  // vertex with 1D texture coordinate
56  struct s_vertex_1d
57  {
58      float m_pos[3];
59      float m_tex;
60
61      static const DWORD FVF;
62  };
63
64  // vertex with 2D texture coordinate
65  struct s_vertex_2d
66  {
```

```
67      float m_pos[3];
68      float m_tex[2];
69
70      static const DWORD FVF;
71  };
72
73  // vertex with 3D texture coordinate
74  struct s_vertex_3d
75  {
76      float m_pos[3];
77      float m_tex[3];
78
79      static const DWORD FVF;
80  };
81
82  // functor object used with std::transform
83  // to set z & w values in a 3D vertex
84  class fn_set_z_w
85  {
86  public:
87      fn_set_z_w(float z = 0.f, float w = 0.f)
88          : m_z(z), m_w(w)
89      {}
90      ~fn_set_z_w()
91      {}
92
93      s_vertex_3d operator()(const s_vertex_3d &vtx)
94      {
95          s_vertex_3d result = vtx;
96          result.m_pos[2] = m_z;
97          result.m_tex[2] = m_w;
98          return result;
99      }
100
101 private:
102     float m_z;
103     float m_w;
104 };
105
106 // functor object used with std::generate
107 // to generate a sequence of increasing integers
108 // as vertex indices fed to D3DXCreateMeshFVF
109 class gen_integers
110 {
111 public:
112     gen_integers(int start = 0)
```

```
113             : m_pos(start)
114         {}
115         ~gen_integers()
116         {}
117
118         int operator()()
119         {
120             return m_pos++;
121         }
122
123     private:
124         int m_pos;
125     };
126
127     // 1D coords: mesh is a square
128     static const s_vertex_1d g_verts_1d[6] =
129     {
130         -1.5, -1.5, 0,  -2,
131          1.5,  1.5, 0,   2,
132          1.5, -1.5, 0,   2,
133
134         -1.5, -1.5, 0,  -2,
135         -1.5,  1.5, 0,  -2,
136          1.5,  1.5, 0,   2
137     };
138
139     // 2D coords: mesh is a square
140     static const s_vertex_2d g_verts_2d[6] =
141     {
142         -1.5, -1.5, 0,  -2,  2,
143          1.5,  1.5, 0,   2, -2,
144          1.5, -1.5, 0,   2,  2,
145
146         -1.5, -1.5, 0,  -2,  2,
147         -1.5,  1.5, 0,  -2, -2,
148          1.5,  1.5, 0,   2, -2
149     };
150
151     // 3D coords: mesh is a stack of planes,
152     // with the bottom-most plane defined here
153     // and other planes generated with code.
154     static const s_vertex_3d g_verts_3d[] =
155     {
156         // bottom-most plane
157         -1.5, -1.5, 1,  -2,  2, -2,
158          1.5,  1.5, 1,   2, -2, -2,
```

```
159       1.5, -1.5, 1,   2,  2, -2,
160
161      -1.5, -1.5, 1,  -2,  2, -2,
162      -1.5,  1.5, 1,  -2, -2, -2,
163       1.5,  1.5, 1,   2, -2, -2
164  };
165
166  // FVF initializers for the vertex types.
167  const DWORD s_vertex_1d::FVF =
168      D3DFVF_XYZ | D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE1(0);
169  const DWORD s_vertex_2d::FVF =
170      D3DFVF_XYZ | D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE2(0);
171  const DWORD s_vertex_3d::FVF =
172      D3DFVF_XYZ | D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE3(0);
173
174  ////////////////////////////////////////////////////////////
175  // CMyD3DApplicatoin::texture_usage
176  //
177  // Return the appropriate D3DUSAGE based on the setting of
178  // automatic mipmap generation and support for mipmap
179  // generation in the current texture format.
180  //
181  DWORD
182  CMyD3DApplication::texture_usage(D3DRESOURCETYPE kind)
183  {
184      const DWORD usage = m_autogen_mipmaps ? D3DUSAGE_AUTOGENMIPMAP : 0;
185      return SUCCEEDED(m_pD3D->CheckDeviceFormat(
186          m_d3dSettings.AdapterOrdinal(),
187          m_d3dSettings.DevType(),
188          m_d3dSettings.DisplayMode().Format,
189          usage, kind, m_scenes[m_scene].m_format)) ? usage : 0;
190  }
191
192  ////////////////////////////////////////////////////////////
193  // CMyD3DApplication::init_1d_texture
194  //
195  // Create a 1D texture consisting of a hue ramp
196  //
197  void
198  CMyD3DApplication::init_1d_texture()
199  {
200      CComPtr<IDirect3DTexture9> texture;
201      if (TS_FILE == m_scenes[0].m_source)
202      {
203          // get texture width from file info
204          D3DXIMAGE_INFO info;
```

```
205            THR(::D3DXGetImageInfoFromFile(
206                m_scenes[0].m_filename.c_str(), &info));
207            THR(::D3DXCreateTexture(m_pd3dDevice, info.Width,
208                1, 0, texture_usage(D3DRTYPE_TEXTURE),
209                m_scenes[0].m_format, D3DPOOL_MANAGED,
210                &texture));
211            CComPtr<IDirect3DSurface9> surf;
212            THR(texture->GetSurfaceLevel(0, &surf));
213            THR(::D3DXLoadSurfaceFromFile(surf, NULL, NULL,
214                m_scenes[0].m_filename.c_str(), NULL,
215                D3DX_DEFAULT, 0, NULL));
216            if (m_autogen_mipmaps)
217            {
218                THR(texture->SetAutoGenFilterType(m_autogen_filter));
219            }
220            else
221            {
222                THR(::D3DXFilterTexture(texture, NULL, 0, D3DX_DEFAULT));
223            }
224        }
225        else
226        {
227            // create the texture
228            THR(::D3DXCreateTexture(m_pd3dDevice, 256, 1, 0,
229                texture_usage(D3DRTYPE_TEXTURE),
230                m_scenes[0].m_format, D3DPOOL_MANAGED,
231                &texture));
232            if (m_autogen_mipmaps)
233            {
234                CComPtr<IDirect3DTexture9> single;
235                THR(::D3DXCreateTexture(m_pd3dDevice, 256, 1, 1,
236                    0, m_scenes[0].m_format, D3DPOOL_SCRATCH,
237                    &single));
238                THR(::D3DXFillTextureTX(single,
239    m_scenes[0].m_texture_shader));
240                CComPtr<IDirect3DSurface9> source, dest;
241                THR(single->GetSurfaceLevel(0, &source));
242                THR(texture->GetSurfaceLevel(0, &dest));
243                THR(::D3DXLoadSurfaceFromSurface(dest, NULL, NULL,
244                    source, NULL, NULL, D3DX_FILTER_NONE, 0));
245                THR(texture->SetAutoGenFilterType(m_autogen_filter));
246            }
247            else
248            {
249                THR(::D3DXFillTextureTX(texture,
250                                    m_scenes[0].m_texture_shader));
```

```
251             }
252         }
253
254         if (m_tint_mipmaps)
255         {
256             rt::tint_mipmaps(texture);
257         }
258         m_scenes[0].m_texture =
259             static_cast<IDirect3DBaseTexture9 *>(texture);
260     }
261
262     ////////////////////////////////////////////////////////////
263     // CMyD3DApplication::init_2d_texture
264     //
265     // Load an image of a textured mandelbrot fractal as the 2D
266     // texture.
267     //
268     void
269     CMyD3DApplication::init_2d_texture()
270     {
271         CComPtr<IDirect3DTexture9> texture;
272         if (TS_FILE == m_scenes[1].m_source)
273         {
274             D3DXIMAGE_INFO info;
275             THR(::D3DXGetImageInfoFromFile(m_scenes[1].m_filename.c_str(),
276                 &info));
277             THR(::D3DXCreateTexture(m_pd3dDevice, info.Width, info.Height, 0,
278                 texture_usage(D3DRTYPE_TEXTURE), m_scenes[1].m_format,
279                 D3DPOOL_MANAGED, &texture));
280             CComPtr<IDirect3DSurface9> surface;
281             THR(texture->GetSurfaceLevel(0, &surface));
282             THR(::D3DXLoadSurfaceFromFile(surface, NULL, NULL,
283                 m_scenes[1].m_filename.c_str(), NULL, D3DX_DEFAULT,
284                 0, NULL));
285             if (m_autogen_mipmaps)
286             {
287                 THR(texture->SetAutoGenFilterType(m_autogen_filter));
288             }
289             else if (texture->GetLevelCount() > 1)
290             {
291                 THR(::D3DXFilterTexture(texture, NULL, 0, D3DX_DEFAULT));
292             }
293         }
294         else
295         {
296             THR(::D3DXCreateTexture(m_pd3dDevice, 256, 256, 0,
```

```
297                 texture_usage(D3DRTYPE_TEXTURE),
298                 m_scenes[1].m_format, D3DPOOL_MANAGED,
299                 &texture));
300             if (m_autogen_mipmaps)
301             {
302                 CComPtr<IDirect3DTexture9> single;
303                 THR(::D3DXCreateTexture(m_pd3dDevice, 256, 256, 1,
304                     0, m_scenes[1].m_format, D3DPOOL_SCRATCH,
305                     &single));
306                 THR(::D3DXFillTextureTX(single, m_scenes[1].m_texture_shader));
307                 CComPtr<IDirect3DSurface9> source, dest;
308                 THR(single->GetSurfaceLevel(0, &source));
309                 THR(texture->GetSurfaceLevel(0, &dest));
310                 THR(::D3DXLoadSurfaceFromSurface(dest, NULL, NULL,
311                     source, NULL, NULL, D3DX_FILTER_NONE, 0));
312                 THR(texture->SetAutoGenFilterType(m_autogen_filter));
313             }
314             else
315             {
316                 THR(::D3DXFillTextureTX(texture, m_scenes[1].m_texture_shader));
317             }
318         }
319 
320         if (m_tint_mipmaps && (texture->GetLevelCount() > 1))
321         {
322             rt::tint_mipmaps(texture);
323         }
324         m_scenes[1].m_texture =
325             static_cast<IDirect3DBaseTexture9 *>(texture);
326     }
327 
328     ////////////////////////////////////////////////////////////
329     // CMyD3DApplication::init_3d_texture
330     //
331     // Create a volume texture that is a grayscale alpha blended
332     // ramp based on the distance from one corner of the texture.
333     // This will create a 1/8th sphere that will appear as one or
334     // more spheres  when mirrored around all texture coordinate
335     // axes.
336     //
337     void
338     CMyD3DApplication::init_3d_texture()
339     {
340         CComPtr<IDirect3DVolumeTexture9> texture;
341         if (TS_FILE == m_scenes[2].m_source)
342         {
```

```
343         THR(::D3DXCreateVolumeTextureFromFileEx(m_pd3dDevice,
344             m_scenes[2].m_filename.c_str(),
345             D3DX_DEFAULT, D3DX_DEFAULT, D3DX_DEFAULT,
346             D3DX_DEFAULT,
347             texture_usage(D3DRTYPE_VOLUMETEXTURE),
348             m_scenes[2].m_format, D3DPOOL_MANAGED,
349             D3DX_DEFAULT, D3DX_DEFAULT, 0, NULL, NULL,
350             &texture));
351     }
352     else
353     {
354         // create volume texture
355         const DWORD usage =
356             texture_usage(D3DRTYPE_VOLUMETEXTURE);
357         const UINT levels = (m_d3dCaps.TextureCaps &
358             D3DPTEXTURECAPS_MIPVOLUMEMAP) ? 0 : 1;
359         THR(::D3DXCreateVolumeTexture(m_pd3dDevice, 64, 64,
360             64, levels, usage, m_scenes[2].m_format,
361             D3DPOOL_MANAGED, &texture));
362
363         THR(::D3DXFillVolumeTextureTX(texture,
364                         m_scenes[2].m_texture_shader));
365     }
366     if (m_tint_mipmaps)
367     {
368         rt::tint_mipmaps(texture);
369     }
370     m_scenes[2].m_texture =
371         static_cast<IDirect3DBaseTexture9 *>(texture);
372 }
373
374 ////////////////////////////////////////////////////////////
375 // CMyD3DApplication::InitDeviceObjects()
376 //
377 // Initialize the scenes used by the application: meshes
378 // textures and the font.
379 //
380 HRESULT CMyD3DApplication::InitDeviceObjects()
381 {
382     m_can_volume_texture = 0 !=
383         (m_d3dCaps.TextureCaps & D3DPTEXTURECAPS_VOLUMEMAP);
384     init_textures();
385     if (!m_can_volume_texture && 2 == m_scene)
386     {
387         m_scene = 1;
388     }
```

```
389        HMENU menu = ::GetMenu(m_hWnd);
390        rt::enable_menu(menu, IDM_DIMENSIONALITY_VOLUMETRIC,
391            m_can_volume_texture);
392
393        m_max_detail = m_scenes[m_scene].m_texture->GetLevelCount()-1;
394        m_detail = min(m_max_detail, m_detail);
395        inc_detail(0);
396
397        // create meshes
398        THR(::D3DXCreateMeshFVF(NUM_OF(g_verts_1d)/3,
399            NUM_OF(g_verts_1d), D3DXMESH_MANAGED |
400            D3DXMESH_WRITEONLY, s_vertex_1d::FVF, m_pd3dDevice,
401            &m_scenes[0].m_mesh));
402        THR(::D3DXCreateMeshFVF(NUM_OF(g_verts_2d)/3,
403            NUM_OF(g_verts_2d), D3DXMESH_MANAGED |
404            D3DXMESH_WRITEONLY, s_vertex_2d::FVF, m_pd3dDevice,
405            &m_scenes[1].m_mesh));
406        const UINT NUM_PLANES = 11;
407        THR(::D3DXCreateMeshFVF(NUM_PLANES*NUM_OF(g_verts_3d)/3,
408            NUM_PLANES*NUM_OF(g_verts_3d), D3DXMESH_MANAGED |
409            D3DXMESH_WRITEONLY, s_vertex_3d::FVF, m_pd3dDevice,
410            &m_scenes[2].m_mesh));
411
412        // copy vertices: 1D/2D texture meshes are copied
413        // from static arrays
414        std::copy(g_verts_1d, g_verts_1d+NUM_OF(g_verts_1d),
415            rt::mesh_vertex_lock<s_vertex_1d>(m_scenes[0].m_mesh).
416                data());
417        std::copy(g_verts_2d, g_verts_2d+NUM_OF(g_verts_2d),
418            rt::mesh_vertex_lock<s_vertex_2d>(m_scenes[1].m_mesh).
419                data());
420
421        // 3D mesh is a stack of planes parallel to camera
422        {
423            rt::mesh_vertex_lock<s_vertex_3d> lock(m_scenes[2].m_mesh);
424
425            const s_vertex_3d *begin = g_verts_3d;
426            const s_vertex_3d *end = begin + NUM_OF(g_verts_3d);
427
428            std::copy(begin, end, lock.data());
429            for (UINT i = 1; i < NUM_PLANES; i++)
430            {
431                std::transform(begin, end,
432                    lock.data() + i*NUM_OF(g_verts_3d),
433                    fn_set_z_w(1.f - i/float(NUM_PLANES-1),
434                               -2 + 4.f*i/float(NUM_PLANES-1)));
```

```
435             }
436         }
437
438         // build indices; assume independent triangles I[i] = i
439         const UINT max_vertices =
440             max(m_scenes[0].m_mesh->GetNumVertices(),
441             max(m_scenes[1].m_mesh->GetNumVertices(),
442                 m_scenes[2].m_mesh->GetNumVertices()));
443         std::vector<WORD> indices(max_vertices);
444         std::generate(indices.begin(), indices.end(),
445             gen_integers());
446         for (UINT i = 0; i < 3; i++)
447         {
448             std::copy(&indices[0],
449                 &indices[m_scenes[i].m_mesh->GetNumVertices()],
450                 rt::mesh_index_lock<WORD>(m_scenes[i].m_mesh).data());
451         }
452
453         // Init the font
454         m_font.InitDeviceObjects(m_pd3dDevice);
455
456         return S_OK;
457     }
458
459     ////////////////////////////////////////////////////////////
460     // CMyD3DApplication::RestoreDeviceObjects()
461     //
462     // Restore device state and adjust UI elements to conform
463     // to device capabilities.
464     //
465     HRESULT CMyD3DApplication::RestoreDeviceObjects()
466     {
467         enable_lod_bias();
468         m_rhs.x1 = m_d3dsdBackBuffer.Width/2;
469         m_rhs.x2 = m_d3dsdBackBuffer.Width;
470         m_rhs.y1 = 0;
471         m_rhs.y2 = m_d3dsdBackBuffer.Height;
472
473         m_anisotropic = (m_d3dCaps.RasterCaps &
474             D3DPRASTERCAPS_ANISOTROPY) != 0;
475         update_filter_menu();
476         update_address_menu();
477         update_format_menu();
478
479         // Set miscellaneous render states
480         const rt::s_rs states[] =
```

```
481         {
482             D3DRS_CULLMODE,         D3DCULL_NONE,
483             D3DRS_DITHERENABLE,     FALSE,
484             D3DRS_SPECULARENABLE,   FALSE,
485             D3DRS_ZENABLE,          D3DZB_TRUE,
486             D3DRS_CLIPPING,         TRUE,
487             D3DRS_LIGHTING,         FALSE,
488             D3DRS_ZFUNC,            D3DCMP_LESS
489         };
490         rt::set_states(m_pd3dDevice, states, NUM_OF(states));
491 
492         // Set the world matrix
493         D3DXMATRIX ident(1, 0, 0, 0,
494                          0, 1, 0, 0,
495                          0, 0, 1, 0,
496                          0, 0, 0, 1);
497         THR(m_pd3dDevice->SetTransform(D3DTS_WORLD, &ident));
498         THR(m_pd3dDevice->SetTransform(D3DTS_TEXTURE0, &ident));
499 
500         // Set up our view matrix with an eye position and
501         // default look at position of the origin and view
502         // up vector of the y-axis.
503         THR(m_pd3dDevice->SetTransform(D3DTS_VIEW,
504             rt::anon(rt::mat_look_at(D3DXVECTOR3(0, 0, -2.5f)))));
505 
506         // Set the projection matrix
507         const float aspect = float(m_d3dsdBackBuffer.Width) /
508             float(m_d3dsdBackBuffer.Height);
509         D3DXMATRIX proj;
510         ::D3DXMatrixPerspectiveFovLH(&proj, D3DX_PI/4.f, aspect,
511             0.25f, 5.0f);
512         THR(m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &proj));
513 
514         // Restore the font
515         m_font.RestoreDeviceObjects();
516 
517         return S_OK;
518     }
519 
520     ////////////////////////////////////////////////////////////
521     // CMyD3DApplication::FrameMove()
522     //
523     // Update the movement parameters based on the current
524     // keyboard input.  Construct a matrix from the movement
525     // parameters.  Apply the matrix to either the world
526     // transform or the texture transform, depending on the
```

```
527 // scene.
528 //
529 HRESULT CMyD3DApplication::FrameMove()
530 {
531     // Update user input state
532     UpdateInput();
533
534     // Update the world state according to user input
535     if (m_animate_view)
536     {
537         m_fWorldRotY += m_fElapsedTime;
538     }
539     else
540     {
541         if (m_input.m_left && !m_input.m_right)
542         {
543             m_fWorldRotY += m_fElapsedTime;
544         }
545         else if (m_input.m_right && !m_input.m_left)
546         {
547             m_fWorldRotY -= m_fElapsedTime;
548         }
549     }
550     if (m_input.m_up && !m_input.m_down)
551     {
552         m_fWorldRotX += m_fElapsedTime;
553     }
554     else if (m_input.m_down && !m_input.m_up)
555     {
556         m_fWorldRotX -= m_fElapsedTime;
557     }
558
559     if (m_fWorldRotX > 2.f*D3DX_PI)
560     {
561         m_fWorldRotX = fmodf(m_fWorldRotX, 2.f*D3DX_PI);
562     }
563     if (m_fWorldRotY > 2.f*D3DX_PI)
564     {
565         m_fWorldRotY = fmodf(m_fWorldRotY, 2*D3DX_PI);
566     }
567
568     const D3DXMATRIX orient = rt::mat_rot_x(m_fWorldRotX)*
569         rt::mat_rot_y(m_fWorldRotY);
570     const D3DXMATRIX ident(1, 0, 0, 0,
571                            0, 1, 0, 0,
572                            0, 0, 1, 0,
```

```
573                               0, 0, 0, 1);
574     // for 3D texture, rotate the texture coordinates and
575     // leave the planes parallel to the viewer
576     if (m_scene == 2)
577     {
578         THR(m_pd3dDevice->SetTransform(D3DTS_WORLD,
579             &ident));
580         THR(m_pd3dDevice->SetTransform(D3DTS_TEXTURE0,
581             &orient));
582     }
583     else
584     // for 1D/2D texture, rotate the plane and leave the texture
585     // coordinates alone.
586     {
587         THR(m_pd3dDevice->SetTransform(D3DTS_WORLD,
588             &orient));
589         THR(m_pd3dDevice->SetTransform(D3DTS_TEXTURE0,
590             &ident));
591     }
592
593     return S_OK;
594 }
595
596 ////////////////////////////////////////////////////////////
597 // CMyD3DApplication::Render()
598 //
599 // Render the scene.  If split screen is enabled, then
600 // use the Z buffer to occlude rendering of one half of the
601 // screen by initializing it to 0.0f to ensure that the Z
602 // test always fails for that half when rendering the
603 // filtered scene.  Then clear it to 1.0f before rendering
604 // the unfiltered scene.
605 //
606 HRESULT CMyD3DApplication::Render()
607 {
608     // Clear the viewport
609     THR(m_pd3dDevice->Clear(0L, NULL, D3DCLEAR_TARGET |
610         D3DCLEAR_ZBUFFER, D3DCOLOR_ARGB(0, 0, 0, 255), 1.0f,
611         0L));
612     THR(m_pd3dDevice->BeginScene());
613
614     if (m_split_screen)
615     {
616         THR(m_pd3dDevice->Clear(1, &m_rhs, D3DCLEAR_ZBUFFER,
617             0, 0.f, 0L));
618     }
```

```
619
620         {
621             const rt::s_rs set[] =
622             {
623                 D3DRS_ALPHATESTENABLE, TRUE,
624                 D3DRS_ALPHAREF, 0,
625                 D3DRS_ALPHAFUNC, D3DCMP_GREATER,
626                 D3DRS_ZENABLE, D3DZB_TRUE,
627                 D3DRS_ZFUNC, D3DCMP_LESS,
628                 D3DRS_ALPHABLENDENABLE, (m_scene == 2),
629                 D3DRS_SRCBLEND, D3DBLEND_ONE,
630                 D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA
631             };
632             rt::set_states(m_pd3dDevice, set, NUM_OF(set));
633             const rt::s_tss tset[] =
634             {
635                 D3DTSS_COLORARG1, D3DTA_TEXTURE,
636                 D3DTSS_COLOROP, D3DTOP_SELECTARG1,
637                 D3DTSS_ALPHAARG1, D3DTA_TEXTURE,
638                 D3DTSS_ALPHAOP, D3DTOP_SELECTARG1,
639                 D3DTSS_TEXTURETRANSFORMFLAGS, m_scene+1,
640             };
641             rt::set_states(m_pd3dDevice, 0, tset, NUM_OF(tset));
642             const rt::s_ss sset[] =
643             {
644                 D3DSAMP_ADDRESSU, m_address_u,
645                 D3DSAMP_ADDRESSV, m_address_v,
646                 D3DSAMP_ADDRESSW, m_address_w,
647                 D3DSAMP_BORDERCOLOR, D3DCOLOR_XRGB(0,255,0),
648                 D3DSAMP_MINFILTER, m_min_filter,
649                 D3DSAMP_MAGFILTER, m_mag_filter,
650                 D3DSAMP_MIPFILTER, m_mip_filter,
651                 D3DSAMP_MAXANISOTROPY, m_anisotropy,
652                 D3DSAMP_MAXMIPLEVEL, m_max_level,
653                 D3DSAMP_MIPMAPLODBIAS,
654                     rt::float_dword(m_lod_bias*0.1f)
655             };
656             rt::set_states(m_pd3dDevice, 0, sset, NUM_OF(sset));
657         }
658
659         THR(m_pd3dDevice->SetTexture(0, m_scenes[m_scene].m_texture));
660         THR(m_scenes[m_scene].m_mesh->DrawSubset(0));
661
662         if (m_split_screen)
663         {
664             THR(m_pd3dDevice->Clear(1, &m_rhs, D3DCLEAR_ZBUFFER,
```

```
665                 0, 1.f, 0L));
666
667             const rt::s_ss sset[] =
668             {
669                 D3DSAMP_MINFILTER, D3DTEXF_POINT,
670                 D3DSAMP_MAGFILTER, D3DTEXF_POINT,
671                 D3DSAMP_MIPFILTER, D3DTEXF_NONE
672             };
673             rt::set_states(m_pd3dDevice, 0, sset, NUM_OF(sset));
674
675             THR(m_scenes[m_scene].m_mesh->DrawSubset(0));
676
677             // draw dividing line down center of the screen
678             THR(m_pd3dDevice->SetFVF(D3DFVF_XYZRHW));
679             THR(m_pd3dDevice->SetRenderState(D3DRS_ZENABLE,
680                 D3DZB_FALSE));
681             const rt::s_tss tset[] =
682             {
683                 D3DTSS_COLOROP, D3DTOP_DISABLE,
684                 D3DTSS_ALPHAOP, D3DTOP_DISABLE
685             };
686             rt::set_states(m_pd3dDevice, 0, tset, NUM_OF(tset));
687             const D3DVECTOR separator[2] =
688             {
689                 float(m_rhs.x1), float(m_rhs.y1), 0,
690                 float(m_rhs.x1), float(m_rhs.y2), 0
691             };
692             THR(m_pd3dDevice->DrawPrimitiveUP(D3DPT_LINELIST, 1,
693                 &separator[0], sizeof(separator[0])));
694         }
695
696         // Render stats and help text
697         if (m_show_text)
698         {
699             RenderText();
700         }
701
702         THR(m_pd3dDevice->EndScene());
703
704         return S_OK;
705     }
```