Programming with OpenGL: Advanced Rendering

Organizer:
Tom McReynolds
Silicon Graphics

SIGGRAPH ‘96 Course

Abstract

This course moves beyond the straightforward images generated by the novice, demonstrating the more sophisticated and novel techniques possible using the OpenGL library.

By explaining the concepts and demonstrating the techniques required to generate images of greater realism and utility, the course provides deeper insights into OpenGL functionality and computer graphics concepts.

Speakers

David Blythe

David Blythe is a Principal Engineer with the Advanced Systems Division at Silicon Graphics. David joined SGI in 1991 to assist Silicon Graphics in the development of the RealityEngine. He has contributed extensively to implementations of the OpenGL graphics library and OpenGL extension specifications. Prior to joining SGI, David was a visualization scientist at the Ontario Centre for Large Scale Computation.

Email: blythe@asd.sgi.com

Tom McReynolds

Tom McReynolds is a software engineer in the IRIS Performer group at Silicon Graphics. Prior to Performer, Tom worked in the OpenGL group, where he implemented OpenGL extensions and did OpenGL performance work. Prior to SGI, he worked at Sun Microsystems, where he developed graphics hardware support software and graphics libraries, including XGL.

He is also an adjunct professor at Santa Clara University, where he teaches courses in computer graphics using the OpenGL library.

Email tomcat@asd.sgi.com

Contents

A. Introduction 5
1. Acknowledgements 5
2. Coure Notes Web Site 5
3. OpenGL 6
4. Modelling Considerations 7
B. Texture Mapping 9
1. Mipmap Generation 11
2. Fine Tuning 13
3. Paging Textures 14
4. Transparency Mapping and Trimming with Alpha 15
5. Rendering Text 16
6. Projective Texutures 16
7. Environment Mapping 17
8. Image Warping and Dewarping 17
C. Blending 18
1. Compositing 18
2. Translucency 18
3. Advanced Blending 19
4. Image Processing 19
5. Painting 20
6. Antil-Aliasing 20
D. The Z Cooridinate and Perspective Projection 22
1. Depth Buffering 23
E. Multipass rendering 25
1. Phong Highlights 25
2. Spotlight Effects using Projective Textures 26
3. Reflections and Refractions using the Stencil Buffer 28
1. Reflections and Refractions
F. Using the Stencil Buffer 31
1. Dissolves with Stencil 33
2. Decals with Stencil 34
3. Finding Depth Complexity with the Stencil Buffer 35
4. Constructive Solid Geometry with the Stencil Buffer 36
5. Compositing Images with Depth 40
E. Using the Accumulation Buffer 43
1. Motion Blur 44
2. Soft Shadows 45
3. Antialiasing 45
3. Depth of Field 47
4. Convolving with the Accumulation Buffer 49
F. Shadows, 3 Different Flavors 51
1. Projection Shadows 51
2. Shadow Volumes 53

3. Shadow Maps 56
G. Line Rendering Techniques 58
1. Hidden Lines 58
2. Haloed Lines 59
3. Silhouette Lines 60
H. List of Demo Programs 61
I. Equation Appendix 63
J. References 66

Introduction

Since its first release in 1992, OpenGL has been rapidly adopted as the graphics API of choice for real-time interactive 3D graphics applications. The OpenGL state machine is conceptually easy to understand, but its simplicity and orthogonality enable a multitude of interesting effects to be generated. The goal of this course is to demonstrate how to generate more satisfying images using OpenGL. There are three general areas of discussion: generating more athestically pleasing or realistic looking images simply by developing a better understanding of OpenGL's base capabilities and how to exploit them, computing interesting effects or operations by using individual or combining multiple OpenGL operations in straightforward ways, and finally generating more sophisticated images by combining some of OpenGL's capabilities in less obvious ways.

We have assumed that the attendees have a strong working knowledge of OpenGL As much as possible we have tried to include interesting examples involving only those commands in OpenGL 1.0, but we have not restricted ourselves to this version of OpenGL. OpenGL is an evolving standard and we have taken the liberty of incorporating material that uses some multivendor extensions and some vendor specific extensions. The course notes include reprints of selected papers describing rendering techniques relevant to OpenGL, but may refer to other APIs such as OpenGL's predecessor, Silicon Graphics' IRIS GL. For new material developed for the course notes we have attempted to use terminology consistent with other OpenGL documentation.

Acknowledgments

The authors have tried to compile together more than a decade worth of experience, tricks, hacks and wisdom that has often been communicated by word of mouth, code fragments or the occasional magazine or journal article. We are indebted to our colleagues at Silicon Graphics for providing us with interesting material, references, suggestions for improvement, sample programs and cool hardware.

We'd particularly like to thank Allen Akin and Mark Segal for their helpful comments, Rob Wheeler for coding prototype algorithms, Chris Everett for his invaluable production expertise and assistance, And the IRIS Performer Team for covering for one of us while these notes were being written.

Course Notes Web Site

We've created a webpage for this course in SGI's OpenGL web site. It contains an html version of the course notes and downloadable source code for the demo programs mentioned in the text. The web address is:

`http://www.sgi.com/Technology/openGL/advanced_sig96.html`

OpenGL

Before getting into the intricacies of using OpenGL, we will begin by making a few comments about the philosophy behind the OpenGL API and some of the caveats that come with it.

OpenGL is a procedural rather than descriptive interface. In order to get a rendering of a red sphere the program must specify the appropriate sequence of commands to set up the camera view and modelling transformations, draw the geometry for a sphere with a red color. etc. Other APIs such as Renderman [12] are descriptive and one would simply specify that a red sphere should be drawn at coordinates (x,y). The disadvantage of using a procedural interface is that the application must specify all of the operations in exacting detail and in the correct sequence to get the desired result. The advantage of this approach is that it allows great flexibility in the process of generating the image. The application is free to make tradeoffs between rendering speed and image quality by changing the steps through which the image is drawn. The easiest way to demonstrate the power of the procedural interface is to note that a descriptive interface can be built on top of a procedural interface, but not vice-versa.

A second aspect of OpenGL is that the specification is not pixel exact. This means that two different OpenGL implementations are very unlikely to render the same image. The motivation for this is to allow OpenGL to be implemented across a range of hardware platforms. If the specification were too exact, it would limit the kinds of hardware acceleration that could be used and limit its usefulness as a standard. In practice, the lack of exactness need not be a burden - unless you plan to build a rendering farm from a set of machines with different implementations.

The lack of pixel exactness shows up even within a single implementation, in that different paths through the implementation may not generate the same set of fragments although the specification does mandate a set of invariance rules to guarantee repeatable behavior across a variety of circumstances. A concrete example that one might encounter is an implementation that does not accelerate texture mapping operations, but otherwise accelerates all other operations. When texture mapping is enabled the fragment generation is performed on the host and as a consequence all other steps that precede texturing likely also occur on the host possibly resulting in either different algorithms being invoked or the use of arithmetic with different precision than that used in the hardware accelerator. As a result, when texturing is enabled slightly different pixels in the window may be written to compared to when texturing is disabled. For some of the algorithms presented in this course such variability can cause problems, so it is important to understand a little about the underlying details of the OpenGL implementation you are using.

Modelling Considerations

OpenGL is a renderer not a modeler. There are utility libraries such as the OpenGL Utility Library (GLU) which can assist with modelling tasks, but for all practical purposes you are on your own for modelling. Fortunately (or unfortunately), the image quality you will achieve will be directly related to the quality of the modelling. Some of the techniques we will describe can alleviate or hide the effects of, say, undertessellation, but it is inescapable that limitations in a model will show up as artifacts in the rendered image. For example, undertessellated geometry will render poor silhouette edges. Other artifacts can occur due to a combination of the model and the manner in which some of the processing is implemented in OpenGL. For example, interpolation of colors determined as a result of evaluation of a lighting equation at the vertices can result in less than pleasing specular highlight, local viewer, spotlight, and other effects if the geometry is not sufficiently sampled. We include a short list of modelling considerations with which OpenGL programmers should be familiar:

1. Consider using triangles, triangle strips and perhaps triangle fans. Other primitives such as polygons and quads are decomposed into triangles before rasterization. OpenGL does not provide controls over how this decomposition is done, so it is best for the application to do it itself. It is also more efficient if the same model is to be drawn multiple times (e.g., multiple times per frame or for multiple frames). The second release of the GLU library (version 1.1) includes a very good general polygon tesselator; it is highly recommended.

2. Avoid T-intersections (also called T-vertices). T-intersections occur when one or more triangles share (or attempt to) share a partial edge with another triangle. In OpenGL there is no guarantee that a partial edge will share the same pixels since the two edges may be rasterized differently. This problem typically manifests itself during animations when the model is moved and cracks along the edges appear and disappear. In order to avoid the problem, shared edges should share the same vertex coordinates so that the edge equations are the same. Note that this condition needs to be true when seemingly separate models are sharing an edge. For example, an application may have modelled the walls and ceiling of the interior of a room independently, but they do share common edges where they meet. In order to avoid cracking when the room is rendered from different viewpoints, the walls and ceilings should use the same vertex coordinates for any triangles along the shared edges.

3. The T-intersection problem immediately brings up the issue of view-dependent tessellation. Imagine drawing an object in extreme perspective so that some part of the object maps to a large part of the screen and an equally large part of the object (in object coordinates) maps to a small portion of the screen. To minimize the rendering time for this object, one would be inclined to tessellate the object to varying degrees depending on the portion of the screen that it covers. This would ensure that time is not wasted drawing many triangles that cover only a small portion of the screen. This is a difficult mechanism to implement correctly if the view of the object is changing since the changes in tessellation from frame to frame may result in noticeable motion artifacts. The problem is equally difficult within an object and for the case of a single object, such as sphere, being drawn with different screen-space projections. Often it is best to either undertessellate and live with those artifacts or overtessellate and live with the reduced performance. The GLU NURBS package is an example of a library which does implement view-dependent tessella-

tion and provides substantial control over the sampling method and tolerances for the tessellation.

4. Another problem related to the T-intersection problem occurs with careless specification of surface boundaries. If a surface is intended to be closed, it should share the same vertex coordinates where the surface specification starts and ends. A simple example of this would be drawing a sphere by subdividing the interval $(0..2*\pi)$ to generate the vertex coordinates It is important to ensure that the vertex at 0.0 is the same as the one at $2*\pi$. Note that the OpenGL specification is very strict in this regard as even the `MapGrid` routine is specified as evaluating exactly at the boundaries to ensure that evaluated surfaces can be properly stitched together.

5. Another consideration is the quality of the attributes that are specified with the vertex coordinates, in particular, the vertex (or face) normals and texture coordinates. If these attributes are not accurate then shading techniques such as environment mapping will exaggerate the errors resulting in unacceptable artifacts.

6. The final suggestion is to be consistent about the orientation of polygons. That is, ensure that all polygons on a surface are oriented in the same direction (clockwise or counterclockwise) when viewed from the outside. There are a couple of valuable reasons for maintaining this consistency. First the OpenGL face culling method can be used as an efficient form of hidden surface elimination for closed surfaces and, second, several algorithms can exploit the ability to selectively draw only the frontfacing or backfacing polygons of a surface.

Texture Mapping

Texture mapping is one of the main techniques to improve the appearance of objects shaded with OpenGL's simple lighting model. Texturing is typically used to provide color detail for intricate surfaces., e.g, woodgrain, by modifying the surface color. Environment mapping is a view dependent texture mapping technique which modifies the specular and diffuse reflection, i.e. the environment is reflected in the object. More generally texturing can be thought of as a method of perturbing parameters to the shading equation such as the surface normal (bump mapping), or even the coordinates of the point being shaded (displacement mapping). OpenGL 1.0 readily supports the first two techniques (surface color manipulation and environment mapping). Texture mapping can also be used to solve some rendering problems in less obvious ways. This section will review some of the details of OpenGL texturing support, outline some considerations when using texturing and suggest some interesting algorithms using texturing.

Review

OpenGL supports texture images which are 1D or 2D and have dimensions that are a power of two. Some implementations have been extended to support 3D and 4D textures. Texture coordinates are assigned to the vertices of all primitives (including pixel images). The texture coordinates are part of a three dimensional homogeneous coordinate system (s,t,r,q). When a primitive is rasterized a texture coordinate is computed for each pixel fragment. The texture coordinate is used to look up a texel value from the currently enabled texture map. The coordinates of the texture map range from [0,1]. OpenGL can treat coordinate values outside the range of [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. The texel value that results from the lookup can be used to modify the original surface color value in one of several ways, the simplest being to replace 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.

OpenGL also 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 will choose some filter which does interpolation. For magnification, OpenGL 1.0 only supports linear interpolation between four texel values. Some vendors have also added support for bicubic filtering in which the a weighted sum of 4x4 array of texels is used (Filter4 is a more appropriate name for it since it is only performing cubic filtering when used as a magnification filter). For minification, OpenGL 1.0 supports various types of mipmapping [26], with the most useful (and computationally expensive) being trilinear mipmapping (4 samples taken from each of the nearest two mipmap levels and then interpolating the two sets of samples). OpenGL does not provide any built-in commands for generating mipmaps, but the GLU provides some simple routines for generating mipmaps using a simple box filter.

The process by which the final fragment color value is derived is referred to as the texture environment function (`TexEnv`) Several methods exist for computing the final color, each capable of producing a particular effect. One of the most commonly used is the modulate function. For all practical purposes the modulate function can be thought of multiplying or modulating the original fragment color with the texel color. Typically, applications generate white polygons, light them, and then use this lit value to modulate the texture image to effectively produce a lit, textured surface. Unfortunately when the lit polygon includes a specular highlight, the resulting modulated texture will not look correct since the specular highlight simply changes the brightness of the texture at that point rather than the desired effect of adding in some specular illumination. Some vendors have tried to address this problem with extensions to perform specular lighting after texturing. We will discuss some other techniques which can be used to address this problem later on.

The 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 luminance (L) and luminance alpha (LA) textures. Conversely, the blend environment function is only defined for L and LA textures, using the luminance value as a blending factor to combine the incoming fragment color and a constant texture environment color..

Another useful (and sometimes misunderstood) feature of OpenGL is the concept of a texture border. OpenGL supports either a constant texture border color or a portion of the edge of the texture image. The key to understanding texture borders is understanding how textures are sampled when the texture coordinate values are near the edges of the [0,1] range and the texture wrap mode is set to `CLAMP`. For point sampled filters, the computation is quite simple: the border is never sampled. However, when the texture filter is linear and the texture coordinate reaches the extremes (0.0 or 1.0) the resulting texel value will be a 50% mix of the border color and the outer texel of the texture image at that edge.

This is most useful when attempting to use a single high resolution texture image which is too large for the OpenGL implementation to support as a single texture map. For this case, the texture can be broken up into multiple tiles, each with a 1 pixel wide border from the neighboring tiles. The texture tiles can then be loaded and used for rendering in several passes. For example, if a 1K by 1K texture is broken up into 4 512 by 512 images, the 4 images would correspond to the texture coordinate ranges (0-0.5,0-0.5), (0.5,1.0,0-0.5), (0-0.5,0.5,1.0) and (.5-1.0,.5-1.0). As each tile is loaded, only the portions of the geometry that correspond to the appropriate texture coordinate ranges for a given tile should be drawn. If we had a single triangle whose texture coordinates were (.1,.1), (.1,.7), and (.8,.8) we would clip the triangle against the 4 tile regions and draw only the portion of the triangle that intersects with that region as shown in Figure 1. At the same time, the original texture coordinates need to be adjusted to correspond to the scaled and translated texture space represented by the tile. This transformation can be easily performed by loading the appropriate scale and translation onto the texture matrix stack.

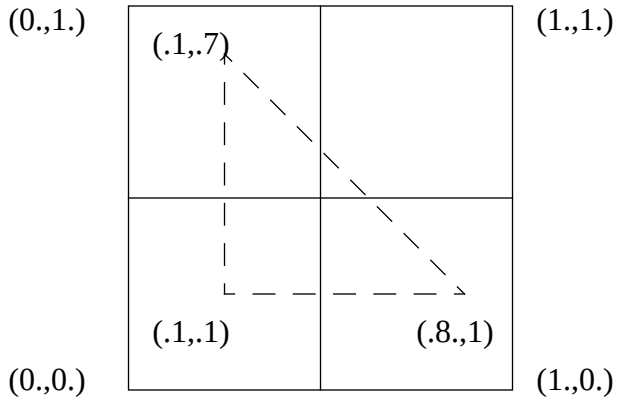


Figure 1: Texture Tiling

Unfortunately, OpenGL doesn't provide much assistance for performing the clipping operation. If the input primitives are quads and they are appropriately aligned in object space with the texture, then the clipping operation is trivial; otherwise, it is substantially more work. One method to assist with the clipping would involve using stenciling to control which textured fragments are kept. Then we are left with the problem of setting the stencil bits appropriately. The easiest way to do this is to produce alpha values which are proportional to the texture coordinate values and use `AlphaFunc` to reject alpha values that we do not wish to keep. Unfortunately, we can't easily map a multidimensional texture coordinate value (e.g. s,t) to an alpha value by simply interpolating the original vertex alpha values, so it would be best to use a multidimensional texture itself which has some portion of the texture with zero alpha and some portion with it equal to one. The texture coordinates are then scaled so that the textured polygon will map to texels with an alpha of 1.0 for pixels to be retained and 0.0 for pixels to be rejected.

Mipmap Generation

Having explored the possibility of tiling low resolution textures to achieve the effect of high resolution textures, we can next examine methods for generating better texturing results without resorting to tiling. Again, OpenGL supports a modest collection of filtering algorithms, the best of the minification algorithms being `LINEAR_MIPMAP_LINEAR`. OpenGL does not specify a method for generating the individual mipmap levels (LODs). Each level can be loaded individually, so it is possible, but probably not desirable, to use a different filtering algorithm to generate each mipmap level.

The GLU library provides a very simple interface (`gluBuild2DMipmaps`) for generating all of the 2D levels required. The algorithm currently employed by most implementations is a box filter. There are a number of advantages to using the box filter; it is simple, efficient, and can be repeat-

edly applied to the current level to generate the next level without introducing filtering errors. However, the box filter has a number of limitations that can be quite noticeable with certain textures. For example, if a texture contains very narrow features (e.g., lines), then aliasing artifacts may be very pronounced.

The best choice of filter functions for generating mipmap levels is somewhat dependent on the manner in which the texture will be used and it is also somewhat subjective. Some possibilities include using a linear filter (sum of 4 pixels with weights [1/8,3/8,3/8,1/8]) or a cubic filter (weighted sum of 8 pixels). Mitchell and Netravali [11] propose a family of cubic filters for general image reconstruction which can be used for mipmap generation. The advantage of the cubic filter over the box is that it can have negative side lobes (weights) which help maintain sharpness while reducing the image. This can help reduce some of the blurring effect of filtering with mipmaps.

When attempting to use a different filtering algorithm, it is important to keep a couple of things in mind. The highest resolution image of the mipmap (LOD 0) should always be used as the input image source for each level to be generated. For the box filter, the correct result is generated when the preceding level is used as the input image for generating the next level, but this is typically not true for other filter functions. This means each time a new level is generated, the filter needs to be scaled to twice the width of the previous version of the filter. A second consideration is that in order to maintain a strict factor of two reduction the filters with widths wider than 2 will need to sample outside the boundaries of the image. This is commonly dealt with by using the value for the nearest edge pixel when sampling outside the image. However, a more correct algorithm can be selected depending on whether the image is to be used in a texture in which a repeat or clamp wrap mode is to be used. In the case of repeat, requests for pixels outside the image should wrap around to the appropriate pixel counted from the opposite edge, effectively repeating the image.

Mipmaps may be generated using the host processor or using the OpenGL pipeline to perform some of the filtering operations. For example, the `LINEAR` minification filter can be used to draw an image of exactly half the width and height of an image which has been loaded into texture memory, by drawing a quad with the appropriate transformation (i.e., the quads projects to a rectangle one fourth the area of the original image). This effectively filters the image with a box filter. The resulting image can then be read from the color buffer back to host memory for later use as LOD 1. This process can be repeated using the newly generated mipmap level to produce the next level and so on until the coarsest level has been generated.

The above scheme seems a little cumbersome since each generated mipmap level needs to be read back to the host and then loaded into texture memory before it can be used to create the next level. Some vendors have added an OpenGL extension to allow an image in the color buffer to be copied directly to texture memory. This `CopyTexture` capability has also been added to the core capability of OpenGL 1.0.

This process can still be slightly difficult since OpenGL 1.0 only allows a single texture of a given dimension (1D, 2D) to exist at any one time, making it difficult to build up the mipmap texture while using the non-mipmapped texture for drawing. This problem is solved by the texture object extension which allows multiple texture definitions to coexist at the same time. However, it would

be much simpler if we could simply use the most recent level loaded as part of the mipmap as the current texture used for drawing. OpenGL 1.0 only allows *complete* textures to be used for texturing, meaning that all mipmap levels need to be defined. Some vendors have added yet another extension which can deal with this problem (though that was not the original intent behind the extension). This third extension, the texture lod extension, limits the selection of mipmap image arrays to a subset of the arrays that would normally be considered; that is, it allows an application to specify a contiguous subset of the mipmap levels to be used for texturing. If the subset is *complete* then the texture can be used for drawing. Therefore we can use this extension to limit the mipmap images to the level most recently created and use this to create the next smaller level. The other capability of the lod extension is the ability to clamp the LOD to a specified floating point range so that the entire filtering operation can be restricted. This extension will be discussed in more detail later on.

The above method outlines an algorithm for generating mipmap levels using the existing texture filters. There are other mechanisms within the OpenGL pipeline that can be combined to do the filtering. Convolution can be implemented using the accumulation buffer (this will be discussed in more detail in the section on the accumulation buffer). A texture image can be drawn using a point sampling filter (NEAREST) and the result added to the accumulation buffer with the appropriate weighting. Different pixels (texels) from an NxN pattern can be selected from the texture by drawing a quad that projects to a region 1/N x 1/N of the original texture width and height with a slight offset in the s and t coordinates to control the nearest sampling. Each time a textured quad is rendered to the color buffer it is accumulated with the appropriate weight in the accumulation buffer. Combining point sampled texturing with the accumulation buffer allows the implementation of nearly arbitrary filter kernels. Sampling outside the image, however, still remains a difficulty for wide filter kernels. If the outside samples are generated by wrapping to the opposite edge, then the REPEAT wrap mode can be used.

Fine Tuning

In addition to issues concerning the maximum texture resolution and the methods used for generating texture images there are also some pragmatic details with using texturing. Many OpenGL implementations hardware accelerate texture mapping and have finite storage for texture maps being used. Many implementations will virtualize this resource so that an arbitrarily large set of texture maps can be supported within an application, but as the resource becomes oversubscribed performance will degrade. In applications that need to use multiple texture maps there is a tension between the available storage resources and the desire for improved image quality.

This simply means that its unlikely that every texture map can have an arbitrarily high resolution and still fit within the storage constraints; therefore, applications need to anticipate how textures will be used in scenes to determine the appropriate resolution to use. Note that texture maps need not be square; if a texture is typically used with an object that is projected to a non-square aspect ratio then the aspect ratio of the texture can be scaled appropriately to make more efficient use of the available storage.

Paging Textures

Imagine trying to draw an object which is covered by a portion of an arbitrary large 2D texture. This type of problem typically occurs when rendering terrain or attempting to pan over very large images. If the texture is arbitrarily large it will not entirely fit into texture memory unless it is dramatically reduced in size. Rather than suffer the degradation in image quality by using the smaller texture, it might be possible to only use the subregion of the texture that is currently visible. This is somewhat similar to the texture tiling problem discussed earlier, but rather than sequence through all of the tiles for each frame only the set of tiles necessary to draw the image need to be used [17].

There are two different approaches that could be used to address the problem. The first is to subdivide the texture into fixed sized tiles and selectively draw the portion of the geometry that intersects each tile as that tile is loaded. As discussed previously, this is difficult for `LINEAR` filters since the locations where the geometry crosses tile boundaries need to be resampled properly. The problem could be addressed by clipping the geometry so that the texture coordinates are kept within the [0.0, 1.0] range and then use borders to handle the edges, or a single large texture consisting of all of the tiles could be used and the clipping step could be avoided.

This latter solution is not very practical with OpenGL 1.0 since the entire texture needs to be reloaded each time a new tile needs to be added, but it is addressed by the incremental loading capability added to OpenGL 1.1 and added to several OpenGL 1.0 implementations as an extension. This `TexSubImage` routine allows a subregion within an existing texture image to be updated. This make it simple to load new tiles into areas that are no longer needed to draw the image. The ability to update portions of the texture doesn’t completely solve the problem. Consider the case of a two dimensional image roam, illustrated in Figure 2, in which the view is moving to the right. As the view pans to the right, new texture tiles must be added to the right edge of the current portion of the texture and old tiles could be discarded from the left edge.

Tiles discarded on the right side of the image create holes where new tiles could be loaded into the texture, but there is a problem with the texture coordinates. Tiles loaded at the left end will correspond to low values of the t texture coordinate, but the geometry may be drawn with a single command or perhaps using automatic texture coordinate generation expecting to index those tiles with higher values of the t coordinate. The solution to the problem is to use the repeat texture mode and let the texture coordinates for the geometry extend past 1.0
The texture memory simply wraps back onto itself in a toroidal topology. The origin of the texture coordinates for the geometry must be adjusted as the leading and trailing edges of the tiles cycle through texture memory. The translation of the origin can be done using the texture matrix.

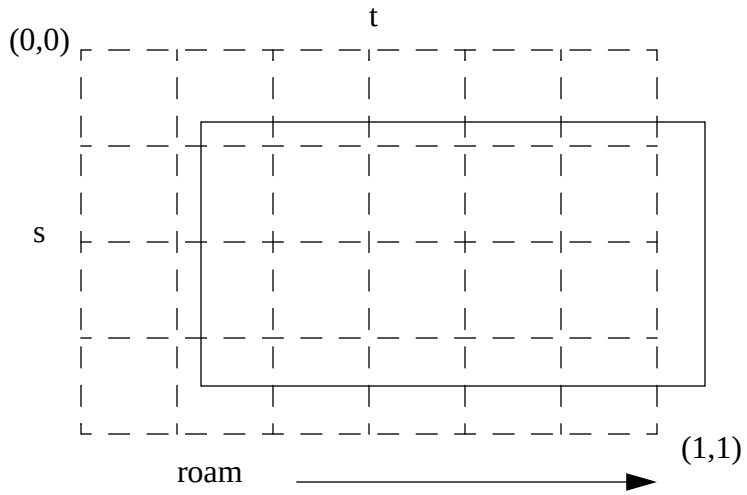


Figure 2: 2D image roam

The algorithm works for both mipmap and non-mipmapped textures but for the former, tiles corresponding to each level of detail must be loaded together.

The ability to load subregions within a texture has other uses besides these paging applications. Without this capability textures must be loaded in their entirety and their widths and heights must be powers of two. In the case of video data, the images are typically not powers of two so an texture of the nearest larger power of two can be created and only the relevant subregion needs to be loaded. When drawing geometry, the texture coordinates are simply constrained to fraction of the texture which is occupied with valid data. Mipmapping can not easily be used with non-power-of-two image data since the coarser levels will contain image data from the invalid region of the texture.

Transparency Mapping and Trimming with Alpha

The alpha component in textures can be used to solve a number of interesting problems. Intricate shapes such as an image of a tree can be stored in texture memory with the alpha component acting as a matte (1.0 where there the image is opaque, 0. where it is transparent, and a fractional value along the edges). When the texture is applied to geometry, blending can be used to composite the image into the color buffer or the alpha test can be used to discard pixels with a zero alpha component using the EQUALS test. The advantage of using the alpha test over blending is that typically blending degrades the performance of fragment processing. With alpha testing fragments with zero alpha are rejected before they get to the color buffer. A disadvantage of alpha testing is that the edges will not be blended into the scene so the edges will not be properly anti-aliased.

The alpha component of a texture can be used in other ways, for example, to cut holes in polygons

or to trim surfaces. An image of the trim region is stored in a texture map and when it is applied to the surface, alpha testing or blending can be used to reject the trimmed region. This method can be useful for trimming complex surfaces in scientific visualization applications.

Rendering Text

A novel use for texturing is rendering anti-aliased text [27] Characters are stored in a 2D texture map as for the tree image described above. When a character is to be rendered, a polygon of the desired size is texture mapped with the character image. Since the texture image is filtered as part of the texture mapping process the quality of the rendered character can be quite good. Text strings can be drawn efficiently by storing an entire character set within a single texture. Rendering a string then becomes rendering a set of quads with the vertex texture coordinates determined by the position of each character in the texture image. Another advantage of this method is that strings of characters may be arbitrarily oriented and positioned in three dimensions by orienting and position the polygons.

The competing methods for drawing text in OpenGL include bitmaps, vector fonts, and outline fonts rendered as polygons. The texture method is typically faster than bitmaps and comparable to vector and outline fonts. A disadvantage of the texture method is that the texture filtering may make the text appear somewhat blurry. This can be alleviated by taking more care when generating the texture maps (e.g. sharpening them). If mipmaps are constructed with multiple characters stored in the same texture map, care must be taken to ensure that map levels are clamped to the level where the image of a character has been reduced to 1 pixel on a side. Characters should also be spaced far enough apart that the color from one character does not contribute to that of another when filter the images to produce the levels of detail.

Projective Textures

Projective textures [10] use texture coordinates which are computed as the result of a projection. The result is that the texture image can be subjected to a separate independent projection from the viewing projection. This technique may be used to simulate effects such as slide projector or spotlight illumination, to generate shadows, and to reproject a photograph of an object back onto the geometry of the object. Several of these techniques are described in more detail in later sections of these notes.

OpenGL generalizes the two component texture coordinate (s,t) to a four component homogeneous texture coordinate (s,t,r,q). The q coordinate is analogous to the w component in the vertex coordinates. The r coordinate is used for three dimensional texturing in implementations that support that extension and is iterated in manner similar to s and t. The addition of the q coordinate adds very little extra work to the usual texture mapping process. Rather than iterating (s,t,r) and dividing by 1/w at each pixel, the division becomes a division by q/w. Thus, in implementations that perform perspective correction there is no extra rasterization burden associated with processing q.

Environment Mapping

OpenGL directly supports environment mapping using spherical environment maps. A sphere maps is a single texture of a perfectly reflecting sphere in the environment where the viewer is infinitely far from the sphere. The environment behind the viewer (a hemisphere) is mapped to a circle in the center of the map. The hemisphere in front of the viewer is mapped to a ring surrounding the circle. Sphere maps can be generated using a camera with an extremely wide-angle (or fish eye) lens. Sphere map approximations can also be generated from a six-sided (or cube) environment map by using texture mapping to project the six cube faces onto a sphere.

OpenGL provides a texture generation function (`SPHERE_MAP`) which maps a vertex normal to a point on the sphere map. Applications can use this capability to do simple reflection mapping (shade totally reflective surfaces) or use the framework to do more elaborate shading such as Phong lighting [28]. We discuss this algorithm in a later section.

Image Warping and Dewarping

Image warping or dewarping may be implemented using texture mapping by defining a correspondence between a uniform polygonal mesh and a warped mesh. The points of the warped mesh are assigned the corresponding texture coordinates of the uniform mesh and the mesh is texture mapped with the original image. Using this technique simple transformations such as zoom, rotation or shearing can be efficiently implemented. The technique also easily extends to much higher order warps such as those needed to correct distortion in satellite imagery.

Blending

OpenGL provides a rich set of blending operations which can be used to implement transparency, compositing, painting, etc. Rasterized fragments are linearly combined with pixels in the selected color buffers, clamped to 1.0 and written back to the color buffers. The glBlendFunc command is used to select the source and destination blend factors. The most frequently used factors are `ZERO`, `ONE`, `SRC_ALPHA` and `ONE_MINUS_SRC_ALPHA`. OpenGL 1.0 specifies additive blending, but other vendors have added extensions to allow other blending equations such as subtraction and reverse subtraction

Most OpenGL implementations use fixed point representations for color throughout the fragment processing path. The color component resolution is typically 8 or 12 bits. Resolution problems typically show up when attempting to blend many images into the color buffer, for example in some volume rendering techniques or multilayer composites. Some of these problems can be alleviated using the accumulation buffer instead, but the accumulation buffer does not provide the same richness of methods for building up results.

OpenGL does not require that implementations support a destination alpha buffer (storage for alpha). For many applications this is not a limitation, but there is a class of multipass operations where maintaining the current computed alpha value is necessary.

Compositing

The OpenGL blending operation does not directly implement the compositing operations described by Porter and Duff [3]. The difference is that in their compositing operations the colors are premultiplied by the alpha value and the resulting factors used to scale the colors are simplified when this scaling has been done. It has been proposed that OpenGL be extended to include the ability to premultiply the source color values by alpha to better match the Porter and Duff operations. In the meantime, its certainly possible to achieve the same effect by computing the premultiplied values in the color buffer itself. For example, if there is an image in the color buffer, a new image can be generated which multiplies each color component by its alpha value and leaves the alpha value unchanged by performing a CopyPixels operation with blending enabled and the blending function set to (SRC_ALPHA,ZERO). To ensure that the original alpha value is left intact, use the ColorMask command to disable updates to the alpha component during the copy operation. Further, on implementations which support the ColorMatrix extension it is simple to configure the color matrix to multiply source red, green and blue by alpha and leave alpha unchanged, before the pixels are sent to texture memory or the color buffer.

Translucency

To draw translucent geometry, the most common technique is to draw the objects from back to front using the blending factors SRC_ALPHA, ONE_MINUS_SRC_ALPHA. As each object is drawn the alpha value for each fragment should reflect the translucency of that object. If lighting is not being used, then the alpha value can be set using a 4 component color command. If lighting is enabled, then the ambient and diffuse reflectance coefficients of the material should correspond

to the translucency of the object.

The sorting step itself can be quite complicated. The sorting should be done in eye coordinates, so it is necessary to transform the geometry to eye coordinates in some fashion. If translucent objects interpenetrate then the individual triangles should be sorted and drawn from back to front. Ideally, polygons which interpenetrate should be tessellated along their intersections, sorted and drawn independently, but this is typically not required to get good results. In fact, frequently only crude or perhaps no sorting at all gives acceptable results. Its the usual time versus quality tradeoff.

Drawing depth buffered opaque objects mixed with translucent objects takes somewhat more care. The usual trick is to draw the background and opaque objects first in any order with depth testing enabled and depth buffer updates enabled and blending disabled Next the translucent objects are drawn from back to front with blending enabled, depth testing enabled but depth buffer updates disabled so that translucent objects do not occlude each other.

Advanced Blending

OpenGL 1.0 blending only allows simple additive combinations of the source and destination color components. Two ways in which the blending operations have been extended by other vendors include the ability to blend with a constant color and the ability to use other blending equations. The blend color extension adds a constant RGBA color state variable which can be used as a blending factor in the blend equation. This capability can be very useful for implementing blends between two images without needing to specify the individual source and destination alpha components on a per pixel basis.

The blend equation extension provides the framework for specifying alternate blending equations, for example subtractive rather than additive. In OpenGL 1.0, the accumulation buffer is the only mechanism which allows pixel values to be subtracted, but there is no easy method to include a per-pixel scaling factor such as alpha, so its easy to imagine a number of uses for a subtractive blending equation. Other equations which have been implemented include min and max functions which can be useful in image processing algorithms (e.g., for computing maximum intensity projections).

Image Processing

Some interesting image enhancement operations can be implemented using the linear interpolation (or blending function) $A*\alpha+B*(1-\alpha)$ [29]. The method described by Haeberli extends the value of α outside the normal [0.0, 1.0] range to perform extrapolation as well as interpolation. Since OpenGL 1.0 (and 1.1) do not support alpha values out of the [0.0, 1.0] range these techniques need to be implemented using the accumulation buffer. Some of the examples include

1. Brightness - by combining an image with a black (degenerate) image, interpolation darkens the image and extrapolation brightens it. In both cases, brighter pixels are affected more. If a white image is used instead of black, then interpolation lightens and desaturates the image

while extrapolation darkens it. In this case the darker pixels are more affected.
2. Contrast - by combining an image with a constant gray image equal to the average image luminance, interpolation reduces contrast and extrapolation boosts it. Negative α generates inverted images with varying contrast. The average luminance always remains constant.
3. Saturation - using a black and white image as the degenerate image, saturation can be decreased using interpolation and increased using extrapolation. Negative α preserves luminance but inverts the hue.
4. Color Balance - color balance can be altered by using a tinted version of the image as a degenerate one. The tinted (degenerate) image should have scaled RGB components but preserve the luminance in order to maintain the global color balance.
5. Sharpening - using a blurred image as the degenerate image, interpolation attenuates high frequencies and extrapolation boosts them (sharpening the image by unsharp masking). α acts as a kernel scale factor, so a series of convolutions differing in scale can be easily done, independent of the size of the kernel.

The image processing capability in OpenGL is experiencing the fastest growth. Multiple vendors have implemented extensions to perform convolution, histogramming, minimum and maximum pixel value calculations, lookup table variants (in addition to pixel maps).

Painting

Two dimensional painting applications can make interesting use of texturing and blending. An arbitrary image can be used as a paint brush, using blending to accumulate the contribution over time. The image (or paint brush) source can be geometry, a pixel image. A texture mapped quad under an orthographic projection can be used in the same way as a pixel image and often more efficiently (when texture mapping is hardware accelerated).

An interesting way to implement the painting process is to precompute the effect of painting the entire image with the brush and then use blending to selectively expose the painted area as the brush passes by the area. This can be implemented efficiently with texturing by using the fully painted image as a texture map, blending the image of it mapped on the brush with the current image stored in the color buffer. The brush is some simple geometric shape and the (s,t) texture coordinates track the (x,y) coordinates as they move across the image. The main advantage of this techniques is that elaborate paint/brush combinations can be efficiently computed across the entire image all at once rather than performing localized computations in the area covered by the brush.

Anti-Aliasing

Most of the methods for rendering anti-aliased primitives in OpenGL rely on blending the primitives in sorted order (either back to front or front to back). Anti-aliased points and lines typically use a lookup table technique to compute the fraction of each pixel covered by the primitive. The coverage value is then used to determined an alpha value to be used in the blending computation. Blended lines and points typically require some care when mixing them with depth buffered primitives. Usually the depth buffered primitives are drawn first and then the points and lines are

drawn with the depth test still enabled and the depth buffer updates disabled so that the points and lines are correctly depth buffered against the rest of the geometry. This is a similar algorithm to that used for drawing a mixture of opaque and translucent surfaces with depth buffering. If anti-aliased lines are drawn with the normal depth buffering algorithm a halo artifact may be visible at the intersections of lines. This halo is a result of the anti-aliased lines being drawn several pixels wide with the pixels along the edges of the line having attenuated alpha values which can also occlude pixels with larger depth values (i.e., parts of other lines). When drawing anti-aliased lines it is often necessary to adjust the gamma of the monitor to get the best results.

Anti-aliased polygons can be rendered using a similar technique to the ones for lines and points in which coverage for the edge pixels are computed and converted to an alpha value for subsequent blending. Polygons should be drawn in front to back order with the blend function (SRC_ALPHA_SATURATE, ONE). Since the accumulated coverage is stored in the color buffer, destination alpha is required for this algorithm to work.

There are also some alternatives to these anti-aliasing methods. One of these, the accumulation buffer, is described in more detail later on. Another method using super-sampling is available as an OpenGL 1.0 extension from one vendor [5]. In this technique additional subpixel storage is maintained as part of the color, depth and stencil buffers. Instead of using alpha for coverage, coverage masks are computed to help maintain sub-pixel coverage information for all pixels. The implementations available so far support four, eight, and sixteen samples per pixel. The method allows for full scene anti-aliasing at a modest performance penalty but a more substantial storage penalty (since sub-pixel samples of color, depth, and stencil need to be maintained for every pixel). This technique does not entirely replace the methods described above, but is complementary. Anti-aliased lines and points using alpha coverage can be mixed with super-sampling as well as the accumulation buffer anti-aliasing method.

The *Z* Coordinate and Perspective Projection

The *z* coordinates are treated in the same fashion as the *x* and *y* coordinates. After transformation, clipping and perspective division, they occupy the range -1.0 through 1.0. The `DepthRange` mapping specifies a transformation for the *z* coordinate similar to the viewport transformation used to map x and y to window coordinates. The `DepthRange` mapping is somewhat different from the viewport mapping in that the hardware resolution of the depth buffer is hidden from the application. The parameters to the DepthRange call are in the range [0.0, 1.0]. The *z* or depth associated with a fragment represents the distance to the eye. By default the fragments nearest the eye (the ones at the near clip plane) are mapped to 0.0 and the fragments farthest from the eye (those at the far clip plane) are mapped to 1.0. Fragments can be mapped to a subset of the depth buffer range by using smaller values in the `DepthRange` call. The mapping may be also be reversed so that fragments furthest from the eye are at 0.0 and fragments closest to the eye are at 1.0 simply by calling `DepthRange(1.0,0.0)`. While this reversal is possible, it may not be practical for the implementation. Parts of the underlying architecture may have been tuned for the forward mapping and may not produce results of the same quality when the mapping is reversed.

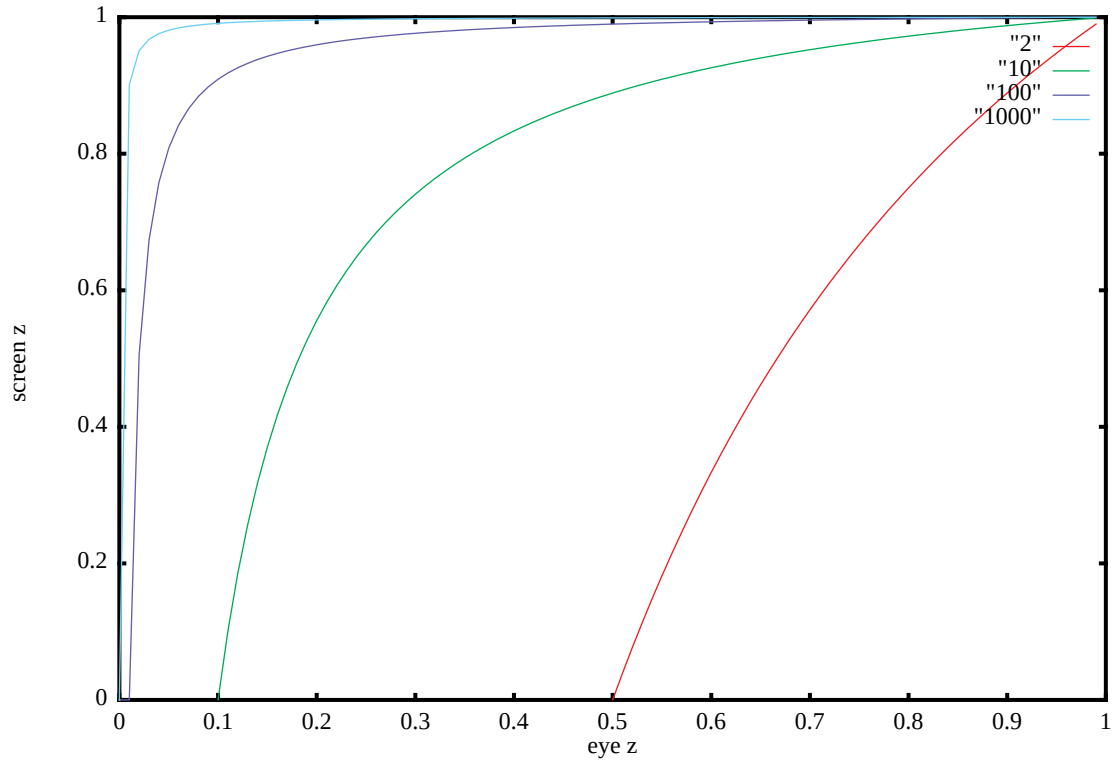


Figure 3: The relationship of screen z (depth) to eye z for different far/near ratios

To understand why there might be this disparity in the rendering quality, its important to understand the characteristics of the screen space *z* coordinate. The z value does specify the distance from the fragment to the eye. The relationship between distance and *z* is linear in an orthographic projection, but not in a perspective projection. In the case of a perspective projection the relation-

ship is non-linear and the degree of the non-linearity is proportional to the ratio of *far* to *near* in the `Frustum` call (or `zFar` to `zNear` in the `gluPerspective` call). Figure 4 plots the screen space *z* value as a function of the eye-to-pixel distance for several ratios of *far* to *near.*
The non-linearity increases the precision of the z-values when they are close to the near clipping plane, increasing the effectiveness of the depth buffer, but decreasing the precision throughout the rest of the viewing frustum, thus decreasing the accuracy of the depth buffer in this part of the viewing volume. Empirically it has been observed that ratios greater than 1000 have this undesired result.

The simplest solution is to improve the far to near ratio by moving the near clipping plane away from the eye. The only negative effect of moving the near clipping plane further from the eye is that objects which are rendered very close to the eye may be clipped away, but this is seldom a problem in typical applications. The position of the near clipping plane has no effect on the projection of the x and y coordinates and therefore has minimal effect on the image.

In addition to depth buffering, the *z* coordinate is also used for fog computations. Some implementations may perform the fog computation on a per-vertex basis using eye *z* and then interpolate the resulting colors whereas other implementations may perform the computation for each fragment. In this case the implementation may use the screen *z* to perform the fog computation. Implementations may also choose to convert the computation into a cheaper table lookup operation which can also cause difficulties with the non-linear nature of screen *z* under perspective projections. If the implementation uses a linearly indexed table, large far to near ratios will leave few table entries for the large eye *z* values.

Depth Buffering

We have discussed some of the caveats of using the screen z value in computations, but there are several other aspects of OpenGL rasterization and depth buffering that are worth mentioning [24]. One big problem is that the rasterization process uses inexact arithmetic so it is exceedingly difficult to deal with primitives that are coplanar unless they share the same plane equation. This problem is further exacerbated by the finite precision of depth buffer implementations. Many solutions have been proposed to deal with a class of problems that potentially involve coplanar primitives such as:

1. decaling
2. hidden line elimination
3. outlined polygons
4. shadows

Many of these problems have elegant solutions involving the stencil buffer, but it is still worth describing some alternative methods if only to get more insight into the uses of the depth buffer.

The decal problem can be solved rather simply by using the painter’s algorithm (i.e. drawing from back to front) combined with color buffer and depth buffer masking, assuming the decal is contained entirely with in the underlying polygon. The steps are:

1. draw the underlying polygon with depth testing enabled but depth buffer updates disabled.
2. draw the top layer polygon (decal) also with depth testing enabled and depth buffer updates still disabled.
3. draw the underlying polygon one more time with depth testing and depth buffer updates enabled, but color buffer updates disabled.
4. enable color buffer updates and continue on.

Outlining a polygon and drawing hidden lines are similar problems. If we have an algorithm to outline polygons hidden lines can be drawn by outline polygons with one color and drawing the polygons with the background color. Ideally a polygon could be outlined by simply connecting the vertices together with line primitives. This is similar to the decaling problem except that edges of the polygon being outlined may be shared with other polygons and those polygons may not be coplanar with the outlined polygon, so the decaling algorithm can not be used since it relies on the coplanar decal being contained within the polygon. The solution most frequently suggested for this problem is to simply draw the outline as a series of lines and translate the outline a small amount towards the eye, or translate the polygon away from the eye. Besides not being a particularly elegant solution, there is a problem in determining the amount to translate the polygon (or outline). In fact, in the general case there is no constant amount that can be expressed as a simple translation of the z object coordinate that will work for all polygons in a scene.

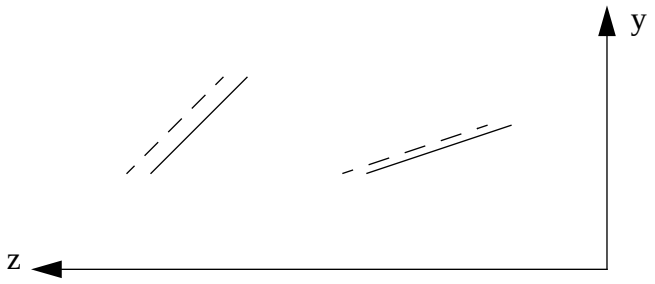


Figure 4: Polygon and Outline Slopes

Figure 3 shows two polygons (solid) with outlines (dashed) in the screen space y-z plane. One of the primitive pairs has a 45-degree slope in the y-z plane and the other has a very steep slope. During the rasterization process a the depth value for a given fragment may be derived from a sample point nearly an entire pixel away from the edge of the polygon,. Therefore the translation must be as large as the maximum absolute change in depth for any single pixel step on the face of the polygon. From the figure its fairly obvious that the steeper the depth slope the larger the required translation. If an unduly large constant value is used to deal with steep depth slopes, then for polygons which have a shallower slope there is an increased likelihood that another neighboring polygon might end up interposed between the outline and the polygon. So it seems that a translation proportional to the depth slope is necessary. However, a translation proportional to slope is not sufficient for a polygon that has constant depth (zero slope) since it would not be translated at all. Therefore a bias is also needed. Many vendors have implemented an extension which provides a scaled slope plus bias capability for solving outline problems such as these and for other applications. A modified versions of the polygon offset extension has been added to the core of OpenGL 1.1 as well.

Multipass Rendering

A number of the techniques that we have described thus far are examples of multipass techniques. The final image is computed by rendering some or all of the objects in the scene multiple times (passes) carefully controlling the processing steps used in the OpenGL pipeline for each element rendered. We can think of a pass as being one rendering traversal through the scene geometry. Multipass techniques are useful when the order of operations in the pipeline do not agree with the order of operations that are required to render the image. The OpenGL pipeline can be simply viewed as a sequence of optional operations and each time a pass is added there is another opportunity to use one or more of the operations in the pipeline. This flexibility is one of the features that makes OpenGL so powerful. In the following examples we will demonstrate some more sophisticated shading techniques using multipass methods.

Phong Highlights

One of the problems with the OpenGL lighting model is that specular reflectance is computed before textures are applied in the normal pipeline sequence. To achieve more realistic looking results, specular highlights should be computed and added to image after the texture has been applied. This can be accomplished by breaking the shading process into two passes. In the first pass diffuse reflectance is computed for each surface and then modulated by the texture colors to be applied to the surface and the result written to the color buffer. In the second pass the specular highlight is computed for each polygon and added to the image in the frame using a blending function which sums 100% of the source fragment and 100% of the destination pixels. For this particular example we will use an infinite light and a local viewer. The steps to produce the image are as follows:

1.define the material with appropriate diffuse and ambient reflectance and zero for the specular reflectance coefficients
2.define and enable lights
3. define and enable texture to be combined with diffuse lighting
4.define modulate texture environment
5.draw lit, textured object into the color buffer with the vertex colors set to 1.0
6.define new material with appropriate specular and shininess and zero for diffuse and ambient reflectance.
7.disable texturing, enable blending, set the blend function to `ONE`, `ONE`
8.draw the specular-lit, non-textured geometry.
9.disable blending

This implements the basic algorithm, but the Gouraud shaded specular highlight still leaves something to be desired. We can improve on the specular highlight by using environment mapping to generate a higher quality highlight. We generate a sphere map consisting only of a Phong highlight [4] and then use the `SPHERE_MAP` texture coordinate generation mode to generate texture coordinates which index this map. For each polygon in the object, the reflection vector is computed at each vertex. Since the coordinates of the vector are interpolated across the polygon and used to lookup the highlight, a much more accurate sampling of the highlight is achieved compared to interpolation of the highlight value itself. The sphere map image for the texture map of

the highlight can be computed by rendering a highly tessellated sphere lit with only a specular highlight using the regular OpenGL pipeline. Since the light position is effectively encoded in the texture map, the texture map needs to be recomputed whenever the light position is changed.

The nine step method outlined above needs minor modifications to incorporate the new lighting method:

6. disable lighting.
7. load the sphere map texture, enable the sphere map texgen function.
8. enable blending, set the blend function to `ONE`, `ONE`.
9. draw the unlit, textured geometry with vertex colors set to 1.0.
10. disable texgen, disable blending.

With a little work the technique can be extended to handle multiple light sources.

Spotlight Effects using Projective Textures

The projective texture technique described earlier can be used to generate a number of interesting illumination effects. One of the possible effects is spotlight illumination. The OpenGL lighting model already includes a spotlight illumination model, providing control over the cutoff angle (spread of the cone), the exponent (concentration across the cone), direction of the spotlight, and attenuation as a function of distance. The OpenGL model typically suffers from undersampling of the light. Since the lighting model is only evaluated at the vertices and the results are linearly interpolated, if the geometry being illuminated is not sufficiently tessellated incorrect illumination contributions are computed. This typically manifests itself by a dull appearance across the illuminated area or irregular or poorly defined edges at the perimeter of the illuminated area. Since the projective method samples the illumination at each pixel the undersampling problem is eliminated.

Similar to the Phong highlight method, a suitable texture map must be generated. The texture is an intensity map of a cross-section of the spotlight's beam. The same type of exponent parameter used in the OpenGL model can be incorporated or a different model entirely can be used. If 3D textures are available the attenuation due to distance can be approximated using a 3D texture in which the intensity of the cross-section is attenuated along the r-dimension. When geometry is rendered with the spotlight projection, the r coordinate of the fragment is proportional to the distance from the light source.

In order to determine the transformation needed for the texture coordinates, it is easiest to think about the case of the eye and the light source being at the same point. In this instance the texture coordinates should correspond to the eye coordinates of the geometry being drawn. The simplest method to compute the coordinates (other than explicitly computing them and sending them to the pipeline from the application) is to use an `EYE_LINEAR` texture generation function with an `EYE_PLANE` equation. The planes simply correspond to the vertex coordinate planes (e.g. the s coordinate is the distance of the vertex coordinate from the y-z plane, etc.). Since eye coordinates are in the range [-1.0, 1.0] and the texture coordinates need to be in the range [0.0, 1.0], a scale

and translate of .5 is applied to s and t using the texture matrix. A perspective spotlight projection transformation can be computed using `gluPerspective` and combined into the texture transformation matrix. The transformation for the general case when the eye and light source are not in the same position can be computed by incorporating into the texture matrix the inverse of the transformations used to move the light source away from the eye position.

With the texture map available the method for rendering the scene with the spotlight illumination is as follows:

1. initialize the depth buffer
2. clear the color buffer to a constant value which represents the scene ambient illumination
3. draw the scene with depth buffering enabled and color buffer writes disabled
4. load and enable the spotlight texture, set the texture environment to `MODULATE`
5. enable the texgen functions, load the texture matrix
6. enable blending and set the blend function to `ONE`, `ONE`
7. disable depth buffer updates and set the depth function to `EQUAL`
8. draw the scene with the vertex colors set to 1.0
9. disable the spotlight texture, texgen and texture transformation
10. set the blend function to `DST_COLOR`
11. draw the scene with normal illumination

There are three passes in the algorithm. At the end of the first pass the ambient illumination has been established in the color buffer and the depth buffer contains the resolved depth values for the scene. In the second pass the illumination from the spotlight is accumulated in the color buffer. By using the `EQUAL` depth function, only visible surfaces contribute to the accumulated illumination. In the final pass the scene is drawn with the colors modulated by the illumination accumulated in the first two passes to arrive at the final illumination values.

The algorithm does not restrict the use of texture on objects, since the spotlight texture is only used in the second pass and only the scene geometry is needed in this pass. The second pass can be repeated multiple time with different spotlight textures and projections to accumulate the contributions of multiple light sources.

There are a couple of considerations that also should be mentioned. Texture projection along the negative line-of-sight of the texture (back projection) can contribute undesired illumination. This can be eliminated by positioning a clip plane at the near plane of the line-of-site. OpenGL does not guarantee pixel exactness when various modes are enabled or disabled. This can manifest itself in undesirable ways during multipass algorithms. For example enabling texture coordinate generation may cause fragments with different depth values to be generated compared to the case when texture coordinate generation is not enabled. This problem can be overcome by re-establishing the depth buffer values between the second and third pass. This is done by redrawing the scene with color buffer updates disabled and the depth buffering configured the same as for the first pass.

It is also possible that the entire scene can be rendered in a single pass. If none of the objects in the scene are textured the complete image could be rendered in a single pass assuming the ambi-

ent illumination can be summed with spotlight illumination in a single pass. Some vendors have added an additive texture environment function as an extension which would make this operation feasible. A cruder method that works in OpenGL 1.0 involves illuminating the scene using normal OpenGL lighting with the spotlight texture modulating this illumination.

Reflections and Refractions using the Stencil Buffer

Drawing a scene containing a planar mirror surface with a properly rendered reflection in the mirror can be done by constructing the scene in two passes. The angle of incidence from the view direction to the mirror is used to compute the reflected ray. This angle is used to compute a new view to render the reflected version of the scene. This view of the scene is rendering into the polygons corresponding to the mirror. The scene from the viewer's perspective is rendered last and the stencil buffer is used to mask out the mirror so that the reflected image in the mirror is not overwritten. This process can be generalized to handle multiple mirrors by rendering the scene from the viewpoint of the reflected ray from each mirror. If second order reflections are to be included then more passes will be required.

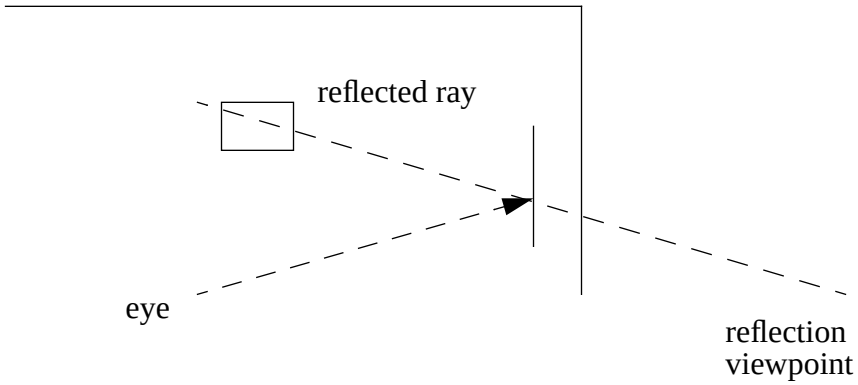


Figure 5: Mirror Reflection

The basic steps to calculate the image from the vantage point of the mirror are:

1. draw the mirror polygon, setting the stencil buffer to 1.
2. compute the angle between the viewer and the mirror.
3. compute the angle of the reflected ray and the viewer
4. move the viewing position to a point along the reflected ray
5. draw the scene from the viewpoint along the reflected ray
6. set the stencil test function to draw where the stencil value is not equal to 1
7. move the viewpoint back to the eye position and draw the scene again

By moving the viewpoint along the reflected ray the mirror projection will produce a magnified or

minified reflection. If the position is the same distance as the eye point from the mirror then an image of the same scale will result.

In addition to reflections a similar, though somewhat simpler, procedure can be used to approximate refraction in semi-transparent surfaces (refractors). In order to do this the scene must include some geometry on the other side of the refractive surface. Again the process is broken into multiple passes. In the first pass the refractive view of the scene is computed and drawn and in the second pass the scene from the original eye point is drawn, again using the stencil buffer to avoid overwriting the portion of the refractive view contained within the refractive surface.

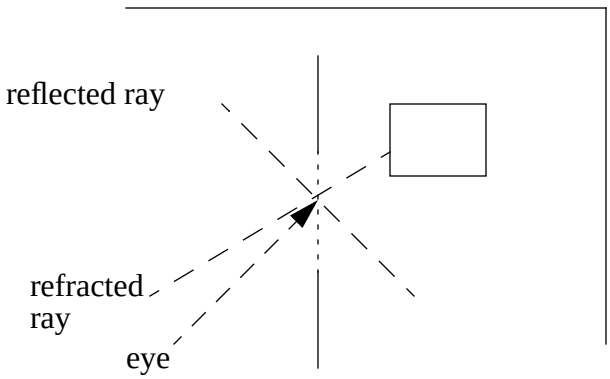


Figure 6: Refraction

The steps to compute the complete image are as follows:

1. draw the geometry of the refractor setting the stencil buffer to 1
2. compute the angle between the viewer and the refractor
3. compute the refracted ray
4. move the viewpoint to a position along the refracted ray
5. draw the scene from the viewpoint along the refracted ray
6. set the stencil test function to draw where the stencil value is not equal to 1
7. move the viewpoint back to the original eye position and draw the scene again

A combination of the reflection and refraction techniques can be used to deal with surfaces that are partially reflective and refractive. Alternatively, the surface may be both reflective and diffuse, e.g., a shiny marble surface. In the previous section on Phong highlights we discussed separating the specular highlight computation from the diffuse and ambient computation. A similar algorithm can be used to render a planar marble surface as well. One final step is added to the mirror algorithm in which the diffuse reflectance is computed for the marble polygon and added to the contribution already stored in the color buffer using blending.

These particular techniques for dealing with reflection and refraction are fairly simplistic, i.e., they apply to planar surfaces. It is possible to handle more complicated reflective and refractive

surfaces using environment mapping techniques.

Using the Stencil Buffer

A stencil buffer is a set of planes in the frame buffer containing one or more stencil bits per pixel. The contents of the stencil buffer can't be seen directly, but the values in the stencil buffer can be updated as a side effect of rendering images, and control, though the stencil test in the fragment operations, whether the corresponding pixel color values are updated [13].

Stencil buffer actions are part of OpenGL's fragment operations. Stencil testing occurs immediately after alpha tests. If `GL_STENCIL_TEST` is enabled, and the visual has a stencil buffer, then the application can control what happens under three different scenarios:

8. The stencil test fails
9. The stencil test passes, but the depth test fails
10. Both the stencil and the depth test pass.

Whether a stencil test for a given fragment passes or fails has nothing to do with the color or depth value of the fragment. Instead, it's a comparison operation between the value of the stencil buffer at the fragment's pixel location, and the current *stencil reference value*. Both the reference value and the comparison function are set using the `glStencilFunc()` call. In addition, the `glStencilFunc()` call also provides a mask value that used as part of the comparison. The comparison functions available are:

Table 1 Stencil Buffer Comparisons

Comparison	Description of comparison test between reference and stencil value
GL_NEVER	always fails
GL_ALWAYS	always passes
GL_LESS	passes if reference value is less than stencil buffer
GL_LEQUAL	passes if reference value is less than or equal to stencil buffer
GL_EQUAL	passes if reference value is equal to stencil buffer
GL_GEQUAL	passes if reference value is greater than or equal to stencil buffer
GL_GREATER	passes if reference value is greater than stencil buffer
GL_NOTEQUAL	passes if reference value is not equal to stencil buffer

These comparisons are done on a pixel by pixel basis. For each fragment, the stencil value for the target pixel is first masked against the current stencil mask, then compared against the current stencil reference value, using the current stencil comparison function.

If the stencil test fails, two things happen. The fragment for that pixel is discarded, and the color and depth values for that pixel remain unchanged. Then the *stencil operation* associated with the stencil test failing is applied to that stencil value. If the stencil test passes, then the depth

test is applied (if depth testing is disabled, and/or the visual doesn't have a depth buffer, it's as if the depth test always passes). If the depth test passes, the fragment continues on the pixel pipeline, and the stencil operation corresponding to both stencil and depth passing is applied to the stencil value for that pixel, if the depth test fails, the stencil operation set for stencil passing but depth failing is applied to the pixel's stencil value. Thus the stencil buffer both applies tests to control which fragments get to continue towards the frame buffer, and is updated by the results of those tests, in concert with the results of the depth buffer tests.

The stencil operations available are described in the table below:

Table 2

Stencil Operation	Results of Operation on Stencil Values
GL_KEEP	stencil value unchanged
GL_ZERO	stencil value set to zero
GL_REPLACE	stencil value set to current stencil reference value
GL_INCR	stencil value incremented by one
GL_DECR	stencil value decremented by one
GL_INVERT	stencil value bitwise inverted

The `glStencilOp()` call sets the stencil operation for all three stencil test results; stencil fail, stencil pass/depth buffer fail, and stencil pass/depth buffer pass. The `glStencilMask()` call can be used to globally enable or disable writing to the stencil buffer. This allows an application to apply stencil tests without the results affecting the stencil values. This can also be done by calling `glStencilOp()` with `GL_KEEP` for all three parameters.

There are two other important ways of controlling the stencil buffer. Every stencil value in the buffer can be set to a desired value using a combination of `glClearStencil()` and `glClear()` with the `GL_STENCIL_BUFFER_BIT` set. The contents of the stencil buffer can be read into system memory using `glReadPixels()` with the format parameter set to `GL_STENCIL_INDEX`. The stencil buffer can also be written to from system memory using `glDrawPixels()`.

Different machines support a different number of stencil bits per pixel. Use `glGetIntegerv()` with `GL_STENCIL_BITS` as the pname argument, to see how many bits the visual supports. The mask argument to `glStencilFunc()` can be used to divide up the stencil bits in each pixel into a number of different sections. This allows the application to store the stencil values of different tests separately within the same stencil value.

The following sections describe how to use the stencil buffer functionality in a number of useful multipass rendering techniques.

Dissolves with Stencil

Stencil buffers can be used to mask selected pixels on the screen. This allows for pixel by pixel compositing of images. Any image you can draw can be used as a pixel-by pixel mask to control what is shown on the color buffer. One way to use this capability is to use the stencil buffer to composite multiple images.

A common film technique is the “dissolve”, where one image or animated sequence is replaced with another, in a smooth sequence. The stencil buffer can be used to implement arbitrary dissolve patterns.

The basic approach to a stencil buffer dissolve is to render two different images, using the stencil buffer to control where each image can draw to the frame buffer. This can be done very simply by defining a stencil test and associating a different reference value with each image. The stencil buffer is initialized to a value such that the stencil test will pass with one of the image’s reference value, and fail with the other.

At the start of the dissolve, the stencil buffer is all one value, allowing only one of the images to be drawn to the frame buffer. Frame by frame, the stencil buffer is progressively changed (in an application defined pattern) to a different value, one that passes when compared against the second image’s reference value, but fails on the first’s. As a result, more and more of the first image is replaced by the second.

Over a series of frames, the first image “dissolves” into the second, under control of the evolving pattern in the stencil buffer.

Here is a step-by-step description of a dissolve.

1. Clear the stencil buffer with `glClear()`
2. Disable writing to the color buffer, using `glColorMask(GL_FALSE)`
3. If you don’t want the depth buffer changed, use `glDepthMask(GL_FALSE)`

Now you can write without changing the color or the depth buffer. You need to set the stencil functions and operations to set the stencil buffer to the value you want everywhere a pixel would have been written. To do this, call `glStencilFunc()` and `glStencilOp()`. For this example, we’ll have the stencil test always fail, and set the reference value to the stencil buffer: `glStencilFunc(GL_NEVER, 1, 1)`, `glStencilOp(GL_REPLACE, GL_KEEP, GL_KEEP)`. You also need to turn on stenciling: `glEnable(GL_STENCIL_TEST)`.

4. Turn on stenciling; `glEnable(GL_STENCIL_TEST)`
5. Set stencil function to always fail; `glStencilFunc(GL_NEVER, 1, 1)`
6. Set stencil op to write 1 on stencil test failure; `glStencilOp(GL_REPLACE, GL_KEEP, GL_KEEP)`
7. Write to the frame buffer with the pattern you want to stencil with.
8. Disable writing to the stencil buffer with `glStencilMask(GL_FALSE)`
9. Set stencil function to pass on 0; `glStencilFunc(GL_EQUAL, 0, 1)`.

10.Enable color buffer for writing with `glColorMask()`
11.If you're depth testing, turn depth buffer writes back on with `glDepthMask()`
12.Draw the first image. It will only be written where the stencil buffer values are 0.
13.Change the stencil test so only values that are 1 pass; glStencilFunc(GL_EQUAL, 1, 1).
14. Draw the second image. Only pixels with stencil value of 1 will change.
15. Repeat the process, updating the stencil buffer, so that more and more stencil values are 1, using your dissolve pattern, and redrawing image 1 and 2, until the entire stencil buffer has 1's in it, and only image 2 is visible.

If you choose your dissolve pattern so that at each step in the dissolve, the new dissolve pattern is a superset of the former, you don't have to re-render image1. This is because once a pixel of image 1 is replaced with image 2, image 1 will never be redrawn there. Designing your dissolve patterns with this restriction could be used to improve dissolve performance on low-end systems.

A demo program showing dissolves is on the website. It's named `dissolve.c`

Decaling with Stencil

In the dissolve example, we used the stencil buffer to control where pixels were drawn on an entire scene. We can be more targeted, and control the pixels that are drawn on a particular primitive. This technique can help solve a number of important problems:

1.Drawing depth-buffered, co-planar polygons without z-buffering artifacts.
2.Decaling multiple textures on a primitive.

The idea is similar to a dissolve: write values to the stencil buffer that mask the area you want to decal. Then use the stencil mask to control two separate draw steps; one for the decaled region, one for the rest of the polygon.

A useful example that illustrates the technique is rendering co-planar polygons. If one polygon is to be rendered on top of another, the depth buffer can't be relied upon to produce a clean separation between the two. This is due to the limited precision of the depth buffer. Since the polygons have different vertices, the rendering algorithms can produce z values that are rounded to the wrong depth buffer value, producing "stitching" or "bleeding" artifacts.

Decaled polygons can be done with the following steps:

1.Turn on stenciling; `glEnable(GL_STENCIL_TEST)`
2.Set stencil function to always pass; `glStencilFunc(GL_ALWAYS, 1, 1)`
3.Set stencil op to set 1 if depth passes, 0 if it fails; `glStencilOp(GL_KEEP, GL_ZERO, GL_REPLACE)`.
4. Draw the base polygon.
5.Set stencil function to pass when stencil is 1; `glStencilFunc(GL_EQUAL, 1, 1`
6. Disable writes to stencil buffer; `glStencilMask(GL_FALSE)`.
7. Turn off depth buffering; `glDisable(GL_DEPTH_TEST)`

8. Render the decal polygon.

The stencil buffer doesn't have to be cleared to an initial value; all the initializing is done as a side effect of writing the base polygon. Stencil values will be one where the base polygon was successfully written into the frame buffer. The stencil buffer becomes a mask, ensuring that the decal polygon can only affect the pixels that were touched by the base polygon. This is important if the decal polygon isn't completely contained within the base polygon, or there are other primitives partially obscuring them.

There are a few limitations to this technique; First, it assumes that the decal doesn't extend beyond the edge of the base polygon. If it does, you'll have to clear the entire stencil buffer before drawing the base polygon, which is expensive on some machines. If you are careful to redraw the base polygon with the stencil operations set to zero the stencil after you've drawn each decaled polygon, you will only have to clear the entire stencil buffer once, for any number of decaled polygons.

Second, you'll have to make sure that the screen extents of each polygon you're decaling don't overlap. If they do, the stencil value changes will affect each other. You'll have to do each polygon separately. Again, this will hurt performance.

This process can be extended to allow a number of overlapping decal polygons, the number of decals limited by the number of stencil bits available for the visual. The decals don't have to be sorted. The procedure is the similar to the previous algorithm, with these extensions

Assign a stencil bit for each decal and the base polygon. The lower the number, the higher the priority of the polygon. Render the base polygon as before, except instead of setting its stencil value to one, set it to the largest priority number. For example, if there were three decal layers, the base polygon would have a value of 8.

When you render a decal polygon, only draw it if the decal's priority number is lower than the pixels it's trying to change. For example, if the decal's priority number was 1, it would be able to draw over every other decal and the base polygon; `glStencilFunc(GL_LESS, 1, ~0)` and `glStencilOp(GL_KEEP, GL_REPLACE, GL_REPLACE)`.

Decals with the lower priority numbers will be drawn on top of decals with higher ones. Since the region not covered by the base polygon is zero, no decals can write to it. You can draw multiple decals at the same priority level. If you overlap them, however, the last one drawn will overlap the previous ones at the same priority level.

A simple demo program showing stencil decaling is on the website. It's named `decal.c`

Finding Depth Complexity with the Stencil Buffer

Finding depth complexity, or how many times a particular pixel was rendered in a depth buffered scene, is important for analyzing graphics performance. It indicates how many pixels are being

sent to the frame buffer at each pixel, in order to construct a scene.

One way to show depth complexity is to change the color values of the pixels in the scene to indicate the number of times a pixel was written. The image can then be displayed. It is relatively easy to draw an image representing depth complexity with the stencil buffer. The basic approach is simple; increment a stencil value everytime a pixel is written; when the scene is finished, read back the stencil buffer, and display it in the color buffer, color coding the different stencil values.

Here's the procedure in more detail:

1.Clear the depth and stencil buffer;
glClear(GL_STENCIL_BUFFER_BIT|GL_DEPTH_BUFFER_BIT).
2. Enable stenciling; `glEnable(GL_STENCIL_TEST)`.
3. Set up the proper stencil parameters; `glStencilFunc(GL_ALWAYS, 0, 0)`, `glStencilOp(GL_KEEP, GL_INCR, GL_INCR)`.
4. Draw the scene.
5.Read back the stencil buffer with `glReadPixels()`, using `GL_STENCIL_INDEX` as the *format* argument.
6. Draw the stencil buffer to the screen using glDrawPixels() with GL_COLOR_INDEX as the format argument.

You can control the mapping by calling glPixelTransferi(GL_MAP_COLOR, GL_TRUE) to turn on the color mapping, and set the appropriate pixel transfer maps with calls to glPixelMap(). You can map the stencil values to either RGBA or color index values, depending on the type of visual you're writing to.

There is a demo program showing depth complexity on the website. It's called `complexity.c`

Constructive Solid Geometry with the Stencil Buffer

Constructive Solid Geometry (CSG) is the process of building complex solids from logical combinations of simpler ones [1]. Basic logical operations, such as union, intersection, and inverse are used as relationship operators between pairs of three dimensional solids. In these algorithms, the solids are assumed to be defined by closed surfaces composed of polygons. The polygons composing the primitives are defined in a consistent way so that the front face of each polygon faces out from the surface. CSG is used as a method to create complex solids as part of a computer aided mechanical design process.

Constructive solid geometry can be created using the stencil buffer functionality. It can be thought of as a ray-casting technique, counting the intersections the pixels of an solid's surface makes with other surfaces as it's projected onto the screen. The techniques shown here are image space techniques and are limited to visualizing CSG solids. They don't produce a list of vertices that could be used for further processing. They also tend to leave the depth buffer with values that don't match the visible surfaces. As a result, CSG solids often can't be used as components of a complex depth-buffered scene.

Creating CSG solids is a multipass technique. Two objects are chosen, and the desired boolean relationship between the two is discovered. The basic CSG stencil operation is the ability to find the part of one solid's surface that exists in the volume defined by another solid. Pieces of each solids surface is progressively rendered, until the entire solid is completed. The solids created with CSG are rendered as pieces of the surfaces of component solids; no new surfaces are created.

The component solids are rendered multiple times; the solid's front and back surfaces rendered in separate passes. The process requires rendering a surface into the depth buffer, then using the other solid in combination with stencil plane operations, to determine the part of the surface inside the other solid. Then either the inside or outside piece of the surface is rendered to the color buffer, controlled by the values in the stencil buffer.

Here is the sequence of operations needed to determine which parts of a solids surface are inside another. Note that this is just a variant of the shadow volume technique described in these notes. Two solids, labeled A and B are used. In this example, we'll show how to find the parts of the front surface of A that are inside, and outside of B's volume.

1. Turn off the color buffer with `glColorMask()`.
2. Clear the stencil buffer
3. Turn on depth testing
4. Enable face culling using `glEnable(GL_CULL_FACE)`
5. Set back face culling using `glCullFace(GL_BACK)`
6. Render A

Now the depth buffer has been updated with the front surface of A.

7. Set the stencil buffer to increment if the depth test passes with `glStencilFunc(GL_ALWAYS, 0, 0)` and `glStencilOp(GL_KEEP, GL_KEEP, GL_INCR)`
8. Render B
9. Set the stencil buffer to decrement if the depth test passes; `glStencilOp(GL_KEEP, GL_KEEP, GL_DECR)`
10. Now cull the front face of B; `glCullFace(GL_FRONT)`
11. Render B again

Now the stencil buffer contains a non-zero value whereever the front surface of A was enclosed by B's volume. Now you can use the stencil buffer test to either render the part of A's front surface that's inside B, or the part that's outside.

Here's how to do the inside part:

12. Set the stencil buffer to pass if the stencil value isn't zero; `glStencilTest(GL_NOTEQUAL, 0, ~0)`.
13. Disable the stencil buffer updates; `glStencilMask()`.
14. Turn on back face culling; `glCullFace(GL_BACK)`
15. Turn off depth testing, since it's not needed.
16. Render A

To render just the part of A's front face that is outside of B's volume, change the stencil test from GL_NOTEQUAL to GL_EQUAL.

Note that although only a part of A's front surface was used to update the color buffer, all of A's front surface is in the depth buffer. This will cause problems if you are trying to combine CSG solids in with other depth buffered objects in a scene.

Here are three basic CSG operations using the mechanism described above. We show union, intersection, and subtraction. From these basic operations it's possible to build an expression tree to create any arbitrary CSG solid [18]. The expression tree must be ordered in such a way as to follow a number of restrictions, but it can be shown that an arbitrary expression tree of unions, intersections, and subtractions can always be converted into the more restrictive form. The details for creating CSG expression trees is beyond the scope of these notes, however; please refer to the references [1] for more details.

Union

The union of two solids, A and B, means rendering the parts of A not enclosed by B, and the parts of B not enclosed by A. To render the union of two solids, simply requires rendering them in either order using depth testing. The stencil buffer isn't required.

Intersection

Rendering the intersection of two solids, A and B, means rendering the volume that is enclosed by both A and B. The algorithm to find this intersection can be thought of as finding the parts of B that are in A's volume, and drawing them, then finding the parts of A that are in B's volume, and drawing them. It consists of two passes of the basic interior algorithm described above.

The final result is a combination of the existing surfaces of the original primitives. No new surfaces are created. Since the visible results of the will consist solely of the front surfaces of the primitives, only the front faces of a volume that are interior to the other need to be rendered.

Here are the steps needed to render the intersection of A and B.

1. Turn off writing the color buffer
2. Clear the stencil buffer to zero
3. Turn on depth testing
4. Draw the front faces of A
5. Disable writing to the depth buffer with glDepthMask(GL_FALSE)
6. Set the stencil buffer to increment if the depth test passes
7. Render the front faces of B
8. Set the stencil buffer to decrement if the depth test passes
9. Render the back faces of B

10. Set the stencil buffer to pass if the stencil values aren't zero
11. Turn off writing to the stencil buffer
12. Turn on writing to the color buffer
13. Turn off depth testing
14. Render the front faces of A again

Now the front faces of A that are inside of B has been rendered. To finish the intersection, the front faces of B that are inside A have to be rendered. There is a problem, however. Only the inside part of A has been rendered to the color buffer, but all of A is in the depth buffer. The outside part of A could mask off parts of B that should be visible. There are a number ways to fix this. The way we'll do it is to re-render the pixels of the depth buffer covered by B's front face, setting the depth test to always pass.

15. Turn on writing to the depth buffer
16. Turn on depth testing
17. Change the depth test to GL_ALWAYS
18. Render the front face of B

Now the depth buffer values under B are correct. All that is needed is to find the parts of B's front surface that are inside A, and render them.

19. Clear the stencil buffer to zero again
20. Disable writing to the depth buffer
21. Set the stencil buffer to increment if the depth test passes
22. Render the front faces of A
23. Set the stencil buffer to decrement if the depth test passes
24. Render the back faces of A
25. Set the stencil buffer to pass if the stencil values aren't zero
26. Turn off writing to the stencil buffer
27. Turn on writing to the color buffer
28. Turn off depth testing
29. Render the front face of B again

Depth testing is disabled when writing to the color buffer avoids any problems with coplanar polygons that are shared by both A and B.

Difference

The difference between solids A and B (A - B) can be thought of as A intersected with the inverse of B. Put another way, only draw the parts of A that aren't also a part of B. In terms of our stencil algorithm, this means drawing the front parts of A that are outside of B. There is more to it than that, however. In order to create a solid that looks closed, we also have to draw parts of B, specifically, the back part of B in A.

Here is the algorithm:

1. Turn off writing the color buffer
2. Clear the stencil buffer to zero
3. Turn on depth testing
4. Draw the back faces of A
5. Disable writing to the depth buffer with glDepthMask(GL_FALSE)
6. Set the stencil buffer to increment if the depth test passes
7. Render the front faces of B
8. Set the stencil buffer to decrement if the depth test passes
9. Render the back faces of B
10. Set the stencil buffer to pass if the stencil values aren't zero
11. Turn off writing to the stencil buffer
12. Turn on writing to the color buffer
13. Turn off depth testing
14. Render the back faces of A again

At this stage, you've drawn the parts of the back facing polygons of A that are inside of B. You've "lined the holes" of any pieces of B that A will cut out. Now the front face of B has to be drawn, using the same technique to fix the depth buffer:

15. Turn on writing to the depth buffer
16. Turn on depth testing
17. Change the depth test to GL_ALWAYS
18. Render the front face of B

Now the depth buffer values under B are correct. All that is needed is to find the parts of B's front surface that are *outside* of A, and render them.

19. Clear the stencil buffer to zero again
20. Disable writing to the depth buffer
21. Set the stencil buffer to increment if the depth test passes
22. Render the front faces of A
23. Set the stencil buffer to decrement if the depth test passes
24. Render the back faces of A
25. Set the stencil buffer to pass if the stencil values are zero
26. Turn off writing to the stencil buffer
27. Turn on writing to the color buffer
28. Turn off depth testing
29. Render the front face of B again

A demo program showing CSG with stencil buffer is on the website. It's called `csg.c`

Compositing Images with Depth

Compositing separately images together is a useful technique for increasing the complexity of a scene [8]. An image can be saved to memory, then drawn to the screen using `glDrawPix-`

els().
Both the color and depth buffer contents can be copied into the frame buffer. This is sufficient to do 2D style composites, where objects are drawn on top of each other to create the final scene. To do true 3D compositing, it's necessary to use the color and depth values simultaneously, so that depth testing can be used to determine which surfaces are obscured by others.

The stencil buffer can be used to do true 3D compositing in a two pass operation. The color buffer is disabled for writing, the stencil buffer is cleared, and the saved depth values are copied into the frame buffer. Depth testing is enabled, insuring that only depth values that are closer to the original can update the depth buffer. Stenciling is set to set a stencil buffer bit if the depth test passes.

The stencil buffer now contains a mask of pixels that were closer to the view than the pixels of the original image. The stencil function is changed to accomplish this masking operation, the color buffer is enabled for writing, and the color values of the saved image are drawn to the frame buffer.

This technique works because the fragment operations, in particular the depth test and the stencil test, are part of both the geometry and imaging pipelines in OpenGL. Here is the technique in more detail. It assumes that both the depth and color values of an image have been saved to system memory, and are to be composited using depth testing to an image in the frame buffer:

1. Clear the stencil buffer using `glClear()`, or'ing in `GL_STENCIL_BUFFER_BIT`
2. Disable the color buffer for writing with `glColorMask()`
3. Set stencil values to 1 when the depth test passes by calling `glStencilFunc(GL_ALWAYS, 1, 1)`, and `glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE)`
4. Ensure depth testing is set; `glEnable(GL_DEPTH_TEST)`, `glDepthFunc(GL_LESS)`
5. Draw the depth values to the frame buffer with `glDrawPixels()`, using `GL_DEPTH_COMPONENT` for the format argument.
6. Set the stencil buffer to test for stencil values of 1 with `glStencilFunc(GL_EQUAL, 1, 1)` and `glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP)`.
7. Disable the depth testing with `glDisable(GL_DEPTH_TEST)`
8. Draw the color values to the frame buffer with `glDrawPixels()`, using `GL_RGBA` as the format argument.

At this point, both the depth and color values have been merged, using the depth test to control which pixels from the saved image can update the frame buffer. Compositing can still be problematic when merging images with coplanar polygons.

This process can be repeated to merge multiple images. The depth values of the saved image can be manipulated by changing the values of `GL_DEPTH_SCALE` and `GL_DEPTH_BIAS` with `glPixelTransfer()`. This technique could allow you to squeeze the incoming image into a limited range of depth values within the scene.

There is a demo program showing image compositing with depth on the website. It's called

`zcomposite.c`

Using the Accumulation Buffer

The accumulation buffer is designed for integrating multiple images. Instead of simply replacing pixel values with incoming pixel fragments, the fragments are scaled, then added to the existing pixel value. In order to maintain accuracy over a number of blending operations, the accumulation buffer has a higher number of bits per color component than a typical visual.

The accumulation buffer can be cleared like any other buffer. You can use `glClearAccum()` to set the red, green and blue components of the color it's cleared to. Clear the accumulation buffer by bitwise or'ing in the `GL_ACCUM_BUFFER_BIT` value to the parameter of the `glClear()` command.

You can't render directly into the accumulation buffer. Instead you render into a selected buffer, then use `glAccum()` to accumulate the current image in that buffer into the accumulation buffer. The glAccum() command uses the currently selected read buffer to copy from. You can set the buffer you want it to read from using the `glReadBuffer()` command.

The glAccum() command takes two arguments, *op* and *value*. The op value can be one of the following values:

Table 3 `glAccum()` op values

Op Value	Action
GL_ACCUM	read from selected buffer, scale by *value*, then add into accumulation buffer
GL_LOAD	read from selected buffer, scale by *value*, then use image to replace contents of accumulation buffer
GL_RETURN	scale image by *value*, then copy into buffers selected for writing
GL_ADD	add *value* to R, G, B, and A components of every pixel in accumulation buffer
GL_MULT	clamp *value* to range -1 to 1, then scale R, G, B, and A components of every pixel in accumulation buffer.

Since you must render to another buffer before accumulating, a typical approach to accumulating images is to render images to the back buffer some number of times, accumulating each image into the accumulation buffer. When the desired number of images have been accumulated, the contents are copied back into the back buffer, and the buffers are swapped. This way, only the final, accumulated image is displayed.

Here is an example procedure for accumulating *n* images:

1. Call `glDrawBuffer(GL_BACK)` to render to the back buffer only

2.Call `glReadBuffer(GL_BACK)` so that the accumulation buffer will read from the back buffer.

Note that the first two steps are only necessary if the application has changed the selected draw and read buffers. If the visual is double buffered, these selections are the default.

3.Clear the back buffer with `glClear(`*`bitfield`*`)`, then render the first image
4.Call `glAccum(GL_LOAD, 1.f/n)`; this allows you to avoid a separate step to clear the accumulation buffer.
5.Alter the parameters of your image, and re-render it
6.Call `glAccum(GL_ACCUM,1.f/n)` to add the second image into the first.
7.Repeat the previous two steps n - 2 more times...
8.Call `glAccum(GL_RETURN, 1.f)` to copy the completed image into the back buffer
9.Call `glutSwapBuffers()` if your using GLUT, or whatever's appropriate to swap the front and back buffers.

The accumulation buffer provides a way to do "multiple exposures" in a scene, while maintaining good color resolution. There are a number of image effects that can be done using the accumulation buffer to improve the realism of a rendered image [23,6]. Among them are; antialiasing, motion blur, soft shadows, and depth of field. To create these effects, the image is rendered multiple times, making small, incremental changes to the scene position (or selected objects within the scene), and accumulating the results.

Motion Blur

This is probably one of the easiest effects to implement. Simply re-render a scene multiple times, incrementing the position and/or orientation of an object in the scene. The object will appear blurred, suggesting motion. This effect can be incorporated in the frames of an animation sequence to improve its realism, especially when simulating high-speed motion.

The apparent speed of the object can be increased by dimming its blurred path. This can be done by accumulating the scene without the moving object, using setting the value parameter to be larger than 1/n. Then re-render the scene with the moving object, setting the value parameter to something smaller than 1/n. For example, to make a blurred object appear 1/2 as bright, accumulated over 10 scenes, do the following:

1.Render the scene without the moving object, using `glAccum(GL_LOAD,.5f)`
2.Accumulate the scene 10 more times, with the moving object, using `glAccum(GL_ACCUM,.05f)`

Choose the values to ensure that the non-moving parts of the scene retain the same overall brightness.

It's also possible to use different values for each accumulation step. This technique could be used to make an object appear to be accelerating or decelerating. As before, ensure that the overall scene brightness remains constant.

If you are using motion blur as part of a real-time animated sequence, and your value is constant, you can improve the latency of each frame after the first n dramatically. Instead of accumulating n scenes, then discarding the image and starting again, you can subtract out the first scene of the sequence, add in the new one, and display the result. In effect, you're keeping a "running total" of the accumulated images.

The first image of the sequence can be "subtracted out" by rendering that image, then accumulating it with `glAccum(GL_ACCUM, -1.f/n)`. As a result, each frame only incurs the latency of drawing two scenes; adding in the newest one, and subtracting out the oldest.

A demo program showing motion blur is on the website. It is called `motionblur.c`

Soft Shadows

Most shadow techniques create a very "hard" shadow edge; surfaces in shadow, and surfaces being lit are separated by a sharp, distinct boundary, with a large change in surface brightness. This is an accurate representation for distant point light sources, but is unrealistic for many real-world lighting environments.

An accumulation buffer can let you render softer shadows, with a more gradual transition from lit to unlit areas. These soft shadows are a more realistic representation of area light sources, which create shadows consisting of an umbra (where none of the light is visible) and penumbra, where part of the light is visible.

Soft shadows are created by rendering the shadowed scene multiple times, and accumulating into the accumulation buffer. Each scene differs in that the position of the light source has been moved slightly. The light source is moved around within the volume where the physical light being modeled would be emitting energy. To avoid aliasing artifacts, it's best to move the light in an irregular pattern.

Shadows from multiple, separate light sources can also be accumulated. This allows the creation of scenes containing shadows containing non-trivial patterns of light and dark, resulting from the light contributions of all the lights in the scene.

A demo program showing soft shadows is on the website, it is called `softshadow.c`

Antialiasing

Accumulation buffers can be used to antialias a scene without having to depth sort the primitives before rendering. A supersampling technique is used, where the entire scene is offset by small, subpixel amounts in screen space, and accumulated. The jittering can be accomplished by modifying the transforms used to represent the scene.

One straightforward jittering method is to modify the projection matrix, adding small translations

in x and y. Care must be taking to compute the translations so that they shift the scene the appropriate amount in window coordinate space. Fortunately, computing these offsets is straightforward. To compute a jitter offset in terms of pixels, divide the jitter amount by the dimension of the object coordinate scene, then multiply by the appropriate viewport dimension. The example code fragment below shows how to calculate a jitter value for an orthographic projection; the results are applied to a translate call to modify the modelview matrix:

```
void ortho_jitter(GLfloat xoff, GLfloat yoff)
{
     GLint viewport[4];
     GLfloat ortho[16];
     GLfloat scalex, scaley;

     glGetIntegerv(GL_VIEWPORT, viewport);
     /* this assumes that only a glOrtho() call has been
     applied to the projection matrix */
     glGetFloatv(GL_PROJECTION_MATRIX, ortho);

     scalex = (2.f/ortho[0])/viewport[2];
     scaley = (2.f/ortho[5])/viewport[3];
     glTranslatef(xoff * scalex, yoff * scaley, 0.f);
}
```

If the projection matrix is not simply the result of an ortho call, use the arguments that were made to glOrtho() instead, and compute the value of right-left, and top-bottom. Use these values in place of ortho[0] and ortho[5], respectively.

The code is very similar for jittering a perspective projection. In this example, we jitter the frustum itself:

```
void frustum_jitter(GLdouble left, GLdouble right,
                    GLdouble bottom, GLdouble top,
                    GLdouble near, GLdouble far,
                    GLdouble xoff, GLdouble yoff)
{
     GLfloat scalex, scaley;
     GLint viewport[4];

     glGetIntegerv(GL_VIEWPORT, viewport);
     scalex = (right - left)/viewport[2];
     scaley = (top - bottom)/viewport[3];

     glFrustum(left - xoff * scalex,
               right - xoff * scalex,
               top - yoff * scaley,
               bottom - yoff * scaley,
```

```
                near, far);
}
```

The jittering values you choose should fall in an irregular pattern; this reduces aliasing artifacts by making them “noisy”. Selected subpixel jitter values, organized by the number of samples needed, are taken from the OpenGL Programming Guide, and are shown below:

Count	Values
2	{0.25, 0.75}, {0.75, 0.25}
3	{0.5033922635, 0.8317967229}, {0.7806016275, 0.2504380877}, {0.2261828938, 0.4131553612}
4	{0.375, 0.25}, {0.125, 0.75}, {0.875, 0.25}, {0.625, 0.75}
5	{0.5, 0.5}, {0.3, 0.1}, {0.7, 0.9}, {0.9, 0.3}, {0.1, 0.7}
6	{0.4646464646, 0.4646464646}, {0.1313131313, 0.7979797979}, {0.5353535353, 0.8686868686}, {0.8686868686, 0.5353535353}, {0.7979797979, 0.1313131313}, {0.2020202020, 0.2020202020}
8	{0.5625, 0.4375}, {0.0625, 0.9375}, {0.3125, 0.6875}, {0.6875, 0.8125}, {0.8125, 0.1875}, {0.9375, 0.5625}, {0.4375, 0.0625}, {0.1875, 0.3125}
9	{0.5, 0.5}, {0.1666666666, 0.9444444444}, {0.5, 0.1666666666}, {0.5, 0.8333333333}, {0.1666666666, 0.2777777777}, {0.8333333333, 0.3888888888}, {0.1666666666, 0.6111111111}, {0.8333333333, 0.7222222222}, {0.8333333333, 0.0555555555}
12	{0.4166666666, 0.625}, {0.9166666666, 0.875}, {0.25, 0.375}, {0.4166666666, 0.125}, {0.75, 0.125}, {0.0833333333, 0.125}, {0.75, 0.625}, {0.25, 0.875}, {0.5833333333, 0.375}, {0.9166666666, 0.375}, {0.0833333333, 0.625}, {0.583333333, 0.875}
16	{0.375, 0.4375}, {0.625, 0.0625}, {0.875, 0.1875}, {0.125, 0.0625}, {0.375, 0.6875}, {0.875, 0.4375}, {0.625, 0.5625}, {0.375, 0.9375}, {0.625, 0.3125}, {0.125, 0.5625}, {0.125, 0.8125}, {0.375, 0.1875}, {0.875, 0.9375}, {0.875, 0.6875}, {0.125, 0.3125}, {0.625, 0.8125}

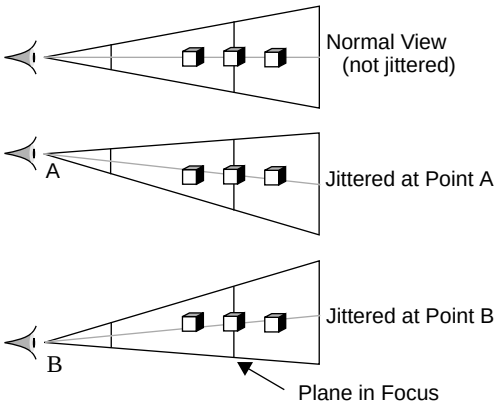
Table 4 Sample Jittering Values

A demo program showing antialiasing is on the website. It’s called `accumaa.c`

Depth of Field

OpenGL’s perspective projections simulate a pinhole camera; everything in the scene is in perfect focus. Real lenses have a finite area, which causes only objects within a limited range of distances to be in focus. Objects closer or farther from the camera are progressively more blurred.

The accumulation buffer can be used to create depth of field effects by jittering the eye point and the direction of view. These two parameters change in concert, so that one plane in the frustum doesn't change. This distance from the eyepoint is thus in focus, while distances nearer and farther become more and more blurred.



To create depth of field blurring, the perspective transform change described in the antialiasing section are expanded somewhat. This code modifies the frustum as before, but adds in an additional offset. This offset is also used to change the modelview matrix; the two acting together change the eyepoint and the direction of view:

```
void frustum_depthoffield(GLdouble left, GLdouble right,
                  GLdouble bottom, GLdouble top,
                  GLdouble near, GLdouble far,
                  GLdouble xoff, GLdouble yoff,
                  GLdouble focus)
{
     GLfloat scalex, scaley;
     GLint viewport[4];

     glGetIntegerv(GL_VIEWPORT, viewport);
     scalex = (right - left)/viewport[2];
     scaley = (top - bottom)/viewport[3];

     glFrustum(left - xoff * near/focus,
               right - xoff * near/focus,
               top - yoff * near/focus,
               bottom - yoff * near/focus,
               near, far);

     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();
     glTranslatef(-xoff, -yoff);
}
```

The variables xoff and yoff now jitter the eyepoint, not the entire scene. The focus variable describes the distance from the eye where objects will be in perfect focus. Think of the eyepoint jittering as sampling the surface of a lens. The larger the lens, the greater the range of jitter values, and the more pronounced the blurring. The more samples taken, the more accurate a sampling of the lens. You can use the jitter values given in the scene antialiasing section.

This function assumes that the current matrix is the projection matrix. It sets the frustum, then changes to the modelview matrix, resets it, and loads it with a translate. The usual modelview transformations could then be applied to the modified modelview matrix stack. The translate would become the last logical transform to be applied.

A simple demo program showing depth of field is on the website. It's called `field.c`

Convolving with the Accumulation Buffer

Convolution is a very powerful technique for processing images. OpenGL has been extended to support convolution in the pixel path by some OpenGL licensees, and has direct hardware support on some new system. But there is currently no convolution support in core library. Convolution can be done with the accumulation buffer, however, if you observe some precautions.

In order to discuss convolving with the accumulation buffer, it's helpful to review convolution itself. Convolving requires a filter kernel, a 1d or 2d array signed real numbers, and an image to convolve.

To convolve an image, the filter array is aligned with an equal sized subset of the image. Every element in the convolution kernel array corresponds to a pixel in the subimage. At each convolve, step, the color values of each pixel corresponding to a kernel array element are read, then scaled by their corresponding kernel element. The resulting values are all summed together into a single value.

Thus, every element of the kernel, and every pixel under them, contributes values that are combined into a single convolved pixel color. One of the kernel array elements corresponds to the location where this output value is written back to update the output image.

Generally, convolving is done with separate input and output images, so that the input image is read-only, and the outputs of the individual convolution steps don't affect each other.

After each convolution step, the convolution kernel filter position is shifted by one, covering a slightly different set of pixels in the input image, and a new convolution step is performed. The cycle continues, convolving consecutive pixels in a scanning pattern, until the entire image has been convolved.

The convolution filter could have a single element per pixel, where the RGBA components are scaled by the same value, or have separate red, green, blue, and alpha values for each kernel element.

The accumulation buffer can perform all the operations needed to perform convolutions. The input image is the image the `glAccum()` function reads from, the output image is built up in the accumulation buffer, and the value argument used in multiple calls of glAccum(GL_ACCUM, value) correspond to the values in the convolution kernel. In accumulation buffer convolution, the convolution steps don't take place one pixel at a time, using all the contributions for the convolution filter kernel. Instead all the pixels are convolved at once, adding a single kernel element contribution at a time.

Convolution Example

Here is an example of using the accumulation buffer to convolve using a Sobel filter, commonly used to do edge detection. This filter is used to find horizontal edges, and looks like this:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The operations needed to simulate this filter are the following. Assume that the translate step shifts the image an integer number of pixel in the direction indicated.

1. `glAccum(GL_LOAD, 1.)`
2. Translate -1 in x, 0 in y
3. `glAccum(GL_ACCUM, 2.)`
4. Translate -1 in x, 0 in y
5. `glAccum(GL_ACCUM, 1.)`
6. Translate +2 in x, -2 in y
7. `glAccum(GL_ACCUM, -1.)`
8. Translate -1 in x, 0 in y
9. `glAccum(GL_ACCUM, -2.)`
10. Translate -1 in x, 0 in y
11. `glAccum(GL_ACCUM, -1.)`

In this example, the output point is the lower left-hand corner of the kernel (where the value is 1). At each step, the image is shifted so that the pixel that would have been under the kernel element with the value used is under the lower left corner.

The order of the accumulations can be re-arranged to minimize color precision errors. The order can be changed to ensure that the color values don't overflow, or become negative.

A demo program showing convolution with the accumulation buffer is on the website. It's called `convolve.c`

Shadows: 3 Different Flavors

Shadows are an important way to add realism to a scene. There are a number of trade-offs possible when rendering a scene with shadows. Just as with lighting, there are increasing levels of realism possible, paid for with decreasing levels of rendering performance.

Shadows are composed of two parts, the umbra and the penumbra. The umbra is the area of a shadowed object that isn't visible from any part of the light source. The penumbra is the areas of a shadowed object that can receive some, but not all of the light. A point source light would have no penumbra, since no part of a shadowed object can receive *part* of the light.

Penumbras form a transition region between the umbra and the lighted parts of the object; they vary as function of the geometry of the light source and the shadowing object. Since shadows tend to have high contrast edges, They are more unforgiving to aliasing artifacts and other rendering errors.

Although OpenGL doesn't support shadows directly, there are a number of ways to implement them with the library. They vary in difficulty to implement, and quality of results. The quality varies as a function of two parameters. The complexity of the shadowing object, and the complexity of the scene that is being shadowed.

Projection Shadows

An easy-to-implement type of shadow can be created using projection transforms [14]. An object is simply projected onto a plane, then rendered as a separate primitive. Computing the shadow involves applying a orthographic or perspective projection matrix to the modelview transform, then rendering the projected object in the desired shadow color.

Here is the sequence needed to render an object that has a shadow cast from a directional light on the y axis down onto the x, y plane:

1. Render the scene, including the shadowing object in the usual way
2. Set the modelview matrix to identity, then call `glScalef(1.f, 0.f, 1.f)`
3. Make the rest of the transformation calls necessary to position and orient the shadowing object.
4. Set the OpenGL state necessary to create the correct shadow color
5. Render the shadowing object

In the last step, above, the second time the object is rendered, the tranform flattens it into the object's shadow. This simple example can be expanded by applying additional transforms before the `glScalef()` call to position the shadow onto the appropriate flat object. Applying this shadow is similar to decaling a polygon with another co-planar one. Depth buffering aliasing must be taken into account. To avoid depth aliasing problems, the shadow can be slightly offset from the base polygon using polygon offset, the depth test can be disabled, or the stencil buffer can be used to ensure correct shadow decaling. The best approach is probably depth buffering with poly-

gon offset. This way the depth buffering will minimize the amount of clipping you'll have to do to the shadow.

The direction of light source can be altered by applying a shear transform after the `glScalef()` call. This technique is not limited to directional light sources. A point source can be represented by adding a perspective transform to the sequence.

Although you can construct an arbitrary shadow from a sequence of transforms, it might be easier to just construct a projection matrix directly. The function below takes an arbitrary plane, defined as a plane equation in Ax + By + Cz + D = 0 form, and a light position in homogeneous coordinates. If the light is directional, the w value should be 0. The function concatenates the shadow matrix onto the top element of the current matrix stack.

```
static void
myShadowMatrix(float ground[4], float light[4])
{
    float  dot;
    float  shadowMat[4][4];

    dot = ground[0] * light[0] +
          ground[1] * light[1] +
          ground[2] * light[2] +
          ground[3] * light[3];

    shadowMat[0][0] = dot - light[0] * ground[0];
    shadowMat[1][0] = 0.0 - light[0] * ground[1];
    shadowMat[2][0] = 0.0 - light[0] * ground[2];
    shadowMat[3][0] = 0.0 - light[0] * ground[3];

    shadowMat[0][1] = 0.0 - light[1] * ground[0];
    shadowMat[1][1] = dot - light[1] * ground[1];
    shadowMat[2][1] = 0.0 - light[1] * ground[2];
    shadowMat[3][1] = 0.0 - light[1] * ground[3];

    shadowMat[0][2] = 0.0 - light[2] * ground[0];
    shadowMat[1][2] = 0.0 - light[2] * ground[1];
    shadowMat[2][2] = dot - light[2] * ground[2];
    shadowMat[3][2] = 0.0 - light[2] * ground[3];

    shadowMat[0][3] = 0.0 - light[3] * ground[0];
    shadowMat[1][3] = 0.0 - light[3] * ground[1];
    shadowMat[2][3] = 0.0 - light[3] * ground[2];
    shadowMat[3][3] = dot - light[3] * ground[3];

    glMultMatrixf((const GLfloat*)shadowMat);
}
```

A demo program showing projective shadows is on the website. It's called `projshadow.c`

Projection Shadow Trade-offs

This method of shadow volume is limited in a number of ways. First, it's very difficult to use this method to shadow onto anything other than flat surfaces. Although you could project onto a polygonal surface, by carefully casting the shadow onto the plane of each polygon face, you would then have to clip the result to the polygon's boundaries. Sometimes depth buffering can do the clipping for you; casting a shadow to the corner of a room composed of just a few perpendicular polygons is feasible with this method.

The other problem with projection shadows is controlling the shadow's color. Since the shadow is a squashed version of the shadowing object, not the polygon being shadowed, there are limits to how well you can control the shadow's color. Since the normals have been squashed by the projection operation, trying to properly light the shadow is impossible. A shadowed polygon with an interpolated color won't shadow correctly either, since the shadow is a copy of the shadowing object.

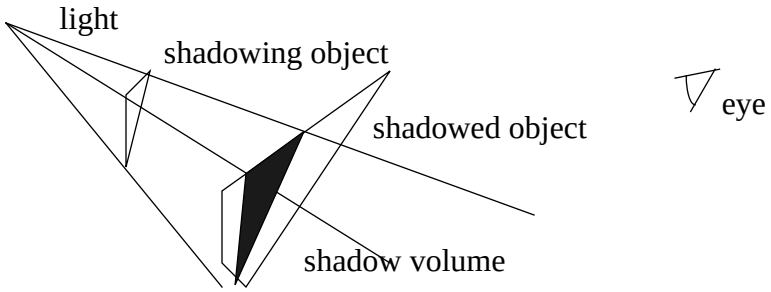
Shadow Volumes

This technique treats the shadows cast by objects as polygonal volumes. The stencil buffer is used to find the intersection between the polygons in the scene and the shadow volume [19].

The shadow volume is constructed from rays cast from the light source, intersecting the vertices of the shadowing object, then continuing outside the scene. Defined in this way, the shadow volumes are semi-infinite pyramids, but the same results can be obtained by truncating the base of the shadow volume beyond any object that might be shadowed by it. This gives you a polygonal surface, whose interior volume contains shadowed objects or parts of shadowed objects. The polygons of the shadow volume are defined so that their front faces point out from the shadow volume itself.

The stencil buffer is used to compute which parts of the objects in the scene are in the shadow volume. It uses a non-zero winding rule technique. For every pixel in the scene, the stencil value is incremented as it crosses a shadow boundary going into the shadow volume, and decrements as it crosses a boundary going out. The stencil operations are set so this increment and decrement only happens when the depth test passes. As a result, pixels in the scene with non-zero stencil values

identify the parts of an object in shadow.



Since the shadow volume shape is determined by the vertices of the shadowing object, it's possible to construct a complex shadow volume shape. Since the stencil operations will not wrap past zero, it's important to structure the algorithm so that the stencil values are never decremented past zero, or information will be lost. This problem can be avoided by rendering all the polygons that will increment the stencil count first; i.e. the front facing ones, then rendering the back facing ones.

Another issue with counting is the position of the eye with respect to the shadow volume. If the eye is inside a shadow volume, the count of objects outside the shadow volume will be -1, not zero. This problem is discussed in more detail in the shadow volume trade-offs section. The algorithm takes this case into account by initializing the stencil buffer to 1 if the eye is inside the shadow volume.

Here's the algorithm for a single shadow and light source:

1. The color buffer and depth buffer are enabled for writing, and depth testing is on
2. Set attributes for drawing in shadow. Turn off the light source
3. Render the entire scene
4. Compute the polygons enclosing the shadow volume
5. Disable the color and depth buffer for writing, but leave the depth test enabled
6. Clear the stencil buffer to 0 if the eye is outside the shadow volume, or 1 if inside
7. Set the stencil function to always pass
8. Set the stencil operations to increment if the depth test passes
9. Turn on back face culling
10. Render the shadow volume polygons
11. Set the stencil operations to decrement if the depth test passes
12. Turn on front face culling
13. Render the shadow volume polygons
14. Set the stencil function to test for equality to 0.
15. Set the stencil operations to do nothing
16. Turn on the light source
17. Render the entire scene

When the entire scene is rendered the second time, only pixels that have a stencil value equal to zero are updated. Since the stencil values were only changed when the depth test passes, this value represents how many times the pixel's projection passed into the shadow volume minus the

number of times it passed out of the shadow volume before striking the closest object in the scene (after that the depth test will fail). If the shadow boundary crossings balanced, the pixel projection hit an object that was outside the shadow volume. The pixels outside the shadow volume can therefore “see” the light, which is why it is turned on for the second rendering pass.

For a complicated shadowing object, it make sense to find its silhouette vertices, and only use them for calculating the shadow volume. These vertices can be found by looking for any polygon edges that either (1) Surrounds a shadowing object composed of a single polygon, or (2) is shared by two polygons, one which is facing towards the light source, one which is facing away. You can determine how the polygons are facing by taking a dot product of the polygon’s facet normal with the direction of the light source, or by a combination of selection and front/back face culling

A demo program showing shadow volumes is on the website. It’s called `shadowvol.c`

Multiple Light Sources

The algorithm can be easily extended to handle multiple light sources. For each light source, repeat the second pass of the algorithm, clearing the stencil buffer to “zero”, computing the shadow volume polygons, and rendering them to update the stencil buffer. Instead of replacing the pixel values of the unshadowed scenes, choose the appropriate blending function and add that lights contribution to the scene for each light. If more color accuracy is desired, use the accumulation buffer.

The accumulation buffer can also be used with this algorithm to create soft shadows. Jitter the light source position and repeat the steps described above for multiple light sources.

Shadow Volume Trade-offs

Shadow volumes can be very efficient if the shadowing object is simple. Difficulties occur when the shadowing object is a complex shape, making it difficult to compute a shadow volume. Ideally, the shadow volume should be generated from the vertices along the silhouette of the object, as seen from the light. This isn’t a trivial problem for complex shadowing objects.

Since the stencil count for objects in shadow depends on whether the eyepoint is in the shadow or not, making the algorithm independent of eye position is more difficult. One solution is to intersect the shadow volume with the view frustum, and use the result as the shadow volume. This can be a non-trivial CSG operation.

In certain pathological cases, the shape of the shadow volume may cause a stencil value underflow even if you render the front facing shadow polygons first. To avoid this problem, you can choose a “zero” value in the middle of the stencil values representable range. For an 8 bit stencil buffer, you could choose 128 as the “zero” value. The algorithm would be modified to initialize and test for this value instead of zero. The “zero” should be initialized to “zero” + 1 if the eye is inside the shadow volume.

Shadow volumes will test your polygon renderer's handling of adjacent polygons. If there are any rendering problems, the stencil count can get messed up, leading to grossly incorrect shadows.

There is a demo program showing shadow volumes on the website. It's called `shadowvol.c`

Shadow Maps

Shadow maps use the depth buffer and projective texture mapping to create a screen space method for shadowing objects [2, 10]. It's performance is not directly dependent on the complexity of the shadowing object.

The scene is transformed so that the eyepoint is at the light source. The objects in the scene are rendered, updating the depth buffer. The depth buffer is read back, then written into a texture map. This texture is mapped onto the primitives in the original scene, as viewed from the eyepoint, using the texture transformation matrix, and eye space texture coordinate generation. The value of the texture's texel values, the texture's "intensity", is compared against the texture coordinate's r value at each pixel. This comparison is used to determine whether the pixel is shadowed from the light source. If the r value of the texture coordinate is greater than texel value, the object was in shadow. If not, it was lit by the light in question.

This procedure works because the depth buffer records the distances from the light to every object in the scene, creating a *shadow map.* The smaller the value, the closer the object is to the light. The transform and texture coordinate generation is chosen so that x, y, and z locations of objects in the scene map to the s and t coordinates of the proper texels in the shadow texture map, and to r values corresponding to the distance from the light source. Note that the r values and texel values must be scaled so that comparisons between them are meaningful.

Both values measure the distance from an object to the light. The texel value is the distance between the light and the first object encountered along that texel's path. If the r distance is greater than the texel value, this means that there is an object closer to the light than this one. Otherwise, there is nothing closer to the light than this object, so it is illuminated by the light source. Think of it as a depth test done from the light's point of view.

Shadow maps can almost be done with the OpenGL 1.0 implementation. What's missing is the ability to compare the texture's r component against the corresponding texel value. There is an OpenGL extension, SGIX_shadow, that performs the comparison. As each texel is compared, the results set the fragment's alpha value to 0 or 1. The extension can be described as using the shadow texture/r value test to mask out shadowed areas using alpha values.

Shadow Map Trade-offs

Shadow maps have an advantage, being an image space technique, that they can be used to shadow any object that can be rendered. You don't have to find the silhouette edge of the shadow-

ing object, or clip the shadowed one. This is similar to the argument made for depth buffering vs. an object based technique hidden surface removal technique, such as depth sort.

The same image space drawbacks are also true. Since the shadow map is point sampled, then mapped onto objects from an entirely different point of view, aliasing artifacts are a problem. When the texture is mapped, the shape of the original shadow texel shape doesn't necessarily map cleanly to the pixel. Two major types of artifacts result from these problems; aliased shadow edges, and self-shadowing "shadow acne" effects.

These effects can't be fixed by simply averaging shadow map texel values. These values encode distances. They must be compared against r values, and generate a boolean result. Averaging the texel values would result in distance values that are simply incorrect. What needs to be blended is the boolean results of the r and texel comparison. The SGIX_shadow extension does this, blending four adjacent comparison results to produce an alpha value. Other techniques can be used to suppress aliasing artifacts:

Increase shadowmap/texture spatial resolution. Silicon graphics supports off-screen buffers on some systems, called a p-buffer, whose resolution is not tied to the window size. It can be used to create a higher resolution shadowmap.

Jitter the shadow texture by modifying the projection in the texture transformation matrix. The r/texel comparisons can then be averaged to smooth out shadow edges.

Modify the texture projection matrix so that the r values are biased by a small amount. Making the r values a little smaller is equivalent to moving the objects a little closer to the light. This prevents sampling errors from causing a curved surface from self shadowing. This r biasing can also be done with polygon offset.

One more problem with shadow maps should be noted. It is difficult to use the shadowmap technique to cast shadows from a light surrounded by objects. This is because the shadowmap is created by rendering the entire scene from the light's point of view. It's not always possible to come up with a transform to do this, depending on geometric relationship between the light and the objects in the scene.

A demo program using shadowmaps with the SGIX_shadow extension is on the website. It's called `shadowmap.c`

Line Rendering Techniques

Hidden Lines

This technique allows you to draw wireframe objects with the hidden lines removed, or drawn in a style different from the ones that are visible. This technique can clarify complex line drawings of objects, and improve their appearance [15].

The algorithm assumes that the object is composed of polygons. The algorithm first renders the polygons of the objects, then the edges themselves, which make up the line drawing. During the first pass, only the depth buffer is updated. During the second pass, the depth buffer only allows edges that are not obscured by the objects polygons to be rendered.

Here's the algorithm in detail:

1. Disable writing to the color buffer with `glColorMask()`
2. Enable depth testing with `glEnable(GL_DEPTH_TEST)`
3. Render the object as polygons
4. Enable writing to the color buffer
5. Render the object as edges

In order to improve the appearance of the edges (which might show z-buffer aliasing artifacts), you should use polygon offset or stencil decaling techniques to draw the polygon edges. One such technique, although not completely general, works well. Use the stencil buffer to mask off where all the lines, both hidden and visible, are. Then use the stencil function to prevent the polygon rendering to update the depth buffer where the stencil values have been set. When the visible lines are rendered, there is no depth value conflict, since the polygons never touched those pixels. Here's the modified algorithm:

1. Disable writing to the color buffer with `glColorMask()`
2. Disable depth testing; `glDisable(GL_DEPTH_TEST)`
3. Enable stenciling `glEnable(GL_STENCIL_TEST)`
4. Clear the stencil buffer
5. Set the stencil buffer to set the stencil values to 1 where pixels are drawn; `glStencilFunc(GL_ALWAYS, 1, 1)` `glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE)`
6. Render the object as edges
7. Use the stencil buffer to mask out pixels where the stencil value is 1; `glStencilFunc(GL_EQUAL, 1, 1)` and `glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP)`
8. Render the object as polygons
9. Turn off stenciling `glDisable(GL_STENCIL_TEST)`
10. Enable writing to the color buffer
11. Render the object as edges

The only problem with this algorithm is if the hidden and visible lines aren't all the same color, or interpolate colors between endpoints. In this case, it's possible for a hidden and visible line to overlap, in which case the most recent line will be the one that is drawn.

Instead of removing hidden lines, sometimes it's desirable to render them with a different color or pattern. This can be done with a modification of the algorithm:

1. Leave the color depth buffer enabled for writing
2. Set the color and/or pattern you want for the hidden lines
3. Render the object as edges
4. Disable writing to the color buffer
5. Render the object as polygons
6. Set the color and/or pattern you want for the visible lines
7. Render the object as edges

In this technique, all the edges are drawn twice; first with the hidden line pattern, then with the visible one. Rendering the object as polygons updates the depth buffer, preventing the second pass of line drawing to affect the hidden lines.

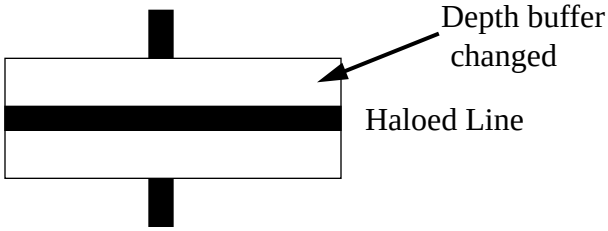
There is a demo program showing hidden lines on the website. It's called `hiddenline.c`

Haloed Lines

Haloing lines makes it easier to understand a wireframe drawing. Lines that pass behind other lines stop short a little before passing behind. It makes it clearer which line is in front of the other.

Haloed lines are easy to draw using the depth buffer. The technique has two passes. First disable writing to the color buffer; the first pass only updated the depth buffer. Set the line width to be greater than the normal line width you're using. The width you choose will determine the extent of the halos. Render the lines. Now set the line width back to normal, and enable writing to the color buffer. Render the lines again. Each line will be bordered on both sides by a wider "invisible line" in the depth buffer. This wider line will mask out other lines as they pass beneath it.

1. Disable writing to the color buffer
2. Enable the depth buffer for writing
3. Increase line width
4. Render lines
5. Restore line width
6. Enable writing to the color buffer
7. Ensure that depth testing is on, passing on `GL_LESS`
8. Render lines



There is a program showing haloed lines on the website. It's called `haloed.c`

Silhouette Lines

The stencil buffer can be used to render silhouette lines around an object. With this technique, you can render the object, then draw a silhouette around it, or just draw the silhouette itself [20].

The object is drawn 4 times; each time displaced by one pixel in the x or y direction. This offset must be done in window coordinates. An easy way to do this is to change the viewport coordinates each time, changing the viewport transform. The color and depth values are turned off, so only the stencil buffer is affected.

Every time the object covers a pixel, it increments the pixel's stencil value. When the four passes have been completed, the perimeter pixels of the object will have stencil values of 2 or 3. The interior will have values of 4, and all pixels surrounding the object exterior will have values of 0 or 1.

Here is the algorithm in detail:

1. If you want to see the object itself, render it in the usual way.
2. Clear the stencil buffer to zero.
3. Disable writing to the color and depth buffers
4. Set the stencil function to always pass, set the stencil operation to increment
5. Translate the object by +1 pixel in y, using `glViewport()`
6. Render the object
7. Translate the object by -2 pixels in y, using `glViewport()`
8. Render the object
9. Translate by +1 pixel x and +1 pixel in y
10. Render
11. Translate by -2 pixel in x
12. Render
13. Translate by +1 pixel in x. You should be back to the original position.
14. Turn on the color and depth buffer
15. Set the stencil function to pass if the stencil value is 2 or 3. Since the possible values range from 0 to 4, the stencil function can pass if stencil bit 1 is set (counting from 0).
16. Rendering any primitive that covers the object will draw only the pixels of the silhouette. For a solid color silhouette, render a polygon of the color desired over the object.

There is a demo program showing silhouette lines on the website. It's called `silhouette.c`

List of Demo Programs

This list shows all the demonstration programs available on the advanced rendering with opengl web site. The programs are grouped by the sections they're discussed in. Each line gives a short description of what the program demonstrates.

Texture Mapping

- mipmap_lines.c - shows different mipmap generation filters
- genmipmap.c - shows how to use the OpenGL pipeline to generate mipmaps
- textile.c - shows how to tile textures
- texpage.c - shows how to page textures
- textrim.c - shows how to trim textures
- textext.c - shows how draw characters with texture maps
- envmap.c - shows hows to do reflection mapping
- warp.c - shows how to warp images with textures

Blending

- comp.c - shows Porter/Duff compositing
- transp.c - shows how to draw transparent objects
- imgproc.c - shows image processing operations

The Z Coordinate and Perspective Projection

- depth.c - compare screen and eye space z

Multipass Rendering

- envphong.c - shows how to draw phong highlights with environment mapping
- projtex.c - shows how do spotlight illumination using projective textures
- mirror.c - shows how to do planar mirror reflections

Using the Stencil Buffer

- dissolve.c - shows how to do dissolves with the stencil buffer
- decal.c - shows how to decal coplanar polygons with the stencil buffer
- csg.c - shows how to render CSG solids with the stencil buffer
- zcomposite.c - shows how to composite depth-buffered images with the stencil buffer

Using the Accumulation Buffer

- motionblur.c - shows how to do motion blur with the accumulation buffer
- softshadow.c - shows how to do soft shadows with the accumulation buffer
- accumaa.c - shows how to antialias a scene with the accumulation buffer
- field.c - shows how to a depth of field effects with the accumulation buffer
- convolve.c - shows how to convolve with the accumulation buffer

Shadows: 3 Different Flavors

- projshadow.c - shows how to render projection shadows
- shadowvol.c - shows how to render shadows with shadow volumes
- shadowmap.c - shows how to render shadows with shadow maps

Line rendering techniques

- hiddenline.c - shows how to render wireframe objects with hidden lines
- haloed.c - shows how to draw haloed lines using the depth buffer
- silhouette.c - shows how to draw the silhouette edge of an object with the stencil buffer

Equation Appendix

This Appendix describes some important formula and matrices referred to in the text.

Projection Matrices

Perspective Projection

The call glFrustum(*l, r, b, t, n, f*) generates R, where:

$$R = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \text{ and } R^{-1} = \begin{bmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{-(f-n)}{2fn} & \frac{f+n}{2fn} \end{bmatrix}$$

R is defined as long as $l \neq r$, $t \neq b$, and $n \neq f$.

Orthographic Projection

The call glOrtho*(*l, r, b, t, u, f*) generates R, where:

$$R = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } R^{-1} = \begin{bmatrix} \frac{r-l}{2} & 0 & 0 & \frac{r+l}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\ 0 & 0 & \frac{f-n}{-2} & \frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

R is defined as long as $l \neq r$, $t \neq b$, and $n \neq f$.

Lighting Equations

Attenuation Factor

The *attenuation factor* is defined to be:

$$\text{attenuation factor} = \frac{1}{k_c + k_l d + k_q d^2}$$

where

d = distance between the light's position and the vertex

k_c = GL_CONSTANT_ATTENUATION

k_l = GL_LINEAR_ATTENUATION

k_q = GL_QUADRATIC_ATTENUATION

If the light is a directional one, the attenuation factor is 1.

Spotlight Effect

The *spotlight effect* evaluates to one of three possible values, depending on whether the light is actually a spotlight and whether the vertex lies inside or outside the cone of illumination produced by the spotlight:

- 1 if the light isn't a spotlight (GL_SPOT_CUTOFF is 180.0).
- 0 if the light is a spotlight but the vertex lies outside the cone of illumination produced by the spotlight.
- $(\max \{v \times d, 0\})^{\text{GL\_SPOT\_EXPONENT}}$ where:

  v = (v_x, v_y, v_z) is the unit vector that points from the spotlight (GL_POSITION) to the vertex.

  d = (d_x, d_y, d_z) is the spotlight's direction (GL_SPOT_DIRECTION), assuming the light is a spotlight and the vertex lies inside the cone of illumination produced by the spotlight.

  The dot product of the two vectors v and d varies as the cosine of the angle between them; hence, objects directly in line get maximum illumination, and objects off the axis have their illumination drop as the cosine of the angle.

To determine whether a particular vertex lies within the cone of illumination, OpenGL evaluates (max {v x d, 0}) where v and d are as defined above. If this value is less than the cosine of the spotlight's cutoff angle (GL_SPOT_CUTOFF), then the vertex lies outside the cone; otherwise, it's inside the cone.

Ambient Term

The ambient term is simply the ambient color of the light scaled by the ambient material property:

$\text{ambient}_{\text{light}} * \text{ambient}_{\text{material}}$

Diffuse Term

The diffuse term needs to take into account whether light falls directly on the vertex, the diffuse color of the light, and the diffuse material property:

$(\max \{l \cdot n, 0\}) * \text{diffuse}_{\text{light}} * \text{diffuse}_{\text{material}}$ where:

$l = (l_x, l_y, l_z)$ is the unit vector that points from the vertex to the light position (GL_POSITION).

$n = (n_x, n_y, n_z)$ is the unit normal vector at the vertex.

Specular Term

The specular term also depends on whether light falls directly on the vertex. If $l \cdot n$ is less than or equal to zero, there is no specular component at the vertex. (If it's less than zero, the light is on the wrong side of the surface.) If there's a specular component, it depends on the following:

- The unit normal vector at the vertex (n_x, n_y, n_z).
- The sum of the two unit vectors that point between (1) the vertex and the light position and (2) the vertex and the viewpoint (assuming that GL_LIGHT_MODEL_LOCAL_VIEWER is true; if it's not true, the vector (0, 0, 1) is used as the second vector in the sum). This vector sum is normalized (by dividing each component by the magnitude of the vector) to yield $s = (s_x, s_y, s_z)$.
- The specular exponent (GL_SHININESS).
- The specular color of the light ($GL\_SPECULAR_{light}$).
- The specular property of the material ($GL\_SPECULAR_{material)}$.

Using these definitions, here's how OpenGL calculates the specular term:

$(\max \{s \times n, 0\})^{shininess} * specular_{light} * specular_{material}$

However, if $1 \times n = 0$, the specular term is 0.

Putting It All Together

Using the definitions of terms described in the preceding paragraphs, the following represents the entire lighting calculation in RGBA mode.

$$\text{vertex color} = \text{emission}_{material} +$$

$$\text{ambient}_{light\ model} * \text{ambient}_{material} +$$

$$\sum_{i=0}^{i-1} \left(\frac{1}{k_c + k_1 d + k_q d^2} \right) (\text{spotlight effect})_i$$

$$[\text{ambient}_{light} * \text{ambient}_{material} +$$

$$(\max \{l \cdot n, 0\}) * \text{diffuse}_{light} * \text{diffuse}_{material} +$$

$$(\max \{s \cdot n, 0\})^{shininess} * \text{specular}_{light} * \text{specular}_{material}]_i$$

References

[1]J Goldfeather, J.P. Hultquist. “Fast Constructive Geometry Display in the Pixel-Powers Graphics System”, *Computer Graphics*, Volume 20, No. 4, pp. 107-116, 1986.
[2]W. T. Reeves, D. H. Salesin, R.L. Cook. “Rendering Antialiased Shadows with Depth Maps”, *Computer Graphics*, Volume 21, No. 4, pp. 283-291, 1987.
[3]T. Duff, T. Porter. “Compositing Digital Images”, *Computer Graphics*, Volume 18, No. 3, pp. 253-259, 1984.
[4]B. T. Phong. “Illumination for computer generated pictures”, *Communications of the ACM*, No. 18, pp. 311-317, 1978
[5]K. Akeley. “Reality Engine Graphics”, *Computer Graphics* , Volume 27, pp. 109-116, 1993
[6]T. Nishita, E. Nakamae, “Method of Displaying Optical Effects within Water using Accumulation Buffer”, *Computer Graphics*, Volume 28, pp. 373-379, 1994.
[7]D. Voorhies, J. Foran, “Reflection Vector Shading Hardware”, *Computer Graphics*, Volume 28, pp. 163-166, 1994.
[8]T. Duff, “Compositing 3-D Rendered Images”, *Computer Graphics*, Volume 19, pp. 41-44, 1985.
[9]G. Gardner, “Visual Simulation of Clouds”, *Computer Graphics*, Volume 19, pp. 297-303, 1985.
[10]M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, P. Haeberli, “Fast Shadows and Lighting Effects Using Texture Mapping”, *Computer Graphics*, Volume 26, pp. 249-252, 1992.
[11]D. Mitchell, A. Netravali, “Reconstruction Filters in Computer Graphics”, *Computer Graphics*, Volume 22, pp. 221-228, 1988.
[12]S. Upstill, *The Renderman Companion*, Addison-Wesley, Menlo Park, 1990.
[13]J. Neider, T. Davis, M. Woo, *OpenGL Programming Guide*, Addison-Wesley, Menlo Park, 1993.
[14] T. Tessman, “Casting Shadows on Flat Surfaces”, *Iris Universe*, Winter, pp. 16, 1989.
[15]Y. Attarwala, “Rendering Hidden Lines”, *Iris Universe*, Fall, pp. 39, 1988.
[16] Y. Attarwala, M. Kong, “Picking From the Picked Few”, *Iris Universe*, Summer, pp. 40-41, 1989.
[17]P. Rustagi, “Image Roaming with the Help of Tiling and Memory-Mapped Files”, *Iris Universe*, Number 15, pp. 12-13.
[18] J. Kichury, “Stencil Functions for Data Visualization”, *Iris Universe*, Number 18, pp. 12-15.
[19] T. Heidmann, “Real Shadows Real Time”, *Iris Universe*, Number 18, pp. 28-31.
[20] P. Rustagi, “Silhouette Line Display from Shaded Models”, Iris Universe, Fall, pp. 42-44, 1989.
[21] M. Schulman, “Rotation Alternatives”, *Iris Universe*, Spring, pp. 39, 1989.
[22] K. Akeley, “OpenGL Philosophy and the Philosopher’s Drinking Song”, Personal Communiation.
[23] P. Haeberli, K. Akeley, “The Accumulation Buffer: Hardware Support for High-Quality Rendering”, Computer Graphics, Volume 24, No. 4, pp. 309-318, 1990.
[24]K. Akeley, “The Hidden Charms of Z-Buffer”, *Iris Universe*, No. 11, pp 31-37.
[25]R. A. Drebin, L. Carpenter, P. Hanrahan, “Volume Rendering”, *Computer Graphics*, Volume 22, No. 4, pp. 65-74, 1988.
[26]L. Williams, “Pyramidal Parametrics”, *Computer Graphics*, Volume 17, No. 3, pp. 1-11. 1983.

[27]P. Haeberli, M. Segal, “Texture Mapping as a Fundamental Drawing Primitive”, *Proceedings of the fourth Eurographics Workshop on Rendering*, pp. 259-266, 1993.
[28]M. Teschner, “Texture Mapping: New Dimensions in Scientific and Technical Visualization”, *Iris Universe*, No. 29, pp. 8-11.
[29]P. Haeberli, D. Voorhies, “Image Processing by Linear Interpolation and Extrapolation”, *Iris Universe*, No. 28, pp. 8-9.

Building an OpenGL Volume Renderer

by Todd Kulick, Applications Marketing Engineer

The ability to produce volume-rendered images interactively opens the door to a host of new application capabilities. Volumetric data is commonplace today. Radiologists use magnetic resonance images (MRI) and computed tomography (CT) data in clinical diagnoses. Geophysicists map and study three-dimensional voxel Earth models. Environmentalists examine pollution clouds in the air and plumes underground. Chemists and biologists visualize potential fields around molecules and meteorologists study weather patterns. With so many disciplines actively engaging in the study and examination of three-dimensional data, today's software developers need to understand techniques used to visualize this data. You can use three-dimensional texture mapping, an extension of two-dimensional texture mapping, as the basis for building fast, flexible volume renderers.

This article tells you how to build an interactive, texture mapping-based volume renderer in OpenGL. The article also includes a pseudo-coded volume renderer to help illustrate particular concepts.

Understanding Volume Rendering

Volume rendering is a powerful rendering technique for three-dimensional data volumes that does not rely on intermediate geometric representation. The elements of these volumes, the three-dimensional analog to pixels, are called *voxels*. The power of volume-rendered images is derived from the direct treatment of these voxels. Contrasting volume rendering with isosurface methods reveals that the latter methods are computationally expensive and show only a small portion of the data. On the other hand, volume rendering lets you display more data, revealing fine detail and global trends in the same image. Consequently, volume rendering enables more direct understanding of visualized data with fewer visual artifacts.

All volume-rendering techniques accomplish the same basic tasks: coloring the voxels; computing voxel-to-pixel projections; and combining the colored, projected voxels. Lookup tables and lighting color each voxel based on its visual properties and data value. You determine a pixel's color by combining all the colored voxels that project onto it. This combining takes many forms, often including summing and blending calculations. This variability in coloring and combining allows volume-rendered images to emphasize, among other things, a particular data value, the internal data gradient, or both at once. Traditionally, volume renderers produce their images by one of three methods:

- Ray casting [Sab88]
- Splatting [UpK88]
- Forward projection [DCH88]

Using the OpenGL Graphics API

In contrast to the three traditional methods just cited, this article discusses a volume-rendering method that uses the OpenGL graphics API. The primitives and techniques that this method integrates are common graphics

principles, not generalized mathematical abstractions.

The basic approach of this method is to define a three-dimensional texture and render it by mapping it to a stack of slices that are blended as they are rendered back to front. Figure 1 shows a stack of slices. The first step of this technique involves defining one or more three-dimensional textures containing the volumetric data. The next step involves using lookup tables to color textures. A stack of slices is drawn through the volume with the colored texture data mapped onto the slices. The slices are blended in front-to-back order into the framebuffer. You can vary the blending type and lookup tables to produce different kinds of volume renderings.

The advantage of this technique is that you can implement it with presently available hardware-accelerated, OpenGL graphics architectures. As a result, you can produce volume-rendering applications that run at interactive rates. This method also preserves quality and provides considerable flexibility in the rendering scheme.

Building the Texture Data

Before you can render volumetric data, you must define it as a texture. You can load the data as one three-dimensional texture into OpenGL if it is not too large for the texturing subsystem, but this is uncommon since hardware texture mapping OpenGL architectures typically place a limit on the size of the largest texture. This article assumes that size is an issue, because the hardware size limit is often significantly smaller than that of the average data set that you might volume render.

You can work around this limit by dividing your data into multiple textures. The most common method of doing this is to divide the volume into small bricks that are each defined as a single texture. These bricks are rectangular subvolumes of the original data, which are often created by cutting the volume in half (or thirds) in one or more dimensions.

Since you composite each slice on top of the others with order-dependent blending operations, you must ensure a total ordering of all the rendered slices for each pixel. If you do not maintain the order, the blending operations used to combine slices will produce an incorrect result. The most common ordering is back to front. Texels farthest from the eye point are drawn first and those closest are drawn last.

To preserve the total ordering of all slices at each pixel, the bricks are rendered one at a time. To preserve the ordering, the bricks are sorted and rendered from back to front. The bricks farthest from the eye are rendered before closer ones.

Coloring the Texture Data

Perhaps the most important feature afforded by volume rendering is *classification*, which is the process of grouping similar objects inside a volume. You can use classification to color these groups. For example, in a volume of CT density data, you could color bone white and tissue red. Not only can you color objects, but you can also remove the objects in a given group

from the scene altogether. You can also make groups semi-transparent, enabling you to view both the structure of the top group as well as the groups beneath it.

One method of data classification involves passing the original volumetric data through lookup tables. These tables select the color and opacity to represent each data value. OpenGL supports many different lookup tables at several places along the imaging pipeline. You want to use the texture-lookup tables, which occur after the texturing system. Using these tables allows the texturing system to interpolate the *actual* input data and then look up the result, rather than interpolating the *looked-up* color and opacity values. For architectures that do not support these lookup tables, use the OpenGL color table or pixel map.

Rendering the Texture Data

Once you classify the volumetric data and make it available as texture data, you render it by texture mapping it to a collection of polygons. The best way to sample the texture data is to render the volume as parallel, equidistant texture-mapped slices that are orthogonal to the viewing direction. You clip these slices to the extents of the volume data to create polygons. Figure 1A shows these sampling slices untextured; Figure 1B shows the same slices textured. The object in the volume (a skull) is vaguely discernible. Adding more slices would improve the image.

The number of slices rendered is directly related to both quality and speed. The more slices that you render, the more accurate the results, but the longer the rendering time. The number of slices required to adequately sample the data in the viewing direction is a function of the data itself as dictated by the *Nyquist rate*. The texture data is always sampled at screen pixel resolution in the directions parallel to the viewing plane, due to the nature of the OpenGL texture-mapping capability. To increase the quality of the results, have OpenGL resample the texture linearly.

This viewer-orthogonal sampling technique has two major advantages:

- The number of slices drawn is the same for each viewing angle.

  This means that the frame rate and the sampling characteristics of the renderer will not change with angle.

- The viewer-orthogonal slices do not require the textures to be perspectively corrected as they are mapped.

  Architectures that do not support corrective mapping will not suffer from the associated geometric distortion that otherwise would be created.

Blending the slices computes the discrete form of an integral. The result is equivalent to integrating along a ray leaving the eye and passing through the screen and then the volumetric data. Each slice represents an area along this ray. To compute this integral correctly, the slices must be attenuated when the area that they represent decreases. This means that as more slices are rendered, each one must get dimmer in order to maintain a constant brightness for the image. If you examine the integral, this attenuation works out to be the exponential of the distance between slices. You can compute this attenuation and use it as the alpha value for the slice

polygons when you render them.

Another way to render the volumetric data is called *multi-planar reconstruction* (*MPR*). Only one slice is drawn in an MPR rendering. You can orient this slice at any angle and then use it to inspect the interior structures in the data. Maintaining a high-quality (at least linear) resampling of the data during MPR rendering is crucial. Multi-planar reconstruction can be fast and informative, but the resulting images are generally not considered to be volume renderings.

Determining the Texture Coordinates

When mapping the texture data to slices, you must be very careful. OpenGL texture space mapping can be tricky, and failure to map the texture data to your slices correctly can produce visual artifacts. Even though OpenGL specifies that textures lie in a [0.0,1.0] coordinate space in each dimension, the texture data you provide is not mapped onto the entire region. Figure 2 shows a one-dimensional, four-texel texture. Notice that the [0.0,1.0] space is divided into four regions, one for each texel. The texels are defined at the center of each region. Coordinates between two centers are colored by combining the values of both nearby texels. The texture coordinates before the first texel center or after the last are colored by both the corresponding edge texel and the border. *Volume slices should never be colored by the border*. *Therefore, you should avoid specifying coordinates before the first texel center or after the last*. Also notice that the location of the edge texels' centers move as the texture size changes.

A more subtle texture-coordinate problem is caused by using multiple textures. Using multiple three-dimensional textures causes seams between adjacent bricks. The seams are created because the edges of adjacent bricks are not interpolated. Figure 3A shows the result when a four-texel, one-dimensional checkerboard texture is mapped to a simple rectangle. The seam in Figure 3B is created when the single texture is divided into two two-texel textures. These seams also appear between adjacent bricks when volume rendering. To correct this problem, duplicate the edge texels in both bricks so that each one appears twice. This causes both bricks to grow by one to three texels. By adjusting the texture coordinates at the vertices—so that each brick handles half of the previously missing region—you can remove the seam. Figure 3C shows these two textures with the corrected texel coordinates. The seam is gone.

Compositing the Textured Slices

After creating the brick textures, mapping them to slices, and segmenting the data, you are ready to composite the slices. The most common compositing scheme for back-to-front volume rendering is called *over blending*. Each compositing scheme is good at accenting different kinds of internal data structures, and the ability to switch between them interactively is extremely useful.

OpenGL supports a collection of framebuffer blending operations that you can use to construct different compositing operations. You can simulate over blending by specifying *addition* as the OpenGL blend equation. The blend

function multiplies the incoming slice by its alpha value, and multiplies the framebuffer pixels by one minus the source alpha value. You can produce maximum-intensity blending by setting the OpenGL blending equation to the maximum operator. You can produce other blending types by using different combinations of the OpenGL blend function and blend-equation parameters.

Mixing in Geometry

You can display related opaque geometric data in concert with volume-rendered data. Geometric data is correctly sorted and blended if it lies inside the volume. You can also combine traditional geometric data renderings with volumes, producing, for example, a surface embedded in a cloud of data.

The OpenGL zbuffer allows you to sort opaque objects. Semi-transparent objects are more complex because, unlike opaque objects, their rendering result is order-dependent. However, if your scene contains only opaque geometry with the volumetric data, then you can combine the two by doing the following:

1. Activate the zbuffer.
2. Enable the zbuffer for reading and writing.
3. Render the opaque geometry.
4. Disable zbuffer writing, but leave reading enabled.
5. Render the texture bricks.

This data is correctly sorted and blended on top of the geometric data where appropriate.

Getting Examples

You can examine examples of the techniques described in this article as source-coded implementations. The volume renderers are available using anonymous ftp. V*ox* is a compact, simple renderer perfect for studying the basic principles of a volume-rendering application. *Volren*, which is larger and more complex, supports all the functions discussed in this article plus quite a bit more. However, the volren code is significantly more complex. You can find these volume renderers at the following sites:

Anonymous FTP site: **sgigate.sgi.com**

Vox distribution: **/pub/demos/vox.tar.Z**

Volren distribution: **/pub/demos/volren-<version>.tar.Z**

Pseudo-Coding a Volume Renderer

```
Draw()
{
  /* enable the depth buffer for reading and writing */
  glEnable(GL_DEPTH_TEST);
  glDepthMask(TRUE);

  /* draw opaque geometry */
  ...

  /* enable the depth buffer for reading only */
  glDepthMask(FALSE);

  /* load the classification lookup tables */
  glColorTableSGI(GL_TEXTURE_COLOR_TABLE_SGI, GL_RGBA, ...);
  glEnable(GL_TEXTURE_COLOR_TABLE_SGI);

  /* Compute the per slice alpha weight */
  alphaWeight = exp(interSliceDistance);
  glColor4f(1.0, 1.0, 1.0, alphaWeight);

  /* setup the compositing function */
  if over blending then {
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glBlendEquationEXT(GL_FUNC_ADD_EXT);
  }
```

```
if maximum intensity blending then {
  glBlendEquationEXT(GL_MAX_EXT);
}

/* sort the bricks back-to-front from eye point */
...

/* enable texturing and blending */
glEnable(GL_TEXTURE_3D_EXT);
glEnable(GL_BLEND);

/* render the bricks */
for each brick B in sorted order do {

  /* load brick B */
  glTexParameter(GL_TEXTURE_3D_EXT, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameter(GL_TEXTURE_3D_EXT, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexImage3DEXT(GL_TEXTURE_3D_EXT, ...);

  /* sort the slices back-to-front from eye point */
  ...

  for each slice S in brick B in sorted order do {

    /* compute spatial coordinates of slice S */
    intersect slice with brick extents
```

```
      /* compute texture coordinates of slice S */
      compute corresponding texture domain coordinates

      /* adjust texture coordinates to account for seams */
      scale back texture coordinates

      /* render the slice */
      render slice S
    }
  }
}
```

Conclusion

Volume rendering is a powerful, flexible way to visualize data. Previously available only through slow, software solutions, it is now available through OpenGL, a standardized, platform-independent programming interface. OpenGL enables volume-rendering solutions that are fast, portable, cost-effective, and maintainable. Using OpenGL, you can produce software products that combine geometry, vectors, polygons, and volumes with the ability to integrate new features rapidly.

References

Sabella, Paolo, “A Rendering Algorithm for Visualizing 3D Scalar Fields,” Computer Graphics (SIGGRAPH ’88 Proceedings) 22(4) pp. 51-58 (August 1988).

Upson, Craig and Keeler, Michael, “V-BUFFER: Visible Volume Rendering,” Computer Graphics (SIGGRAPH ’88 Proceedings) 22(4) pp. 59-64 (August 1988).

Drebin, Robert A., Carpenter, Loren, and Hanrahan, Pat, “Volume Rendering,” Computer Graphics (SIGGRAPH ’88 Proceedings) 22(4) pp. 65-74 (August 1988).

Fast Shadows and Lighting Effects Using Texture Mapping

Mark Segal
Carl Korobkin
Rolf van Widenfelt
Jim Foran
Paul Haeberli
Silicon Graphics Computer Systems*

Abstract

Generating images of texture mapped geometry requires projecting surfaces onto a two-dimensional screen. If this projection involves perspective, then a division must be performed at each pixel of the projected surface in order to correctly calculate texture map coordinates.

We show how a simple extension to perspective-correct texture mapping can be used to create various lighting effects. These include arbitrary projection of two-dimensional images onto geometry, realistic spotlights, and generation of shadows using shadow maps[10]. These effects are obtained in real time using hardware that performs correct texture mapping.

CR Categories and Subject Descriptors: I.3.3 **[Computer Graphics]:** Picture/Image Generation; I.3.7 **[Computer Graphics]:** Three-Dimensional Graphics and Realism - *color, shading, shadowing, and texture*

Additional Key Words and Phrases: lighting, texture mapping

1 Introduction

Producing an image of a three-dimensional scene requires finding the projection of that scene onto a two-dimensional screen. In the case of a scene consisting of texture mapped surfaces, this involves not only determining where the projected points of the surfaces should appear on the screen, but also which portions of the texture image should be associated with the projected points.

*2011 N. Shoreline Blvd., Mountain View, CA 94043

If the image of the three-dimensional scene is to appear realistic, then the projection from three to two dimensions must be a perspective projection. Typically, a complex scene is converted to polygons before projection. The projected vertices of these polygons determine boundary edges of projected polygons.

Scan conversion uses iteration to enumerate pixels on the screen that are covered by each polygon. This iteration in the plane of projection introduces a homogeneous variation into the parameters that index the texture of a projected polygon. We call these parameters *texture coordinates*. If the homogeneous variation is ignored in favor of a simpler linear iteration, incorrect images are produced that can lead to objectionable effects such as texture "swimming" during scene animation[5]. Correct interpolation of texture coordinates requires each to be divided by a common denominator for each pixel of a projected texture mapped polygon[6].

We examine the general situation in which a texture is mapped onto a surface via a projection, after which the surface is projected onto a two dimensional viewing screen. This is like projecting a slide of some scene onto an arbitrarily oriented surface, which is then viewed from some viewpoint (see Figure 1). It turns out that handling this situation during texture coordinate iteration is essentially no different from the more usual case in which a texture is mapped linearly onto a polygon. We use *projective textures* to simulate spotlights and generate shadows using a method that is well-suited to graphics hardware that performs divisions to obtain correct texture coordinates.

2 Mathematical Preliminaries

To aid in describing the iteration process, we introduce four coordinate systems. The *clip* coordinate system is a homogeneous representation of three-dimensional space, with x, y, z, and w coordinates. The origin of this coordinate system is the viewpoint. We use the term clip coordinate system because it is this system in which clipping is often carried out. The *screen* co-

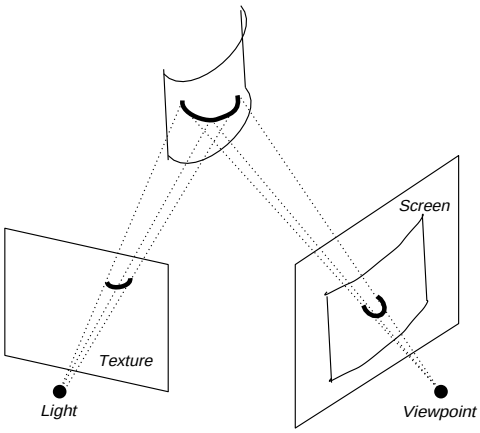


Figure 1. Viewing a projected texture.

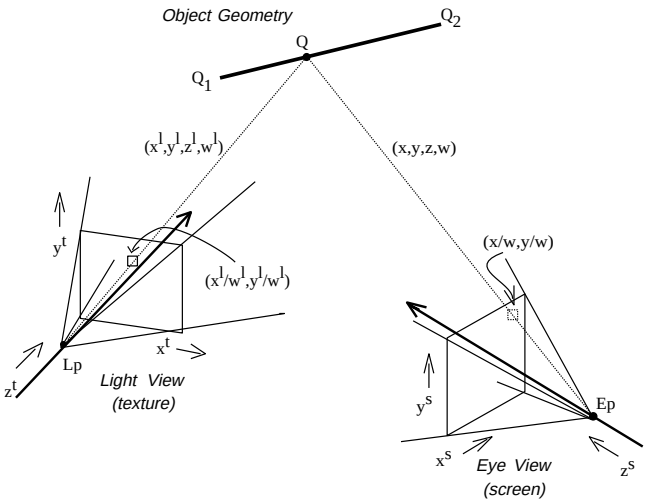


Figure 2. Object geometry in the light and clip coordinate systems.

ordinate system represents the two-dimensional screen with two coordinates. These are obtained from clip coordinates by dividing x and y by w, so that screen coordinates are given by $x^s = x/w$ and $y^s = y/w$ (the s superscript indicates screen coordinates). The *light* coordinate system is a second homogeneous coordinate system with coordinates x^l, y^l, z^l, and w^l; the origin of this system is at the light source. Finally, the *texture* coordinate system corresponds to a texture, which may represent a slide through which the light shines. Texture coordinates are given by $x^t = x^l/w^l$ and $y^t = y^l/w^l$ (we shall also find a use for $z^t = z^l/w^l$). Given (x^s, y^s), a point on a scan-converted polygon, our goal is to find its corresponding texture coordinates, (x^t, y^t).

Figure 2 shows a line segment in the clip coordinate system and its projection onto the two-dimensional screen. This line segment represents a span between two edges of a polygon. In clip coordinates, the endpoints of the line segment are given by

$$\mathbf{Q}_1 = (x_1, y_1, z_1, w_1) \qquad \text{and} \qquad \mathbf{Q}_2 = (x_2, y_2, z_2, w_2).$$

A point $\mathbf{Q}$ along the line segment can be written in clip coordinates as

$$\mathbf{Q} = (1-t)\mathbf{Q}_1 + t\mathbf{Q}_2 \tag{1}$$

for some $t \in [0, 1]$. In screen coordinates, we write the corresponding projected point as

$$\mathbf{Q}^s = (1-t^s)\,\mathbf{Q}_1^s + t^s\,\mathbf{Q}_2^s \tag{2}$$

where $\mathbf{Q}_1^s = \mathbf{Q}_1/w_1$ and $\mathbf{Q}_2^s = \mathbf{Q}_2/w_2$.

To find the light coordinates of $\mathbf{Q}$ given $\mathbf{Q}^s$, we must find the value of t corresponding to t^s (in general $t \neq t^s$). This is accomplished by noting that

$$\mathbf{Q}^s = (1-t^s)\,\mathbf{Q}_1/w_1 + t^s\,\mathbf{Q}_2/w_2 = \frac{(1-t)\mathbf{Q}_1 + t\mathbf{Q}_2}{(1-t)w_1 + tw_2} \tag{3}$$

and solving for t. This is most easily achieved by choosing a and b such that $1-t^s = a/(a+b)$ and $t^s = b/(a+b)$; we also choose A and B such that $(1-t) = A/(A+B)$ and $t = B/(A+B)$. Equation 3 becomes

$$\mathbf{Q}^s = \frac{a\mathbf{Q}_1/w_1 + b\mathbf{Q}_2/w_2}{(a+b)} = \frac{A\mathbf{Q_1} + B\mathbf{Q}_2}{Aw_1 + Bw_2}. \tag{4}$$

It is easily verified that $A = aw_2$ and $B = bw_1$ satisfy this equation, allowing us to obtain t and thus $\mathbf{Q}$.

Because the relationship between light coordinates and clip coordinates is affine (linear plus translation), there is a homogeneous matrix M that relates them:

$$\mathbf{Q}^l = M\,\mathbf{Q} = \frac{A}{A+B}\mathbf{Q}_1^l + \frac{B}{A+B}\mathbf{Q}_2^l \tag{5}$$

where $\mathbf{Q}_1^l = (x_1^l, y_1^l, z_1^l, w_1^l)$ and $\mathbf{Q}_2^l = (x_2^l, y_2^l, z_2^l, w_2^l)$ are the light coordinates of the points given by $\mathbf{Q}_1$ and $\mathbf{Q}_2$ in clip coordinates.

We finally obtain

$$\begin{aligned} \mathbf{Q}^t &= \mathbf{Q}^l/w^l \\ &= \frac{A\mathbf{Q}_1^l + B\mathbf{Q}_2^l}{Aw_1^l + Bw_2^l} \\ &= \frac{a\mathbf{Q}_1^l/w_1 + b\mathbf{Q}_2^l/w_2}{a(w_1^l/w_1) + b(w_2^l/w_2)}. \end{aligned} \tag{6}$$

Equation 6 gives the texture coordinates corresponding to a linearly interpolated point along a line segment in screen coordinates. To obtain these coordinates at a pixel, we must linearly interpolate x^l/w, y^l/w, and w^l/w, and divide at each pixel to obtain

$$x^l/w^l = \frac{x^l/w}{w^l/w} \qquad \text{and} \qquad y^l/w^l = \frac{y^l/w}{w^l/w}. \tag{7}$$

(For an alternate derivation of this result, see [6].)

If w^l is constant across a polygon, then Equation 7 becomes

$$s = \frac{s/w}{1/w} \qquad \text{and} \qquad t = \frac{t/w}{1/w}, \tag{8}$$

where we have set $s = x^l/w^l$ and $t = y^l/w^l$. Equation 8 governs the iteration of texture coordinates that have simply been assigned to polygon vertices. It still implies a division for each pixel contained in a polygon. The more general situation of a projected texture implied by Equation 7 requires only that the divisor be w^l/w instead of $1/w$.

3 Applications

To make the various coordinates in the following examples concrete, we introduce one more coordinate system: the *world* coordinate system. This is the coordinate system in which the three-dimensional model of the scene is described. There are thus two transformation matrices of interest: M_c transforms world coordinates to clip coordinates, and M_l transforms world coordinates to light coordinates. Iteration proceeds across projected polygon line segments according to equation 6 to obtain texture coordinates (x^t, y^t) for each pixel on the screen.

3.1 Slide Projector

One application of projective texture mapping consists of viewing the projection of a slide or movie on an arbitrary surface[9][2]. In this case, the texture represents the slide or movie. We describe a multi-pass drawing algorithm to simulate film projection.

Each pass entails scan-converting every polygon in the scene. Scan-conversion yields a series of screen points and corresponding texture points for each polygon. Associated with each screen point is a color and z-value, denoted c and z, respectively. Associated with each corresponding texture point is a color and z-value, denoted c_τ and z_τ. These values are used to modify corresponding values in a framebuffer of pixels. Each pixel, denoted p, also has an associated color and z-value, denoted c_p and z_p.

A color consists of several indepenedent components (e.g. red, green, and blue). Addition or multiplication of two colors indicates addition or multiplication of each corresponding pair of components (each component may be taken to lie in the range $[0, 1]$).

Assume that z_p is initialized to some large value for all p, and that c_p is initialized to some fixed ambient scene color for all p. The slide projection algorithm consists of three passes; for each scan-converted point in each pass, these actions are performed:

Pass 1 If $z < z_p$, then $z_p \leftarrow z$ *(hidden surface removal)*

Pass 2 If $z = z_p$, then $c_p \leftarrow c_p + c_\tau$ *(illumination)*

Pass 3 Set $c_p = c \cdot c_p$ *(final rendering)*

Pass 1 is a z-buffering step that sets z_p for each pixel. Pass 2 increases the brightness of each pixel according to the projected spotlight shape; the test ensures that portions of the scene visible from the eye point are brightened by the texture image only once (occlusions are not considered). The effects of multiple film projections may be incorporated by repeating Pass 2 several times, modifying M_l and the light coordinates appropriately on each pass. Pass 3 draws the scene, modulating the color of each pixel by the corresponding color of the projected texture image. Effects of standard (i.e. non-projective) texture mapping may be incorporated in this pass. Current Silicon Graphics hardware is capable of performing each pass at approximately 10^5 polygons per second.

Figure 3 shows a slide projected onto a scene. The left image shows the texture map; the right image shows the scene illuminated by both ambient light and the projected slide. The projected image may also be made to have a particular focal plane by rendering the scene several times and using an accumulation buffer as described in [4].

The same configuration can transform an image cast on one projection plane into a distinct projection plane. Consider, for instance, a photograph of a building's facade taken from some position. The effect of viewing the facade from arbitrary positions can be achieved by projecting the photograph back onto the building's facade and then viewing the scene from a different vantage point. This effect is useful in walk-throughs or fly-bys; texture mapping can be used to simulate buildings and distant scenery viewed from any viewpoint[1][7].

3.2 Spotlights

A similar technique can be used to simulate the effects of spotlight illumination on a scene. In this case the texture represents an intensity map of a cross-section of the spotlight's beam. That is, it is as if an opaque screen were placed in front of a spotlight and the intensity at each point on the screen recorded. Any conceivable spot shape may be accommodated. In addition, distortion effects, such as those attributed to a shield or a lens, may be incorporated into the texture map image.

Angular attenuation of illumination is incorporated into the intensity texture map of the spot source. Attenuation due to distance may be approximated by applying a function of the depth values $z^t = z^l/w^l$ iterated along with the texture coordinates (x^t, y^t) at each pixel in the image.

This method of illuminating a scene with a spotlight is useful for many real-time simulation applications, such

Figure 3. Simulating a slide projector.

as aircraft landing lights, directable aircraft taxi lights, and automotive headlights.

3.3 Fast, Accurate Shadows

Another application of this technique is to produce shadows cast from any number of point light sources. We follow the method described by Williams[10], but in a way that exploits available texture mapping hardware.

First, an image of the scene is rendered from the viewpoint of the light source. The purpose of this rendering is to obtain depth values in light coordinates for the scene with hidden surfaces removed. The depth values are the values of z^l/w^l at each pixel in the image. The array of z^t values corresponding to the hidden surface-removed image are then placed into a texture map, which will be used as a *shadow map*[10][8]. We refer to a value in this texture map as z_τ.

The generated texture map is used in a three-pass rendering process. This process uses an additional framebuffer value α_p in the range $[0, 1]$. The initial conditions are the same as those for the slide projector algorithm.

Pass 1 If $z < z_p$, then $z_p \leftarrow z$, $c_p \leftarrow c$ *(hidden surface removal)*

Pass 2 If $z_\tau = z^t$, then $\alpha_p \leftarrow 1$; else $\alpha_p \leftarrow 0$ *(shadow testing)*

Pass 3 $c_p \leftarrow c_p +$ (c modulated by α_p) *(final rendering)*

Pass 1 produces a hidden surface-removed image of the scene using only ambient illumination. If the two values in the comparison in Pass 2 are equal, then the point represented by p is visible from the light and so is not in shadow; otherwise, it is in shadow. Pass 3, drawn with full illumination, brightens portions of the scene that are not in shadow.

In practice, the comparison in Pass 2 is replaced with $z_\tau > z^t + \epsilon$, where ϵ is a bias. See [8] for factors governing the selection of ϵ.

This technique requires that the mechanism for setting α_p be based on the result of a comparison between a value stored in the texture map and the iterated z^t. For accuracy, it also requires that the texture map be capable of representing large z_τ. Our latest hardware posseses these capabilites, and can perform each of the above passes at the rate of at least 10^5 polygons per second.

Correct illumination from multiple colored lights may be produced by performing multiple passes. The shadow effect may also be combined with the spotlight effect described above, as shown in Figure 4. The left image in this figure is the shadow map. The center image is the spotlight intensity map. The right image shows the effects of incorporating both spotlight and shadow effects into a scene.

This technique differs from the hardware implementation described in [3]. It uses existing texture mapping hardware to create shadows, instead of drawing extruded shadow volumes for each polygon in the scene. In addition, percentage closer filtering [8] is easily supported.

4 Conclusions

Projecting a texture image onto a scene from some light source is no more expensive to compute than simple texture mapping in which texture coordinates are assinged to polygon vertices. Both require a single division per-pixel for each texture coordinate; accounting for the texture projection simply modifies the divisor.

Viewing a texture projected onto a three-dimensional scene is a useful technique for simulating a number of effects, including projecting images, spotlight illumination, and shadows. If hardware is available to perform texture mapping and the per-pixel division it requires, then these effects can be obtained with no performance penalty.

Figure 4. Generating shadows using a shadow map.

Acknowledgements

Many thanks to Derrick Burns for help with the texture coordinate iteration equations. Thanks also to Tom Davis for useful discussions. Dan Baum provided helpful suggestions for the spotlight implementation. Software Systems provided some of the textures used in Figure 3.

References

[1] Robert N. Devich and Frederick M. Weinhaus. Image perspective transformations. *SPIE*, 238, 1980.

[2] Julie O'B. Dorsey, Francois X. Sillion, and Donald P. Greenberg. Design and simulation of opera lighting and projection effects. In *Proceedings of SIGGRAPH '91*, pages 41–50, 1991.

[3] Henry Fuchs, Jack Goldfeather, and Jeff P. Hultquist, et al. Fast spheres, shadows, textures, transparencies, and image enhancements in pixels-planes. In *Proceedings of SIGGRAPH '85*, pages 111–120, 1985.

[4] Paul Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Proceedings of SIGGRAPH '90*, pages 309–318, 1990.

[5] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, UC Berkeley, June 1989.

[6] Paul S. Heckbert and Henry P. Moreton. Interpolation for polygon texture mapping and shading. In David F. Rogers and Rae A. Earnshaw, editors, *State of the Art in Computer Graphics: Visualization and Modeling*, pages 101–111. Springer-Verlag, 1991.

[7] Kazufumi Kaneda, Eihachiro Nakamae, Tomoyuki Nishita, Hideo Tanaka, and Takao Noguchi. Three dimensional terrain modeling and display for environmental assessment. In *Proceedings of SIGGRAPH '89*, pages 207–214, 1989.

[8] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Proceedings of SIGGRAPH '87*, pages 283–291, 1987.

[9] Steve Upstill. *The RenderMan Companion*, pages 371–374. Addison Wesley, 1990.

[10] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH '78*, pages 270–274, 1978.

Texture Mapping in Technical, Scientific and Engineering Visualization

Michael Teschner[1] and Christian Henn[2]

[1]Chemistry and Health Industry Marketing,
Silicon Graphics Basel, Switzerland

[2]Maurice E. Mueller−Institute for Microscopy,
University of Basel, Switzerland

Executive Summary

As of today, texture mapping is used in visual simulation and computer animation to reduce geometric complexity while enhancing realism. In this report, this common usage of the technology is extended by presenting application models of real−time texture mapping that solve a variety of visualization problems in the general technical and scientific world, opening new ways to represent and analyze large amounts of experimental or simulated data.

The topics covered in this report are:

- Abstract definition of the texture mapping concept
- Visualization of properties on surfaces by color coding
- Information filtering on surfaces
- Real−time volume rendering concepts
- Quality−enhanced surface rendering

In the following sections, each of these aspects will be described in detail. Implementation techniques are outlined using pseudo code that emphasizes the key aspects. A basic knowledge in GL programming is assumed. Application examples are taken from the chemical market. However, for the scope of this report no particular chemical background is required, since the data being analyzed can in fact be replaced by any other source of technical, scientific or engineering information processing.

Note, that this report discusses the potential of released advanced graphics technology in a very detailed fashion. The presented topics are based on recent and ongoing research and therefore subjected to change.

The methods described are the result of a team−work involving scientists from different research areas and institutions, and is called the *Texture Team,* consisting of the following members:

- Prof. Juergen Brickmann, Technische Hochschule, Darmstadt, Germany
- Dr. Peter Fluekiger, Swiss Scientific Computing Center, Manno, Switzerland
- Christian Henn, M.E. Mueller−Institute for Microscopy, Basel, Switzerland
- Dr. Michael Teschner, Silicon Graphics Marketing, Basel, Switzerland

Further support came from SGI's Advanced Graphics Division engineering group.

Colored pictures and sample code are available from sgigate.sgi.com via anonymous ftp. The files will be there starting November 1st 1993 and will be located in the directory pub/SciTex.

For more information, please contact:

Michael Teschner	*(41) 61 67 09 03*	*(phone)*
SGI Marketing, Basel	*(41) 61 67 12 01*	*(fax)*
Erlenstraesschen 65		
CH−4125 Riehen, Switzerland	*micha@basel.sgi.com*	*(email)*

1 Introduction

2 Abstract definition of the texture mapping concept

3 Color−coding based application solutions
3.1 Isocontouring on surfaces
3.2 Displaying metrics on arbitrary surfaces
3.3 Information filtering
3.4 Arbitrary surface clipping
3.5 Color−coding pseudo code example

4 Real−time volume rendering techniques
4.1 Volume rendering using 2−D textures
4.2 Volume rendering using 3−D textures

5 High quality surface rendering
5.1 Real−time Phong shading
5.1 Phong shading pseudo code example

6 Conclusions

1 Introduction

Texture mapping [1,2] has traditionally been used to add realism in computer generated images. In recent years, this technique has been transferred from the domain of software based rendering systems to a hardware supported feature of advanced graphics workstations. This was largely motivated by visual simulation and computer animation applications that use texture mapping to map pictures of surface texture to polygons of 3−D objects [3].

Thus, texture mapping is a very powerful approach to add a dramatic amount of realism to a computer generated image without blowing up the geometric complexity of the rendered scenario, which is essential in visual simulators that need to maintain a constant frame rate. E.g., a realistically looking house can be displayed using only a few polygons with photographic pictures of a wall showing doors and windows being mapped to. Similarly, the visual richness and accuracy of natural materials such as a block of wood can be improved by wrapping a wood grain pattern around a rectangular solid.

Up to now, texture mapping has not been used in technical or scientific visualization, because the above mentioned visual simulation methods as well as non−interactive rendering applications like computer animation have created a severe bias towards what texture mapping can be used for, i.e. wooden [4] or marble surfaces for the display of solid materials, or fuzzy, stochastic patterns mapped on quadrics to visualize clouds [5,6].

It will be demonstrated that hardware−supported texture mapping can be applied in a much broader range of application areas. Upon reverting to a strict and formal definition of texture mapping that generalizes the texture to be a general repository for pixel−based color information being mapped on arbitrary 3−D geometry, a powerful and elegant framework for the display and analysis of technical and scientific information is obtained.

2 Abstract definition of the texture mapping concept

In the current hardware implementation of SGI [7], texture mapping is an additional capability to modify pixel information during the rendering procedure, after the shading operations have been completed. Although it modifies pixels, its application programmers interface is vertex−based. Therefore texture mapping results in only a modest or small increase in program complexity. Its effect on the image generation time depends on the particular hardware being used: entry level and interactive systems show a significant performance reduction, whereas on third generation graphics subsystems texture mapping may be used without any performance penalty.
Three basic components are needed for the texture mapping procedure: (1) the texture, which is defined in the texture space, (2) the 3−D geometry, defined on a per vertex basis and (3) a mapping function that links the texture to the vertex description of the 3−D object.

The texture space [8,9] is a parametric coordinate space which can be 1,2 or 3 dimensional. Analogous to the pixel (picture element) in screen space, each element in texture space is called texel (texture element). Current hardware implementations offer flexibility with respect to how the information stored with each texel is interpreted. Multi−channel colors, intensity, transparency or even lookup indices corresponding to a color lookup table are supported.

In an abstract definition of texture mapping, the texture space is far more than just a picture within a parametric coordinate system: the texture space may be seen as a special memory segment, where a variety of information can be deposited which is then linked to object representations in 3−D space. Thus this information can efficiently be used to represent any parametric property that needs to be visualized.

Although the vertex−based nature of 3−D geometry in general allows primitives such as points or lines to be texture−mapped as well, the real value of texture mapping emerges upon drawing filled triangles or higher order polygons.

The mapping procedure assigns a coordinate in texture space to each vertex of the 3−D object. It is important to note that the dimensionality of the texture space is independent from the dimensionality of the displayed object. E.g., coding a simple property into a 1−D texture can be used to generate isocontour lines on arbitrary 3−D surfaces.

3 Color−coding based application solutions

Color−coding is a popular means of displaying scalar information on a surface [10]. E.g., this can be used to display stress on mechanical parts or interaction potentials on molecular surfaces.

The problem with traditional, Gouraud shading−based implementations occurs when there is a high contrast color code variation on sparsely tesselated geometry: since the color coding is done by assigning RGB color triplets to the vertices of the 3−D geometry, pixel colors will be generated by linear interpolation in RGB color space.

As a consequence, all entries in the defined color ramp laying outside the linear color ramp joining two RGB triplets are never taken into account and information will be lost. In Figure 1, a symmetric grey scale covering the property range is used to define the color ramp. On the left hand side, the interpolation in the RGB color space does not reflect the color ramp. There is a substantial loss of information during the rendering step.

With a highly tessellated surface, this problem can be reduced. An alignment of the surface vertices with the expected color code change or multi−pass rendering may remove such artifacts completely. However, these methods demand large numbers of polygons or extreme algorithmic complexity, and are therefore not suited for interactive applications.

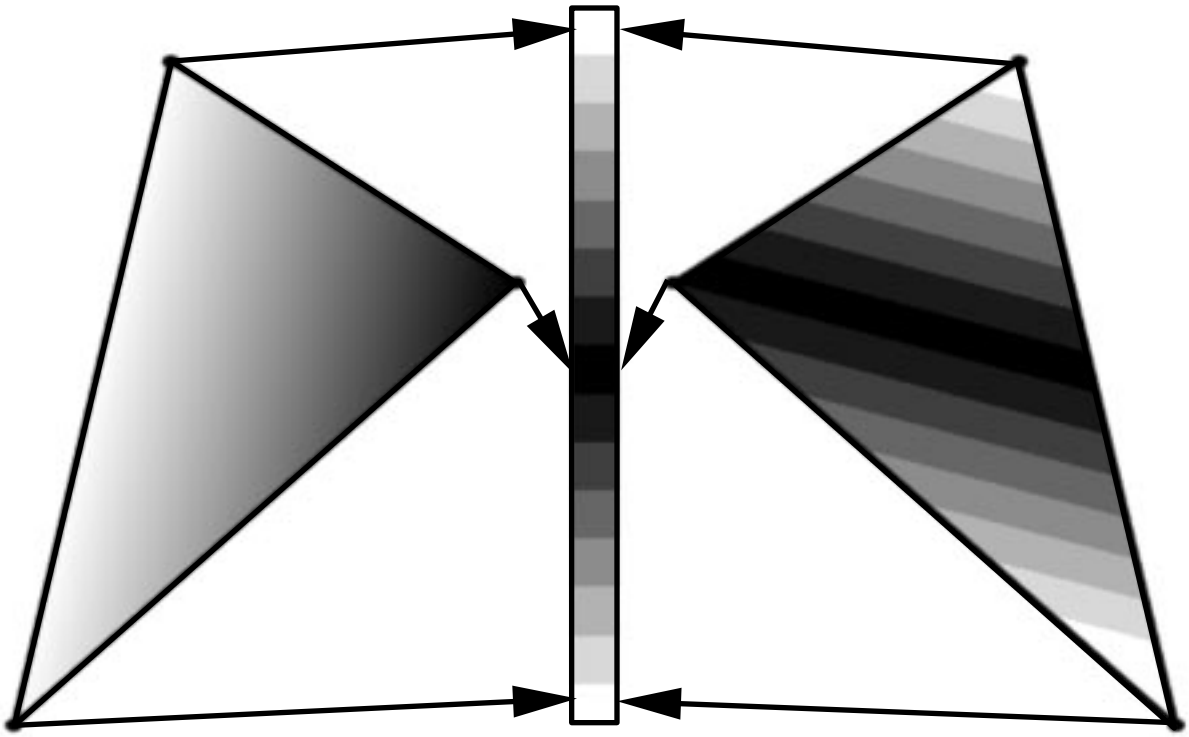


Figure 1: *Color coding with RGB interpolation (left) and texture mapping (right).*

This problem can be solved by storing the color ramp as a 1−D texture. In contrast to the above described procedure, the scalar property information is used as the texture coordinates for the surface vertices. The color interpolation is then performed in the texture space, i.e. the coloring is evaluated at every pixel (Figure 1 right). High contrast variation in the color code is now possible, even on sparsely tessellated surfaces.

It is important to note that, although the texture is one−dimensional, it is possible to tackle a 3−D problem. The dimensionality of texture space and object space is independent, thus they do not affect each other. This feature of the texture mapping method, as well as the difference between texture interpolation and color interpolation is crucial for an understanding of the applications presented in this report.

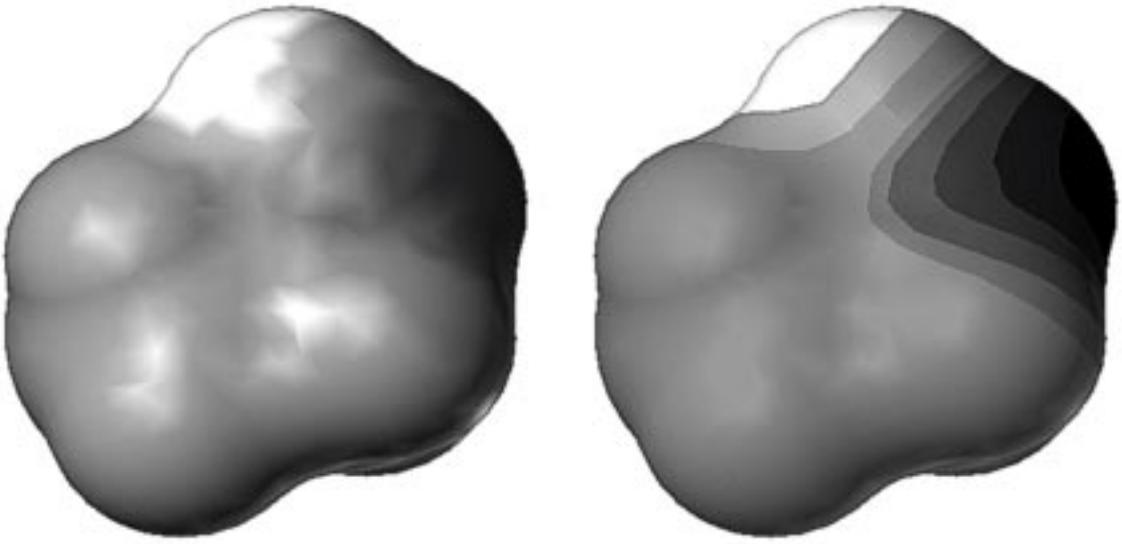


Figure 2: *Electrostatic potential coded on the solvent accessible surface of ethanol.*

Figure 2 shows the difference between the two procedures with a concrete example: the solvent accessible surface of the ethanol molecule is colored by the electrostatic surface potential, using traditional RGB color interpolation (left) and texture mapping (right).

The independence of texture and object coordinate space has further advantages and is well suited to accommodate immediate changes to the meaning of the color ramp. E.g., by applying a simple 3−D transformation like a translation in texture space the zero line of the color code may be shifted. Applying a scaling transformation to the texture adjusts the range of the mapping. Such modifications may be performed in real−time.

With texture mapping, the resulting sharp transitions from one color−value to the next significantly improves the rendering accuracy. Additionally, these sharp transitions help to visually understand the object's 3−D shape.

3.1 Isocontouring on surfaces

Similar to the color bands in general color−coding, discrete contour lines drawn on an object provide valuable information about the object's geometry as well as its properties, and are widely used in visual analysis applications. E.g., in a topographic map they might represent height above some plane that is either fixed in world coordinates or moves with the object [11]. Alternatively, the curves may indicate intrinsic surface properties, such as an interaction potential or stress distributions.

With texture mapping, discrete contouring may be achieved using the same setup as for general color coding. Again, the texture is 1−D, filled with a base color that represents the objects surface appearance. At each location of a contour threshold, a pixel is set to the color of the particular threshold. Figure 3 shows an application of this texture to display the hydrophobic potential of Gramicidine A, a channel forming molecule as a set of isocontour lines on the surface of the molecular surface.

Scaling of the texture space is used to control the spacing of contour thresholds. In a similar fashion, translation of the texture space will result in a shift of all threshold values. Note that neither the underlying geometry nor the texture itself was modified during this procedure. Adjustment of the threshold spacing is performed in real−time, and thus fully interactive.

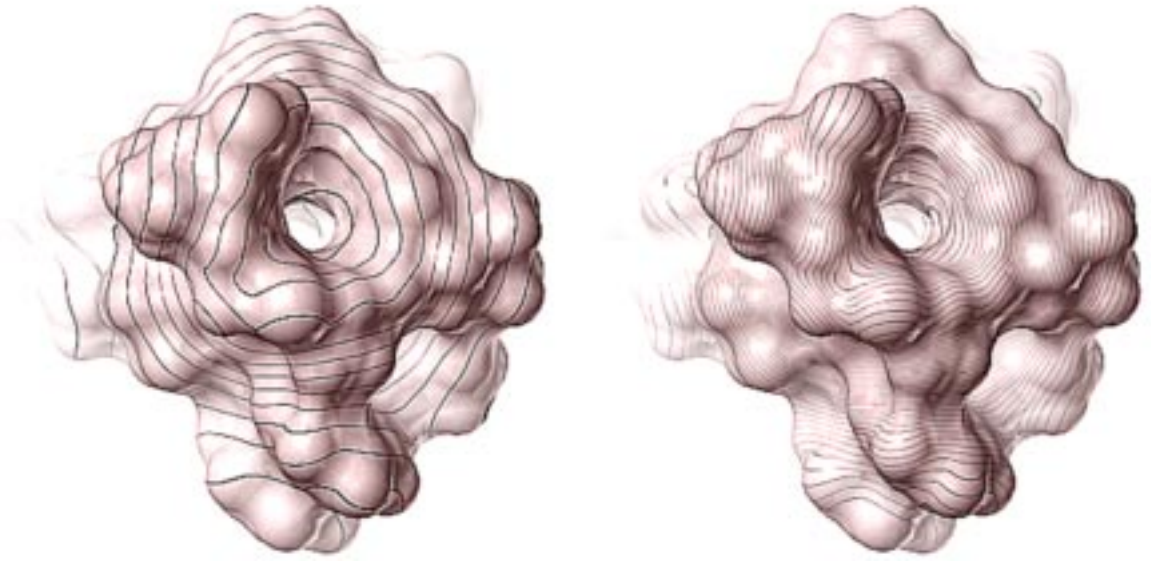


Figure 3: *Isocontour on a molecular surface with different scaling in texture space.*

3.2 Displaying metrics on arbitrary surfaces

An extension of the concept presented in the previous section can be used to display metrics on an arbitrary surface, based on a set of reference planes. Figure 4 demonstrates the application of a 2−D texture to attach tick marks on the solvent accessible surface of a zeolithe.

In contrast to the property−based, per vertex binding of texture coordinates, the texture coordinates for the metric texture are generated automatically: the distance of an object vertex to a reference plane is calculated by the harware and on−the−fly translated to texture coordinates. In this particular case two orthogonal planes are fixed to the orientation of the object's geometry. This type of representation allows for exact measurement of sizes and distance units on a surface.

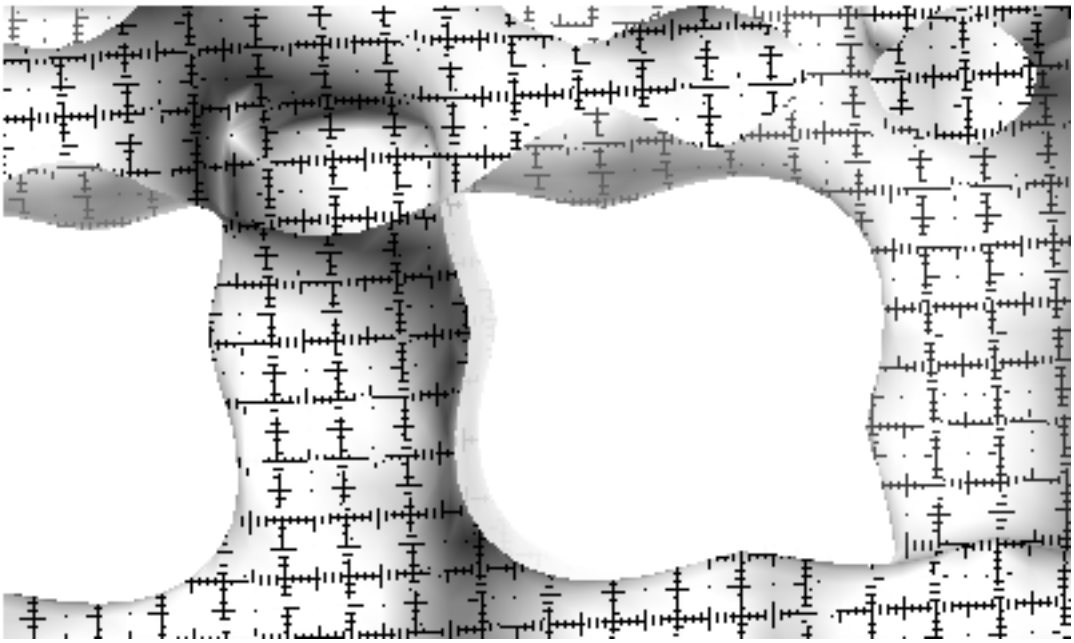


Figure 4: *Display of metrics on a Zeolithe's molecular surface with a 2−D texture.*

3.3 Information filtering

The concept of using a 1−D texture for color−coding of surface properties may be extended to 2−D or even 3−D. Thus a maximum of three independent properties can simultaneously be visualized. However, appropriate multidimensional color lookup tables must be designed based on a particular application, because a generalization is either non−trivial or eventually impossible. Special care must be taken not to overload the surface with too much information.

One possible, rather general solution can be obtained by combining a 1−D color ramp with a 1−D threshold pattern as presented in the isocontouring example, i.e. color bands are used for one property, whereas orthogonal, discrete isocontour lines code for the second property. In this way it is possible to display two properties simultaneously on the same surface, while still being capable of distinguishing them clearly.

Another approach uses one property to filter the other and display the result on the objects surface, generating additional insight in two different ways: (1) the filter allows the scientist to distinguish between important and irrelevant information, e.g. to display the hot spots on an electrostatic surface potential, or (2) the filter puts an otherwise qualitative property into a quantitative context, e.g., to use the standard deviation from a mean value to provide a hint as to how accurate a represented property actually is at a given location on the object surface.

A good role model for this is the combined display of the electrostatic potential (ESP) and the molecular lipophilic potential (MLP) on the solvent accessible surface of Gramicidine A. The electrostatic potential gives some information on how specific parts of the molecule may interact with other molecules, the molecular lipophilic potential gives a good estimate where the molecule has either contact with water (lipophobic regions) or with the membrane (lipophilic regions). The molecule itself is a channel forming protein, and is loacted in the membrane of bioorganisms, regulating the transport of water molecules and ions. Figure 5 shows the color−coding of the solvent accessible surface of Gramicidine A against the ESP filtered with the MLP. The texture used for this example is shown in Figure 8.

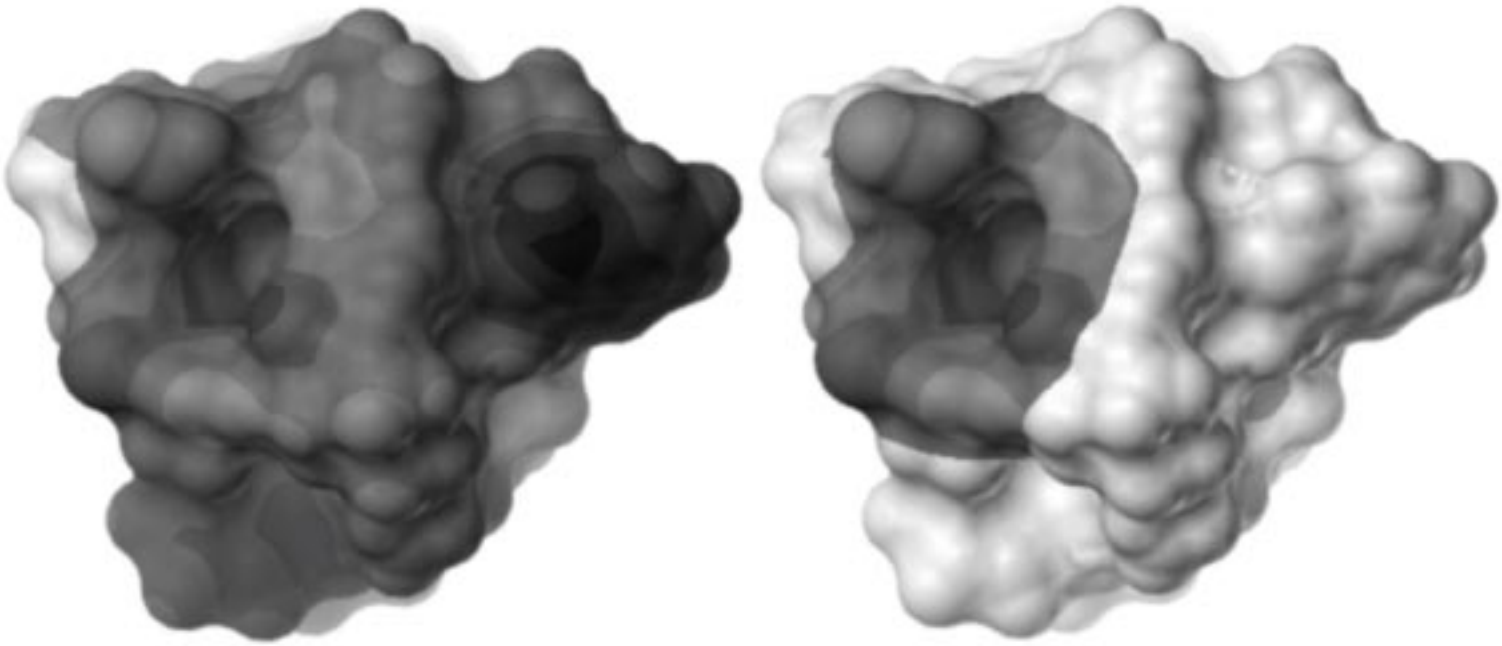


Figure 5: *Solvent accessible surface of Gramicidine A, showing the ESP filtered with the MLP.*

The surface is color−coded, or grey−scale as in the printed example, only at those loactions, where the surface has a certain lipophobicity. The surface parts with lipophilic behavior are clamped to white. In this example the information is filtered using a delta type function, suppressing all information not exceeding a specified threshold. In other cases, a continouos filter may be more appropriate, to allow a more fine grained quantification.

Another useful application is to filter the electrostatic potential with the electric fileld. Taking the absolute value of the electric field, the filter easily pinpoints the areas of the highest local field gradient, which helps in identifying the binding site of an inhibitor without further interaction of the scientist. With translation in the texture space, one can interactively modify the filter threshold or change the appearance of the color ramp.

3.4 Arbitrary surface clipping

Color−coding in the sense of information filtering affects purely the color information of the texture map. By adding transparency as an additional information channel, a lot of flexibility is gained for the comparison of multiple property channels. In a number of cases, transparency even helps in geometrically understanding of a particular property. E.g., the local flexibility of a molecule structure according to the crystallographically determined B−factors can be visually represented: the more rigid the structure is, the more opaque the surface will be displayed. Increasing transparency indicates higher floppyness of the domains. Such a transparency map may well be combined with any other color coded property, as it is of interest to study the dynamic properties of a molecule in many different contexts.

An extension to the continuous variation of surface transparency as in the example of molecular flexibility mentioned above is the use of transparency to clip parts of the surface away completely, depending on a property coded into the texture. This can be achieved by setting the alpha values at the appropriate vertices directly to zero. Applied to the information filtering example of Gramicidine A, one can just clip the surface using a texture where all alpha values in the previously white region a set to 0, as is demonstrated in Figure 6.

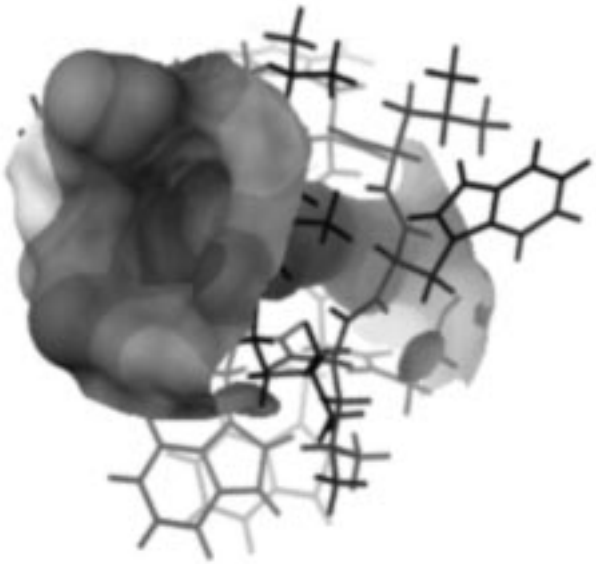


Figure 6: *Clipping of the solvent accessible surface of Gramicidine A according to the MLP.*

There is a distinct advantage in using alpha texture as a component for information filtering: irrelevant information can be completely eliminated, while geometric information otherways hidden within the surface is revealed directly in the context of the surface. And again, it is worthwhile to mention, that by a translation in texture space, the clipping range can be changed interactively!

3.5 Color−coding pseudo code example

All above described methods for property visualization on object surfaces are based upon the same texture mapping requirements. Neither are they very demanding in terms of features nor concerning the amount of texture memory needed.

Two options are available to treat texture coordinates that fall outside the range of the parametric unit square. Either the texture can be clamped to constant behaviour, or the entire texture image can be periodically repeated. In the particular examples of 2−D information filtering or property clipping, the parametric s coordinate is used to modify the threshold (clamped), and the t coordinate is used to change the appearance of the color code (repeated). Figure 7 shows different effects of transforming this texture map, while the following pseudo code example expresses the presented texture setup. GL specific calls and constants are highlighted in **boldface**:

```
texParams = {
   TX_MINIFILTER, TX_POINT,
   TX_MAGFILTER,  TX_POINT,
   TX_WRAP_S,     TX_CLAMP,
   TX_WRAP_T,     TX_REPEAT,
   TX_NULL
};

texdef2d(
   texIndex,numTexComponents,
   texWidth,texHeight,texImage,
   numTexParams,texParams
);

texbind(texIndex);
```

The texture image is an array of unsigned integers, where the packing of the data depends on the number of components being used for each texel.

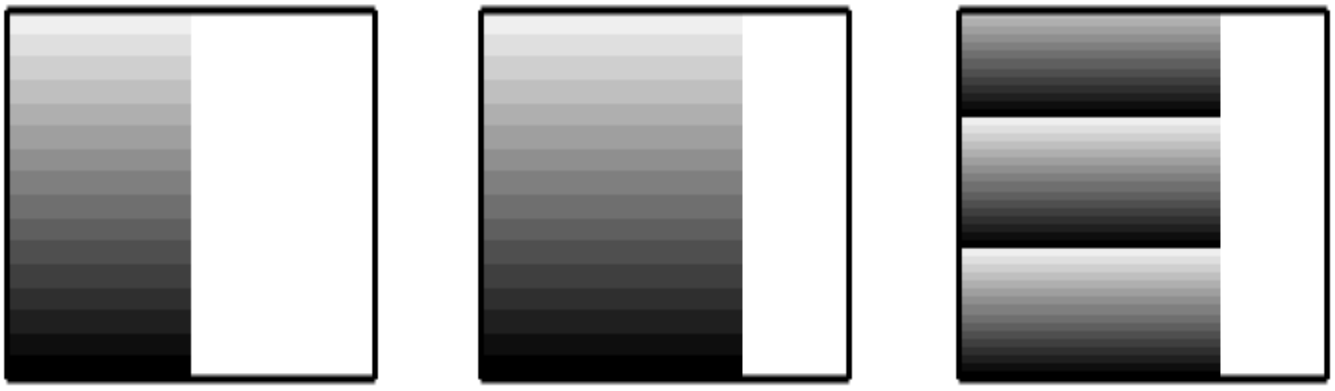


Figure 7: *Example of a 2−D texture used for information filtering, with different transformations applied: original texture (left), translation in s coordinates to adjust filter threshold (middle) and scaling along in t coordinates to change meaning of the texture colors (right).*

The texture environment defines how the texure modifies incoming pixel values. In this case we want to keep the information from the lighting calculation and modulate this with the color coming from the texture image:

```
texEnvParams = {
   TV_MODULATE, TV_NULL
};

tevdef(texEnvIndex,numTexEnvParams,texEnvParams);
tevbind(texEnvIndex);
```

Matrix transformations in texture space must be targeted to a matrix stack that is reserved for texture modifications:

```
mmode(MTEXTURE);
   translate(texTransX,0.0,0.0);
   scale(1.0,texScaleY,1.0);
mmode(MVIEWING);
```

The drawing of the object surface requires the binding of a neutral material to get a basic lighting effect. For each vertex, the coordinates, the surface normal and the texture coordinates are traversed in form of calls to **v3f**, **n3f** and **t2f**.

The **afunction()** call is only needed in the case of surface clipping. It will prevent the drawing of any part of the polygon that has a texel color with alpha = 0:

```
pushmatrix();
   loadmatrix(modelViewMatrix);
   if(clippingEnabled) {
       afunction(0,AF_NOTEQUAL);
   }
   drawTexturedSurface();
popmatrix();
```

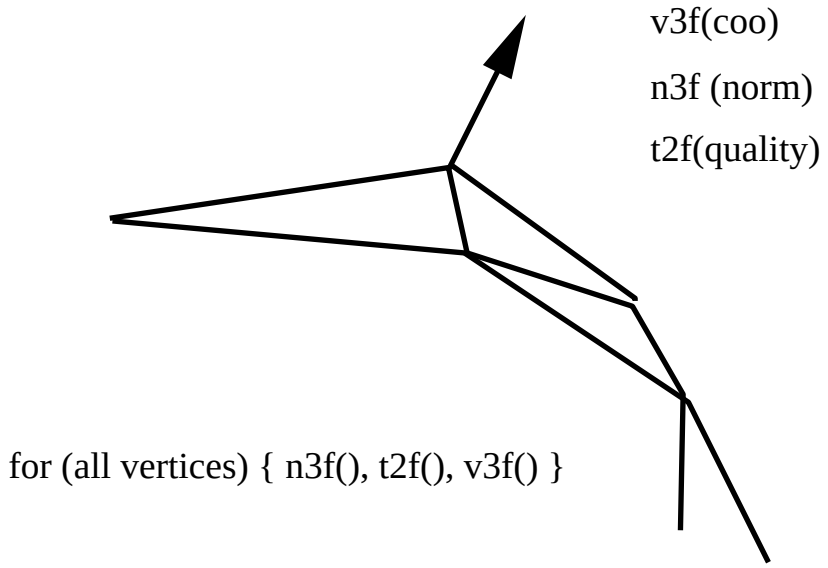


Figure 8: *Schematic representation of the* `drawTexturedSurface()` *routine.*

4 Real−time volume rendering techniques

Volume rendering is a visualization technique used to display 3−D data without an intermediate step of deriving a geometric representation like a solid surface or a chicken wire. The graphical primitives being characteristic for this technique are called voxels, derived from volume element and analog to the pixel. However, voxels describe more than just color, and in fact can represent opacity or shading parameters as well.

A variety of experimental and computational methods produce such volumetric data sets: computer tomography (CT), magnetic resonance imaging (MRI), ultrasonic imaging (UI), confocal light scanning microscopy (CLSM), electron microscopy (EM), X−ray crystallography, just to name a few. Characteristic for these data sets are a low signal to noise ratio and a large number of samples, which makes it difficult to use surface based rendering technique, both from a performance and a quality standpoint.

The data structures employed to manipulate volumetric data come in two flavours: (1) the data may be stored as a 3−D grid, or (2) it may be handled as a stack of 2−D images. The former data structure is often used for data that is sampled more or less equally in all the three dimensions, wheras the image stack is preferred with data sets that are high resolution in two dimensions and sparse in the third.

Historically, a wide variety of algorithms has been invented to render volumetric data and range from ray tracing to image compositing [12]. The methods cover an even wider range of performance, where the advantage of image compositing clearly emerges, where several images are created by slicing the volume perpendicular to the viewing axis and then combined back to front, thus summing voxel opacities and colors at each pixel.

In the majority of the cases, the volumetric information is stored using one color channel only. This allows to use lookup tables (LUTs) for alternative color interpretation. I.e., before a particular entry in the color channel is rendered to the frame buffer, the color value is interpreted as a lookup into a table that aliases the original color. By rapidly changing the color and/or opacity transfer function, various structures in the volume are interactively revealed.

By using texture mapping to render the images in the stack, a performance level is reached that is far superior to any other technique used today and allows the real−time manipulation of volumetric data. In addition, a considerable degree of flexibility is gained in performing spatial transformations to the volume, since the transformations are applied in the texture domain and cause no performance overhead.

4.1 Volume rendering using 2−D textures

As a linear extension to the original image compositing algotrithm, the 2−D textures can directly replace the images in the stack. A set of mostly quadrilateral polygons is rendered back to front, with each polygon binding its own texture if the depth of the polygon corresponds to the location of the sampled image. Alternatively, polygons inbetween may be textured in a two−pass procedure, i.e. the polygon is rendered twice, each time binding one of the two closest images as a texture and filtering it with an appropriate linear weighting factor. In this way, inbetween frames may be obtained even if the graphics subsystem doesn’t support texture interpolation in the third dimension.

The resulting volume looks correct as long as the polygons of the image stack are alligned parallel to the screen. However, it is important to be able to look at the volume from arbitrary directions. Because the polygon stack will result in a set of lines when being oriented perpendicular to the screen, a correct perception of the volume is no longer possible.

This problem can easily be soved. By preprocessing the volumetric data into three independent image stacks that are oriented perpendicular to each other, the most appropriate image stack can be selected for rendering based on the orientation of the volume object. I.e., as soon as one stack of textured polygons is rotated towards a critical viewing angle, the rendering function switches to one of the two additional sets of textured polygons, depending on the current orientation of the object.

4.2 Volume rendering using 3−D textures

As described in the previous section, it is not only possible, but almost trivial to implement real−time volume rendering using 2−D texture mapping. In addition, the graphics subsystems will operate at peak performance, because they are optimized for fast 2−D texture mapping. However, there are certain limitations to the 2−D texture approach: (1) the memory required by the triple image stack is a factor of three larger than the original data set, which can be critical for large data sets as they are common in medical imaging or microscopy, and (2) the geometry sampling of the volume must be aligned with the 2−D textures concerning the depth, i.e. arbitrary surfaces constructed from a triangle mesh can not easily be colored depending on the properties of a surrounding volume.

For this reason, advanced rendering architectures support hardware implementations of 3−D textures. The correspondence between the volume to be rendered and the 3−D texture is obvious. Any 3−D surface can serve as a sampling device to monitor the coloring of a volumetric property. I.e., the final coloring of the geometry reflects the result of the intersection with the texture. Following this principle, 3−D texture mapping is a fast, accurate and flexible technique for looking at the volume.

The simplest application of 3−D textures is that of a slice plane, which cuts in arbitrary orientations through the volume, which is now represented directly by the texture. The planar polygon being used as geometry in this case will then reflect the contents of the volume as if it were exposed by cutting the object with a knife, as shown in Figure 9: since the transformation of the sampling polygon and that of the 3−D texture is independent, it may be freely oriented within the volume. The property visualized in Figure 9 is the probability of water beeing distributed around a sugar molecule. The orientation of the volume, that means the transformation in the texture space is the same as the molecular structure. Either the molecule, together with the volumetric texture, or the slicing polygon may be reoriented in real−time.

An extension of the slice plane approach leads to complete visualization of the entire volume. A stack of slice planes, oriented in parallel to the computer screen, samples the entire 3−D texture. The planes are drawn back to front and in sufficiently small intervals. Geometric transformations of the volume are performed by manipulating the orientation of the texture, keeping the planes in screen−parallel orientation, as can be seen in Figure 10, which shows a volume rendered example of a medical application.

This type of volume visualization is greatly enhanced by interactive updates of the color lookup table used to define the texture. In fact a general purpose color ramp editor may be used to vary the lookup colors or the transparency based on the scalar information at a given point in the 3−D volume.

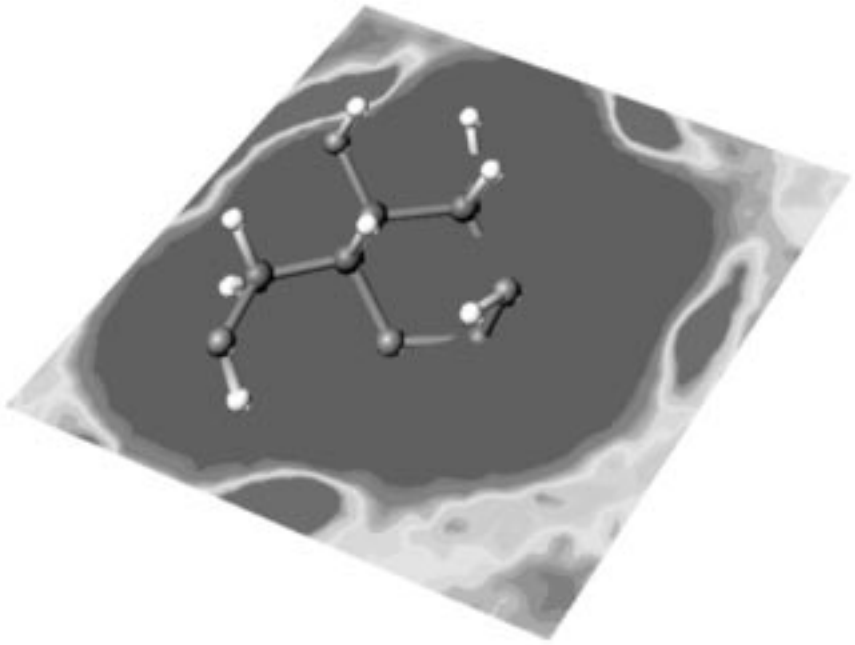


Figure 9: *Slice plane through the water density surrounding a sugar molecule.*

The slice plane concept can be extended to arbitrarily shaped objects. The idea is to probe a volumetric property and to display it wherever the geometric primitives of the probing object cut the volume. The probing geometry can be of any shape, e.g. a sphere, collecting information about the property at a certain distance from a specified point, or it may be extended to describe the surface of an arbitrary object.

The independence of the object's transformation from that of the 3−D texture, offers complete freedom in orienting the surface with respect to the volume. As a further example of a molecular modeling application, this provides an opportunity to look at a molecular surface and have the information about a surrounding volumetric property updated in real−time, based on the current orientation of the surface.

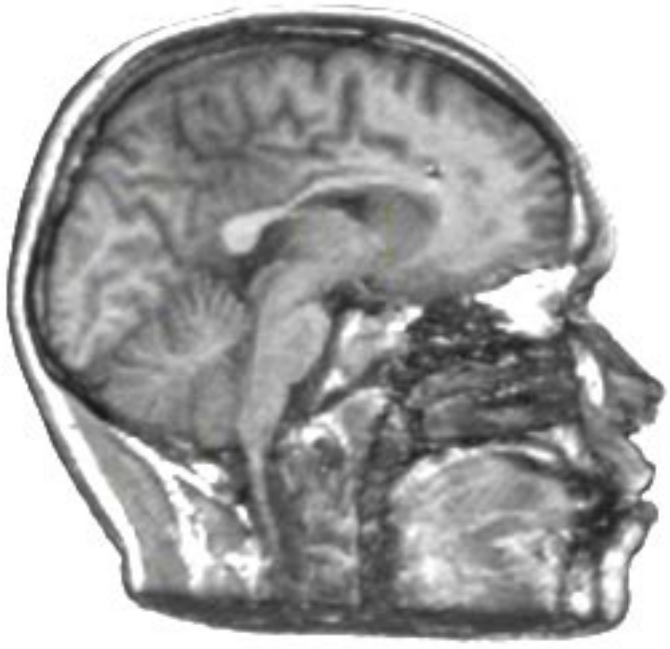


Figure 10: *Volume rendering of MRI data using a stack of screen−parallel sectioning planes, which is cut in half to reveal detail in the inner part of the object.*

5 High quality surface rendering

The visualization of solid surfaces with a high degree of local curvature is a major challenge for accurate shading, and where the simple Gouraud shading [13] approach always fails. Here, the lighting calculation is performed for each vertex, depending on the orientation of the surface normal with respect to the light sources. The output of the lighting calculations is an RGB value for the surface vertex. During rasterization of the surface polygon the color value of each pixel is computed by linear interpolation between the vertex colors. Aliasing of the surface highlight is then a consequence of undersampled surface geometry, resulting in moving Gouraud banding patterns on a surface rotating in real−time, which is very disturbing. Moreover, the missing accuracy in shading the curved surfaces often leads to a severe loss of information on the object's shape, which is not only critical for the evaluation and analysis of scientific data, but also for the visualization of CAD models, where the visual perception of shape governs the overall design process.

Figure 11 demonstrates this problem using a simple example: on the left, the sphere exhibits typical Gouraud artifacts, on the right the same sphere is shown with a superimposed mesh that reveals the tessellation of the sphere surface. Looking at these images, it is obvious how the shape of the highlight of the sphere was generated from linear interpolation. When rotating the sphere, the highlight begins to oscillate, depending on how near the surface normal at the brightest vertex is with respect to the precise highlight position.

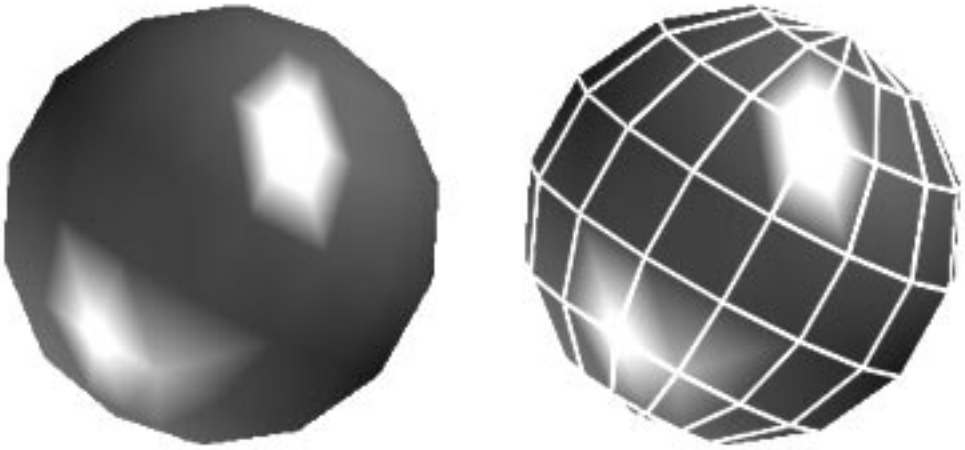


Figure 11: *Gouroud shading artifacts on a moderately tessellated sphere.*

Correct perception of the curvature and constant, non oscillating highlights can only be achieved with computationally much more demanding rendering techniques such as Phong shading [14]. In contrast to linear interpolation of vertex colors, the Phong shading approach interpolates the normal vectors for each pixel of a given geometric primitive, computing the lighting equation in the subsequent step for each pixel. Attempts have been made to overcome some of the computationally intensive steps of the procedure [15], but their performance is insufficient to be a reasonable alternative to Gouraud shading in real−time applications.

5.1 Real−time Phong shading

With 2−D texture mapping it is now possible to achieve both, high performance drawing speed and highly accurate shading. The resulting picture compares exactly to the surface computed with the complete Phong model with infinite light sources.

The basic idea is to use the image of a high quality rendered sphere as texture. The object's unit length surface normal is interpreted as texture coordinate. Looking at an individual triangle of the polygonal surface, the texture mapping process may be understood as if the image of the perfectly rendered sphere would be wrapped piecewise on the surface polygons. In other words, the surface normal serves as a lookup vector into the texture, acting as a 2−D lookup table that stores precalculated shading information.

The advantage of such a shading procedure is clear: the interpolation is done in texture space and not in RGB, therefore the position of the highlight will never be missed. Note that the tessellation of the texture mapped sphere is exactly the same as for the Gouraud shaded reference sphere in Figure 11.

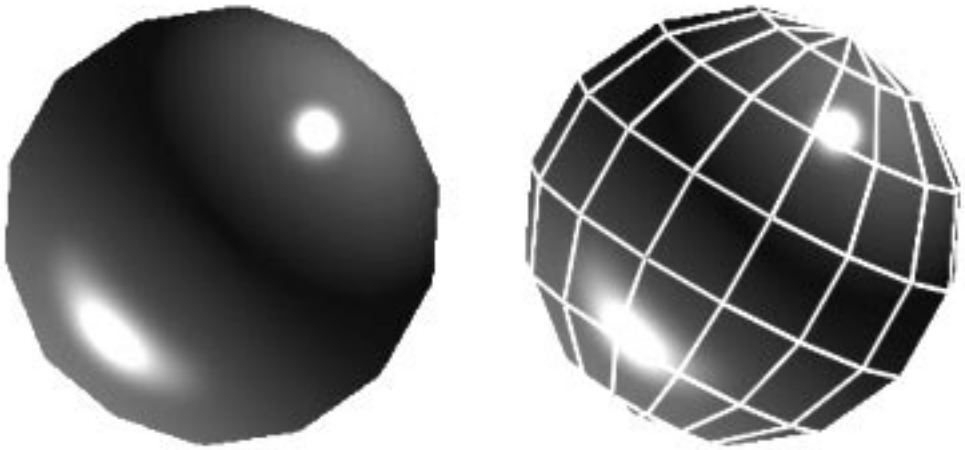


Figure 12: *Phong shaded sphere using surface normals as a lookup for the texture coordinate.*

As previously mentioned, this method of rendering solid surfaces with highest accuracy can be applied to arbitrarily shaped objects. Figure 13 shows the 3−D reconstruction of an electron microscopic experiment, visualizing a large biomolecular complex, the asymmetric unit membrane of the urinary bladder. The difference between Gouraud shading and the texture mapping implementation of Phong shading is obvious, and for the sake of printing quality, can be seen best when looking at the closeups. Although this trick is so far only applicable for infinitely distant light sources, it is a tremendous aid for the visualization of highly complex surfaces.

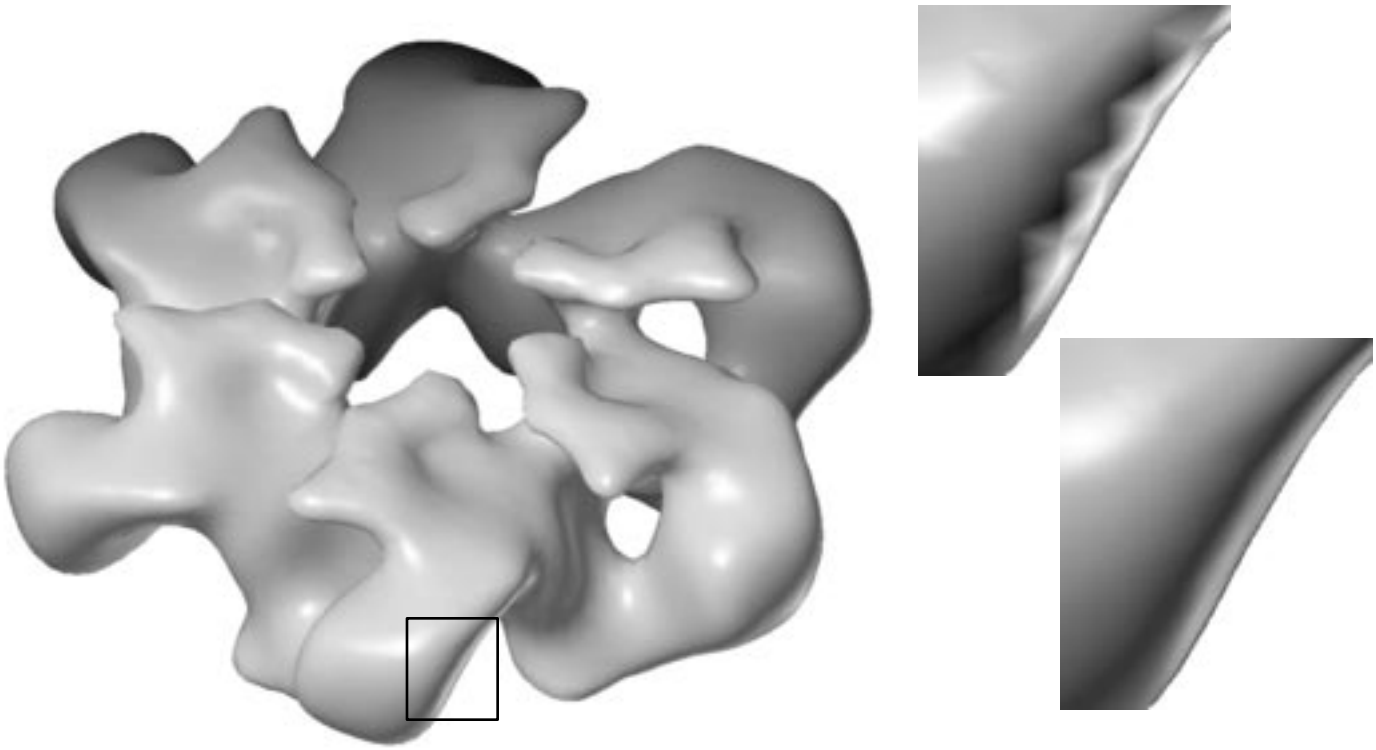


Figure 13: *Application of the texture mapped Phong shading to a complex surface representing a biomolecular structure. The closeups demonstrate the difference between Gouraud shading (above right) and Phong shading (below right) when implemented using texture mapping*

5.2 Phong shading pseudo code example

The setup for the texture mapping as used for Phong shading is shown in the following code fragment:

```
texParams = {
   TX_MINIFILTER, TX_POINT,
   TX_MAGFILTER,  TX_BILINEAR,
   TX_NULL
};

texdef2d(
   texIndex,numTexComponents,
   texWidth,texHeight,texImage,
   numTexParams,texParams
);
```

```
texbind(texIndex);

texEnvParams = { TV_MODULATE, TV_NULL };

tevdef(texEnvIndex,numTexEnvParams,texEnvParams);
tevbind(texEnvIndex);
```

As texture, we can use any image of a high quality rendered sphere either with RGB or one intensity component only. The RGB version allows the simulation of light sources with different colors.

The most important change for the vertex calls in this model is that we do not pass the surface normal data with the **n3f** command as we normally do when using Gouraud shading. The normal is passed as texture coordinate and therefore processed with the **t3f** command.

Surface normals are transformed with the current model view matrix, although only rotational components are considered. For this reason the texture must be aligned with the current orientation of the object. Also, the texture space must be scaled and shifted to cover a circle centered at the origin of the s/t coordinate system, with a unit length radius to map the surface normals:

```
mmode(MTEXTURE);
   loadmatrix(identityMatrix);
   translate(0.5,0.5,0.0);
   scale(0.5,0.5,1.0);
   multmatrix(rotationMatrix);
mmode(MVIEWING);

drawTexPhongSurface();
```

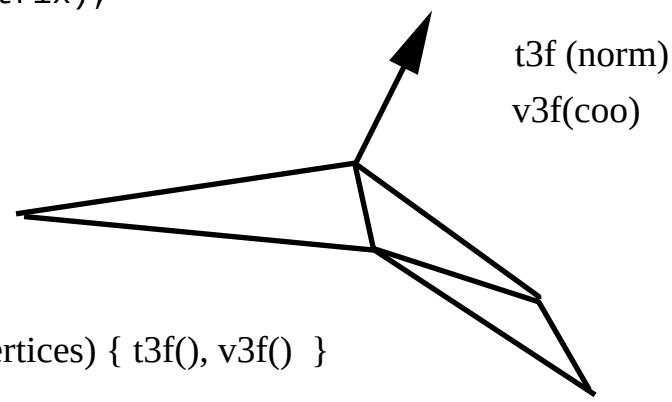


Figure 15: *Schematic representation of the* `drawTexPhongSurface()` *routine.*

6 Conclusions

Silicon Graphics has recently introduced a new generation of graphics subsystems, which support a variety of texture mapping techniques in hardware without performance penalty. The potential of using this technique in technical, scientific and engineering visualization applications has been demonstrated.

Hardware supported texture mapping offers solutions to important visualization problems that have either not been solved yet or did not perform well enough to enter the world of interactive graphics applications. Although most of the examples presented here could be implemented using techniques other than texture mapping, the tradeoff would either be complete loss of performance or an unmaintainable level of algorithmic complexity.

Most of the examples were taken from the molecular modelling market, where one has learned over the

years to handle complex 3−D scenarios interactively and in an analytic manner. What has been shown here can also be applied in other areas of scientific, technical or engineering visualization. With the examples shown in this report, it should be possible for software engineers developing application software in other markets to use the power and flexibility of texture mapping and to adapt the shown solutions to their specific case.

One important, general conclusion may be drawn from this work: one has to leave the traditional mind set about texture mapping and go back to the basics in order to identify the participating components and to understand their generic role in the procedure. Once this step is done it is very simple to use this technique in a variety of visualization problems.

All examples were implemented on a Silicon Graphics Crimson Reality Engine [7] equipped with two raster managers. The programs were written in C, either in mixed mode GLX or pure GL.

7 References

[1] Blinn, J.F. and Newell, M.E. *Texture and reflection in computer generated images,* Communications of the ACM 1976, **19**, 542−547.

[2] Blinn, J.F. *Simulation of wrinkled surfaces* Computer Graphics 1978, **12**, 286−292.

[3] Haeberli, P. and Segal, M. *Texture mapping as a fundamental drawing primitive,* Proceedings of the fourth eurographics workshop on rendering, 1993, 259−266.

[4] Peachy, D.R. *Solid texturing of complex surfaces*, Computer Graphics 1985, **19**, 279−286.

[5] Gardner, G.Y. *Simulation of natural scenes using textured quadric surfaces,* Computer Graphics 1984, **18**, 11−20.

[6] Gardner, G.Y. *Visual simulations of clouds*, Computer Graphics 1985, **19,** 279−303.

[7] Akeley, K. *Reality Engine Graphics*, Computer Graphics 1993, **27**, 109−116.

[8] Catmull, E.A. *Subdivision algorithm for computer display of curved surfaces,* Ph.D. thesis University of Utah, 1974.

[9] Crow, F.C. *Summed−area tables for texture mapping*, Computer Graphics 1984, **18**, 207−212.

[10] Dill, J.C. *An application of color graphics to the display of surface curvature*, Computer Graphics 1981, **15**, 153−161.

[11] Sabella, P. *A rendering algorithm for visualizing 3d scalar fields,* Computer Graphics, 1988 **22**, 51−58.

[12] Drebin, R. Carpenter, L. and Hanrahan, P. *Volume Rendering,* Computer Graphics, 1988, **22**, 65−74.

[13] Gouraud, H. *Continuous shading of curved surfaces*, IEEE Transactions on Computers, 1971, **20**, 623−628.

[14] Phong, B.T. *Illumination for computer generated pictures,* Communications of the ACM 1978, **18**, 311−317.

[15] Bishop, G. and Weimer, D.M. *Fast Phong shading*, Computer Graphics, 1986, **20**, 103−106.

Appeared in Proc. Fourth Eurographics Workshop on Rendering,
Michael Cohen, Claude Puech, and Francois Sillion, eds.
Paris, France, June, 1993. pp. 259-266.

Texture Mapping as a Fundamental Drawing Primitive

Paul Haeberli
Mark Segal
Silicon Graphics Computer Systems*

Abstract

Texture mapping has traditionally been used to add realism to computer graphics images. In recent years, this technique has moved from the domain of software rendering systems to that of high performance graphics hardware.

But texture mapping hardware can be used for many more applications than simply applying diffuse patterns to polygons.

We survey applications of texture mapping including simple texture mapping, projective textures, and image warping. We then describe texture mapping techniques for drawing anti-aliased lines, air-brushes, and anti-aliased text. Next we show how texture mapping may be used as a fundamental graphics primitive for volume rendering, environment mapping, color interpolation, contouring, and many other applications.

CR Categories and Subject Descriptors: I.3.3 **[Computer Graphics]:** Picture/Image Generation; I.3.7 **[Computer Graphics]:** Three-Dimensional Graphics and Realism - *color, shading, shadowing, texture-mapping, line drawing, and anti-aliasing*

1 Introduction

Texture mapping[Cat74][Hec86] is a powerful technique for adding realism to a computer-generated scene. In its basic form, texture mapping lays an image (the texture) onto an object in a scene. More general forms of texture mapping generalize the image to other information; an "image" of altitudes, for instance, can be used to control shading across a surface to achieve such effects as bump-mapping.

Because texture mapping is so useful, it is being provided as a standard rendering technique both in graphics software interfaces and in computer graphics hardware[HL90][DWS+88]. Texture mapping can therefore be used in a scene with only a modest increase in the complexity of the program that generates that scene, sometimes with little effect on scene generation time. The wide availability and high-performance of texture mapping makes it a desirable rendering technique for achieving a number of effects that are normally obtained with special purpose drawing hardware.

After a brief review of the mechanics of texture mapping, we describe a few of its standard applications. We go on to describe some novel applications of texture mapping.

2 Texture Mapping

When mapping an image onto an object, the color of the object at each pixel is modified by a corresponding color from the image. In general, obtaining this color from the image conceptually requires several steps[Hec89]. The image is normally stored as a sampled array, so a continuous image must first be reconstructed from the samples. Next, the image must be warped to match any distortion (caused, perhaps, by perspective) in the projected object being displayed. Then this warped image is filtered to remove high-frequency components that would lead to aliasing in the final step: resampling to obtain the desired color to apply to the pixel being textured.

In practice, the required filtering is approximated by one of several methods. One of the most popular is *mipmapping*[Wil83]. Other filtering techniques may also be used[Cro84].

There are a number of generalizations to this basic texture mapping scheme. The image to be mapped need not be two-dimensional; the sampling and filtering techniques may be applied for both one- and three-dimensional images[Pea85]. In the case of a three-dimensional image, a two-dimensional slice must be selected to be mapped onto an object's boundary, since the result of rendering must be two-dimensional. The

*2011 N. Shoreline Blvd., Mountain View, CA 94043 USA

image may not be stored as an array but may be procedurally generated[Pea85][Per85]. Finally, the image may not represent color at all, but may instead describe transparency or other surface properties to be used in lighting or shading calculations[CG85].

3 Previous Uses of Texture Mapping

In basic texture mapping, an image is applied to a polygon (or some other surface facet) by assigning texture coordinates to the polygon's vertices. These coordinates index a texture image, and are interpolated across the polygon to determine, at each of the polygon's pixels, a texture image value. The result is that some portion of the texture image is mapped onto the polygon when the polygon is viewed on the screen. Typical two-dimensional images in this application are images of bricks or a road surface (in this case the texture image is often repeated across a polygon); a three-dimensional image might represent a block of marble from which objects could be "sculpted."

3.1 Projective Textures

A generalization of this technique projects a texture onto surfaces as if the texture were a projected slide or movie[SKvW+92]. In this case the texture coordinates at a vertex are computed as the result of the projection rather than being assigned fixed values. This technique may be used to simulate spotlights as well as the reprojection of a photograph of an object back onto that object's geometry.

Projective textures are also useful for simulating shadows. In this case, an image is constructed that represents distances from a light source to surface points nearest the light source. This image can be computed by performing z-buffering from the light's point of view and then obtaining the resulting z-buffer. When the scene is viewed from the eyepoint, the distance from the light source to each point on a surface is computed and compared to the corresponding value stored in the texture image. If the values are (nearly) equal, then the point is not in shadow; otherwise, it is in shadow. This technique should not use mipmapping, because filtering must be applied *after* the shadow comparison is performed[RSC87].

3.2 Image Warping

Image warping may be implemented with texture mapping by defining a correspondence between a uniform polygonal mesh (representing the original image) and a warped mesh (representing the warped image)[OTOK87]. The warp may be affine (to generate rotations, translations, shearings, and zooms) or higher-order. The points of the warped mesh are assigned the corresponding texture coordinates of the uniform mesh, and the mesh is texture mapped with the original image. This technique allows for easily-controlled interactive image warping. The technique can also be used for panning across a large texture image by using a mesh that indexes only a portion of the entire image.

3.3 Transparency Mapping

Texture mapping may be used to lay transparent or semi-transparent objects over a scene by representing transparency values in the texture image as well as color values. This technique is useful for simulating clouds[Gar85] and trees for example, by drawing appropriately textured polygons over a background. The effect is that the background shows through around the edges of the clouds or branches of the trees. Texture map filtering applied to the transparency and color values automatically leads to soft boundaries between the clouds or trees and the background.

3.4 Surface Trimming

Finally, a similar technique may be used to cut holes out of polygons or perform domain space trimming on curved surfaces[Bur92]. An image of the domain space trim regions is generated. As the surface is rendered, its domain space coordinates are used to reference this image. The value stored in the image determines whether the corresponding point on the surface is trimmed or not.

4 Additional Texture Mapping Applications

Texture mapping may be used to render objects that are usually rendered by other, specialized means. Since it is becoming widely available, texture mapping may be a good choice to implement these techniques even when these graphics primitives can be drawn using special purpose methods.

4.1 Anti-aliased Points and Line Segments

One simple use of texture mapping is to draw anti-aliased points of any width. In this case the texture image is of a filled circle with a smooth (anti-aliased) boundary. When a point is specified, it's coordinates indicate the center of a square whose width is determined by the point size. The texture coordinates at the

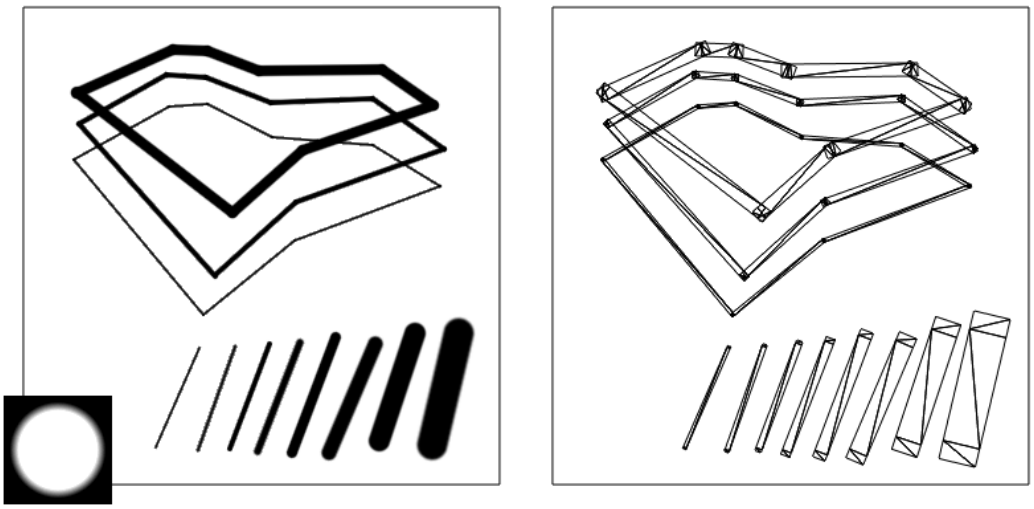


Figure 1. Anti-aliased line segments.

square's corners are those corresponding to the corners of the texture image. This method has the advantage that any point shape may be accommodated simply by varying the texture image.

A similar technique can be used to draw anti-aliased, line segments of any width[Gro90]. The texture image is a filtered circle as used above. Instead of a line segment, a texture mapped rectangle, whose width is the desired line width, is drawn centered on and aligned with the line segment. If line segments with round ends are desired, these can be added by drawing an additional textured rectangle on each end of the line segment (Figure 1).

4.2 Air-brushes

Repeatedly drawing a translucent image on a background can give the effect of spraying paint onto a canvas. Drawing an image can be accomplished by drawing a texture mapped polygon. Any conceivable brush "footprint", even a multi-colored one, may be drawn using an appropriate texture image with red, green, blue, and alpha. The brush image may also easily be scaled and rotated (Figure 2).

4.3 Anti-aliased Text

If the texture image is an image of a character, then a polygon textured with that image will show that character on its face. If the texture image is partitioned into an array of rectangles, each of which contains the image of a different character, then any character may be displayed by drawing a polygon with appropriate texture coordinates assigned to its vertices. An advantage of this method is that strings of characters may be arbitrarily positioned and oriented in three dimensions by appropriately positioning and orienting the textured polygons. Character kerning is accomplished simply by positioning the polygons relative to one another (Figure 3).

Antialiased characters of any size may be obtained with a single texture map simply by drawing a polygon of the desired size, but care must be taken if mipmapping is used. Normally, the smallest mipmap is 1 pixel square, so if all the characters are stored in a single texture map, the smaller mipmaps will contain a number of characters filtered together. This will generate undesirable effects when displayed characters are too small. Thus, if a single texture image is used for all characters, then each must be carefully placed in the image, and mipmaps must stop at the point where the image of a single character is reduced to 1 pixel on a side. Alternatively, each character could be placed in its own (small) texture map.

4.4 Volume Rendering

There are three ways in which texture mapping may be used to obtain an image of a solid, translucent object. The first is to draw slices of the object from back to front[DCH88]. Each slice is drawn by first generating a texture image of the slice by sampling the data representing the volume along the plane of the slice, and then drawing a texture mapped polygon to produce the slice. Each slice is blended with the previously drawn slices using transparency.

The second method uses 3D texture mapping[Dre92]. In this method, the volumetric data is copied into the 3D texture image. Then, slices perpendicular to the viewer are drawn. Each slice is again a texture mapped

Figure 2. Painting with texture maps.

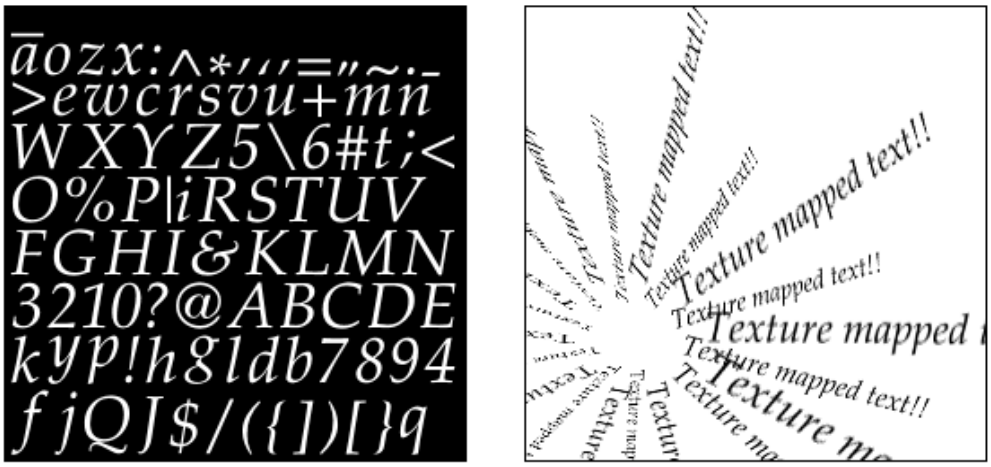


Figure 3. Anti-aliased text.

polygon, but this time the texture coordinates at the polygon's vertices determine a slice through the 3D texture image. This method requires a 3D texture mapping capability, but has the advantage that texture memory need be loaded only once no matter what the viewpoint. If the data are too numerous to fit in a single 3D image, the full volume may be rendered in multiple passes, placing only a portion of the volume data into the texture image on each pass.

A third way is to use texture mapping to implement "splatting" as described by[Wes90][LH91].

4.5 Movie Display

Three-dimensional texture images may also be used to display animated sequences[Ake92]. Each frame forms one two-dimensional slice of a three-dimensional texture. A frame is displayed by drawing a polygon with texture coordinates that select the desired slice. This can be used to smoothly interpolate between frames of the stored animation. Alpha values may also be associated with each pixel to make animated "sprites".

4.6 Contouring

Contour curves drawn on an object can provide valuable information about the object's geometry. Such curves may represent height above some plane (as in a topographic map) that is either fixed or moves with the object[Sab88]. Alternatively, the curves may indicate intrinsic surface properties, such as geodesics or loci of constant curvature.

Contouring is achieved with texture mapping by first defining a one-dimensional texture image that is of con-

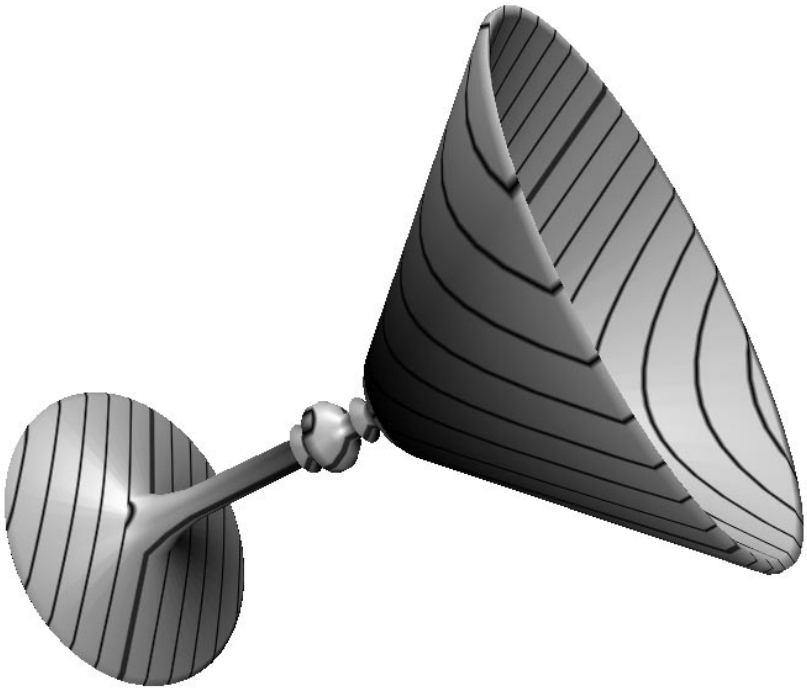


Figure 4. Contouring showing distance from a plane.

stant color except at some spot along its length. Then, texture coordinates are computed for vertices of each polygon in the object to be contoured using a *texture coordinate generation function*. This function may calculate the distance of the vertex above some plane (Figure 4), or may depend on certain surface properties to produce, for instance, a curvature value. Modular arithmetic is used in texture coordinate interpolation to effectively cause the single linear texture image to repeat over and over. The result is lines across the polygons that comprise an object, leading to contour curves.

A two-dimensional (or even three-dimensional) texture image may be used with two (or three) texture coordinate generation functions to produce multiple curves, each representing a different surface characteristic.

4.7 Generalized Projections

Texture mapping may be used to produce a non-standard projection of a three-dimensional scene, such as a cylindrical or spherical projection[Gre86]. The technique is similar to image warping. First, the scene is rendered six times from a single viewpoint, but with six distinct viewing directions: forward, backward, up, down, left, and right. These six views form a cube enclosing the viewpoint. The desired projection is formed by projecting the cube of images onto an array of polygons (Figure 5).

4.8 Color Interpolation in non-RGB Spaces

The texture image may not represent an image at all, but may instead be thought of as a lookup table. Intermediate values not represented in the table are obtained through linear interpolation, a feature normally provided to handle image filtering.

One way to use a three-dimensional lookup table is to fill it with RGB values that correspond to, for instance, HSV (Hue, Saturation, Value) values. The H, S, and V values index the three dimensional tables. By assigning HSV values to the vertices of a polygon linear color interpolation may be carried out in HSV space rather than RGB space. Other color spaces are easily supported.

4.9 Phong Shading

Phong shading with an infinite light and a local viewer may be simulated using a 3D texture image as follows. First, consider the function of x, y, and z that assigns a brightness value to coordinates that represent a (not necessarily unit length) vector. The vector is the reflection off of the surface of the vector from the eye to a point on the surface, and is thus a function of the normal at that point. The brightness function depends on the location of the light source. The 3D texture image is a lookup table for the brightness function given a reflection vector. Then, for each polygon in the scene, the reflection vector is computed at each of the polygon's vertices. The coordinates of this vector are interpolated

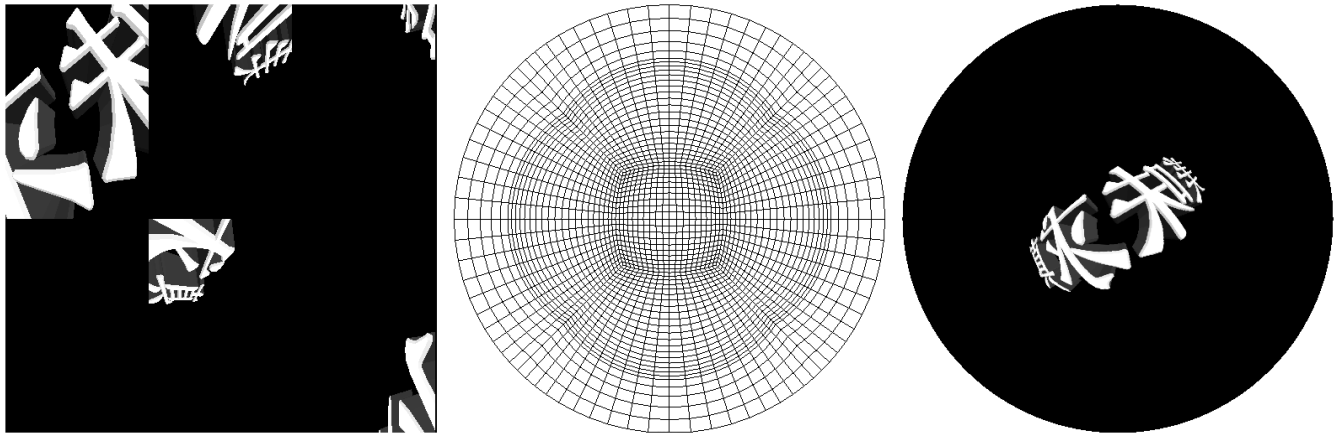


Figure 5. 360 Degree fisheye projection.

across the polygon and index the brightness function stored in the texture image. The brightness value so obtained modulates the color of the polygon. Multiple lights may be obtained by incorporating multiple brightness functions into the texture image.

4.10 Environment Mapping

Environment mapping[Gre86] may be achieved through texture mapping in one of two ways. The first way requires six texture images, each corresponding to a face of a cube, that represent the surrounding environment. At each vertex of a polygon to be environment mapped, a reflection vector from the eye off of the surface is computed. This reflection vector indexes one of the six texture images. As long as all the vertices of the polygon generate reflections into the same image, the image is mapped onto the polygon using projective texturing. If a polygon has reflections into more than one face of the cube, then the polygon is subdivided into pieces, each of which generates reflections into only one face. Because a reflection vector is not computed at each pixel, this method is not exact, but the results are quite convincing when the polygons are small.

The second method is to generate a single texture image of a perfectly reflecting sphere in the environment. This image consists of a circle representing the hemisphere of the environment behind the viewer, surrounded by an annulus representing the hemisphere in front of the viewer. The image is that of a perfectly reflecting sphere located in the environment when the viewer is infinitely far from the sphere. At each polygon vertex, a texture coordinate generation function generates coordinates that index this texture image, and these are interpolated across the polygon. If the (normalized) reflection vector at a vertex is $r = (\ x \quad y \quad z\ )$, and $m = \sqrt{2(z+1)}$, then the generated coordinates are x/m and y/m when the texture image is indexed

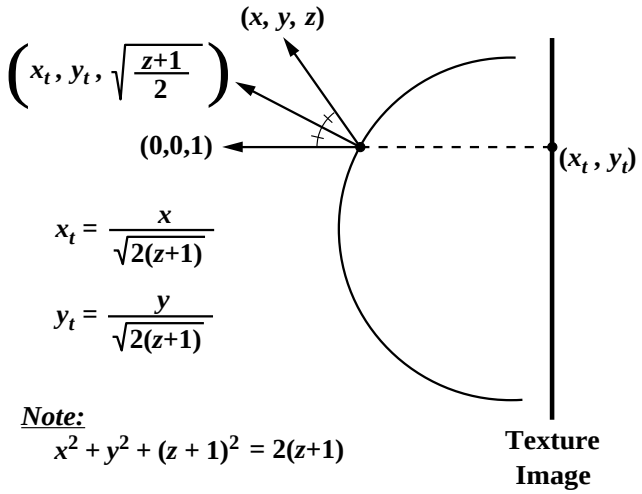


Figure 6. Spherical reflection geometry.

by coordinates ranging from -1 to 1. (The calculation is diagrammed in Figure 6). This method has the disadvantage that the texture image must be recomputed whenever the view direction changes, but requires only a single texture image with no special polygon subdivision (Figure 7).

4.11 3D Halftoning

Normal halftoned images are created by thresholding a source image with a halftone screen. Usually this halftone pattern of lines or dots bears no direct relationship to the geometry of the scene. Texture mapping allows halftone patterns to be generated using a 3D spatial function or parametric lines of a surface (Figure 8). This permits us to make halftone patterns that are bound to the surface geometry[ST90].

Figure 7. Environment mapping.

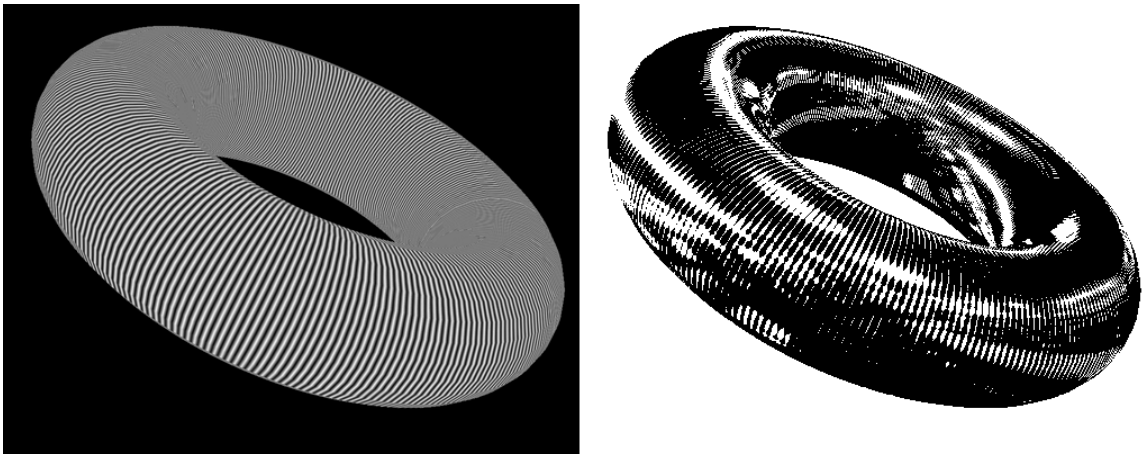


Figure 8. 3D halftoning.

5 Conclusion

Many graphics systems now provide hardware that supports texture mapping. As a result, generating a texture mapped scene need not take longer than generating a scene without texture mapping.

We have shown that, in addition to its standard uses, texture mapping can be used for a large number of interesting applications, and that texture mapping is a powerful and flexible low level graphics drawing primitive.

References

[Ake92] Kurt Akeley. Personal Communication, 1992.

[Bur92] Derrick Burns. Personal Communication, 1992.

[Cat74] Ed Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[CG85] Richard J. Carey and Donald P. Greenberg. Textures for realistic image synthesis. *Computers & Graphics*, 9(3):125–138, 1985.

[Cro84] F. C. Crow. Summed-area tables for texture mapping. *Computer Graphics (SIGGRAPH '84 Proceedings)*, 18:207–212, July 1984.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):65–74, August 1988.

[Dre92] Bob Drebin. Personal Communication, 1992.

[DWS+88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):21–30, August 1988.

[Gar85] G. Y. Gardner. Visual simulation of clouds. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):297–303, July 1985.

[Gre86] Ned Greene. Applications of world projections. *Proceedings of Graphics Interface '86*, pages 108–114, May 1986.

[Gro90] Mark Grossman. Personal Communication, 1990.

[Hec86] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, November 1986.

[Hec89] Paul S. Heckbert. Fundamentals of texture mapping and image warping. M.sc. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, June 1989.

[HL90] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):289–298, August 1990.

[LH91] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):285–288, July 1991.

[OTOK87] Masaaki Oka, Kyoya Tsutsui, Akio Ohba, and Yoshitaka Kurauchi. Real-time manipulation of texture-mapped surfaces. *Computer Graphics (Proceedings of SIGGRAPH '87)*, July 1987.

[Pea85] D. R. Peachey. Solid texturing of complex surfaces. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):279–286, July 1985.

[Per85] K. Perlin. An image synthesizer. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):287–296, July 1985.

[RSC87] William Reeves, David Salesin, and Rob Cook. Rendering antialiased shadows with depth maps. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):283–291, July 1987.

[Sab88] Paolo Sabella. A rendering algorithm for visualizing 3d scalar fields. *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22(4):51–58, August 1988.

[SKvW+92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July 1992.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):197–206, August 1990.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics (SIGGRAPH '90 Proceedings)*, 24(4):367–376, August 1990.

[Wil83] Lance Williams. Pyramidal parametrics. *Computer Graphics (SIGGRAPH '83 Proceedings)*, 17(3):1–11, July 1983.