6 Texture Mapping

Texture mapping is one of the primary techniques to improve the appearance of objects rendered with OpenGL. Texturing is typically used to provide color detail for intricate surfaces by modifying the surface color. For example, a woodgrain supplied by a texture can make a flat polygon appear to be made of wood. Current 3D video games now use texture mapping extensively. Texturing can also be the basis for many more sophisticated rendering algorithms for improving visual realism and quality. For example, environment mapping is a view-dependent texture mapping technique that supplies a specular reflection to the surface of objects. This makes it appear that the environment is reflected in the object. More generally texturing can be thought of as a method of providing (or perturbing) parameters to the shading equation such as the surface normal (bump mapping), or even the coordinates of the point being shaded (displacement mapping) based on a parameterization of the surface defined by the texture coordinates. OpenGL readily supports the first two techniques (surface color manipulation and environment mapping). Texture mapping, using bump mapping, can also solve some rendering problems in less obvious ways. This section reviews some of the details of OpenGL texturing support, outlines some considerations when using texturing and suggests some interesting algorithms using texturing.

6.1 Texturing Basics

6.1.1 The Texture Image

The meat of a texture is the texture's image. This is a array of color values. The color values of a texture are referred to as *texels* (short for texture elements and a pun on the word pixel). The texture image array is typically 1D or 2D, however OpenGL 1.2 adds support for 3D texture images as well.[1] The OpenGL `SGIS_texture4D` extension even provides the option for 4D texture images.

The `glTexImage1D`, `glTexImage2D`, and `glTexImage3D` commands specify a complete texture image. The commands copy the texture image data from the application's address space into texture memory. OpenGL's pixel store unpack state determines how the texture image is arranged in memory. Other OpenGL commands update rectangular subregions of an existing texture image (subtexture loads). Still other texture commands copy color data from the frame buffer into texture memory.

Typically, texture images are loaded from image files stored using a standard 2D image file format such as TIFF or JPEG. To make an image file into a texture for use by OpenGL, the OpenGL application is responsible for reading and decompressing as necessary the image file. Once the image is in memory as an uncompressed array, `glTexImage2D` can be passed the size, format, and pointer to the image in memory. The OpenGL API limits itself to rendering functionality and therefore has no support for loading image files. You can either write an image loader yourself or use one of the numerous image loading libraries that are widely available. In addition to loading image files, applications are free to compute or otherwise procedurally generate texture images. Some techniques for procedural texture generation are discussed in Section 6.20.2. Rendering the image using OpenGL and then copying the image from the framebuffer with `glCopyTexImage2D` is yet another option.

OpenGL's pixel transfer pipeline can process the texture image data when texture images are specified. While typically the pixel transfer pipeline is configured to pass texture image data through unchanged, operations such as color space conversions can be performed during texture image download. When optimized by your OpenGL implementation, the pixel transfer operations can significantly accelerate various common processing operations applied to texture image data. The pixel transfer pipeline is further described in Sections 13.1.1 and 13.1.4.

[1] The phrase *3D texturing* is often used in touting new graphics hardware and software products. The common usage of the phrase is to indicate support for applying a 2D texture to 3D geometry. OpenGL's specification would call that merely *2D texturing*. OpenGL assumes that any type of texturing can be applied to arbitrary 3D geometry so the dimensionality of texture mapping (1D, 2D, or 3D) is based on the dimensionality of the texture image. A 2D texture image (one with width and height) is used for 2D texturing. A 3D image (one with width, height, and depth) is required for 3D texturing in the OpenGL technical sense of the phrase. Unfortunately, the market continues to use the phrase *3D texturing* to mean just `GL_TEXTURE_2D`. To avoid confusion, the phrase *volumetric texturing* unambiguously refers to what OpenGL technically calls 3D texturing. Be aware that the phrases *solid texture* and *hypertexture* are also used in the graphics literature to denote 3D texture images. One more bit of trivia: the term *voxel* is often used to denote the texels of a 3D texture image.

The width, height, and depth of a texture image without a border must be powers of two. A texture with a border has an additional one pixel border around the edge of the texture image proper. Since the border is on each side, the border adds two texels in each texture dimension. The rationale for texture images with borders will be discussed in Section 6.4. The texels that make up the texture image have a particular color format. The color format options are RGB, RGBA, luminance, intensity, and luminance-alpha. Sized versions of the texture color formats permit applications a means to hint to the OpenGL implementation for trading off texture memory requirements with texture color quality.

Internal Texture Formats If you care about the quality of your textures or want to conserve the amount of texture memory your application requires (and often conserving texture memory helps improve performance), you should definitely use appropriate internal formats. Internal texture formats were introduced in OpenGL 1.1. Table 1 lists the available internal texture formats. If your texture is known to be only gray-scale or luminance values, choosing the `GL_LUMINANCE` format instead of `GL_RGB` typically cuts your texture memory usage by one third. Requesting more efficient internal format sizes can also help. The `GL_RGB8` internal texture format requests 8 bits of red, green, and blue precision for each texel. The more space efficient `GL_RGB4` internal texture format uses only 4 bits per component making it require only half the texture memory of the `GL_RGB8` format. Of course, the `GL_RGB4` format only has 16 distinct values per component instead of 256 values for the `GL_RGB8` format. However, if minimizing texture memory usage (and often improving texturing performance too) is more important than better texture quality, the `GL_RGB4` format is a better choice. In the case where the source image for your texture only has 4 bits of color resolution per component, there is absolutely no reason to request a format with more than 4 bits of color resolution.

Some words of advice about internal texture formats: If you do not request a specific internal resolution for your texture image because you requested a `GL_RGBA` internal format instead of a size-specific internal format such as `GL_RGBA8` or `GL_RGBA4`, your OpenGL implementation is free to pick the "most appropriate" format for the particular implementation. If a smaller texture format has better texturing performance, the implementation is free to choose the smaller format. This means if you care about maintaining a particular level of internal format resolution, selecting a size-specific texture format is strongly recommended.

Some words of warning about internal texture formats: Not all OpenGL implementations are expected to support all the available internal texture formats. This means just because you request a `GL_LUMIANCE12_ALPHA4` format (to pick a format that is likely to be obscure) does not mean that your texture is guaranteed to be stored in this format. The size-specific internal texture formats are merely hints. If the best the OpenGL implementation can provide is `GL_LUMINANCE8_ALPHA8`, this will be the format you get, even though is provides less luminance precision and more alpha precision than you requested.

6.1.2 Texture Coordinates

Texture coordinates are the means by which texture image positions are assigned to vertices. The per-vertex assignment of texture coordinates is the key to mapping a texture image to rendered geometry. During rasterization, the texture coordinates of a primitive's vertices are interpolated across the primitive so that each rasterized fragment making up the primitive has an appropriately interpolated texture coordinate. A fragment's texture coordinates are translated into the addresses of one or more texels within the current texture. The texels are fetched and their color values are then filtered into a single texture color value for the fragment. The fragment's texture color is then combined with the fragments color.

The vertices of all primitives (including the raster position of pixel images) have associated texture coordinates. Figure 27 shows how object coordinates have associated texture coordinates that is used to map into a texture image when texture mapping is enabled. The texture coordinates are part of a three-dimensional homogeneous coordinate system (s,t,r,q). Applications often only assign the 2D s and t coordinates, but OpenGL treats this as a special case of the more general 3D homogeneous texture coordinate space. The r and q texture coordinates are vital to techniques that utilize volumetric and projective texturing. When t, r, or q are not explicitly assigned a

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	L bits	I bits
ALPHA4	ALPHA				4		
ALPHA8	ALPHA				8		
ALPHA12	ALPHA				12		
ALPHA16	ALPHA				16		
LUMINANCE4	LUMINANCE					4	
LUMINANCE8	LUMINANCE					8	
LUMINANCE12	LUMINANCE					12	
LUMINANCE16	LUMINANCE					16	
LUMINANCE4_ALPHA4	LUMINANCE_ALPHA				4	4	
LUMINANCE6_ALPHA2	LUMINANCE_ALPHA				2	6	
LUMINANCE8_ALPHA8	LUMINANCE_ALPHA				8	8	
LUMINANCE12_ALPHA4	LUMINANCE_ALPHA				12	4	
LUMINANCE16_ALPHA16	LUMINANCE_ALPHA				16	16	
INTENSITY4	INTENSITY						4
INTENSITY8	INTENSITY						8
INTENSITY12	INTENSITY						12
INTENSITY16	INTENSITY						16
R3_G3_B2	RGB	3	3	2			
RGB4	RGB	4	4	4			
RGB5	RGB	5	5	5			
RGB8	RGB	8	8	8			
RGB10	RGB	10	10	10			
RGB12	RGB	12	12	12			
RGB16	RGB	16	16	16			
RGBA2	RGBA	2	2	2	2		
RGBA4	RGBA	4	4	4	4		
RGB5_A1	RGBA	5	5	5	1		
RGBA8	RGBA	8	8	8	8		
RGB10_A2	RGBA	10	10	10	2		
RGBA12	RGBA	12	12	12	12		
RGBA16	RGBA	16	16	16	16		

Table 1: OpenGL Internal Texture Formats. Each internal texture format has a corresponding base internal format and its *desired* component resolutions.

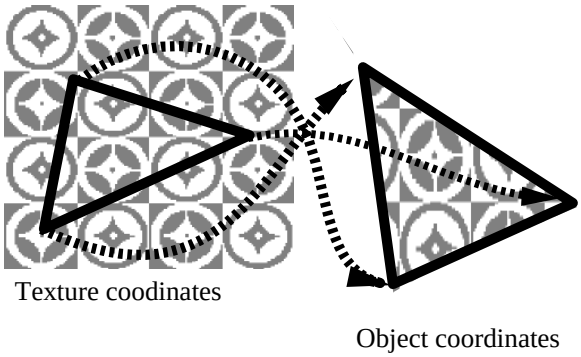

Figure 27. Vertices with Texture Coordinates. Texture coordinates determine how texels in the texture are mapped to the surface of a triangle in object space.

value (as when `glTexCoord1f` is called), their assumed values are 0, 0, and 1 respectively. If the concept of 3D homogeneous texture coordinates is unfamiliar to your, the topic will be revisited in Section 6.16.

OpenGL's interpolation of texture coordinates across a primitive compensates for the appearance of a textured surface when viewed in perspective. While so-called *perspective correct* texture coordinate interpolation is more expensive, failing to account for perspective results in incorrect and unsightly distortion of the texture image across the textured primitive's surface.

Each texture coordinate is assumed to be floating-point value. Each set of texture coordinates must be mapped to a position within the texture image. The coordinates of the texture map range from [0..1] in each dimension. OpenGL can treat coordinate values outside the range [0,1] in one of two ways: clamp or repeat. In the case of clamp, the coordinates are simply clamped to [0,1] causing the edge values of the texture to be stretched across the remaining parts of the polygon. In the case of repeat the integer part of the coordinate is discarded so the texture image becomes an infinitely repeated tile pattern. In the case of clamping, proper filtering may require accounting for border texels or, when no border is specified, the texture border color. OpenGL 1.2 adds a variation on clamping known as *clamp to edge* that clamps such that the border is never sampled.[2] The filtered color value that results from texturing can be used to modify the original surface color value in one of several ways as determined by the *texture environment*. The simplest way replaces the surface color with texel color, either by modulating a white polygon or simply replacing the color value. Simple replacement was added as an extension by some vendors to OpenGL 1.0 and is now part of OpenGL 1.1.

Assigning Texture Coordinates A common question is how texture coordinates are assigned to the vertices of an object. There is no single answer. Sometimes the texture coordinates are some mathematical function of the object coordinates. In other cases, the texture coordinates are manually assigned by the artist that created a given 3D model. Most common 3D object file formats such as VRML or the Wavefront OBJ format contain accompanying texture coordinates. Keep in mind that the assignment of texture coordinates for a particular 3D model is not something that can be done independent of the intended texture to be mapped onto the object.

Optimizing Texture Coordinate Assignment Sloan, Weinstein, and Brederson [93] have explored optimizing the assignment of texture coordinates based on an "importance map" that can encode both intrinsic texture proper-

[2]The *clamp to edge* functionality is also available through the `SGIS_texture_edge_clamp` extension.

ties as well as user-guided highlights. Such importance driven texture coordinate optimization techniques highlight the fact that textured detail is very likely not uniformly distributed for a particular texture image and a particular texture coordinate assignment. Warping the texture image and changing the texture coordinate assignment provides opportunities for improving texture appearance without increasing the texture size.

6.1.3 Texture Coordinate Generation and Transformation

An alternative to assigning texture coordinate explicitly to every vertex is to have OpenGL generate texture coordinates for you. OpenGL's texture coordinate generation (often called *texgen* for short) can generate texture coordinates automatically as a linear function of the eye-space or object-space coordinates or using a special sphere map formula designed for environment mapping.

OpenGL also provides a 4 by 4 *texture matrix* that can be used to transform the per-vertex texture coordinates, whether supplied explicitly or implicitly through texture coordinate generation. The texture matrix provides a means to rescale, translate, or even project texture coordinates before the texture is applied during rasterization.

6.1.4 Filtering

The texture image is a discrete array of texels, but the texture coordinates vary continuously (at least conceptually). This creates a sampling problem. In addition, a fragment can really be thought of as covering some region of the texture image (the fragment's footprint). Filtering also tries to account for a fragment's footprint within the texture image.

OpenGL provides a number of filtering methods to compute the texel value. There are separate filters for magnification (many pixel fragment values map to one texel value) and minification (many texel values map to one pixel fragment). The simplest of the filters is point sampling, in which the texel value nearest the texture coordinates is selected. Point sampling seldom gives satisfactory results, so most applications choose some filter which interpolates. For magnification, OpenGL only supports linear interpolation between four texel values. For minification, OpenGL supports various types of mipmapping [107], with the most useful (and computationally expensive) being tri-linear mipmapping (four samples taken from each of the nearest two mipmap levels and then interpolating the two sets of samples). Some vendors have also added an extension called `SGIS_texture_filter4` that provides a larger filter kernel in which the weighted sum of a 4x4 array of texels is used.

With mipmapping, a texture consists of multiple levels-of-detail (LODs). Each mipmap level is a distinct texture image. The base mipmap level has the highest resolution and is called mipmap level zero. Each subsequent level is half the dimensions (height, width, and depth) until each dimension goes to one and finally all the dimensions reduce to one. For mipmap filtering to work reasonably, each subsequent mipmap level is down-sampled version of the previous mipmap level texture image. Figure 28 shows how texture mipmap levels provide multiple LODs for a base texture image. OpenGL does not provide any built-in commands for generating mipmaps, but the GLU provides some simple routines (`gluBuild1DMipmaps`, `gluBuild2DMipmaps`, and `gluBuild3DMipmaps`[3]) for generating mipmaps using a simple box filter.

During texturing, OpenGL automatically computes (or more likely, approximates) each fragment's LOD parameter based on the partial derivatives of the primitive's mapping of texture coordinates to window coordinates. This LOD parameter is often called lambda (λ). The integer portion of the lambda value determines which mipmap levels to use for mipmap filtering and the fractional portion of the lambda value determines the weighting for selecting or blending mipmaps levels. Because OpenGL handles mipmapping automatically, the details of LOD computation are most interesting to OpenGL implementors, but it is important that users of OpenGL understand the interpolation math so that they will not be surprised by unexpected results.

Additional Control of Texture Level of Detail In OpenGL 1.0 and 1.1, all the mipmap levels of a texture must be specified and consistent. To be consistent, every mipmap level of a texture must be half the dimensions (until

[3] Introduced in GLU version 1.3.

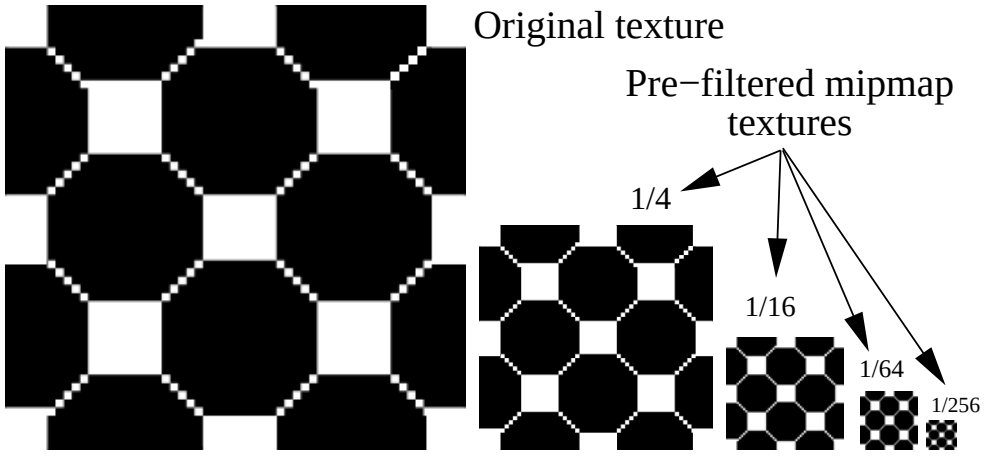

Figure 28. Multiple Levels of Texture Detail using Mipmaps

reaching a dimension of one and excluding border texels) of the previous mipmap LOD, and all the mipmaps must shared the same internal format and borders.

If mipmap filtering is requested for a texture, but all the mipmap levels of a texture are not present or not consistent, OpenGL silently disables texturing. A common pitfall for OpenGL programmers is supplying an inconsistent or incomplete set of mipmap levels and then wondering why texturing does not work. Be sure to specify all the mipmap levels of a texture consistently. If you use the GLU routines for building mipmaps, this is guaranteed.

OpenGL 1.2 relaxes the texture consistency requirement by allowing the application to specify a contiguous range of mipmap levels that must be consistent. This permits an application to still use mipmapping if only the 1x1 through 256x256 mipmap levels of a texture with a 1024x1024 level 0 texture, but not supply the 512x512 and 1024x1024 levels by managing the texture's `GL_TEXTURE_BASE_LEVEL` and `GL_TEXTURE_MAX_LEVEL` parameters. If an application is designed to guarantee a constant frame-rate, one reason the application might constrain the base and maximum LODs in this way is that the application does not have the time to read the 512x512 and 1024x1024 mipmap levels from disk. In this case, the application makes the choice to settle for lower resolution LODs, possibly resulting in blurry textured surfaces, rather than of dropping a frame. Hopefully on subsequent frames, the application can manage to load the full set of mipmap levels for the texture and continue with full texture quality. The OpenGL implementation implements this feature by simply clamping the λ LOD value to the range of available mipmap levels.

Additionally, even when all the mipmap levels are present and consistent, some of the texture images for some levels may be out-of-date if the texture is being dynamically updated using subtexture loads. OpenGL 1.2's `GL_TEXTURE_MIN_LOD` and `GL_TEXTURE_MAX_LOD` texture parameters provide a further means to clamp the λ LOD value to a contiguous range of mipmap levels.[4] Section 6.8 applies this functionality to the task of texture paging.

6.1.5 Texture Environment

The process by which the final fragment color value is derived is called the texture environment function (`glTexEnv`) Several methods exist for computing the final color, each capable of producing a particular effect. One of the most commonly used is the `GL_MODULATE` environment function. The modulate function multiplies or modulates the original fragment color with the texel color. Typically, applications generate polygons with per-vertex lighting

[4]This same functionality for controlling texture level of detail is also available through the `SGIS_texture_lod` extension.

enabled and then modulate the texture image with the fragment's interpolated lit color value to produce a lit, textured surface. The `GL_REPLACE` texture environment[5] is even simpler. The replace function simply replaces the fragment's color with the color from the texture. The same effect as replace can be accomplished in OpenGL 1.1 by using the modulate environment with a constant white current color, though the replace function has a lower computational cost.

The `GL_DECAL` environment function performs simple alpha-blending between the fragment color and an RGBA texture; for RGB textures it simply replaces the fragment color. Decal mode is undefined for other texture formats (luminance, alpha, intensity). The `GL_BLEND` environment function uses the texture value to control the mix of the incoming fragment color and a constant texture environment color.

At the time of this writing, efforts are underway to standardize extensions that enhance the texture environment by adding new functions. For example, there should be a way to add the texture color to the fragment color.

6.1.6 Texture Objects

Most texture mapping applications switch among many different textures during the course of rendering a scene. To facilitate efficient switching among multiple textures and to facilitate texture management, OpenGL uses *texture objects* to maintain texture state.

The state of a texture object consists of the set of texture images for the all mipmap levels of the texture and the texturing parameters such as the texture wrap and minification and magnification filtering modes. Other OpenGL texture-related state such as the texture environment or texture coordinate generation modes are *not* part of a texture object's state. Conceptually, the state of a texture object is just the texture image and the parameters that determine how to filter that image.

As with display lists, each texture object is identified by a 32-bit unsigned integer that serves as the texture's name. Also as with display lists names, the application is free to assign arbitrary unused names to new texture objects. The command `glGenTextures` assists in the assignment of texture object names by returning a set of names guaranteed to be unused. A texture object is bound, prioritized, checked for residency, and deleted by its name. The value zero is reserved to name the default texture of each texture target type. Each texture object has its own texture target type. The three supported texture targets are:

- `GL_TEXTURE_1D`
- `GL_TEXTURE_2D`
- `GL_TEXTURE_3D`

Calling `glBindTexture` binds the named texture object as the current texture for the specified texture target. Instead of creating a texture object explicitly, a texture object is created whenever a texture image or parameter is set for an unused texture object name. Once created a texture object's target (1D, 2D, or 3D) is fixed until the texture object is deleted.

The `glTexImage`, `glTexParameter`, `glGetTexParameter`, `glGetTexLevelParameter`, and `glGetTexImage` commands update or query the state of the currently bound texture of the specified target type. Keep in mind that there are really three current textures, one for each texture target type: 1D, 2D, and 3D. When texturing is fully enabled, the current texture object (i.e., current for the enabled texture target) is used for texturing. When rendering objects with different textures, `glBindTexture` is the way to switch among the available textures.

Keep mind that switching textures is a fairly expensive operation. If a texture is not already resident in dedicated texture memory, switching to a non-resident texture requires that the texture be downloaded to the hardware before use. Even if the texture is already downloaded, caches that maximize texture performance may be invalidated when switching textures. The details of switching textures varies depending on your OpenGL implementation,

[5] Introduced by OpenGL 1.1.

but suffice it to say that OpenGL implementations are inevitably optimized to maximize texturing performance for whatever texture is currently bound so changing textures is something to minimize. Real-world applications often derive significant performance gains by sorting by texture the objects that they render to minimize the number of `glBindTexture` commands required to render the scene. For example, if a scene uses three different tree textures to draw several dozen trees within a scene, it is a good idea to draw all the trees that share a single texture first before switching to a different tree texture.

Texture objects were introduced by OpenGL 1.1. The original OpenGL 1.0 specification did not support texture objects. The thinking at the time was that display lists containing a complete set of texture images and texture parameters could provide a sufficient mechanism for fast texture switches. But display listed textures proved inadequate for several reasons. Recognizing textures embedded in display list efficiently proved difficult. One problem was that a display listed `glTexImage2D` must encapsulate the original image, which might not be the final texture as transformed by the pixel transfer pipeline. Changes to the pixel transfer pipeline state could change the texture image downloaded in subsequent calls of the display list. Unless every pixel transfer state setting was explicitly set in the display list, OpenGL implementations had to maintain the original texture data and be prepared to re-transform it by the current pixel transfer pipeline state when the texture display list is called. Moreover, even if every pixel transfer state setting is explicitly set in the display list, supporting future extensions that add new pixel transfer state would invalidate the optimization. Texture objects store the *post*-pixel transfer pipeline image so texture objects have no such problem. Another issue is that because display lists are not editable, display lists precluded support for subtexture loads as provided by the `glTexSubImage2D` command. Lastly, display lists lack a means to query and update the priority and residency of textures.[6]

6.2 Multitexture

Multitexture refers to the ability to apply two or more distinct textures to a single fragment. Each texture has the ability to supply its own texture color to rasterized fragments. Without multitexture, there is only a single supported texture unit. OpenGL's multitexture support requires that every texture unit be fully functional and maintain state that is independent of any other texture units. Each texture unit has its own texture coordinate generation state, texture matrix state, texture enable state, and texture environment state. However, each texture unit within an OpenGL context shares the same set of texture objects.

Rendering algorithms that require multiple rendering passes can often be reimplemented to use multitexture in operate in less rendering passes. Some effects are only viable with multitexture.

Many OpenGL games such as Quake and Unreal use light maps to improve the lighting quality within their scenes. Without multitexture, light map textures must be modulated into the scene with a second blended rendering pass in addition to a first pass to render the base surface texture. With multitexture, the light maps and base surface texture can be rendered in a single rendering pass. This can cut the transformation overhead almost in half when rendering light maps because a single multitexture rendering pass means that polygons need to only be transformed once. The framebuffer update overhead is also lower when using multitexture to render light maps. When multitexture is used, the overhead of blending in the second rendering pass is completely eliminated. A single multitextured rendering pass can render both the surface texture and the light map texture without any framebuffer blending because the modulation of the surface texture with the light map texture occurs as part of the multitexture texture environment. Light maps are described in more detail in Section 10.2.

The OpenGL 1.2.1 revision of the OpenGL specification [91] includes an Appendix F that introduces the concept of OpenGL Architecture Review Board (ARB) approved extensions and specifies the `ARB_multitexture` extension, the first distinct ARB extension. The original OpenGL 1.2 specification includes an ARB extension called the `ARB_imaging` extension, but the `ARB_imaging` description is intermingled with the core OpenGL 1.2 specification. The `ARB_multitexture` extension is the first ARB extension that is specified in an Appendix distinct from the core specification. The purpose of ARB extensions is to add important new functionality to OpenGL in

[6] While the `SGIX_list_priority` extension does provide a way to prioritize display lists, the concept of querying texture residency, while important to texture objects, is not applicable to display lists.